Running From The Night: Calculating The Lunar Magellan Route in Parallel

Kevin Fang Nikolai Stefanov

May 6, 2024

1 Summary

This project focuses on creating a lightweight and somewhat accurate parallel application using MPI in C++ that when given a set of points of interest located in scientific zones on the lunar surface, finds with a greedy algorithm a relatively short path that visits a point interest in all scientific zones at least once. We chose this approach, because during our past experience with lunar robotics missions at CMU, we wanted a pathfinding algorithm that can be run fast on personal laptops during lunar missions and deliver generally-decent results for the kinds of rough estimations that are made during planning phases. The deliverables are our path creator algorithm and we will present an example output path from this graphed onto a shadow map of the lunar surface.

2 Background

The "Magellan Route" is a theoretical path that maintains constant daylight on the moon's surface. For the sake of this problem, we are implementing a version of the A* pathfinding algorithm which we've adapted to execute in parallel across multiple nodes. In a proposed mission, a rover will traverse this "Magellan Route" and deploy small radio antennas in permanently shadowed regions, with sufficient distance between them. This distance denotes the scientific zones, where at least one antenna must be placed (thus our route must traverse right next to).

Our data structures include:

1. `Node` represents a point at `int (x,y)` coordinate with its related `float cost` to reach the node, a `float heuristic` estimate for distance to goal, a `Node* patent` pointer

to it's parent node, and `bool` flagstfor whether the point is either blocked or shadowed. We compare `Nodes` by checking for the lower `heuristic` value, which represents the `Node` that is closer to the goal coordinate during our A* algorithm. item `grid` is A 2D array of integers, where each cell can be `0` (accessible) or `1` (blocked). This grid is used to initialize the `routeMap` by mapping each cell to a corresponding `Node`. We converted the LROC map for total shaded regions into a 5048 x 5048-sized `grid` which due to its massive size is imported as a include shadowmap.h into our main code.

2. `routeMap` is the 2D vector of `Node` objects mirroring the `Grid`. This structure holds all the path-related data for each point, including costs and heuristic values updated during the execution of the A* algorithm.

3. `antennaList` is a list containing antennas' locations represented as vector pairs, each representing like for `Node int (x,y)` coordinates. These locations are used during pathfinding in order to separate out each "stage" of the pathfinding algorithm which is split up and assigned to processors. Antenna locations aren't set by us when running the program and are determined by the program with each processor finding their ideal antenna locations – more information on this in the Approach section.

4. `Priority Queue:` Inside our implementation of the A* algorithm, we have a priority queue to dynamically fetch the next most promising `Node` to visit based on combined cost + heuristic estimations (priority is given to nodes with lower values).

For operations we perform on these data structures, we first start off by loading in the `Grid` into a `routeMap` at the start of the algorithm. We then find a subset of possible antenna locations (how depends on the specific parallelism approach). A* then one-by-one expands through nodes beginning from the first, updating the `routeMap Node`s with distance estimations and parent `Node`s as it travels through the lunar surface from the starting node(s) to either the next antenna's assigned areas for each processor or to the very end of the image (see Approach for more information).

Our `main` function has the constants that we change to affect how the grid zone assignments of the routeMap are made for the antennaList. It prints the most cost-effective path to get from the start to the destination as found by A*, keeping in mind the need to avoid permanently shaded areas of the moon in relation to the placement of radio antennas in order to maintain signal with the rover.

This program utilizes MPI for parallel processing, taking advantage of the potential paths that can independently calculated at the same time. The significant computational challenge lies in both the pathfinding calculations, particularly in densely blocked or complex areas, making efficient parallelization critical to performance improvement, and also the total number of permutations of waypoints to stop at along a path. Our default, and most optimal, approach is a vertical parallelism across rows – in our Approach section, we will discuss our journey designing the Parallelism Across Columns variation.

3 Approach

Before designing and coding our implementations, we needed to decide what languages, platforms, and frameworks to use, then convert our given data (the .IMG LRO files). We chose C++ and used MPI as our message-passing interface. Initially, our implementation featured a utilization of the Geospatial Data Abstraction Library (GDAL), a C++ library designed to convert the `IMG` file formats from the LRO files into usable data. However, we realized it made no sense to convert the image files while the program was running, as we could simply convert the files ahead of time, as this is not something that needs to be done at runtime. Therefore, our Python-script converted it into shadowmap.h, containing all of the needed permanently shaded region information.

We used the algorithm from https://www.geeksforgeeks.org/a-search-algorithm/ as a starting point for us to implement the A* algorithm, though we heavily modified it and created multiple versions of it to support our project needs. Furthermore, one important note before discussing our two main approaches is that our algorithm started by finding the antenna locations. We did not have set antenna locations, so our implementations changed which antenna locations we were viewing. This will be important for future discussions on optimization. It's also important to note that the amount of permutations in antenna visitation, thus problem size, increase as $(\text{num antennas per zone})^{\text{num antennas}}$. Therefore, we found that we had to limit the number of antennas per zone, otherwise it would take forever. We did most of these tests on one of our personal laptops, which was a 10-core (8 performance, 2 efficiency) mac with 64GB of ram.

Initially, we wanted to use GHC machines in order to run our tests. However, we found out that they had difficulties with opening files that contained a lot of rows and columns (a file would simply be deleted or not be able to be opened). If we had to program for GHC machines, we would consider separating our shadowmap.h into multiple maps loaded in when a processor needs it. However, for our personal devices, our M1 chips/desktop PC have high bandwidth across the interconnect, so loading the entire shadowmap.h was also faster than selectively loading in limited `routeMap` and resizing when needed (as well as being far simpler and less likely to have bugs/issues related to missing pieces of the map.

Although using personal machines helped us with accessing our shadowmap.h, didn't come across disk space/ssh connection issues, and were more comfortable to develop on, one new challenge we had which we had not encountered before in this class was setting up and developing parallel programs for very different devices, as for all previous assignments we used the GHC machines and did not need to choose different versions/installations of

MPI. For the team's devices, one was a desktop PC with an i7 that required installing MS-MPI, while the other was an M1 Macbook Pro that was running OpenMPI. This resulted in test results that differed greatly, and modifications to the code which sped up test results sometimes actually slowed down test results on the other team member's device.

Table 1: Comparing Development Platforms

Feature	**Microsoft MSI Desktop**	**M1 Macbook Pro**
Processor	Intel Core i7-4790	Apple M1
Cores	4 physical, 8 logical	8 performance cores
Operating System	Windows 10	macOS
MPI Implementation	MS-MPI	OpenMPI
IDE	Visual Studio Code	Visual Studio Code
C++ Environment	MinGW-w64	Native Apple clang
Pros	Multithreading	High bandwidth + efficiency cores
Cons	Older architecture	ARM architecture support
Other	Hyperthreading	High bandwidth interconnect

Though this initially presented a great frustration as we individually had to figure out how to set up our C++ and MPI environments and then ensure during development that we made sure to use MPI calls that were universal across MPI implementations, ultimately this turned out to be an interesting development to our project, as similarly to PSC/GHC comparisons we could compare and contrast results and learn more about how processor, MPI implementation, and OS choices impact how parallel code is developed and run.

We ultimately determined to use the Macbook Pro as our primary test/optimizer device. This is because the M1, with its higher number of cores and better parallelism as well as being more modern, contained both high-performance and high-efficiency cores with improved data sharing, more closely represents the future devices which would run our code. For more practical reasons, it ran our tests significantly faster as well, which improved our test/develop cycle speed.

Table 2: Vertical Parallelism - i7

Threads	**Time (ms)**	**Comp Speedup**
1	1056869	1
2	1392248	0.7591
4	749162	1.4107
8	462892	2.2832

Table 3: Vertical Parallelism - M1

Threads	Time (ms)	Comp Speedup
1	141987	1
2	79936	1.7763
4	45609	3.1131
8	33749	4.2071

Once everything was ready, it was time to approach our implementations of the actual A* pathfinding algorithm and how we would incorporate MPI Parallelism. We had two methods: Across Rows and Across Columns.

3.1 Parallelism Across Rows

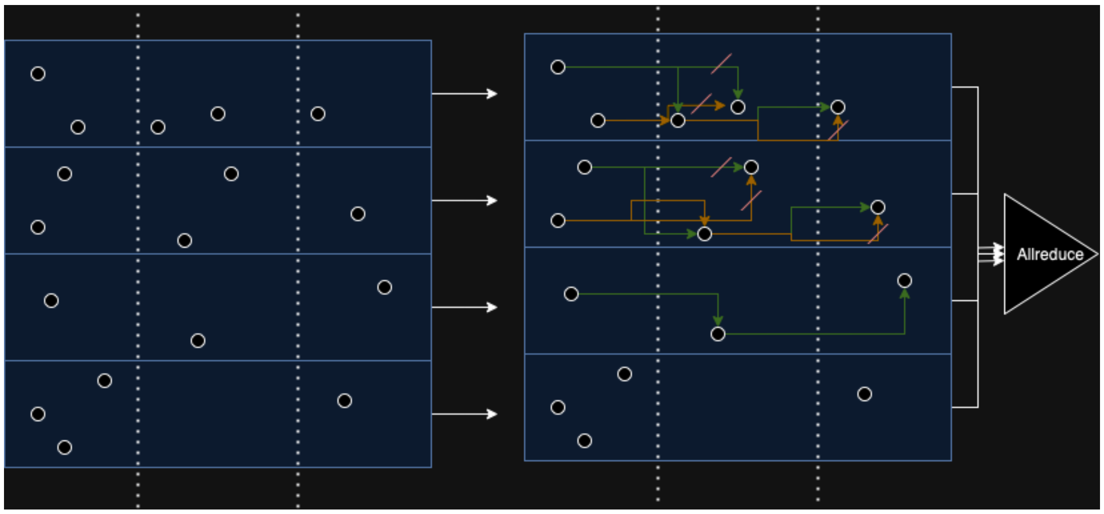


Figure 1: Diagram of the Parallelism Across Rows Approach.

Each blue rectangle represents the portion of an image that a thread will look at. The dotted white lines represent the separation to place the antennas. You see that after it finds the antenna locations, each thread finds the minimum path to the end its section using a greedy algorithm. Note: the final path from the last antenna to the end is excluded in this diagram.

Our first and primary implementation was to have each thread look at a contiguous set of rows at a given time. Each thread would find the antenna locations between their respective starting and ending rows. After each thread finds its antenna locations, it then finds the minimum path to the end (here the end represents the column at the width of the image and the same starting row as the start point) using a greedy algorithm (just picks the minimum path while it's at a given node) for a given start node. It repeats the last step iterating through each potential start node (antenna location in the first section of where antennas could be placed, in the diagram above that corresponds to the first third). After each thread finds its minimum path across all starting nodes, we reduce across all the threads to find the minimum. Afterward, the thread with the minimum will print its cost and its path.

3.1.1 Evolution of Parallelism Across Rows Approach

Some of the biggest evolutions was how exactly we were finding the antenna locations. Originally, we were just looking at the first column in each corresponding antenna section and trying to place an antenna. This lead to a lot of never placing an antenna for certain processors in a section, thus never getting any routes. To counteract this, this was changed to search columns until each processor reached 5 columns with antenna locations in them. The issue with this was that some processors would have many antenna locations across certain columns (image dots of potentially shadowed regions to place the antennas in) while others would not. Even worse for load balance, we saw that some processors would not have 5 columns of antennas until the very end of their section, that they would continue searching for far longer than that of other processors. To fix this issue, we search 30 columns across all threads. This became a hyper-parameter that we altered if some processors were unable to find any antenna locations in the given column bound (i.e. starting off at 5, then 10, then 20, then 50, then 30 which appeared to be our sweet spot for both finding a path and speed).

Secondly, one of the best things about this algorithm is also one of the worst. Since each processor only sees the antenna location in its section, you can't consider the path from an antenna in processor 1 to an antenna in processor 3 for instance. While this may seem like it invalidates our algorithm, we are okay with this loss of accuracy to the incredible speedup this provides. However, we did encounter an issue with this sometimes. Sometimes, a path would not be able to reach an antenna location, in its bounds. We fixed this temporarily by running through the greedy algorithm again, except allowing the processor to use any part of the map if it so desires. This increased our runtime, however we were successful. This was only temporary since applying the final change mentioned in the previous paragraph actually solved this too with very little increase to runtime, so we removed this change.

We iterated many times on the initialization of our routeMap. Initially we started by iterating through the entire image itself, which wasn't too slow. However, we figured we could make it faster since every thread was only going to look at its own section of the image we only had to initialize that part of it. Well, this did speed up our initialization time it saved, maybe one second in total.

We also iterated on our A* algorithm throughout the course of this project. Firstly, we ran into a lot of memory issues so we wanted to ensure that each time we were calling the algorithm it was on properly initialized maps. Therefore between each antenna location, we reinitialized both our own map and our prevented paths (what ensures we don't go over a node twice) to ensure that previous paths didn't impact our finding of the minimum path for this antenna location. We also played around with different heuristics eventually just settling on the Manhattan distance because it was both what was most intuitive (and easier to convert for the other method) and what seemed to work the best. We also realized that we weren't accounting for the final antenna location back to the start, therefore not completing the Magellan route so we had to add a final check that that final location could reach our starting location via a wrap around (though we didn't care too much about the cost of the wrap around).

We also played a lot with our map size to find the optimal size for us to do our test. The image itself is 5028 x 5028 so doing any standard test for both performance and correctness on this would take far too long. We battled against having too small of an image meaning that we were unable to actually find antenna locations in the given size; therefore, not effectively completing our algorithm. For the more to overcome some of our previous mistakes on Homeworks, we wanted to ensure that we always had test cases that didn't evenly divide either the width or the height which did help us actually diagnose some bugs early on and how exactly we were calculating the width per processor.

3.2 Parallelism Across Columns

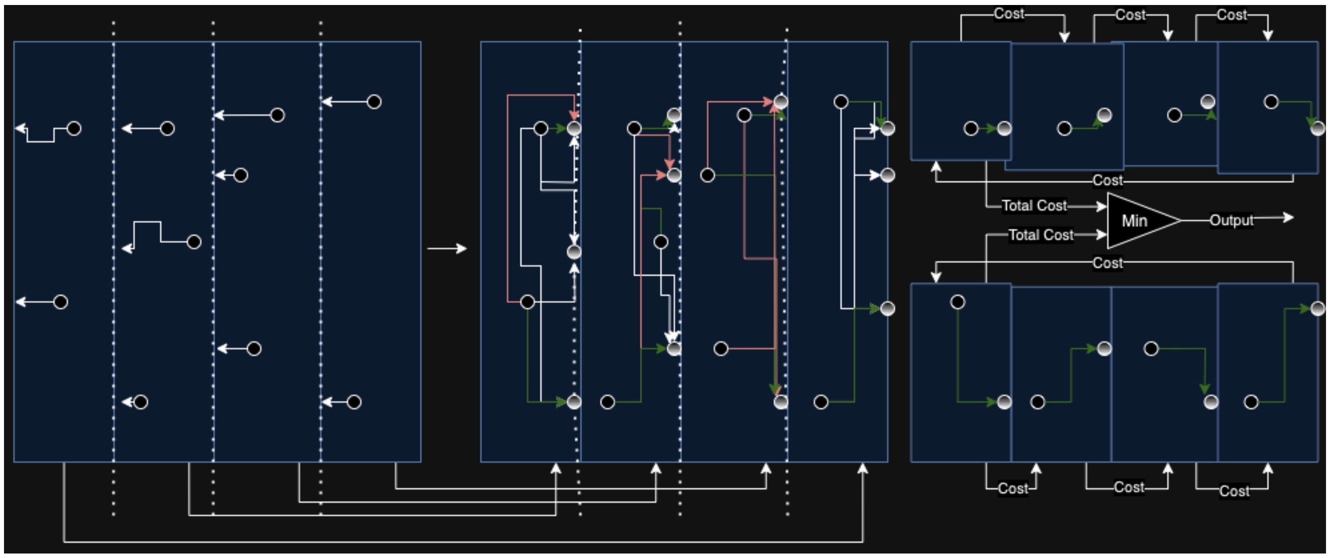


Figure 2: Diagram of the Parallelism Across Columns Approach.

Each blue (vertical) rectangle the portion a thread will look at. We see that initially each thread finds the antenna locations. Which it traces a path to the edge of where the previous processor ends marking down the cost of that traversal and eventually, sending those points to the previous thread as a ghost antenna. After which, each thread with its new set of ghost antennas will calculate the minimum routes (pictured as the green paths) for each one of it starting notes to one of the ghost nodes. after which the first thread will iterate and try to find the minimum path across all the threads using their pre-computed minimum and costs define the overall minimum route.

Our second algorithm is one that separates the image across continuous columns for each processor. We decided to go down this route since we weren't able to figure out a way to make the parallelize across the frontier approach that we specified in our proposal and mid semester report. To make up for that and the fact that our previous algorithm didn't involve a lot of communication between threads, we wanted to see if we could better optimize the path finding parts of our algorithm. Firstly, like the previous algorithm, each thread finds the potential antenna locations. However, the threads are bound by starting width and ending width as opposed to height or ending height. then each thread will find a route to the previous threads ending width to ensure that each antenna location can effectively be reached by the previous thread. Also allowed us to parallelize across our initialization since each thread is now only considering its contiguous set of columns. After all the threads have collected their ghost nodes to send we have each thread sent and receive from its corresponding pair of nodes (i.e. even threads send to odd threads first and then odd threads send to even threads, with the first thread sending to the final thread first as well). Afterwards, each thread will find the minimum destination, ghost node, for each one of it starting nodes. it will then take that information and for each iteration of the thread zero starting node it will receive a destination node the previous thread chose then an send both the cost and the minimum destination to the next node in the series, ending with the last thread sending the first thread the total cost across all threads. Finally thread zero will take each iteration starting point cost and find the minimum across all of them which will be the total cost of the path.

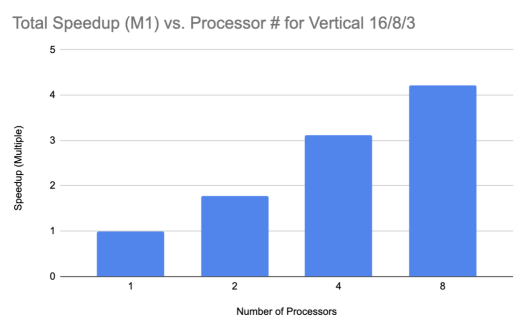
3.2.1 Evolution of Parallelism Across Columns Approach

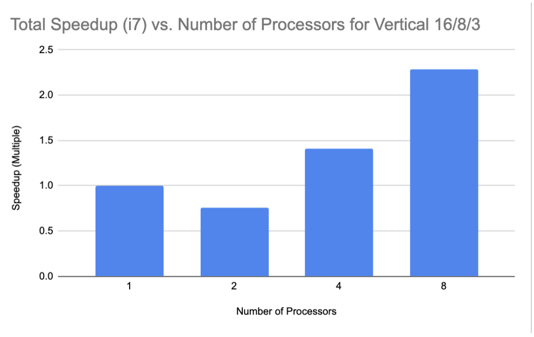
The biggest reason why we wanted to go down this route for another algorithm is because as we increase the width of our image, we recognize that we were increasing the time that each A* search would take. Therefore, by separating out the width across the processors, then we would have the ability to potentially speed up our search time. Furthermore, we felt as though our initial approach didn't it involve much message passing, and we wanted to experiment more with that level of parallelism. However, as we would find out, it didn't quite work out for us.

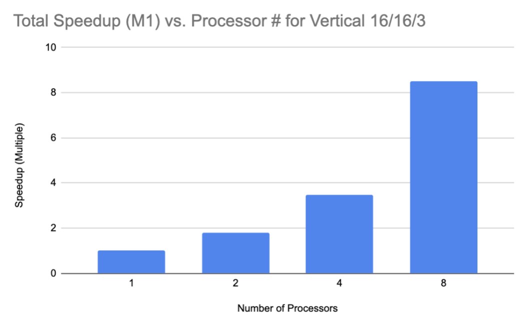
Firstly, we made edits to our initialization, allowing for us to initialize our routeMap in parallel. As was the same case with the Across Rows approach, this isn't tremendously help. Our speed up may be saving a couple of milliseconds at most.
Furthermore, we also optimized our message passing to ensure that as little data as possible was being sent over, for instance instead of sending over a fix-sized array we first send over a message dictating the size of the array then sending the corresponding array. Most of these initial tests were done on the Mac, which had that high bandwidth so we didn't see much improvement as we shrunk the sizes of the messages. Even still, we figured the smaller the better so continued down that route. We also replaced as many broadcasts as possible, going for of a ring style message pass structure wherever it fit.
Lastly, one of the most important things to note about our project is that there is not a predetermined correct answer. We bring this up because it was a constant issue when debugging some of these as the level of parallelism we would implement would dictate what paths we could actually create. While we touch on this in the previous section, it became very apparent here as it was the main reason why we had to give up on this approach. The fundamental issue with this algorithm is that the next node that dictates the minimum path would not necessarily always dictate a path that can successfully end. In other words, what would happen is the last two or so antennas would not be able to actually connect back to our initial starting point (in terms of a wrap around). Therefore the greedy algorithm approach for the message passing would not work. Note it works for the previous algorithm because we were able to see the entire width, whereas here, we cannot. Therefore there were many significant efforts in order to try to make this work. Firstly, we tried to make the number of antennas that each processor would see larger. Therefore, the chance that we would run into a dead end would decrease. While we started to see some success in that (i.e. less paths running into dead-ends) the runtime started to grow far larger than we desired to continue seeing that as a viable option. We eventually settled on an algorithm that was to follow the similar approach to the across rows algorithm, with this algorithm just pre-computing all the costs to the next nodes. However, as we were implementing that approach, we were recognizing that the across rows algorithm that we were concurrently optimizing had already completed before the across columns algorithm finished pre-computing all of the costs. Therefore, then finding the minimum path across all of those costs would result in a longer running algorithm than our across rows approach. It was here we decided to focus our efforts on better optimizing the across rows algorithm and making it more consistent (in terms of workload balance).

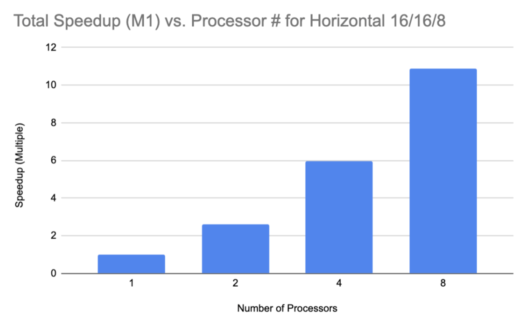
4 Results

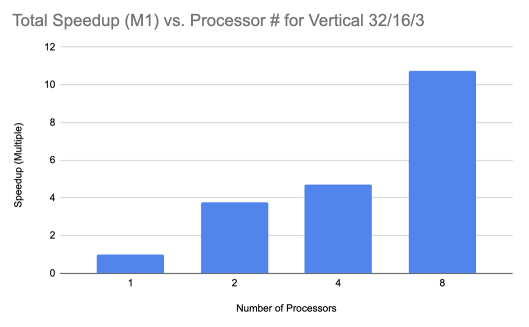
Note: for the following graphs, a/b/c refers to a problem size of width = (image width / a), height = (image height / b), and number of antennas = c.

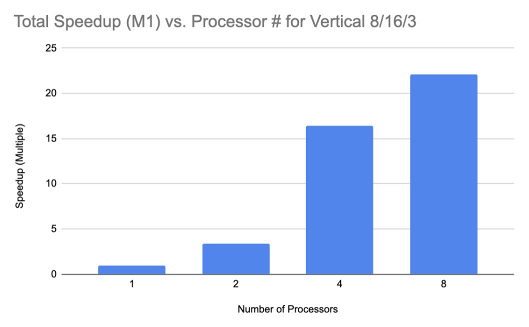












The above are graphs of Total Speedup vs Processor Number for a selection of the tests we ran on various combinations. For the sake of comparison the default speedup graph we will be comparing the rest off of will be the first, our M1 16/8/3. For observations, we saw first that i7 had worse speedup – this is unsurprising given our discussion in the Methods section about the M1's superior performance in parallelism. One interesting note here is that for the i7, the 1 processor actually was faster than 2 processors – this was because the

1 processor test's greedy algorithm failed to find the actual shortest path which all other threads found (cost = 107) and instead quickly returned a more expensive path (cost = 409). This is one interesting limitation to speedup graphs that we encounter in many speedtest scenarios, especially for greedy algorithms (it's very easy to quickly return a wrong answer).

The main difficulty with directly comparing a/b/c values to each other is that when the problem area becomes larger or smaller, we are effectively replacing the current lunar surface map with a completely new one, as only a subset of the previous shaded area now represents the entire new shaded area (decreasing the total area to traverse doesn't shrink all the obstacles as well to fit in the new smaller zone). Increasing the area results in the same testing issue. This is worse when the algorithm is greedy and does not always give the exact shortest path for any image size.

Therefore, we can only give general estimations and reasoning for the effects of larger/smaller image sizes and more/less antennas on speedup. One interesting side effect of parallizing greedy algorithms in the way we did is that the more processors are used, the more accurate the returned paths were (lower cost). Observe an example below for a M1 Vertical 16/4/3 test:

Table 4: Vertical Parallelism - Configuration

Threads	**Cost**	**Time (ms)**	**Speedup**
1	397	2668522	1
2	329	1334261	2
4	125	1369963	1.9479
8	107	541534	4.9277

This makes sense, as the path that is returned is the shortest that is found out of all threads' greedy algos – if there are more threads, it makes sense that discovering a shorter path is more likely. Scaling processors then not only improves speed but also results. Interestingly enough, when there were greater speedups, it was more likely that the Cost would not change between threads. When Cost did see decreases, speedup increases would be less. Our theory is that as Speedup is calculated as a multiplier on the 1-thread result, when the 1-thread result returns fast for its bad answer, the Speedup of a higher-thread run actually is weighed down by the fact that for it finding a better path takes longer time, while when everyone finds the same path, this doesn't apply. Further testing with non-greedy algorithms would be able to verify if this theory for our observed results is correct.

For us, as we were testing and evident in the graphs, we found that as we increased the image height, we gained more possibilities for antenna placements and routes, which meant slower route and antenna finding by 1 or two threads, which meant our speedup was limited by those (as discussed in Speedup Limitations). Image width was far more inconsistent, since, as discussed above, it is a fundamentally new image without increasing the amount of image height for routes, which can result in scientific zones only have a couple of POIs, thus reducing the total amount of possibilities, or having exponentially more, increasing the total runtime. Since we then break up this image into slices per processor and the fact that there is no uniformness to this data, we sometimes have a couple of threads being the sole bearers of the new workload. Increasing antennas decreased Cost and increased total runtime while having mixed results on Speedup. We reason that increasing the number of antennas increases the number of permutations that have to be calculated, while if the additional antennas cause there to be new shorter paths, this could either never be found by the 1-thread (thus it exits with a poor Cost fast) or it means even the 1-thread could find it now, meaning it will exit with the same cost as the other threads and thus take normal time to exit and result in high comparative speedup.

4.1 Algorithm Correctness Limitations

For further discussion on parallelization and problem size, both algorithms limit the problem size as we increase the level of parallelization, but only sometimes. We talked about our journey to limit this as much as possible, but we find that it is fundamentally tied to our problem statement. The across-rows algorithm limits our problem space to only paths that remain in a contiguous set of rows. We are okay with the sacrifice as the chance that a shortest path exists across many sets of contiguous rows seems unlikely. Moreover, we find that when a path can't be found this algorithm returns that very quickly, which allows us to quickly update hyperparameters in our search. With the across cols algorithm, we reduce the problem space to only paths that are the shortest from each antenna. Well, we use a greedy algorithm for both across rows and across cols, it is only across columns that removes all possible next nodes that are not the minimum.

4.2 Speedup Limitations

While our speedup graphs vary wildly sometimes, we believe that that is due to the problem size reduction as we increase parallelism. We, of course, have speedup limitations due to the reduction in across-rows approach and the message passing in the across-columns approach. We focus most of these speedup limitations on our across-rows approach since we focused on optimizing it more. We also see a significant workload imbalance (despite efforts to try to fix it) in the plot below.

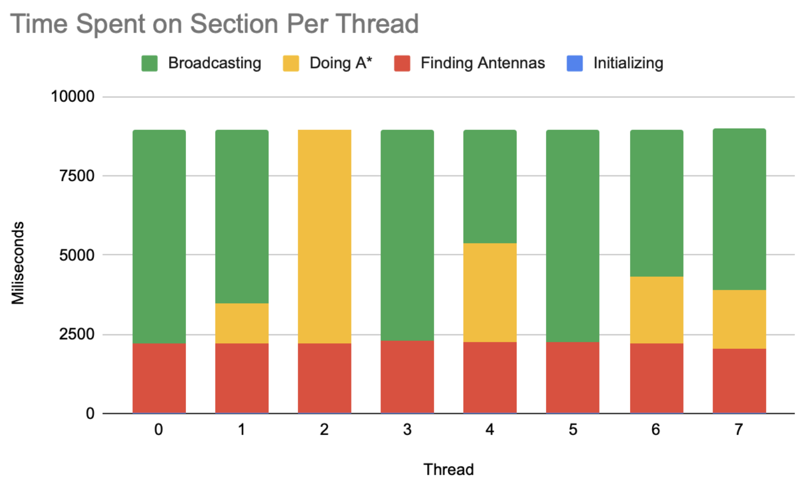


Figure 3: Plot of the time spent by each thread in each section of code for the across rows approach with an 16/16/3 problem size. Note that broadcasting refers to both the time spent doing the reduction and waiting for the broadcast to arrive.

In truth, we have fairly good workload balance when it comes to finding the antennas (which shows that our optimization works). The issue comes down to when a thread either has no antenna or has a very little antennas to run the A* algorithm on. We could dynamically assign these to each thread, although we would then run into issues with having to properly initialize the route map for whoever is getting the data (since they are now dealing with rows that they haven't seen before) would have the potential to increase the runtime with the new overhead (along with the overhead of dynamic scheduling).
Overall, it's hard for us to definitively say how effective our parallelization is in our algorithm. The workload imbalance shows that there's areas to improve, while some of our speedup graphs show performance better than optimal. We found that as we varied our parameters, we created new images and new objectives which make it somewhat challenging to compare between them. Though, we did certainly enjoy every second of this experience, as we learned how to better deal with problem sizes that change as you vary your parameters. We also learned about the different effects of parallelization for greedy algorithms when each thread runs a version on their slice – in cases like ours, it may be that the number of threads actually changes what result we get. In addition, we learned how to install, implement, and develop MPI/C++ programs on our own devices, setting them up and going through processes like adding their Bin to System Environment Variable PATHs, creating our own customer MakeFiles, and writing one-off Python programs to do tasks like graphing paths onto the lunar surface imagery.

References

[1] Lunar Reconnaissance Orbiter imagery, `https://ode.rsl.wustl.edu/moon/`

[2] Nate Otten's PhD thesis, `https://www.ri.cmu.edu/app/uploads/2018/01/thesis_otten.pdf`

5 Work By Each Student

- **Kevin Fang:** Translation of .IMG LROC file and shadowmap.h header. Project coding. Poster design and printing. Testing of i7. Result analysis. Creation of performance charts and tables. Writing report. Maintaining website.
- **Nikolai Stefanov:** A* algorithm + antenna implementer. Project coding. Fixing bugs and issues. Optimization of MPI code and testing for M1. Writing report. Creating per-thread time spent chart. Path graphing script.

Distribution of total credit: 50%-50%.