Final Deliverable

Michael Chladon, Alexander Kluskens, Teddy Pugh, Jacob Zorniak

April 30, 2020

1 Introduction

Pathfinding is a popular area of study in the field of computer science with many applications. These applications include navigation software, video games and large data graph traversals. Our goal for this project was to use the knowledge we have gained in this course to attempt to parallelize four existing pathfinding algorithms and analyze their runtime characteristics. The four pathfinding algorithms we will be analyzing are A*, Dijkstra's, JPS, and HPA*.

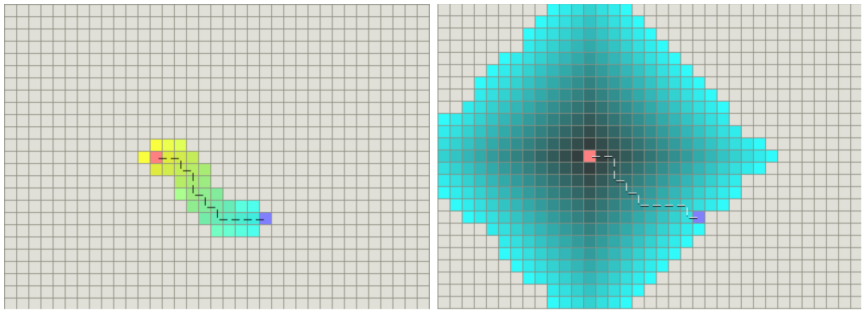
Our repository can be found here.
https://git.cs.jmu.edu/chladomj/cs470-pathfinding-research-project

2 A*

2.1 Background

A* is a pathfinding algorithm that is a common choice for many applications of pathfinding, specifically video games. It is considered to be a very efficient pathfinding algorithm. Our goal in this research project was to come up with a method of parallelizing A*, which would allow all of its current applications to be improved in their efficiency.

A* works by building a list of moves based on what appears to be the next cheapest step towards the end goal. It is very similar to Dijkstra's algorithm in functionality. It essentially follows a similar series of steps to Dijkstra's with an added component of cost to each move. This cost of each move can be determined using a few different equations depending upon the defined ability to traverse a path. For example, if the algorithm is considering all 8 tiles (adjacent and diagonal), then the algorithm uses the distance formula to determine the cost. However if the algorithm is only considering adjacent movements, the cost value is to be determined by the Manhattan Distance (sum of the difference in start to end x values and y values). Using this cost allows the algorithm to make "intelligent" choices and avoid going down paths that would often waste time using Dijkstra's algorithm. Below are pictures representing a search done by each algorithm which highlights the efficiency of A*, with A* on the left and Dijkstra's on the right.



2.2 Methods

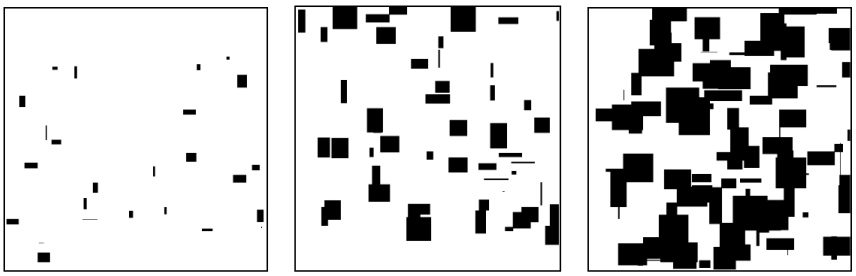
We began with an implementation from an educational article about A*. Here there was a community discussion on the algorithm. A community member by the name of "priya" had simplified the original implementation to consider only the adjacent tiles rather than both adjacent and diagonal tiles. Because this was our group's first experience using C++ and first experience with the algorithm, a simplified version was more ideal than the original implementation from the website.

We settled on using OpenMP for parallelization of the algorithm. The code has two main aspects that need to be considered for parallelization. One is the loop that considers each node on the path, the other is the loop that considers each adjacent node to the current one. The outer loop however does not lend itself to parallelization due to depending on other nodes along the path. When calculating the cost of each move it needs to consider the costs of alternative paths, meaning that there is a loop carried dependence. This leaves only the inner loop to be parallelized. This loop will always run 4 times when searching each of the possible adjacent nodes. This is not ideal for parallelization as this means there is no possibility of scaling the parallelization. As this was our only identified area to parallelize, however, this was our approach. Ideally splitting up the work of this loop between 4 processes will reduce the time spent doing the work. OpenMP was our decided upon method for parallelization due to its simplicity, and it's ability to parallelize for loops in a very trivial way. In order to parallelize this specific for loop there needed to be a few changes to the functionality of the program. The biggest change was working around a break statement from the serial version. The serial version would break from the loop as soon as the final destination was found. Break statements are not allowed inside of OpenMP parallel for regions meaning we needed to create the same

effect while avoiding the break. We solved this issue by using a flag variable for indicating when this break was supposed to occur, and checking the condition of this flag in each process's iteration through the loop.

2.3 Experiments

Our test suite for the A* algorithm consists of three different style maps in four different sizes. The three styles of maps vary in the number of obstacles. Our first map has a low obstacle count, with only approximately two percent of the map being non-traversable. Our second map has a medium obstacle count, with approximately 13 percent of the map being non-traversable. Finally, our last map has a high obstacle count, with approximately, 40 percent of the map being non-traversable.



Low Obstacle Map Medium Obstacle Map High Obstacle Map

For each of these maps, we scaled the map up to two different sizes: 4,096 x 4,096 and 8,192 x 8,192.

All of the maps were generated using a map generator tool. [1]

The times recorded in the results begin after all pre-processing has completed including loading in the file and setting a start and end node. As soon as the search for a path begins the timing begins. The time stops as soon as the final destination is found.

The results are broken up into 3 sections including the minimum time to complete the search, the maximum time to complete the search, and the average time to complete the search. Each map was run 10 times in order to collect sufficient data for both minimum and maximum times.

The minimum and maximum time to complete the search is an interesting metric when considering the nondeterministic nature of the parallel implementation. The serial implementation of A* will always follow the same path to find the destination. When considering whether to make a choice about which adjacent tile it will go to next, it follows the same order every time (Ex: check north tile, then the east tile, then the west tile, and finally the south tile.). The parallel implementation however runs the 4 possible adjacent tiles in parallel meaning that any of them may run first for any given node. Often in the maps tested, there are many adjacent tiles that have the same cost to reach the final destination. This means that the parallel version may have varying paths to reach the final destination. It also may make decisions that appear to cost the same as the other possible options (Ex: moving east and south have the exact same cost) but lead to a faster or slower solution in the end.

This parallel implementation of A* is run with a consistent 4 threads meaning there is no possibility for weak or strong scaling.

2.4 Results

The results from the minimum time test showed that the parallel implementation failed to outperform the serial version in all equivalent tests. In the "High" Obstacle Density Test on the map of size 8192 x 8192, evidence of the possibility described above taking place. Here the serial version did outperform the parallel implementation, however it is significantly closer than any other two equivalent tests. This should be the case as a map with a "High" Obstacle Density should have more possibilities for the non deterministic parallel implementation to find an alternative, yet more efficient, path than the serial implementation. All other tests however seem to indicate that the parallel implementation is not faster than the serial version. This is likely due to the overhead that comes with OpenMP and using threads. The work that each thread must perform is not significant when considering the amount of work that is done elsewhere in the program. Therefore the overhead of spawning and destroying threads is not worth the time saved by parallelizing the computations.

Obstacle Density	4,096 x 4,096	8,192 x 8,192
Low	616.559	1,215.233
Medium	1,051.712	1,217.188
High	861.583	2,910.050

Minimum Time Results for Serial Tests on Varying Map Sizes (in milliseconds)

Obstacle Density	4,096 x 4,096	8,192 x 8,192
Low	1,540.240	3,358.534
Medium	1,439.186	3,392.717
High	1,433.269	3,080.908

Minimum Time Results for Parallel Tests on Varying Map Sizes (in milliseconds)

The maximum time test results further verified the serial version's ability to outperform the parallel version by having a lower maximum time in every equivalent test. The most interesting result from these tests is the "High" Obstacle Density of the map size of 8,192 x 8,192 for the parallel implementation. This test took a significantly longer amount of time compared to all other tests. This is believed to be as a result of the parallel version's non deterministic nature that was previously described above. If the parallel version made several decisions that appeared to be equivalent, but lead to an incorrect or inefficient path then it should increase the overall search time. It is important to note that in the 10 tests, the parallel version had 3 more results above 10^5 milliseconds indicating this maximum time result may not be such an anomaly.

Obstacle Density	4,096 x 4,096	8,192 x 8,192
Low	618.205	1,220.570
Medium	1,053.885	1,221.424
High	864.299	2,917.751

Maximum Time Results for Serial Tests on Varying Map Sizes (in milliseconds)

Obstacle Density	4,096 x 4,096	8,192 x 8,192
Low	1,636.552	3,532.498
Medium	2,510.558	3,562.717
High	20,867.828	132,233.866

Maximum Time Results for Parallel Tests on Varying Map Sizes (in milliseconds)

The average result times provide further evidence of the serial implementations ability to outperform the parallel implementation.

Obstacle Density	4,096 x 4,096	8,192 x 8,192
Low	617.6097	1,217.9436
Medium	1,052.6105	1,220.0830
High	862.4246	2,913.8277

Average Time Results for Serial Tests on Varying Map Sizes (in milliseconds)

Obstacle Density	4,096 x 4,096	8,192 x 8,192
Low	1,571.7703	3,441.9037
Medium	1,752.5895	3,464.5503
High	8,776.1765	52,226.8278

Average Time Results for Parallel Tests on Varying Map Sizes (in milliseconds)

2.5 Discussion

We would have liked to achieve speedup in some aspect for this algorithm, however based on other research that has been done, it is a complex problem. Our attempts at an alternative parallelization rather than the one we decided upon were all unsuccessful due to loop carried dependencies. We believe that if there is a way to parallelize the outer loop involved in the algorithm, one would truly be able to achieve speedup. Our implementation was not able to achieve this. Most of our roadblocks in this program were due to a lack of understanding of C++. After tackling these roadblocks, it became a problem of attempting to find portions of the algorithm that were truly parallelizable. We feel that the A* algorithm may be inherently non parallelizable, however it is unlikely that we considered all possible methods of parallelization.

3 Dijkstra's

3.1 Background

Dijkstra's Shortest Path First algorithm was first published in 1959 by Dutch computer scientist Edsger Dijkstra. This algorithm originally found the shortest path between two given nodes, but was later altered to find the shortest path from a starting node to all other nodes in the graph, creating a shortest-path tree (SPT). [2] In this regard, Dijkstra's algorithm is similar to Prim's Shortest Path algorithm. The only difference between the two algorithms is what is being stored: Prim's stores the minimum cost edge and Dijkstra's stores the total cost from the starting node. There are more efficient pathfinding algorithms such as A*, which uses a heuristic to give priority to nodes that should be better instead of examining every possible path.

Dijkstra's algorithm

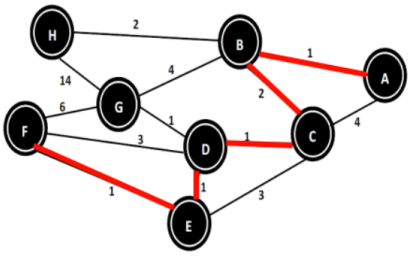


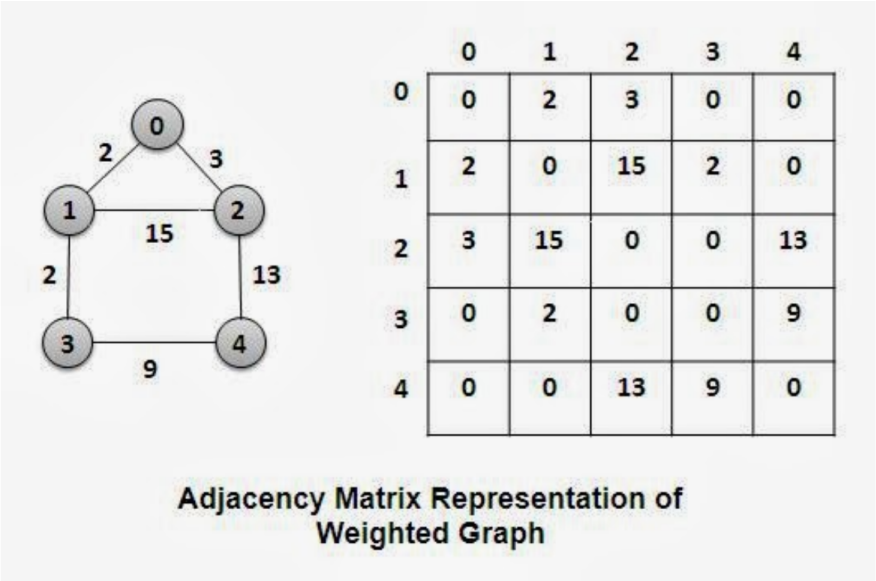
Fig. 2 8-node simple network

	(d, n)	(d, n)	(d, n)	(d, n)	(d, n)	(d, n)	(d, n)
cluster	B	C	D	E	F	G	H
A	1, A	4, A	∞	∞	∞	∞	∞
AB		3, B	∞	∞	∞	5, B	3, B
ABC			4, C	6, C	∞	5, B	3, B
ABCH			4, C	6, C	∞	5, B	
ABCHD				5, D	7, D	5, B	
ABCHDE					6, E	5, B	
ABCHDEG					6, E		
ABCHDEGF							

Table 1. The routing table for node A

[3]

The graphs can be represented as a weighted adjacency matrix, which is what was used for our test files. A weighted adjacency matrix represents a weighted graph by using the format if vertices i and j share an edge, M[i][j] contains the weight of that edge.



[4]

Dijkstra's algorithm can be altered to incorporate obstacle avoidance, which is used in A*, HPA* and JPS. However, we did not manage to implement it in a parallel version so we excluded it.

3.2 Methods

There were many existing serial versions of Dijkstra's algorithm as the algorithm has been around for a very long time, and many variations of the algorithm have been constructed. We managed to find an implementation of the algorithm in C. [5] We had originally found a serial version in C++ but decided to find a C version as we have much more prior experience with C and felt more comfortable using it. This code was modified to read in an adjacency matrix from a file instead of manually providing one to stdin. Timing code was also added to see the run-time of the algorithm itself in its serial version.

After thinking of different ways to parallelize the serial version of the algorithm, all of the most promising implementations had already been created. We found pseudocode for the technique we thought to be the most promising. [3] There was some confusion on how to implement the "parallel prefix" as mentioned in the linked report, so we found code that had a parallel implementation using a parallel prefix. [6] Much of the code was taken from this source as the parallel prefix was tricky to understand and it implemented the exact method we thought to use and for which we found pseudocode. Timing code was added for testing.

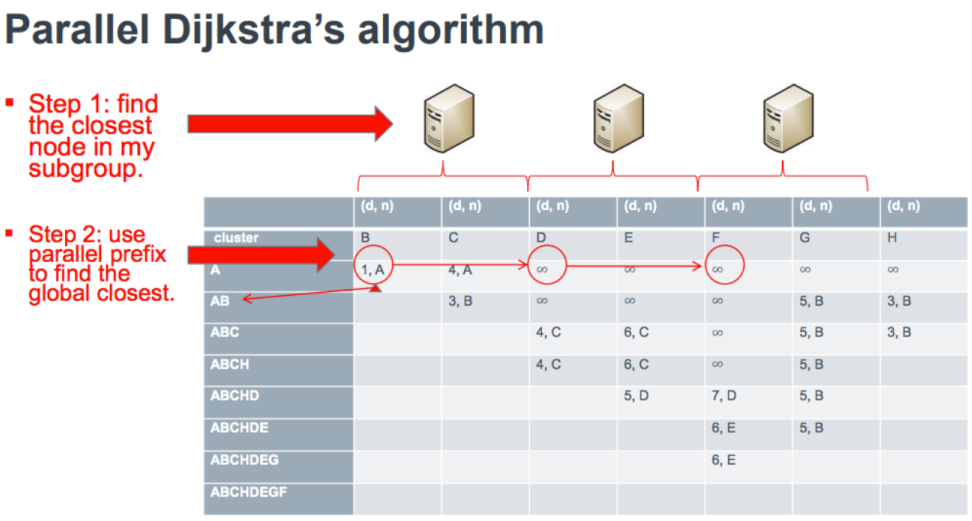


Diagram showing the usage of the parallel prefix [4]

It was clear that Open MPI was the best technology to use for parallelizing Dijkstra's algorithm as the matrix needed to be split up among processes, local results needed to be found, and a global result needed to be formed. This is exactly what is done in our implementation. The matrix is split into an even number of columns among processes, each process finds it's local closest node to the starting node, a parallel prefix (essentially the process of performing some sort of computation locally and using it to compute/find the global result) is applied to find the global closest node, the result is broadcast to all cores and each cluster list is updated.

3.3 Experiments

The test files for Dijkstra's were different from our other algorithms as it uses an adjacency matrix as opposed to a map with obstacles. We could not find any adjacency matrix generators, so the test files were made manually, which is definitely not ideal. This made it very hard to get substantial data sets as inputting a large adjacency matrix by hand would take far too long.

Thus, there are 3 adjacency matrix test files: 8 x 8, 16 x 16, and 32 x 32. The parallel implementation of the algorithm requires the matrix size to be even divisible among the processes. In order to be able to properly conduct weak scaling testing, the sizes of the matrices needed quadruple to get a size that was divisible by 2, 4, and 8 (the number of processes to be run). These are also very small matrices, but without some sort of adjacency matrix generator, it was the best we could do at the moment. Since the test matrices are so small, the results may show inaccuracies that may have not appeared with larger test files. For example, we will not know if the parallel implementation provides speedup with very large test matrices currently.

The timing of the results was done by timing the execution of the algorithm itself in each implementation.

3.4 Results

The first tests were running the largest matrix (32 x 32) using the serial version and then using the parallel version, doubling the number of processes each time. In these tests, we were looking for strong scaling.

Number of Processes	Run Time (s)
1 (serial)	0.0000310
2	0.0000833
4	0.0001257
8	0.0002344

Strong Scaling Test Run Times

It is obvious that the serial version is running faster than the parallel version. The run time also seems to double as the number of processes doubles. Therefore, the parallel version does not exhibit strong scaling. There are many possible reasons for this result. Firstly, as the number of processes increases when using Open MPI, the communication time increases. Another reason could be the way each version of the algorithm is constructed. The parallel implementation has a lot of setup that is done within the method in which the algorithm is run. This setup could potentially be done outside of the method to compare the run times between the two versions more accurately.

The next tests were conducted to test for weak scaling, so as the number of processes doubled, the size of the test matrix doubled.

Number of Processes	Size of Matrix	Run Time (s)
1 (serial)	8 x 8	0.0000141
4	16 x 16	0.0000714
8	32 x 32	0.0033047

Weak Scaling Test Run Times

Clearly the parallel version does not exhibit weak scaling either. The same drawbacks as mentioned for strong scaling can be applied to these tests as well. The small size of the matrices also plays a factor in both. Overall, many improvements could be made in the future to improve the speed and accuracy of the results.

3.5 Discussion

Overall the results are not ideal. In theory the method used to parallelize the algorithm should provide speedup, however many constraints and roadblocks prevented us from demonstrating the potential speedup. The first roadblock was figuring out how the parallel prefix worked for Open MPI to create a block column partitioning of the adjacency matrix. The code for how the datatype is constructed in MPI is still not completely understood. The next and potentially biggest roadblock was trying to find an adjacency matrix generator. Since, there was an obstacle avoidance map generator for the algorithms, it would have been much better if we could also generate large adjacency matrices. Not only was there no adjacency matrix generator, but we couldn't find any publicly available large adjacency matrix files.

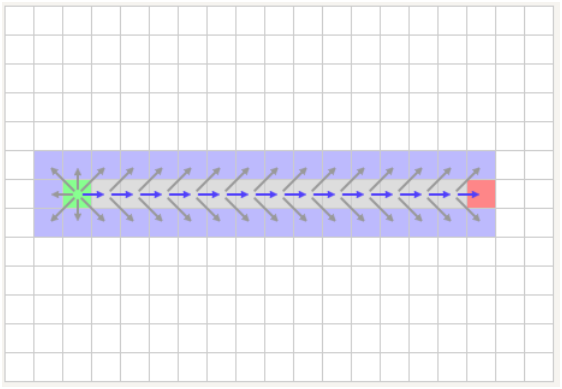
3.6 Future Work

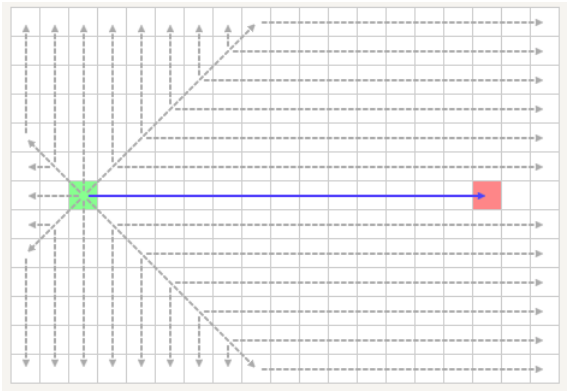
There is a lot of work that could be done in the future to improve the parallel implementation and provide much better results. Firstly, as was mentioned earlier, the current version of the parallel Dijkstra's implementation has a lot of setup done within the method itself. This setup could potentially be done outside of the method, which would reduce the run time and the tests would not lose integrity as those setup steps are not actually part of the algorithm. Secondly, the input format of both versions could be changed to run a map with obstacles. This would require significant change to the code itself, but it would allow us to run the same test suite on all of the algorithms, and provide a better comparison to the other algorithms. It would also solve the problem of not having to use weighted adjacency matrices for which there is currently no generator or publicly available large files. Finally, this is not the only way to parallelize this algorithm and there may be a more efficient way to implement a parallel version. A different parallelization technology could also prove more efficient.

4 JPS

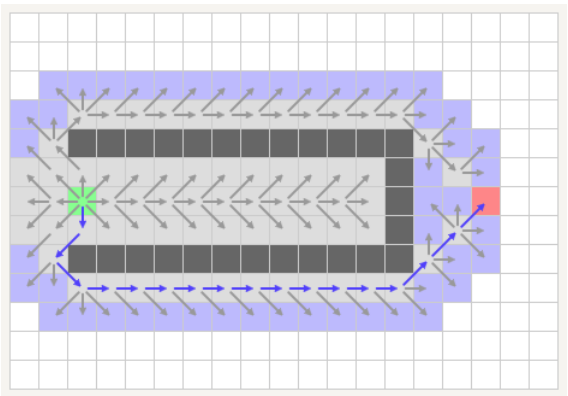
4.1 Background

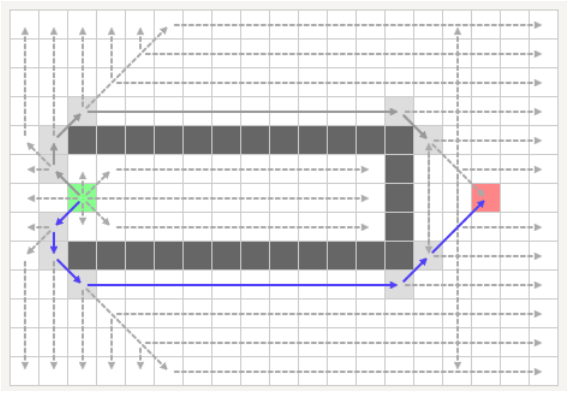
The Jump Point Search Algorithm at its simplest is an optimization of A*. The basic premise is the same; it avoids searching in directions it does not need to, such as away from the goal. The main optimization of the algorithm is the "jump." For graphs with large open spaces, there will be many jumps throughout the execution. This can be done in a straight or diagonal line, and this is part of the algorithms pruning techniques. With either, it jumps until it reaches an obstacle or a jump-point successor. A jump-point successor is a node where the optimal path travels through it, and this could be at a spot with a direction change. With a diagonal jump, it also calculates all of the straight paths towards the target as it goes along. With a either a diagonal or straight jump, the algorithm does not evaluate the nodes it skips through which saves on time and memory. [7]



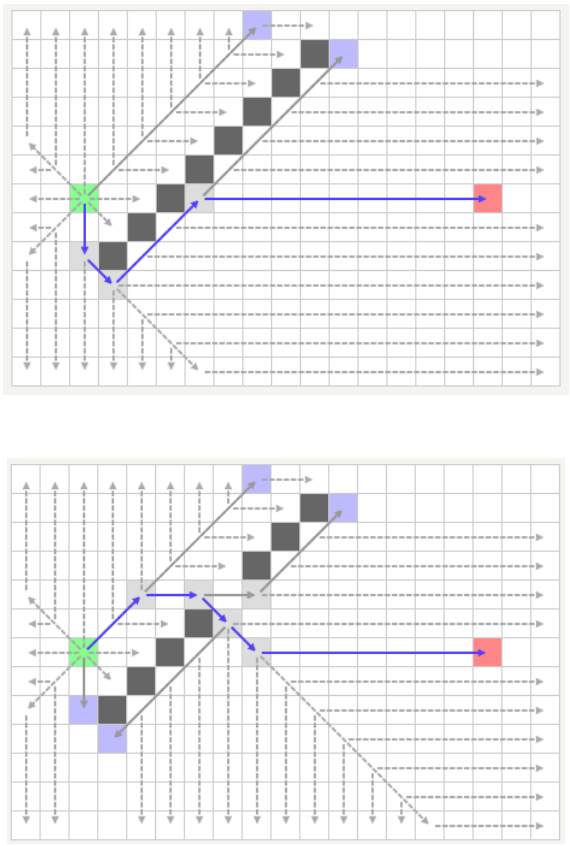


In a very simple example, we can clearly see the optimization of JPS take effect. One step as opposed to the multiple of A*





A simplified “worst case” scenario where the optimization over A* clearly shows.



Outlining how the diagonal jump works.

4.2 Methods

There were not too many existing serial implementations, so we began with an individual's implementation from Github. [8] This was their second attempt at an implementation, and this was the most complete version of the algorithm we found. This was our group's first time working with C++ as well as with the algorithm, which proved especially difficult to understand. Some of our main issues in understanding this implementation of JPS involved not having a good enough understanding of the algorithm, and so with modifying the code we were not sure exactly what we were changing. In addition, there does not seem to be much documentation on JPS as a whole, as it does not have many real world applications, and is used primarily in videogames.In retrospect, we might have had more success writing our own implementation of JPS from scratch, similar to HPA*.

4.3 Discussion and Future Work

As we did not end up testing maps on the serial version or creating a parallel implementation of the algorithm, there is a lot of room for future work on the algorithm. Our main thoughts on methods for parallelization are as follows.

- With each jump-point, divide the four tasks of searching in each of the four directions to multiple threads.
- With each diagonal path, the two straight search paths from each node could be separated and computed separately.
- For more complex graphs, a potential optimization of the algorithm could be to continue searching in spots that are not currently an optimal path, but could be if the optimal path ends up being blocked.

5 HPA*

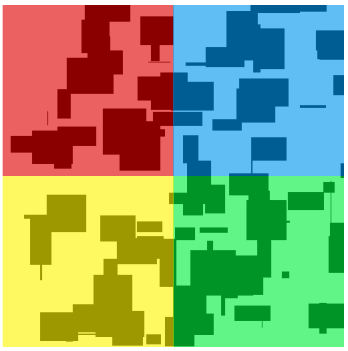
5.1 Background

The HPA* algorithm, which stands for Hierarchical Path-Finding A*, is a pathfinding strategy for grid-based maps that works by dividing a map into equal clusters, finding where to traverse between clusters, and then applying this information to find an overall path across the whole map. The motivation for developing this algorithm was to decrease the need for high memory and CPU resources when dealing with very large maps, compared to traditional A*, and to allow for efficient updates to the map if the map were to change during execution. [9]

Our deliverable for this algorithm includes the preprocessing phase of the HPA* algorithm. The components of this phase are explained in the following diagrams and explanations.

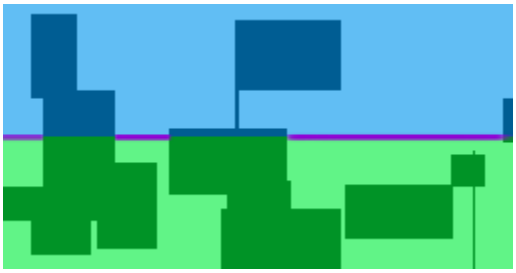
The above image is a 4,096 by 4,096 grid map, where each pixel represents a cell in the grid. The black regions are non-traversable spaces, or, in other words, obstacles. The white regions are traversable spaces. In our implementation, the map is a text file that has a '.' character for each traversable cell and a '@' symbol for a non-traversable space. Once loaded into the program, the map in this form will be referred to as the "concrete graph".

The first step of the HPA* algorithm is to divide this map into equal sized clusters. In reality, the size of the clusters can be any value that evenly divides up the map. However, for simplicity, our implementation will assume the clusters are squares and the total number of clusters in the map is square.

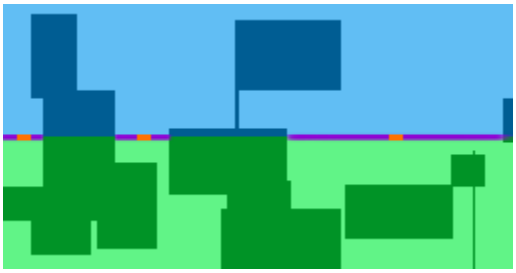


This image is the same map as above, but divided into four clusters. Each color represents one cluster. Each cluster holds 2,048 by 2,048 cells and therefore holds $\frac{1}{4}$ of the data from the concrete graph. This clustered map is one layer of what we refer to as the "abstract graph".

Once the map is divided into clusters, the next step is to find where you can enter one cluster from another cluster. This is done by looking at every border between every neighboring cluster.

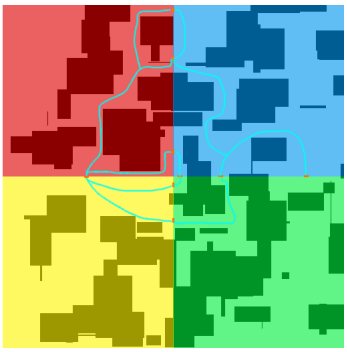


The image above is of the border between the blue and green cluster from the previous map. The algorithm will scan the last row of cells in the blue cluster and the top row of the cells in the green cluster for all locations where the cells in both clusters are traversable. The algorithm will find the regions marked in purple, which we will call "entrances".



For each entrance found, the algorithm will find a single pair of cells within that region that it will mark as a "transition point" between the two clusters. These pairs of cells are also referred to as "nodes". Our implementation will use the midpoint of the entrance for the node for that entrance. Other methods of determining a transition point can be used, such as choosing the far left and far right pairs of cells. All nodes are added to another layer of the abstract graph. The nodes for this graph are marked in orange in the above image. The size of the nodes and are enlarged for demonstration purposes. In the actual algorithm, nodes are only made up of two cells of the map.

Once all nodes are found for all clusters, all nodes within a cluster are connected to each other using traditional A*. This is represented by the light blue paths on the following image. The paths are referred to as "abstract paths".



Using these abstract paths after inserting your start and end points, the algorithm then uses traditional A* again to find a concrete path. More details about this can be found on an external source. [10]

Because of time constraints and other obstacles that will be mentioned later, our deliverable for this algorithm includes implementations for all the steps up until connecting all the nodes together. However, the parts that our deliverable does include are fundamental to what sets HPA* apart from other pathfinding algorithms. These preprocessing steps allow for faster execution when computing various paths in the map and is shown by others to be up to 10 times faster than traditional A*. [9]

5.2 Methods

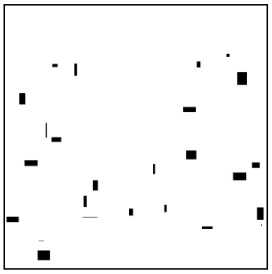
We were unable to find existing code for this algorithm to build our parallel solution on top of. This meant that we had to write a serial implementation from scratch and then build our parallel solution. The serial code for our deliverable for this algorithm was written from pseudocode and written descriptions of basic functionality of the algorithm. [11] However, because of the high levels of abstraction in these resources, we had to determine how to build and implement the underlying representations of all of the steps mentioned in the previous section.

Our implementation is written in C because of our previous knowledge of the language. In summary, the concrete graph is stored in a large array after being read in from a file and the abstract graph, clusters, and nodes are all represented in various C structs containing arrays and other variables.

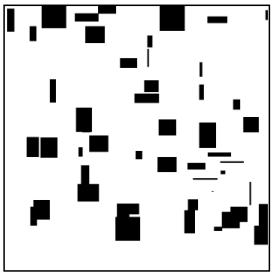
The division of the concrete graph into clusters and further independent operations on each cluster create a great opportunity for parallelization. Our initial idea for parallelization was to have each thread working on the operations for an individual cluster. After building our serial implementation, we noticed an abundance of for-loops. Most of these for-loops perform operations on each cluster and, after careful examination, didn't have any loop-carried dependencies. This, combined with the ability to incrementally parallelize our solution, led us to use OpenMP. We ended up being able to successfully parallelize most steps of the algorithm, including the clustering of the concrete graph and finding the transition points for all clusters, by dividing up the work for each thread in terms of clusters on the map. This was achieved by carefully using parallel and parallel for pragmas, keeping in mind shared and private scoping. We also had to use the critical pragma to protect some shared variables across multiple threads. Overall, our knowledge of C programming, multithreaded programming, and incremental parallelization using OpenMP allowed us to complete our deliverable for this algorithm.

5.3 Experiments

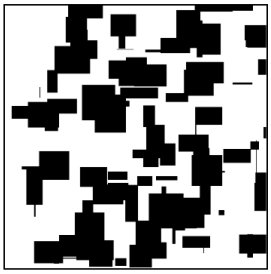
Our test suite for the HPA* algorithm consists of three different style maps in four different sizes. The three styles of maps vary in the number of obstacles. Our first map has a low obstacle count, with only approximately two percent of the map being non-traversable. Our second map has a medium obstacle count, with approximately 13 percent of the map being non-traversable. Finally, our last map has a high obstacle count, with approximately, 40 percent of the map being non-traversable.



Low Obstacle Map



Medium Obstacle Map



High Obstacle Map

For each of these maps, we scaled the map up to three different sizes: 8,192 x 8,192, 16,384 x 16,384 and 32,768 x 32,768.

All of the maps were generated using a map generator tool. [1]

The timing of the algorithm is divided into multiple sections. This allows us to determine which parts of the algorithm are more runtime-intensive and to determine runtime characteristics of these individual sections.

Cluster Contents: The first timing section consists of the operations to divide up the concrete graph into clusters. This includes copying the data from the concrete graph into new arrays representing the clusters.

Cluster Neighbors: The second section consists of the operations to calculate which clusters neighbor each other.

Finding Entrances: The third section consists of the operations to find all of the entrances and transition points between all clusters in the map. This section encompasses a large amount of code in the implementation.

Overall Abstract Graph Building: The fourth and final section consists of the entire runtime to create the abstract graph. This includes the runtime of the previous three sections and a few smaller operations that are not runtime intensive.

We do not include the timing of loading the map from a file into the array, as we are not able to parallelize this, even though this takes a relatively long amount of time.

5.4 Results

Strong Scaling Test

Our first set of tests for our parallelized HPA* algorithm were designed to show if our program demonstrated strong scaling. Our tests for this set consisted of running the algorithm on a map size of 32,768 x 32,768 and dividing the map into 4,096 clusters. We repeated these tests on each level of obstacle counts. We chose these values because they would stress our algorithm and have a relatively long runtime to highlight the potential speedup. We ran this scenario serially and with one, two, four, eight, and sixteen threads. The results are in the tables below.

Number of Threads	Cluster Contents	Cluster Neighbors	Finding Entrances	Overall Abstract Graph Building
Serial	10.4001	0.0878	0.1338	10.622
1 Thread	10.4005	0.0881	0.1336	10.6226
2 Threads	5.8901	0.0505	0.0746	6.0157
4 Threads	3.1343	0.0275	0.0396	3.202
8 Threads	2.2641	0.0248	0.0324	2.3219
16 Threads	1.481	0.0127	0.0202	1.5147

Strong Scaling Testing Results for 32768 x 32768 Low Obstacle Map

Number of Threads	Cluster Contents	Cluster Neighbors	Finding Entrances	Overall Abstract Graph Building
Serial	9.8379	0.0878	0.1251	10.0511
1 Thread	9.8219	0.0881	0.1252	10.0356
2 Threads	5.3476	0.0505	0.0706	5.4692
4 Threads	2.9139	0.0253	0.036	2.9757
8 Threads	1.7161	0.0137	0.0219	1.7522
16 Threads	1.4177	0.0126	0.0194	1.4505

Strong Scaling Testing Results for 32768 x 32768 Medium Obstacle Map

Number of Threads	Cluster Contents	Cluster Neighbors	Finding Entrances	Overall Abstract Graph Building
Serial	9.8591	0.0878	0.1146	10.0617
1 Thread	9.8144	0.0881	0.1153	10.0181
2 Threads	5.0351	0.0495	0.0615	5.1465
4 Threads	2.9011	0.0267	0.0334	2.9616
8 Threads	1.9135	0.025	0.0287	1.9679
16 Threads	1.4242	0.0126	0.0178	1.4552

Strong Scaling Testing Results for 32768 x 32768 High Obstacle Map

For the "Cluster Contents" timing region, there is very strong evidence of strong scaling. The runtime for this region decreases with every increase in thread count, with a minimized decrease when run with four threads and greater. While our program does not appear to demonstrate linear strong scaling for four threads and greater, the speedup when increasing the thread count shows that our parallelization greatly improves the performance of this portion of the algorithm.

The "Cluster Neighbors" and "Finding Entrances" timing regions also demonstrate strong scaling across all thread counts, with near linear strong scaling when run with one, two, and four threads. When run with greater than four threads, the runtime still decreases with each increase in threads, but is not linear.

The "Overall Abstract Graph Building" timing region, which encompasses all of the work to build the abstract graph, demonstrates the same strong scaling pattern as the individual methods: near linear scaling up to four threads and a minimized improvement with greater than four threads.

Compared to the serial version, our parallel implementation is faster than the serial version in all timing regions when run with two or more threads.

Varying Map Size (Weak Scaling Test 1)

Our next set of tests was designed to show if our program demonstrated weak scaling when varying the thread count proportionally with the map size. In order to keep the testing conditions consistent for each map size, we kept the number of total clusters in the map constant for each map size. This means that even though the map size increases, the map will always be divided up into 4,096 clusters. Also note that each size map has the obstacles in the exact same locations; the overall size of the map is just scaled up. Another thing we had to keep in mind when designing these tests is that in order to keep the size increases and thread count increases at the same rate, we needed to increase both at a rate of four. This is also because our map sizes can only be squares of powers of two in order to have equal cluster division. This is a restriction that could be removed in a continuation of the project.

The two main timing regions to analyze with this test are "Cluster Contents" and "Finding Entrances". We are ignoring "Cluster Neighbors" because that function's runtime characteristics are dependent on the number of clusters in the map, which we are not changing in these tests.

Map Size	Num Threads	Cluster Contents - Low Obst.	Cluster Contents - Med Obst.	Cluster Contents - High Obst.
8192	1	0.7312	0.7388	0.7344
16384	4	0.9685	0.8988	0.9041
32768	16	1.4835	1.4302	1.4283

The "Cluster Contents" section does not appear to demonstrate weak scaling, as the runtime increases with each increase in map size and thread count. This is understandable when considering that when the algorithm needs to set up the same number of clusters for a bigger map, the clusters are bigger and therefore there is more data to process for each cluster. Even though we are increasing the thread count to try to combat this increase in processing, our parallelization solution does not appear to demonstrate weak scaling.

Map Size	Num Threads	Finding Entrances - Low Obst.	Finding Entrances - Med Obst.	Finding Entrances - High Obst.
8192	1	0.0322	0.0302	0.028
16384	4	0.0194	0.0176	0.0166
32768	16	0.0202	0.0194	0.0178

The "Finding Entrances" region results for this test appear to show some level of weak scaling, primarily with four and sixteen threads. As the map size increases, the size of the clusters increases too, when keeping the total number of clusters constant. Therefore, there is a larger border to search for entrances on. Because our solution involves each thread searching one border at a time, the work each thread has to do increases.

Map Size	Cluster Contents - Low Obst - Serial.	Cluster Contents - Med Obst. - Serial	Cluster Contents - High Obst. - Serial
8192	0.7347	0.7391	0.7383
16384	2.8313	2.805	2.97
32768	10.4007	9.8524	9.8319

Map Size	Finding Entrances - Low Obst - Serial.	Finding Entrances - Med Obst. - Serial	Finding Entrances - High Obst. -Serial
8192	0.0312	0.0301	0.0274
16384	0.0638	0.0609	0.0561
32768	0.1331	0.1242	0.1145

Compared to the serial version, our parallel implementation when run with greater than one thread is faster than the serial version for all testing levels.

Varying Cluster Size (Weak Scaling Test 2)

Our final set of tests deals with varying the total number of clusters while keeping the overall map size constant at 32,768 by 32,768. The number of clusters increases at the same rate as the number of threads, which is a rate of four in this case. The goal of these tests is to show if our solution demonstrates weak scaling when changing the number of clusters. We will focus on the timing sections for "Cluster Neighbors" and "Finding Entrances".

Cluster Size	Num Threads	Cluster Neighbors - Low Obst.	Cluster Neighbors - Med Obst.	Cluster Neighbors - High Obst.
2048 (256 total clusters)	1	0.0004	0.0004	0.0004
1024 (1024 total clusters)	4	0.0018	0.0018	0.0018
512 (4096 total clusters)	16	0.0126	0.0148	0.0143

As the number of clusters increases, there is more work to be done in the "Cluster Neighbors". This work consists of identifying the neighbors of each cluster. Our parallelized solution does not appear to demonstrate weak scaling for this portion of the algorithm, with the runtime increasing with each increase in the number of clusters.

Cluster Size	Num Threads	Finding Entrances - Low Obst.	Number of Entrances Found Low Obst	Finding Entrances - Med Obst.	Number of Entrances Found Med Obst	Finding Entrances - High Obst.	Number of Entrances Found High Obst
2048 (256 total clusters)	1	0.0373	494	0.0293	488	0.0275	385
1024 (1024 total clusters)	4	0.0241	1985	0.0188	1833	0.0173	1377
512 (4096 total clusters)	16	0.0203	7982	0.0195	7213	0.0178	5196

The "Finding Entrances" section is the most interesting result of this set of tests. As the number of clusters increases, there are more entrances to be found. In our tests, the number of entrances increases at the nearly same rate as the increase in the number of clusters. The runtime for the "Finding Entrances" section appears to demonstrate weak scaling when run with four and sixteen threads, with the runtime staying similar between tests.

Cluster Size	Cluster Neighbors - Low Obst. - Serial	Cluster Neighbors - Med Obst - Serial.	Cluster Neighbors - High Obst. - Serial.
2048 (256 total clusters)	0.0004	0.0004	0.0004
1024 (1024 total clusters)	0.0058	0.0058	0.0058
512 (4096 total clusters)	0.0878	0.0877	0.0878

Cluster Size	Finding Entrances - Low Obst. -Serial	Finding Entrances - Med Obst -Serial.	Finding Entrances - High Obst -Serial.
2048 (256 total clusters)	0.038	0.0289	0.0264
1024 (1024 total clusters)	0.0744	0.0566	0.0516
512 (4096 total clusters)	0.1336	0.1253	0.1146

Once again, our parallel implementation when run with greater than one thread is faster than the serial version for all testing levels.

How does the varying obstacle counts affect runtime?

The varying number of obstacles across the three maps should only affect the timing section for "Finding Entrances", because regardless of the number of obstacles, the cluster sizes and neighbor calculations should be the same.

As the number of obstacles increases, the number of entrances decreases as there are less locations in the map to travel from one cluster to another. This is seen best when comparing the low obstacle map and the high obstacle map. On the largest map size of 32,768 by 32,768, the low obstacle map has 7,982 entrances while the high obstacle map has only 5,196 entrances. This results in the high obstacle map having a slightly shorter runtime for the Finding Entrances section than the low obstacle map.

However, an interesting observation is that the serial and one thread runtime for the "Cluster Contents" section for the medium and high obstacle maps is consistently more than half a second faster than the low obstacle map. We are unsure what causes this discrepancy because the runtime should be consistent for this section for the three different obstacle maps because the cluster size and number of clusters are the same across the three maps. We ran these tests multiple times and the results were consistent.

Other than the two observations above, the obstacle count does not affect the runtime greatly for this portion of the algorithm. If the rest of the algorithm were to be included in these tests and run on more map variations, we could potentially see a bigger difference in runtime across the different obstacle counts.

5.5 Discussion

We are very pleased to have achieved not only a correct parallelization of this phase of the HPA* algorithm, but determine that our parallelization produces a significant decrease in runtime. Most of our roadblocks in this part of the project were with trying to complete the serial version of the algorithm. The code to find the entrances and transition points was the hardest to complete. After the serial version was finished, we ran into a few challenges with parallelization, such as determining whether or not we could parallelize certain parts of the algorithm. This was resolved by carefully tracing through the code. Because we wrote the code from scratch with parallelization in mind, we didn't have to rewrite any code from the serial version in order to achieve parallelization.

5.6 Future Work

The logical next step for working on this algorithm is to finish implementing the rest of the algorithm. Even though we were able to write most of the HPA*-specific code and the remaining code to write primarily utilizes the traditional A* algorithm, we would be able to analyze the full runtime of the algorithm. Furthermore, we would like to be able to use a parallel A* solution in this step.

6 Conclusion

As seen in our results for each algorithm, our parallelization attempts had varying results. Dijkstra's and A* didn't have positive results, but with more time working with the algorithm and different testing scenarios, they could potentially show improved performance. HPA* showed the best results out of the four algorithms with a large speedup compared to the serial version and some scalability. Finally, while we didn't get to test JPS, we believe that our early parallelization ideas could eventually show some improved performance. Ideally, we would have also liked to have been able to compare all algorithms to each other running the same maps.

By working on this project, we have only scratched the surface of pathfinding. There are many other approaches to parallelizing these algorithms and even more pathfinding algorithms in general. Our deliverable for this project shows a strong first attempt at parallelizing pathfinding algorithms and the suggestions in the "Future Work" section of each algorithm set up a strong foundation for continuing this project.

References

[1] djc1024gh. Map generator. https://github.com/djc1024gh/MapGeneratorMT.

[2] Christopher Williams Thaddeus Abiy, Hannah Pang. Dijkstra's shortest path algorithm. https://brilliant.org/wiki/dijkstras-short-path-finder/.

[3] Zilong Ye. An implementation of parallelizing dijkstra's algorithm. https://cse.buffalo.edu/faculty/miller/Courses/CSE633/Ye-Fall-2012-CSE633.pdf.

[4] Neeraj Mishra. Representation of graphs: Adjacency matrix and adjacency list. https://www.thecrazyprogrammer.com/2014/03/representation-of-graphs-adjacency-matrix-and-adjacency-list.html.

[5] Neeraj Mishra. Dijkstra's algorithm in c. https://www.thecrazyprogrammer.com/2014/03/dijkstra-algorithm-for-finding-shortest-path-of-a-graph.html.

[6] Lehmannhen. Mpi-dijkstra. https://github.com/Lehmannhen/MPI-Dijkstra/commit/5991fb8c938c44e5b53da7210a49b28633d11989#diff-04c6e90faac2675aa89e2176d2eec7d8R43.

[7] Alban Grastien Daniel Harabor. Online graph pruning for pathfinding on grid maps. http://users.cecs.anu.edu.au/~dharabor/data/papers/harabor-grastien-aaai11.pdf.

[8] fgenesis. Jps serial c++. https://github.com/fgenesis/tinypile.

[9] Jonathan Schaeffer Adi Botea, Martin Muller. Near optimal hierarchical path-finding. https://webdocs.cs.ualberta.ca/~mmueller/ps/hpastar.pdf.

[10] Daniel Harabor. Near optimal hierarchical path-finding. https://harablog.files.wordpress.com/2009/06/beyondastar.pdf.

[11] Linus van Elswijk. Near optimal hierarchical path-finding, 2013. https://www.cs.ru.nl/bachelors-theses/2013/Linus_van_Elswijk___0710261___Hierarchical_Path-Finding_Theta_star_Combining_HPA_star_and_Theta_star.pdf.