15-418/618
Parallelization of A*: A Graph Search Algorithm

Jacob Zych (Andrew ID `jzych`), William Foy (Andrew ID `wfoy`)

5/5/22

1 Summary

We parallelized the A* graph search algorithm in both C++ and Golang using various techniques and compared the performance of the implementations on differently constructed input graphs. More specifically, in C++, we implemented the algorithm sequentially, using pthreads, and using MPI. In Golang, we implemented the algorithm sequentially and using goroutines. The performance was measured on 2 different sets of hardware, the GHC Machines and the PSC Machines. We also wrote code to test our results against the optimal path and code to visualize the path on a graph.

2 Background

A* search is a graph traversal algorithm used to find the shortest path (using some heuristic function) between a source and target vertex. A* minimizes the path function $f(n) = g(n) + h(n)$ where $g(n)$ is the true cost from the start node to node n and $h(n)$ is the heuristic function estimating the cost from node n to the target node. This algorithm is typically implemented sequentially using priority queues and an algorithm similar to Breadth First Search. The node with the lowest f score, or estimated score to reach the target node, has its neighbors expanded, updating the scores for each one until the target node is reached and the path is reconstructed.

Our implementation of A* was designed to work on square grid based graphs where each element in the grid has up to 4 neighbors adjacent to them. This reason for this is that it doesn't really make sense to use A* on simple graphs because they do not benefit much from heuristic functions. For simple graphs, Dijkstra's algorithm would likely perform better.

Our input files are structured as follows:

```
<dimension>
<x_00> <x_01> <x_02> ... <x_0n>
<x_10> <x_11> ... <x_1n>
...
<x_n0> <x_n1> ... <x_nn>
```

where n is one less than the dimension of the grid and x_{ij} is 0 if that square is unavailable to be moved onto and 1 if it is available.

Our output files are be structured as follows:

```
<path_length>
(r0, c0) (r1, c1) ... (rn, cn)
```

where (r_0, c_0) is the starting node of the path and (r_n, c_n) is the ending node of the path.

There are four data structures shared between the various implementations.

A priority queue ordered by the f score of the node (or the estimated path cost to each the target node). Each implementation has a supporting unordered_set to allow for more efficient look-ups of nodes that are currently waiting to be processed. The priority queue is inserted into and popped from as is the open set with insertions and deletions.

A dictionary that keeps track of the relationships between nodes. It's structured in the following form cameFrom[child] = parent. This data structure is used to reconstruct the path once the target node is found as the path can be traced back by searching for the parent of each node until the source node is found. The dictionary is updated whenever a neighbor is processed, assuming that this would not make a parent come from its child. Thus, before each update is made, keys in the dictionary must be looked up.

Another dictionary that keeps track of the current g score of each node n (or the true path cost of reaching node n from the source node). It is initialized to INT_MAX for every node at the start. A key value pair is updated every time a neighbor is processed where the g score of the current node is less than a neighbor's g score.

A 1 dimensional array of integers representing the 2D grid. Indices into the grid can be converted to their row column counter part and vice versa. This is mainly used for generating neighbors nodes to be processed.

Whenever a node is processed, it updates the key data structures for each of its available neighbors (which can be up to 4 in our implementation). Then, each of those neighbors will expand up to 4 more neighbors. It is easy to see how quickly the amount of computation that needs to be done grows. Moreover, each neighbor can be processed in parallel since the only dependency for processing a node is that one of its neighbors must first be processed. This is the aspect of the algorithm that could benefit the most from parallelization. This algorithm is not data parallel since all of the computations on the data are not known beforehand and there exist dependencies between neighboring nodes. It is, however, task parallel as multiple nodes have the potential to be processed in parallel. Also, there is some locality in array accesses. 2 of the neighbors of each node as adjacent in the grid array; however, the neighbor above and below may be on separate cache lines.

3 Approach

We broke our implementation up into different parallelization strategies and also languages. The two parallelization strategies were a centralized and decentralized approach, as described in *A Survey of Parallel A* (Fukunaga, Botea, Jinnai, Kishimoto)*. We wrote the centralized algorithm in both C++ with pthreads and

mutexes and in Go using goroutines and channels. The decentralized algorithm was written in C++ using Open_MPI. Our target platforms were the GHC machines and PSC machines.

The centralized algorithm was implemented using global shared data structures across threads. Each thread picks up the next node in the open set to expand and iterate upon, so load balancing is done automatically and is balanced as long as the open set has nodes, although this is not always the case and depends on the problem size. Each thread expands one of the best nodes in the current open set and generates its children nodes, updating the shared open set. This approach is severely bottlenecked by the shared global data structure since synchronization needs to be in place.

The C++ pthread solution is largely similar to the sequential version except for the need to know when another thread has found an optimal solution. Threads will go into a waiting state whenever the priority queue that sorts open nodes by current cost is empty or the next best one has a higher cost than the current cost. When every node is in the waiting state, it can be determined that an optimal solution exists and the algorithm can terminate. The tricky part of the pthread version is the use of mutexes to ensure there are no data races. Multiple mutexes were used (one for each data structure) so that as much work as possible could be done concurrently.

The Golang centralized solution mirrored the parallel algorithm of the C++ pthread version. It was implemented in a much different way. We wanted to compare the performance of pthreads and mutexes with Golang goroutines and channel communication. The Golang solution was implemented such that there was one routine running for each mutex in the C++ implementation that had a for select block to synchronize pieces of code that a mutex would do. Every goroutine would send data on channels in order to synchronize the modification of global data structures.

Overall for the centralized applications there was no clear mapping between each thread and the work it was to do. Every thread just took turns pulling off the queue of open nodes to expand them. We iterated upon the centralized algorithm since at one point we had termination sometimes and not others, and this was due to an error in ordering data structure updates. We also experimented with different heuristic functions for the graphs we ran our solutions against. Overall Manhattan distance seemed the best for our 4-direction graph traversal both in terms of solution path cost and speed since Euclidean distance provides a more accurate estimation when diagonal travel is allowed.

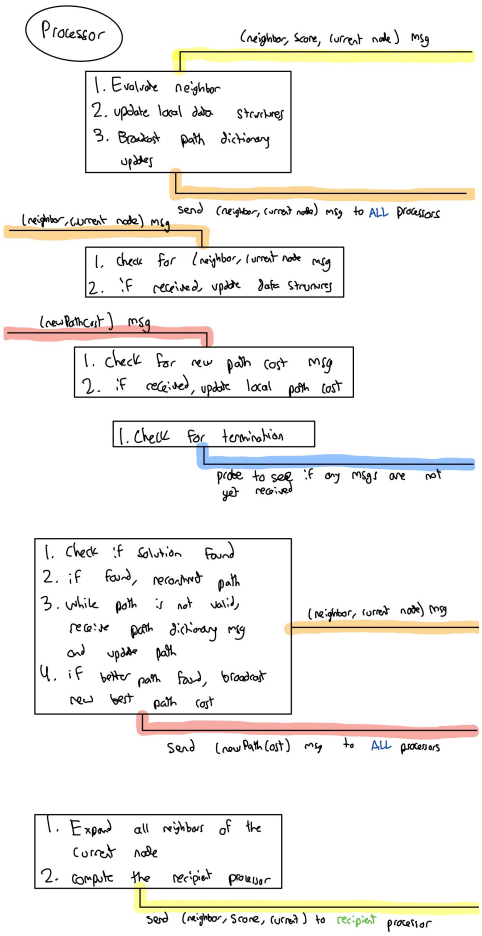
The decentralized version of A* restructures the algorithm more significantly than the centralized version. In this approach, each processor has its own local data structures that it operates on to mitigate the bottleneck of multiple threads frequently reading from and writing to shared data structures. Rather than expanding and processing the neighbors of the current node, the neighbors are sent to be processed by another processor. Moreover, there are 2 other types of messages that must be sent. Since each processor now has its own local data structures, any updates to the dictionary mapping parent and child relationships must be broadcast to all other processors. This is because the processor which evaluates the target node must be able to reconstruct the path to the source node in order to obtain the solution. Furthermore, whenever a processor finds a valid path solution, it must broadcast its cost to all of the other threads so that any threads that find another solution can know whether or not their path is better than the current path found. To implement this, we used Open_MPI with asynchronous message sending.

The termination condition for the decentralized algorithm is also more complex than the centralized counterpart. Not only must each processor have no more nodes to process in their priority queue but there also must not be any messages currently in transit that may lead to a better solution. If the messages in transit

are not checked, it is possible for the algorithm to return a less than optimal solution.

Load balancing becomes quite important in the decentralized approach. Sending a significant proportion of nodes to a single processor becomes a bottleneck in the system. The naive implementation we tried sent messages to processors uniformly at random; however, the random nature of this may cause some processors to have heavier workloads than others, leading to less speedup. In order to resolve this issue, we used a multiplicative hashing function to distribute the nodes relatively equally among all of the processors. The hash function is structured as follows $M(s) = H(k(s)), H(k) = \lfloor p(kA - \lfloor kA \rfloor) \rfloor$ where s is the state, $k(s)$ is the key mapping to the state, p is the number of processors, and A is the golden ratio (used because its been proven to work well for this hash function). Since all of the nodes in our implementation were integers, the key function simply returned the node value itself. The implication of this hashing is that each processor will have a relatively equal subset of nodes that "belong" to it and processors will always send certain nodes to specific processors which "own" that node.

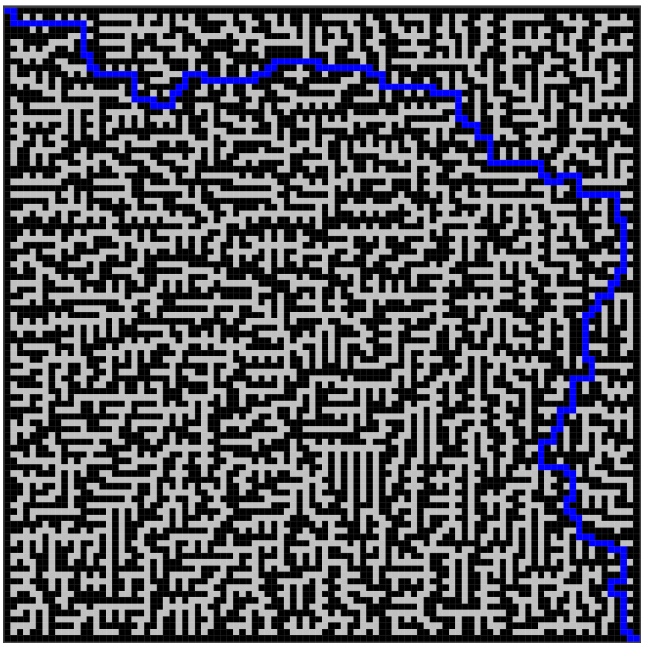
Below is a diagram of the algorithm and communication in the decentralized implementation of A*:



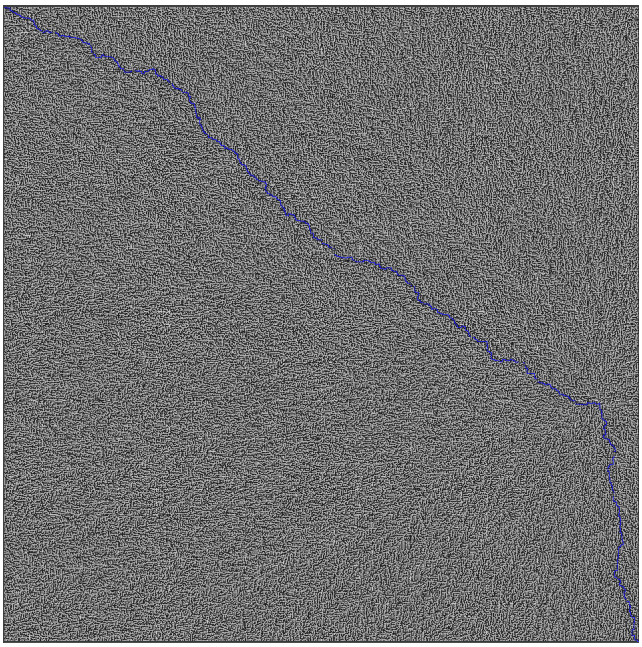
4 Results

In order to test our results we wrote a validating program in Python to ensure the solution returned by our A* implementations is the optimal solution as determined by Dijkstra's algorithm. Our input files varied in size and complexity. The input files were text files with each space separated cell a 1 for a valid move or a 0 for an obstacle. We had small and large input files that were all 1s, and thus a solution from corner to corner should closely mirror a diagonal line. We also had small and large input files that were generated mazes where A* would have to heavily search for a path that led to the end. We typically tested inputs by trying to generate a path from the top left corner to the bottom right corner.

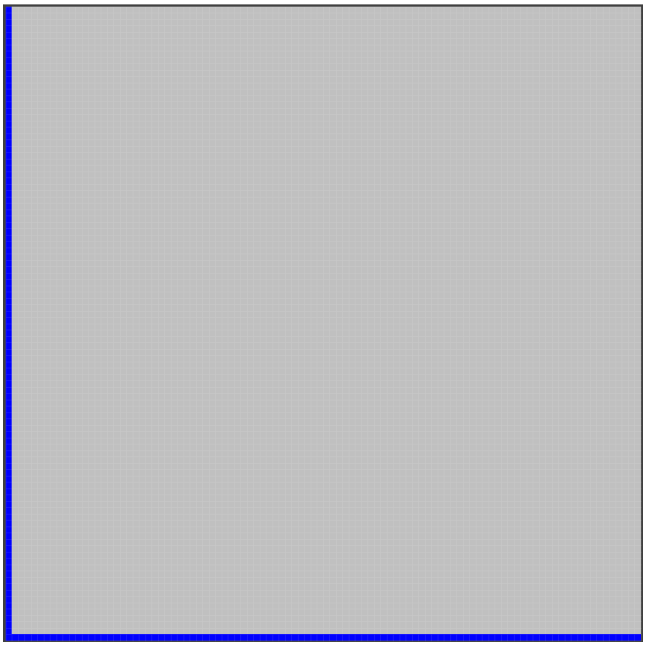
We wrote a Java application to visualize the paths that our A* implementations found. This app mirrored the code from the WireGrapher application from assignments 3 and 4. Below are some outputs that show our generated paths.



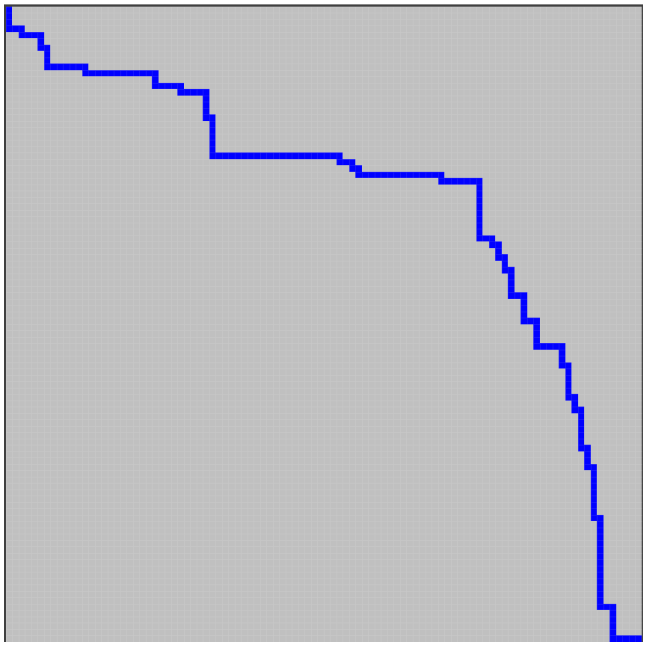
Go Centralized on 100x100 Maze



Go Centralized on 1000x1000 Maze



MPI Decentralized on 100x100 No Obstacles

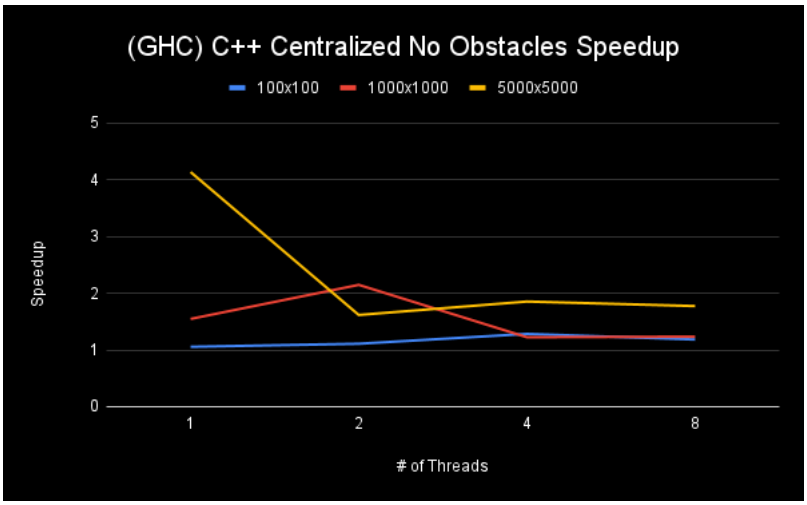


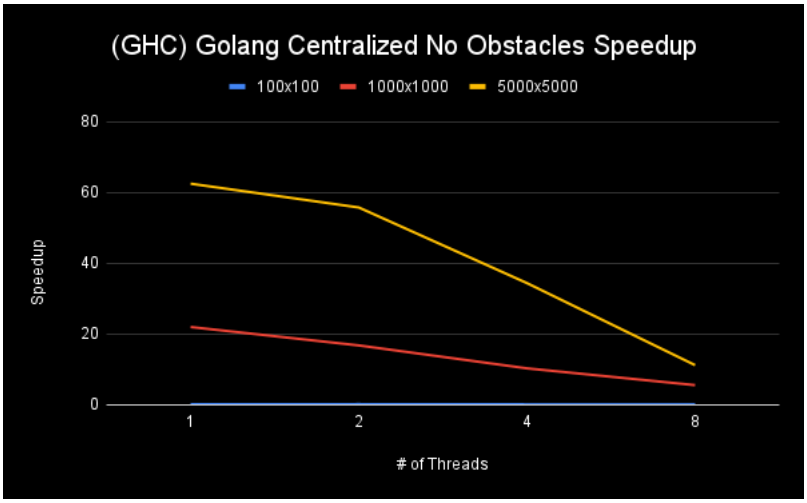
Pthread Centralized on 100x100 No Obstacles

We measured performance by calculating the speedup for thread counts ranging from 1 to 8 threads.

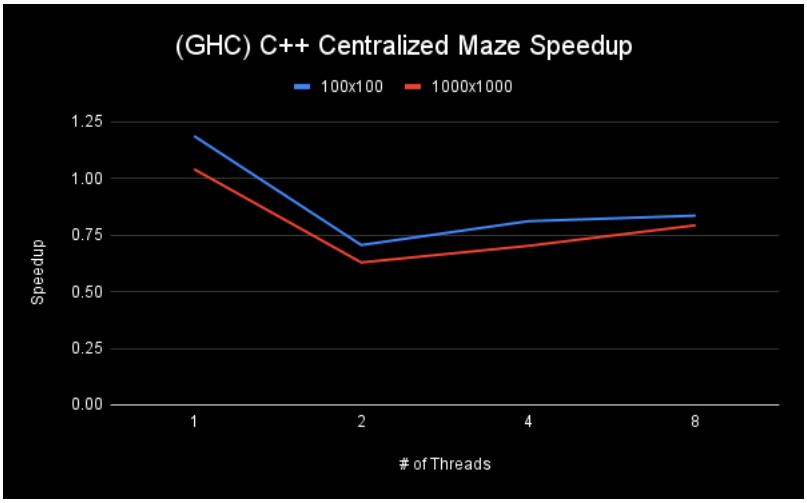
The input sizes were 100x100, 1000x1000, and 5000x5000. The input graphs themselves for each of these sizes were also configured differently to test the performance of our parallel implementations under a variety of conditions. The types were an open grid (no_obstacles), a maze, and a wall across the middle of the grid with only a few nodes available to pass through. Paths were taken from top left to bottom right corner.

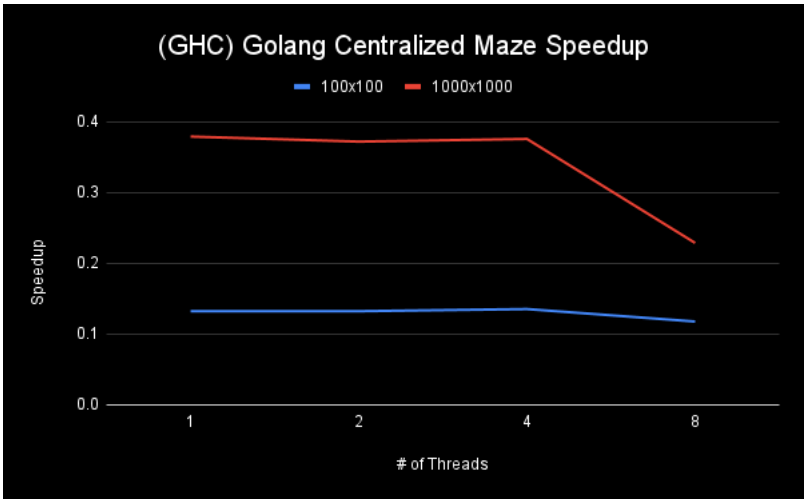
Below are some performance graphs for the various implementations. The baseline for each was the single-threaded CPU code.



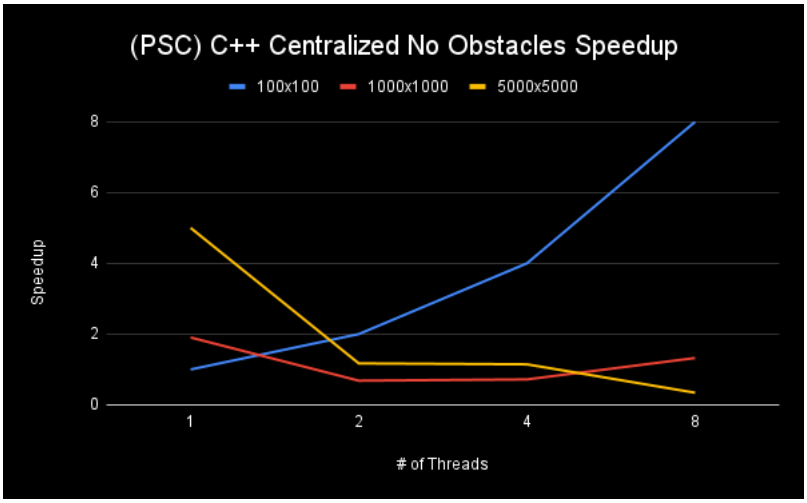


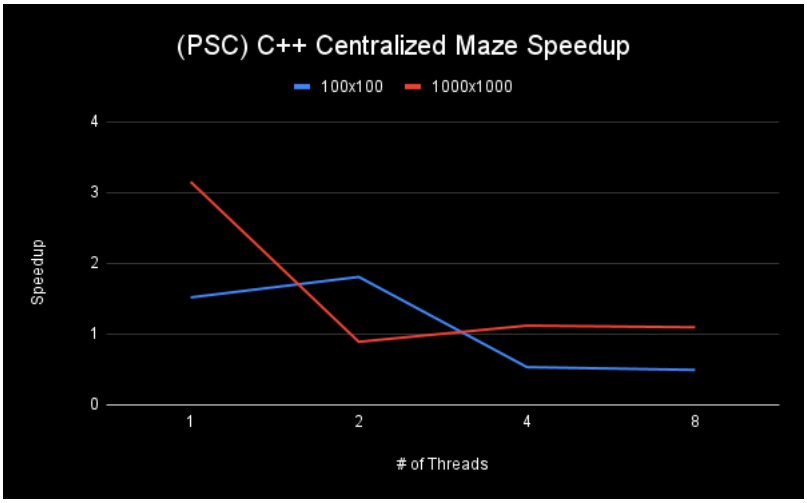
We saw that the Golang channel based parallel implementation saw much greater speedups across all threads than the C++ pthread implementation for the larger input sizes. For the 100x100, we observed that the Golang implementation takes much longer regardless of the threading. This is likely due to the communication overhead of using channels. This is further supported by the fact that the larger input sizes exhibited less speedup as the thread count increased. Contrasting this with the C++ implementation using mutexes, we observed a lower but more consistent speedup across all threads. For the 100x100, we saw little to no speedup relative to the C++ sequential implementation. The larger input sizes once again exhibited the same behavior of lower speedups when thread counts increase. This is likely due to the centralized nature of each implementation. By using locks to read and write to shared data structures, the system is severely bottlenecked as thread counts increase. When only a single thread is operating, we would have expected the parallel implementations to be slower than the sequential; however, we made some optimizations to reduce duplicate node processing in the parallel implementations which is likely why there is still speedup when the parallel implementation is run with a single thread.





The maze input test case features only a single possible path from the top left corner to the bottom right corner. The C++ implementation exihibited little to no speedup while the Golang parallel implementation was slower than its sequential counterpart. For this project, the problem size and specific problem conditions, i.e. what kind of space the path traversal is going through, influence the performance a lot. The lack of speedup demonstrates that the maze tests didn't benefit as much from more threads since there is only one path to the exit with a lot of nodes only having 2 neighbors (one of which being where you came from). The benefit of parallelizing A* comes from being able to process nodes in parallel. With less nodes being available to be processed, the parallelization instead introduces overhead in the form of synchronization or communication overhead (depending on the type of implementation). Thus, for solving mazes, it may be more wise to use a different type of algorithm as the basis for parallelization as A* is better suited for finding paths when most nodes are available.





We found that the speedups for PSC were higher than that on GHC. This is probably due to the fact that the individual cores on PSC are less powerful than that on GHC but when multiple cores run in parallel there is a higher speedup.

We wanted to write the centralized A* algorithm in C++ and Go for multiple reasons. First, we wanted to see how different the difficulty and effort was for programming with Pthreads and using mutexes for synchronization versus launching goroutines that communicate and sync with channels. We found that writing the Go implementation ended up being more code and a little more complex using channels, but the language itself made development much more easier. In Go we also had to implement an interface to create a priority queue while in C++ we could just use the standard library one. In retrospect we could have used mutexes for synchronization in Go which would have been easier and a bit cleaner.

We also wanted to compare which language resulted in better performance. It was a mix depending on the problem size. For smaller inputs and the maze inputs, C++ performed better and computation time was faster. We did find though that for the 1000x1000 and 5000x5000 no_obstacle inputs that Go performed much better than C++. It's worth digging further into how Pthreads and Goroutines work under the hood to better understand why we saw this difference in performance.

Our implementation of the decentralized algorithm did not perform as well as we would have anticipated. The original goal with decentralizing the computation was to mitigate the synchronization overhead we observed in the pthread implementation that used locks. However, there are still a few aspects of the algorithm that require synchronization, namely the cameFrom dictionary used to generate the path and the cost of the path itself. In particular, synchronizing the cameFrom dictionary introduced a massive amount of communication overhead. A majority of the messages being broadcasted were found to be these dictionary update messages. This is because as the input size increases by 1, the number of potential node pairs that could be inserted into the dictionary increases by approximately $4n$ where n is the dimension of the input graph. Moreover, the communication pattern is all to all because there is no guarantee of which processor will be the one to process the solution node.

A further optimization that could be attempted would be only having the processor which the target node hashes to keep track of the path. This could potentially decrease the amount of communication necessary, decreasing this overhead. This is likely to improve the speedup because we found that on average, the de-

centralized implementation spends about 50%-70% of the execution time attempting to generate the path by receiving these dictionary updates. This is an overhead of 50%-70% relative to the sequential implementation which can reconstruct the path immediately upon finding the solution node.

Here are some measurements we took with 4 processors and 8 processors on the 100x100 no obstacles grid with a path from the top left corner to the top right corner.

```
Initialization Time for proc 2: 0.013819.
Initialization Time for proc 3: 0.012804.
Initialization Time for proc 1: 0.014877.
Initialization Time for proc 0: 0.015835.
Total time to find target node for 2: 42.515331
Total time to find path for 2: 98.753193
Computation Time for proc 3: 98.754109
Computation Time for proc 0: 98.754026
Computation Time for proc 1: 98.754791
Computation Time for proc 2: 98.882473
```

```
Initialization Time for proc 2: 0.016524.
Initialization Time for proc 4: 0.014931.
Initialization Time for proc 5: 0.014208.
Initialization Time for proc 7: 0.012786.
Initialization Time for proc 0: 0.018662.
Initialization Time for proc 1: 0.017571.
Initialization Time for proc 3: 0.015625.
Initialization Time for proc 6: 0.013600.
Total time to find target node for 5: 63.629781
Total time to find path for 5: 181.862731
Computation Time for proc 4: 181.864129
Computation Time for proc 3: 181.864104
Computation Time for proc 6: 181.864043
Computation Time for proc 1: 181.864162
Computation Time for proc 0: 181.864294
Computation Time for proc 2: 181.864389
Computation Time for proc 7: 181.865656
Computation Time for proc 5: 182.898804
```

We see that as the number of threads doubles, the proportion of time spent reconstructing the path roughly doubles whereas the time spent to find the solution node only increases by roughly a factor of $\frac{1}{2}$. This would increase the speedup, however, one may notice that the time for even finding the solution node is still much larger than the sequential implementation. This may be due to the smaller input size in part.

5 References

- https://arxiv.org/pdf/1708.05296.pdf
- https://en.wikipedia.org/wiki/A*_search_algorithm
- https://www.mpich.org/documentation/guides/
- http://www.mathcs.emory.edu/~cheung/Courses/561/Syllabus/92-MPI/async.html
- https://github.com/cmu15418s22/Assignment-4
- http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html

6 Work Distribution

Jake's Contribution

- C++ sequential implementation

- Completion of C++ centralized implementation
- C++ MPI decentralized implementation
- Python validation script
- Java visualization application

Will's Contribution

- Go sequential implementation
- Go centralized implementation
- Initial C++ centralized implementation

Estimated Distribution: Jake 60% - Will 40%