Parallel Sampling-Based Motion Planning Using Probabilistic Road Maps (PRMs)

Prithu Pareek[12], Omkar Savkur[1]

Abstract

This paper presents an exploration of implementing the Probabilistic Road Map (PRM) algorithm using principles of parallel computation. Our implementation attempted to parallelize the three phases of PRMs: Generation, Connection, and Query. It utilizes the shared address space memory model of parallelism with the help of the OpenMP programming interface. The planner was tested rigorously with varying map sizes, obstacle densities, sample amounts, and number of threads on the Pittsburgh Supercomputer Center (PSC) machines. Our findings show no benefit from parallelism in the Generation or Query phases, but good speedup for the - most time consuming - Connection phase.

Introduction

Planning the motion of a robot is a challenging task and becomes even more challenging when attempting to plan in unknown environments. To solve this issue several motion planning tools - known as sampling-based planning algorithms - have been developed. Instead of using an explicit representation of the environment, these algorithms sample random points from a robot's state space, check them for validity and use them for planning. The two most popular of these sampling-based motion planning algorithms are Probabilistic Road Maps (PRMs) and Rapidly-exploring Random Trees (RRTs) [3][5]. Our project will focus on PRMs.

PRMs work in three stages: Generation, Connection, and Query [3]. They are discussed in more detail in the following sections.

[1] Prithu Pareek and Omkar Savkur are with the Carnegie Institute of Technology, Carnegie Mellon University. ppareek@andrew.cmu.edu, osavkur@andrew.cmu.edu
[2] Prithu Pareek is with the Robotics Institute, Carnegie Mellon University, ppareek@andrew.cmu.edu

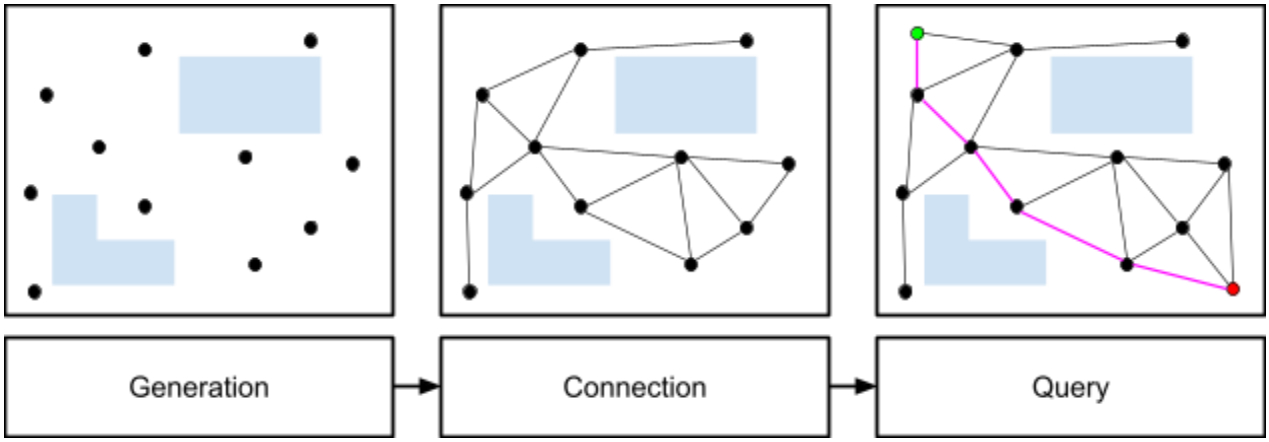

Figure 1: *The three stages of the PRM algorithm: Generation, Connection, and Query. Nodes are shown as black dots, edges as lines. The start and goal states are shown in green and red respectively and the final path is shown in pink.*

Generation Phase

In the Generation phase, the robot's state space is randomly sampled using a normal distribution to generate valid robot states. These states are collision-checked for validity against known obstacles in the environment and will become nodes in the graph searched for a valid path. The PRM algorithm is independent of the collision checker used and thus usually an efficient collision checker tailored to the required configuration space and robot is designed and plugged into the algorithm. This can be a potentially computationally intensive step depending on the dimensions of the robot's state (configuration) space. Furthermore, the quality of your generated path increases with the number of states sampled, which presents a tradeoff between motion optimality and efficiency.

Connection Phase

Once a set of valid nodes is found, the Connection phase begins. In this phase, each node is attempted to be connected to neighboring nodes that are close to it. The nodes that are allowed to be connected to it are determined by a user-tunable heuristic - usually a maximum distance threshold - and all connections between nodes must be collision-free paths. The validity of a path is checked by a local planner that divides up the path into small chunks and runs a collision check on each of them. In this way, there is a large amount of computational load during this stage of PRMs. Finally, we have a graph of nodes connected to each other with edges.

Query Phase

In the query phase, the start and goal states are attempted to be connected to the graph following a similar strategy to the preceding connection phase. Then the graph is searched for a valid path from start to goal using a graph search algorithm (in our case A*) [4]. This stage of the path planning process finds an optimal path from start to goal on the sampled graph. It does

this by intelligently expanding nodes in the graph using a heuristic, such that when it expands the goal node, it would have found the shortest path. When each node is expanded it stores a pointer to its parent so that once the algorithm arrives at the goal state, it can just follow the parent pointer on each node back to the start to determine the complete path. While A* is already a fairly efficient algorithm, there is always room to make it faster.

Given these inefficiencies in all three steps of PRMs, we think they would benefit greatly from parallelization.

Our Approach

We utilized a Shared Address Space model with OpenMP for parallelizing our PRM algorithm. The implementation is targeted for the Pittsburgh Supercomputing Center (PSC) Machines with 128 cores split amongst two AMD EPYC 7742 CPUs, each with 64 cores [CITE Bridges-2].

In the parallel implementation, OpenMP uses the fork-join method to spawn many threads, each executing on a different core of the node that was allocated by the PSC Machine to run this algorithm. This implementation had a global graph that all threads updated and read from. This was stored in an adjacency list format, where each node was stored as an entry in a vector. Each node also had a linked list containing the other nodes that were connected to it (to represent the edges between nodes). Each node also had the workspace coordinates that the node corresponded to within its struct fields.

In the generation phase, each thread randomly sampled a coordinate in the workspace, determined if the point belonged to the freespace by querying the map for if any obstacle occupied that space. If the point was in the freespace, the thread would add the node to the graph. Otherwise, the thread would loop and pick another random pair of coordinates in the workspace. The threads do not have to check if the point has already been sampled before by any other thread. The idea is that the algorithm samples enough points of the freespace that the number of duplicates is kept to a minimum. We used a dynamic scheduling algorithm to determine how many points a thread should sample because we noticed that a static scheduling algorithm - the default being blocked assignment - would have a fairly low utilization of all the cores. We saw that the majority nodes generated in this phase would only require a few iterations to select a point in the freespace. However, a few unlucky attempts would have to loop several more times in order to find a point in the free space. This means that each thread would have an unpredictable completion time for the sampling portion, and if a thread got very unlucky, all of the nodes it was trying to place could all take several attempts. This would mean that all the other threads would have to wait for that thread to complete before they could all move on to the next phase of the planner. We did experiment with a static scheduled assignment and because of our implementation, it would have better cache utilization when adding to the vector of nodes, but the work distribution was very uneven, as mentioned before, so we decided to switch to a dynamic scheduled assignment.

In the connection phase, we parallelized over the nodes in the vector. The algorithm requires each node to look at all the other nodes to determine which ones are the closest and use a local planner to determine which of the closest neighbors can be connected, meaning that there are no obstacles between them. Each thread is assigned a group of nodes in a blocked assignment and loops over all the other nodes to determine which ones are within the distance threshold (we set ours to 25 units) and determine if the nodes could be connected with an 'L' path - either horizontal or vertical first. The local planner did not have to be the most optimal since it is a probabilistic model, meaning that if the algorithm samples more points, it is more likely to find a solution or a better solution. We initially had a critical section around the part where each thread added an edge to the adjacency list graph, but we realized that it only adds 1 direction of the edge and since each thread would look at a different source node at once, each thread would be adding edges to a different linked list at the same time. Therefore, we could get rid of our critical section and improve our speedup for the connection phase by allowing simultaneous updates to the graph.

We did not have to modify the Generation and Connection Phases algorithms much to make them run in parallel [1]. But, we did have to make quite a few modifications to implement A* in parallel - the main one being that we implemented Hashed Distributed A* (HDA*), where each node is assigned to a specific core through a hash function [2]. Then, each core goes through its open list of nodes assigned to it in order of highest priority based on the heuristic. Then, for each of the nodes connected to that node, it pushes that node onto the open list of the processor assigned to that node, provided that the node does not already exist in that processor's closed set. One issue we faced was the exit condition for all of the threads. Since the threads are executing concurrently, there is no guarantee that the shortest path will be found once a thread opens the goal node. In the sequential version, once the algorithm opens the goal node, since the heuristic is admissible and consistent, the shortest path is the one found first. However, in the parallel version, a processor may open the goal node before the other processors can find a shorter path to the goal node, leading to a non-optimal path to the goal [2].

In our first attempt at parallelizing A*, we did not implement HDA* and tried expanding the nodes in parallel. Each thread would look at one neighbor of the node currently open, calculate the heuristic value for that node, and determine if that node has already been seen before by checking the closed set. This did not provide much speedup because there was not enough work for each thread to do, especially considering the overhead required to fork and join threads on different cores. We did not sample enough nodes to have enough neighbors for the threads to compute the heuristic for. We could have sampled more points to guarantee more work for each thread in this model, but the purpose of a PRM is to sample enough of the workspace to get a good model of where the robot can travel and get a path that was optimized enough. Sampling more of the workspace has diminishing returns where it becomes more work, and a more optimal solution is not guaranteed. Also, most of the time spent in A* was accessing memory for checking if a particular node existed already in the closed set, and that had to be done sequentially since there was the possibility that multiple nodes were at the same

coordinates since the generation phase did not check for this. All of these reasons pointed towards us implementing HDA*.

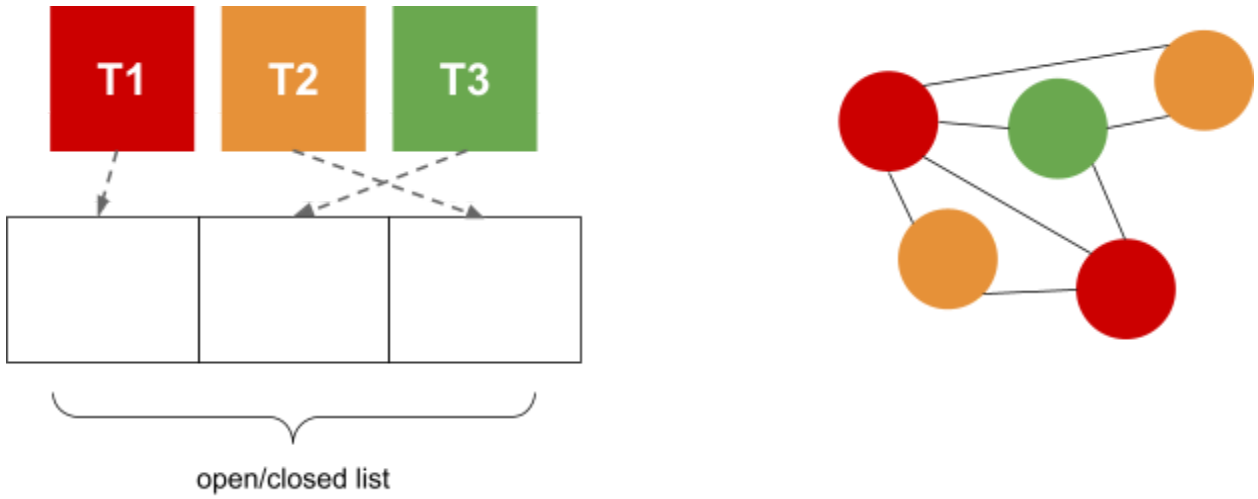

Figure 2: *In HDA each processor is assigned its own open and closed list in shared memory and can access the open and closed lists belonging to other processors. This is shown on the left. Each processor is also assigned a series of nodes in the graph, shown on the right.*

We had to have as many open lists and closed sets as there were cores. And since each core could access any other core's open list or closed set, we needed to add mutexes for each of the open lists and closed sets to guarantee atomic operations on these data structures. To solve the issues of exit condition and optimality mentioned above, we followed a similar model to [2] where we had a barrier after each thread found that their open list was empty. Then, after all threads reach the barrier, all threads check if their open list is still empty. This is because other threads could have inserted more nodes into a core's open list after that core thinks its work is done. Then, if any core still has work to do, all the cores check their open lists and wait at the barrier again. This process continues until all cores find that their open lists are empty. After implementing this and trying to find the most optimal path to the goal given our probabilistic graph, we noticed that the program was taking much longer than the usual sequential version since all processors had to expand all possible paths. We then made the decision to not return the most optimal solution for a given graph, but rather a close heuristic to the best case solution.

We stop our HDA* algorithm after the goal node is expanded all processors think they finished expanding their open lists. This way, we can get much faster execution time at the cost of a slightly worse path to our goal. Since we are running a PRM algorithm, which itself is based on the premise that it will return a "good enough" solution and if you need a better solution, you increase the number of samples and hope you get a better one, even the most optimal path return by A* may not be the globally optimal solution for a given map. This means that we could justify returning a close heuristic to the most optimal solution, which would help improve our execution time. Although the path length does increase (for the most part) as the number of cores increase, the quality of the path does not degrade too much as the number of cores increase.

Experimental Results

Experiment Setup

We measured our performance both in terms of speedup achieved for using multiple cores as well as by how much the path distance increased when adding more cores. We defined our own metric, called Path Degrading Factor, to standardize how much the path distance has increased for different sized maps. With P cores running, we define Path Degrading Factor as the path distance for p threads divided by the path distance for 1 thread. This allows us to compare path degrading for across map sizes. The 1 Thread execution is running HDA* but with only 1 open list and closed set, so it is running normal A*, providing us with the minimum path distance for our given graph. From that, we can determine how much the path distance increased relative to the baseline path distance.

For our experiment, we had 4 different map sizes - small, medium, large, and extra-large - as well as 4 different obstacle densities - empty, sparse, medium-density, and dense. All maps were discretized and assumed 4-point connectivity. The small map was 500x500 units; the medium map was 1000x1000 units; the large map was 2000x2000 units; the extra-large map was 10000x5627 units. The small, medium, and large maps had the same obstacle density as medium and large were tessellations of the small map. The extra-large map was a stress test of our algorithm where we increased both the density of obstacles and the size of the map to see a more “real-world” example of a path planning exercise. The empty, sparse, medium-density, and dense maps were all 1000x1000 units and varied how much of the workspace was occupied by obstacles. The empty map had no obstacles, the sparse map had 25% of the workspace occupied by obstacles; the medium-density map had 50% of the workspace occupied by obstacles; the dense map had 75% of the workspace occupied by obstacles. We sampled all of the maps using 1% sampling rate, meaning that the number of points sampled in the generation phase was equal to 1% of the number of points in the workspace. We also sampled the medium size map using a 2% sampling rate to see if changing the number of nodes in our graph for the same size map had an impact on speedup or our Path Degrading Factor. All experiments were run on the PSC machines with 1, 2, 4, 8, 16, 32, 64, and 128 threads.

Our baseline that we are referencing when calculating speedup is our parallel algorithm running on a single core. This should have a similar execution time to an optimized version of a sequential version of the PRM algorithm because with 1 thread, that thread is assigned all the nodes in HDA*, and the Generation and Connection phases done in parallel utilized the data parallelism inherent to the PRM algorithm.

Results

The results from our trials are presented in graphs below.

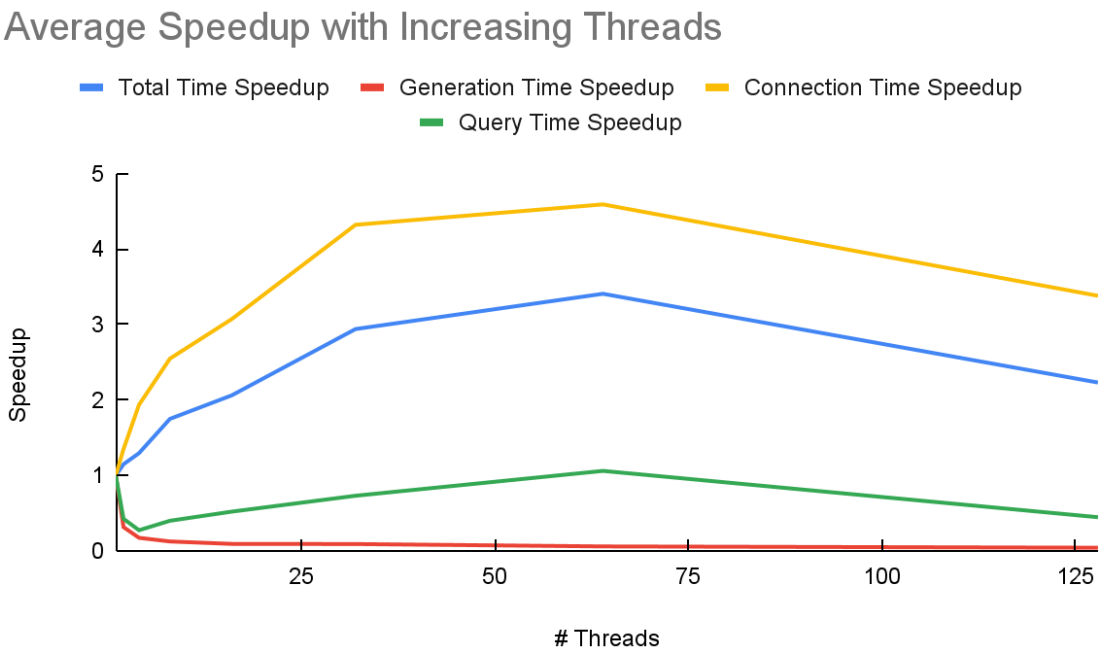

Figure 3: *Graph of the average speedup provided by our parallel implementation over an increasing number of threads. The graph contains the overall speedup for the entire planning process, as well as a breakdown for each of the three stages: Generation. Connection and Query.*

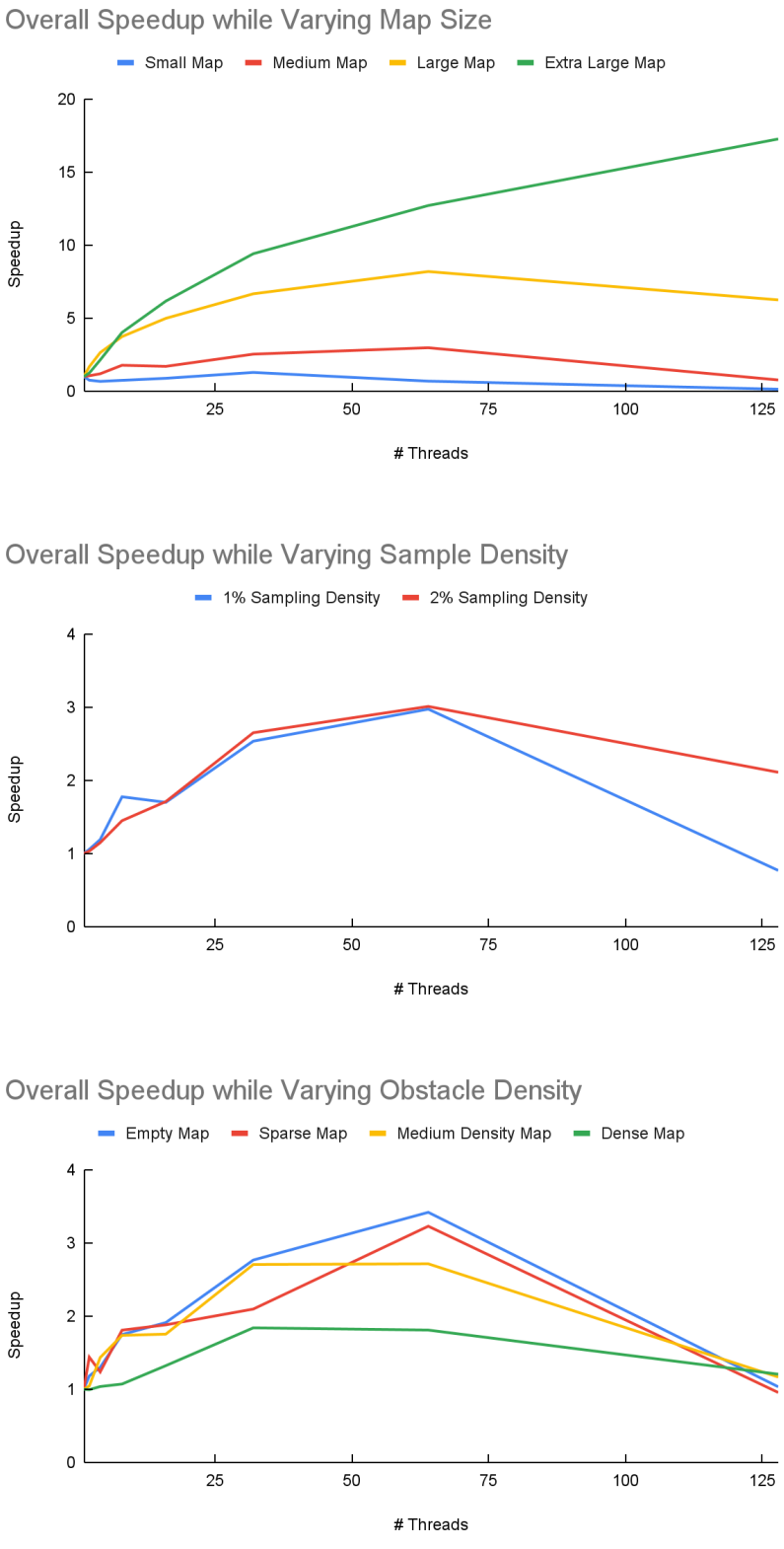

Figure 4: *Graphs showing the change in overall speedup of our parallel planner implementation while varying map size, number of points sampled and map obstacle density.*

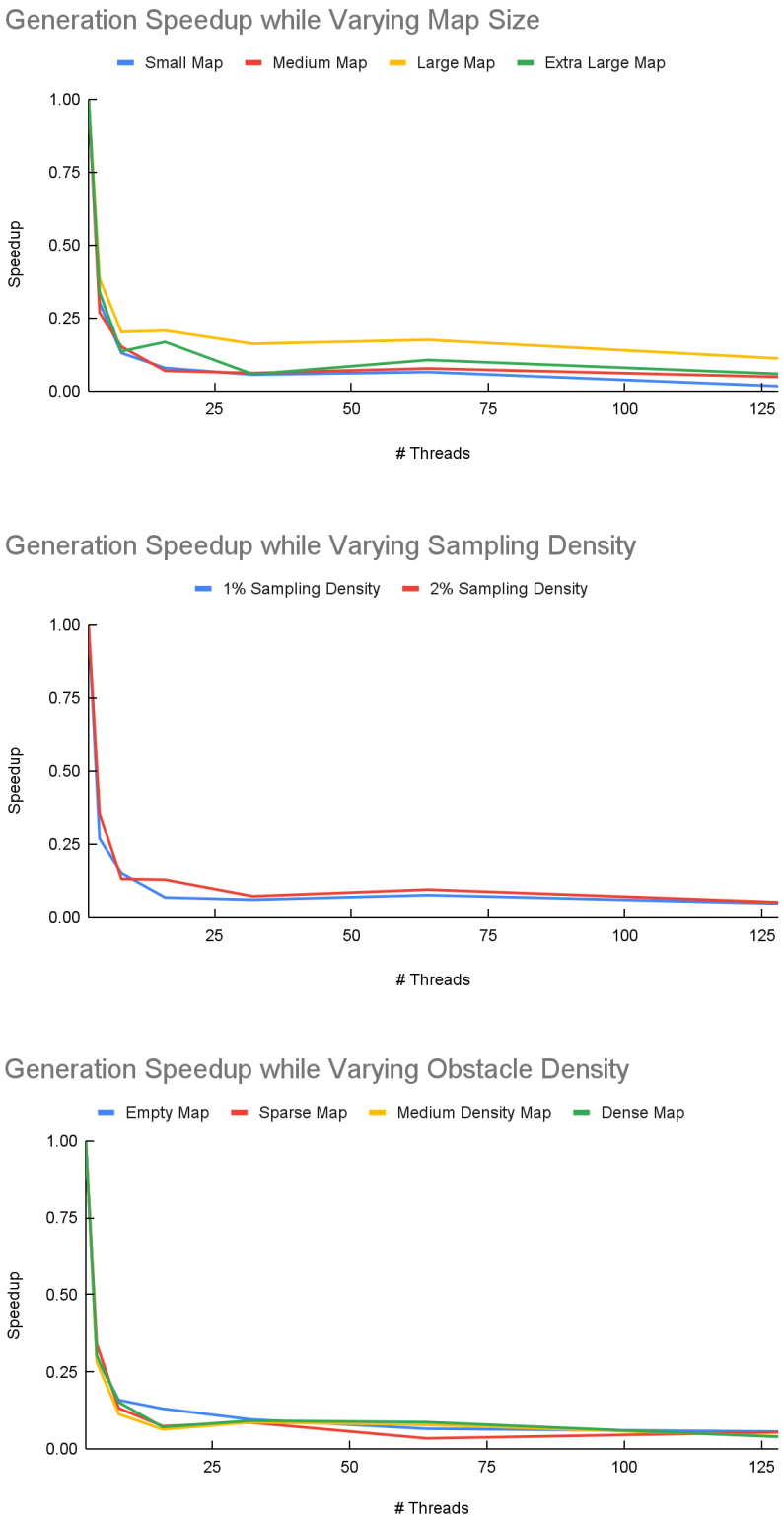

Figure 5: *Graphs showing the change in speedup for the generation phase of the planner while varying map size, number of points sampled and map obstacle density.*

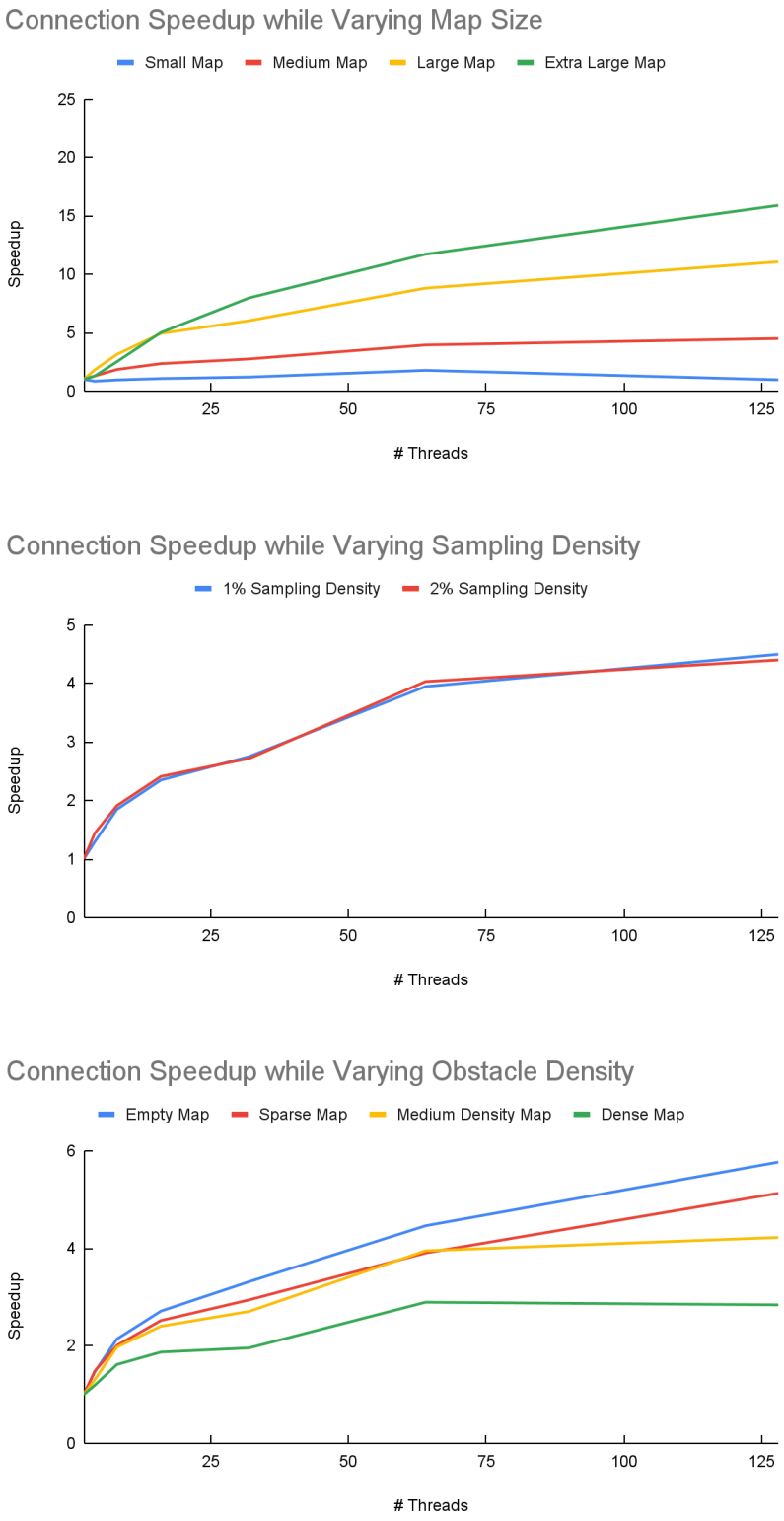

Figure 6: *Graphs showing the change in speedup for the connection phase of the planner while varying map size, number of points sampled and map obstacle density.*

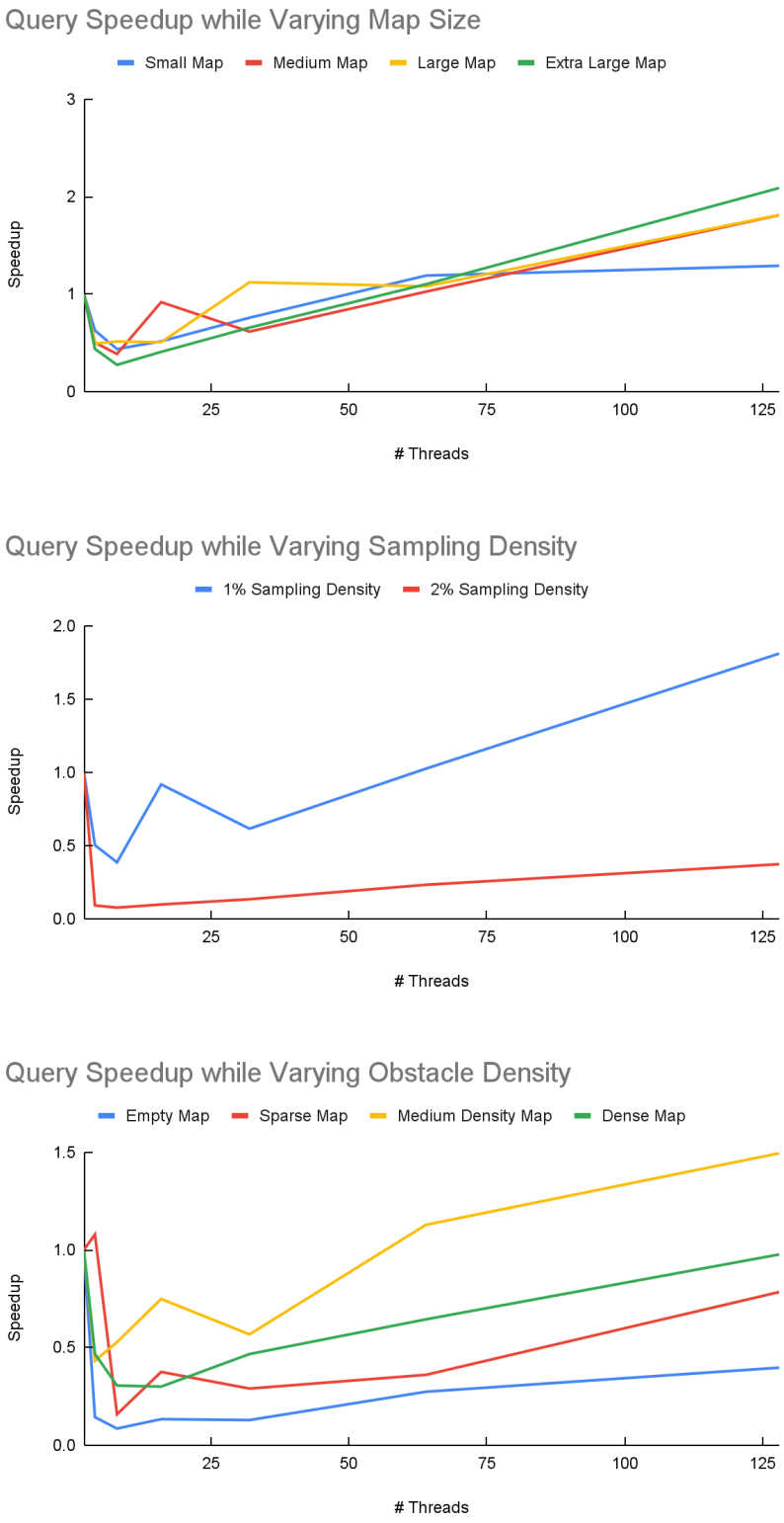

Figure 7: *Graphs showing the change in speedup for the query phase of the planner while varying map size, number of points sampled and map obstacle density.*

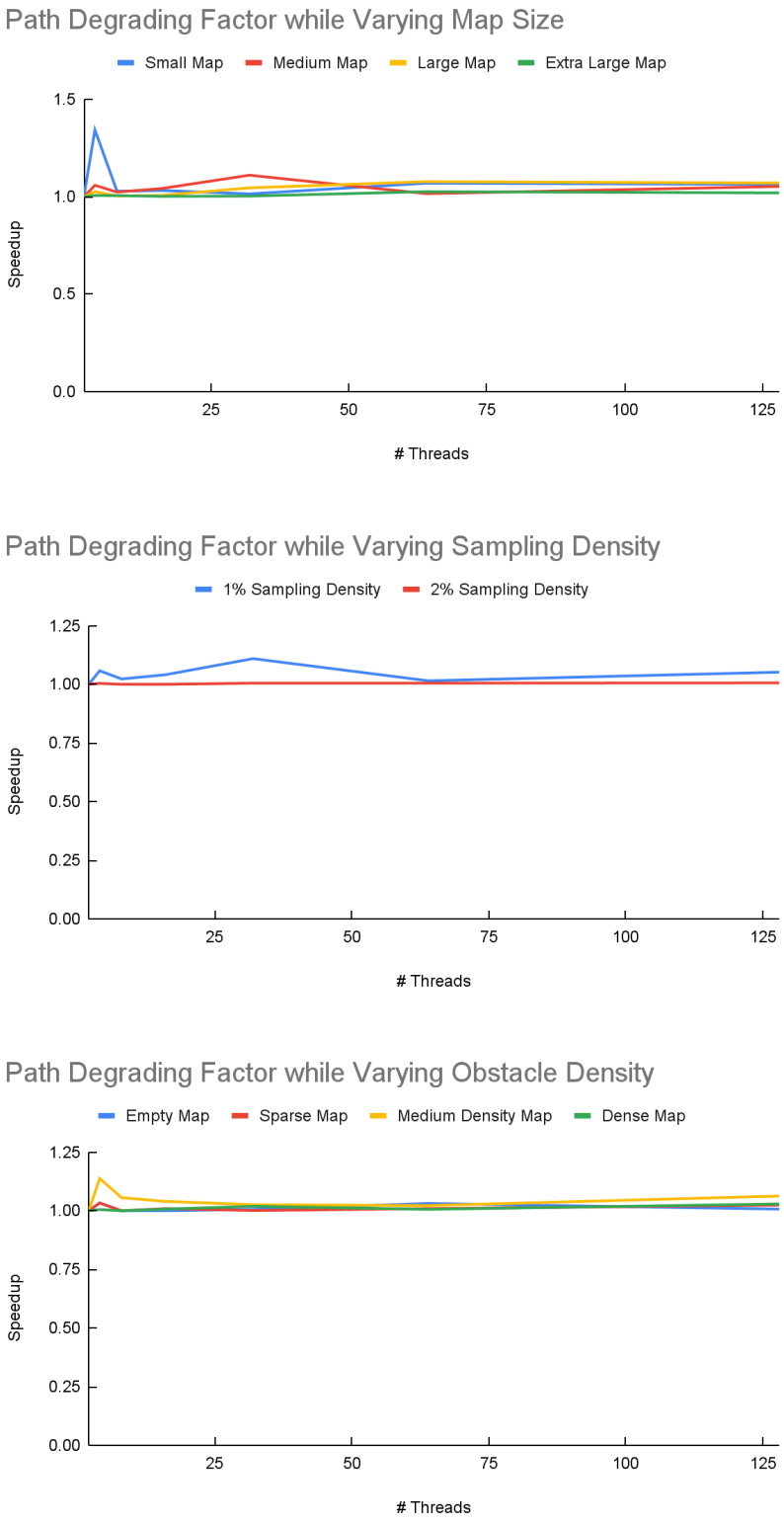

Figure 8: *Graphs showing the change in path quality of the planner while varying map size, number of points sampled and map obstacle density. The path degrading factor is defined as the path distance for p threads divided by the path distance for 1 thread.*

Analysis

Overall Speedup

For the overall speedup of our parallel implementation of the PRM planner we saw a consistent increase in speedup until 64 threads (Figure 3). After this point we started to see a drop in speedup with 128 threads.

When varying map size, our algorithm seems to perform better with larger maps. With the XL map, we see a continuous increase in speedup when using more threads (Figure 4). With 128 threads, this map has a speedup of 17.25x. The medium and large maps achieve their maximum speedup at 64 threads: 2.97x and 8.12x respectively (Figure 4). The small map only sees a speedup at 32 threads (1.28x) and has a continuous drop in performance after that. We believe that these changes are due to the connection phase since most of the time of the algorithm is spent here. With larger maps we are sampling an increased number of points which results in a larger graph size and reduced false sharing.

When looking at the effect of sampling density on overall speedup, for the most part there doesn't seem to be a significant difference between 1 and 2 percent (Figure 4). Similar conclusions can be drawn when planning over maps with different obstacle densities (Figure 4). However it appears that Dense maps have a lower speedup across all thread counts (Figure 4). We believe this has to do with false sharing during the Connection Phase since the sampled points are concentrated in areas very close to each other.

When breaking the speedups down into the speedups for each of the three phases of PRMs - Generation, Connection, and Query - we saw varied results.

Generation Speedup

The trajectories of the Generation Phase are very different than that of the overall speedup (Figure 5). They follow what looks to be like a negative exponential curve and look to be the same regardless of map size, sampling density, and obstacle density (Figure 5). At 2 threads, the speedup for the Generation Phase is around 0.3, and at 4 threads, the speedup is around 0.15. By 128 threads, the speedup for the Generation Phase is around 0.03. The overhead needed to parallelize this phase seems to be increasing with the number of threads. We believe this results from the fact that all threads have to add new nodes to the same graph data-structure, which is adding to the memory access overhead as a result of parallelization.

Connection Speedup

The overall trajectory of the Connection Phase speedups were similar to that of the overall speedup. However, this phase had a speedup 1.5x faster than the overall speedup. We believe that this is because of the natural parallelism that exists in the connection phase since each thread is responsible for finding neighbors of a specific set of nodes in the graph.

They look to vary with map size similar to the overall speedup (Figure 6). Larger maps result in a higher speedup then smaller ones do. In fact on the XL map, the speedup is linear with the number of threads until 8 threads. As we increase the graph size, we believe that this reduces false sharing in the Connection Phase when accessing and writing to the adjacency list that represents the graph.

There doesn't seem to be any change in speedup with respect to the sample density (Figure 6). When looking at varying obstacle density however, there is increased speedup on sparser maps compared to denser ones (Figure 6). This is a result of the sampled points being close to each other in dense maps since there is less free space for the same number of samples. This results in multiple threads having to access the same cache lines in memory, which in turn results in a reduction in speedup.

Overall we think that the memory access to the graph data structure is the main limiting factor in speedup.

Query Speedup

Parallelization of the Query Phase resulted in a sharp slowdown with more than one thread (Figure 7). The speedup then increased (still less than one) until 64 threads before decreasing again (Figure 7).

When looking at different map sizes, speedup tended to be higher for larger maps when using a high number of cores (Figure 7). This is most likely since each node had fewer neighbors for larger maps, which reduced the amount of time that a thread was being blocked by another. Similar results were seen when varying sampling density: less samples resulted in higher speedup. This is probably due to the same reason as above. Finally, when comparing across obstacle density, the highest speedup was found with a medium density map, followed by a desne map, sparse map, and then empty map.

The biggest bottleneck in performance in this phase was the overhead due to mutexes having to lock and unlock data structures shared by multiple threads. We think this is why we finally start to see a boost in performance - albeit a small one - with a large amount of threads that are able to overcome this overhead.

Path Degradation

When increasing the number of threads used, we found that the quality of the path does not significantly increase or decrease (Figure 8).

Conclusions & Future Work

Overall, we think that PRMs would benefit greatly from parallelization: specifically in the Connection Phase. In our trials, we found that a majority of the computation time is taken by this

step. Furthermore, with less than 8 threads we were able to achieve linear speedup. Even with more than 8 threads, there were still gains - albeit not as high. In addition to this, we want to further investigate other ways to parallelize the Generation and Query phase. For the Generation Phase, we want to try adding delays in the collision checker function to try and simulate actual computationally intensive collision checking to see if we do see speedup with our current parallel implementation. For the Query phase, we want to explore using a message-passing implementation to see if we can reduce the overhead we currently have with the shared-memory model. In conclusion, we believe that path planning - especially PRM - can be improved with parallel computation and should be an area of continued research.

References

[1] N.M. Amato and L.K. Dale. 1999. Probabilistic roadmap methods are embarrassingly parallel. In *Proceedings 1999 IEEE International Conference on Robotics and Automation (Cat. No.99CH36288C)*, IEEE, Detroit, MI, USA, 688–694. DOI:https://doi.org/10.1109/ROBOT.1999.770055

[2] Ariana Weinstock and Rachel Holladay. 2016. Parallel A* Graph Search. Retrieved from https://people.csail.mit.edu/rholladay/docs/parallel_search_report.pdf

[3] L.E. Kavraki, P. Svestka, J.-C. Latombe, and M.H. Overmars. 1996. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Trans. Robot. Automat.* 12, 4 (August 1996), 566–580. DOI:https://doi.org/10.1109/70.508439

[4] Peter Hart, Nils Nilsson, and Bertram Raphael. 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Trans. Syst. Sci. Cyber.* 4, 2 (1968), 100–107. DOI:https://doi.org/10.1109/TSSC.1968.300136

[5] Steven M LaValle. 1998. Rapidly-exploring random trees: A new tool for path planning. (1998).

Appendix

Raw Performance Data

Map	# Samples	# Threads	Gen Time (ns)	Conn Time (ns)	Query Time (ns)	Total Time (s)	Path Nodes	Path Distance
Small	2500	1	284745	22070100	9990260	0.0323826	1081	1024
		2	939528	26052800	15923100	0.0429527	1454	1374
		4	2185430	23136300	22899100	0.04825	1110	1052
		8	3603000	20574100	19360300	0.0435712	1116	1056
		16	5063970	18396100	13191500	0.0366843	1095	1038
		32	4372890	12430100	8396520	0.0252362	1154	1094
		64	16532000	22813100	7743730	0.0471278	1146	1088
		128	24271100	22978600	184147000	0.237553	1208	1146
	5000	1	520650	88723900	33322700	0.122623	1058	998
		2	2718980	104669000	71671400	0.179119	1063	1006
		4	3415650	94684500	107323000	0.20547	1057	1002
		8	8176720	87411200	66082000	0.161711	1069	1012
		16	6609380	79200000	71171400	0.157036	1057	1000
		32	9985180	51698900	38519300	0.100245	1096	1038
		64	18712900	53157400	26698100	0.0987526	1068	1014
		128	26066400	75489500	98817400	0.200545	1076	1024
Medium	10000	1	1118500	282664000	44461200	0.328307	2118	2008
		2	4158830	218033000	88425000	0.310678	2241	2124
		4	7354280	152982000	115301000	0.275692	2167	2054
		8	16134300	120299000	48475500	0.184953	2206	2090
		16	18111600	102762000	72278200	0.193224	2351	2228
		32	14452800	71613700	43380200	0.129525	2146	2038
		64	22975700	62852200	24564300	0.110458	2228	2112
		128	45909100	185801000	195156000	0.426956	2341	2220
	20000	1	2523070	1138190000	31059300	1.17185	2112	1998
		2	7084690	788037000	339899000	1.13511	2116	2006
		4	19053300	595419000	406229000	1.02079	2107	1998
		8	19449500	472036000	317316000	0.808879	2110	1998
		16	34245300	418784000	232447000	0.685544	2116	2008
		32	26230600	282256000	133648000	0.44221	2120	2008
		64	47379700	258648000	83294100	0.389413	2117	2010
		128	57114900	373606000	124499000	0.555407	2124	2016
Large	40000	1	8460150	4220200000	137830000	4.36692	4244	4026
		2	21941600	2310810000	279524000	2.61238	4346	4128
		4	41810300	1344700000	268239000	1.65486	4266	4038
		8	40896300	855667000	272206000	1.16886	4265	4048
		16	52230100	700651000	122949000	0.875919	4438	4206
		32	48249800	479176000	127755000	0.655286	4567	4334
		64	75729100	381495000	76088800	0.533429	4539	4306
		128	107910000	393572000	197422000	0.699052	4384	4160
	80000	1	19420900	16771000000	205643000	16.9962	4223	3998
		2	40723200	9323860000	1319670000	10.6844	4221	4004
		4	71741000	5811730000	2906230000	8.78986	4214	4000
		8	81212900	3587950000	1318980000	4.98833	4215	3998
		16	134744000	2939270000	1183020000	4.25719	4218	4004
		32	105119000	2031280000	641175000	2.77779	4216	4000
		64	122063000	1698000000	560880000	2.38118	4218	4004
		128	181514000	1583370000	429905000	2.19507	4217	4006
Extra Large	562700	1	3.83E+08	8.77E+11	1.85E+10	895.808	16514	15629
		2	1.13E+09	6.79E+11	4.23E+10	721.938	16580	15711
		4	2.81E+09	3.49E+11	6.74E+10	419.166	16583	15707
		8	2.28E+09	1.75E+11	4.54E+10	222.842	16533	15645
		16	6.62E+09	1.10E+11	2.82E+10	145.341	16541	15667
		32	3.60E+09	7.49E+10	1.68E+10	95.2319	16901	16021
		64	6.49E+09	5.52E+10	8.86E+09	70.5149	16803	15923
		128	6.09E+09	3.92E+10	6.66E+09	51.9043	16686	15825

Map	# Samples	# Threads	Gen Time (ns)	Conn Time (ns)	Query Time (ns)	Total Time (s)	Path Nodes	Path Distance
Empty	10000	1	978912	274835000	7085130	0.282972	2111	1998
		2	3306470	186981000	49227700	0.239575	2175	2064
		4	6148530	128577000	82906900	0.217717	2105	1998
		8	7463980	101380000	53024500	0.161914	2110	1998
		16	10162900	82931300	54927100	0.148091	2120	2010
		32	14794600	61631000	25859100	0.102337	2171	2060
		64	17221500	47681600	17839800	0.0828087	2121	2012
		128	36419200	65342300	171885000	0.273723	2139	2034
	20000	1	1883500	1120730000	27754000	1.15044	2110	1998
		2	6264140	753707000	167922000	0.92798	2107	1998
		4	10208300	464101000	307325000	0.781715	2102	1998
		8	22597200	383614000	223005000	0.629317	2112	1998
		16	24710400	341690000	158797000	0.525284	2106	1998
		32	25288700	230365000	110389000	0.366114	2106	1998
		64	21360300	187784000	76575800	0.285815	2105	1998
		128	40862500	257837000	127900000	0.426703	2106	2000
Sparse	10000	1	996576	278270000	14522900	0.293848	2106	1998
		2	2924960	187712000	13459000	0.204144	2181	2064
		4	7520940	138647000	91473800	0.23769	2110	1998
		8	13341200	110520000	38684100	0.162587	2127	2016
		16	11563800	94639800	50042500	0.156307	2108	2000
		32	28552700	71328900	40286200	0.140232	2129	2016
		64	18176400	54279700	18522000	0.0910549	2152	2046
		128	40652200	100200000	166528000	0.307438	2241	2128
	20000	1	2019550	1120230000	78178800	1.2005	2113	1998
		2	5672160	776173000	169551000	0.95148	2108	1998
		4	17347200	548370000	386562000	0.952368	2104	1998
		8	17042800	430178000	311014000	0.758312	2107	1998
		16	28793500	382079000	171231000	0.582176	2112	2004
		32	22151900	254453000	105835000	0.382507	2111	1998
		64	42605400	220175000	89558800	0.352636	2107	1998
		128	49848400	334121000	113850000	0.498099	2106	1998
Medium_Density	10000	1	1083470	278887000	37005800	0.317038	2123	2008
		2	3849740	213779000	85639200	0.303332	2412	2284
		4	9528340	141461000	70109400	0.221151	2238	2120
		8	16949600	116182000	49441700	0.182618	2201	2088
		16	12448300	103123000	65204000	0.180841	2174	2060
		32	13694600	70658000	32798200	0.117231	2160	2052
		64	25895900	66090800	24764900	0.116828	2254	2134
		128	36605900	85034700	149847000	0.271598	2260	2148
	20000	1	2329980	1139170000	218300000	1.35988	2115	1998
		2	6105760	773180000	353055000	1.13244	2123	2014
		4	15613700	589032000	414211000	1.01894	2107	1998
		8	20092200	480629000	295120000	0.795927	2108	1998
		16	34001700	417563000	137641000	0.589278	2155	2040
		32	35056800	278314000	133973000	0.447511	2113	2002
		64	43388500	247938000	97808600	0.38927	2119	2012
		128	67193500	346106000	125006000	0.538443	2111	2002
Dense	10000	1	1630970	292439000	39813100	0.333952	2108	1998
		2	5388870	245386000	85061800	0.335906	2118	2008
		4	10717600	180992000	130272000	0.322034	2108	1998
		8	22700100	156293000	132652000	0.311697	2121	2012
		16	17682300	149492000	85316900	0.252561	2153	2036
		32	18636200	101114000	61774800	0.181593	2119	2010
		64	40810200	103019000	40775300	0.184664	2166	2056
		128	55878500	122660000	98323100	0.276974	2180	2070
	20000	1	3100320	1175050000	105603000	1.28385	2118	1998
		2	10331700	954451000	423908000	1.38878	2110	1998
		4	19800000	723223000	724389000	1.4675	2109	1998
		8	41925600	599680000	488035000	1.12973	2111	1998
		16	34557800	588915000	306563000	0.930107	2110	1998
		32	38864700	402925000	215084000	0.656955	2108	1998
		64	48581500	364369000	128511000	0.541544	2106	2000
		128	55243500	598528000	136627000	0.791153	2098	1998

List of Work By Student; Distribution of Credit

We worked on all tasks together and our work was divided basically equally.