Parallel RRT path planning for robots

Summary

We are going to parallelize rapidly exploring random path planning for robots on both CPU and GPU, and perform a detailed analysis of both systems performance. We have implemented a serial 2D version of the algorithm on two maps with obstacles. We have parallelized the algorithm on upto 8 cores using OpenMP and implemented the same algorithm in GPU using CUDA.

Background

Given a description of the robot, an initial configuration, a set of goal configurations, and a set of obstacles, the robot motion planning problem is to find a path that starts from the initial configuration and reaches a goal configuration while avoiding collision with the obstacles. The motion planning problem is known to be challenging from a computational point of view due to the exploratory nature of the problem. The complexity is increased further if the motion of the robot is constrained due to physical and mechanical parameters. Incremental sampling-based algorithms, such as the Rapidly-exploring Random Tree (RRT) algorithm are used to find a path if it exists. This algorithm is *probabilistically complete* in the sense that the probability that these algorithms return a solution, if one exists, converges to one as the number of samples approaches infinity.

Pseudo Code for RRT

```
Qgoal //region that identifies success
Counter = 0 //keeps track of iterations
lim = n //number of iterations algorithm should run for
G(V,E) //Graph containing edges and vertices, initialized as empty
While counter < lim:
    Xnew  = RandomPosition()
    if IsInObstacle(Xnew) == True:
        continue
    Xnearest = Nearest(G(V,E),Xnew) //find nearest vertex
    Link = Chain(Xnew,Xnearest)
    G.append(Link)
    if Xnew in Qgoal:
        Return G
Return G
```

The RRT algorithm is built on top of n-ary tree where each node in the tree hold the (x, y) coordinates on the 2D map. Along with it each node has an unique identification number and each node stores the UID of its parent node. The root node of the n-ary tree is the starting point on the map and the daughter nodes are all the possible coordinates the robot can move from the parent node.

The RRT algorithm starts with the starting point as the root node and then samples a random point in the 2D space. We then find the nearest node from the sampled point where the definition of distance is euclidean distance. Then the direction vector from the nearest node to the random sampled node is computed and a point in that direction at a predefined distance from the nearest node is computed. This point is then checked for collisions and if it is not obstructed, this new node is added to the tree with the parent node as the nearest node. The predefined distance is given by the variable 'q_step_size' and needs to be imputed by the user. This process is repeated unless a node is in the interior of a predefined radius with the target point as the centre. At this stage, we declare the algorithm has found a path. The predefined target radius is declared as 'target_p' and needs to be defined by the user. To summarize, the input arguments needed for the algorithm are starting and the target coordinates followed by 'q_step_size' and 'target_p'. Optionally we can specify the maximum number of nodes the algorithm should sample before it gives up.

The shared data structure is the tree and most computationally expensive task is to find the nearest node. There exists sophisticated data structures like KD-Tree that allows one to find the nearest node in logarithmic time. However we have developed a NaiveTree that iterates through all the nodes in order to find the nearest node. This is not the most efficient method but we are interested in the speed ups and the modifications required to make the algorithm parallel.

The most computational expensive function is that of finding the nearest node by iterating through all the nodes in the tree. This can easily be parallelized using openMP or MPI frameworks as we just need to parallelize a for loop that has a single shared variable that stores the minimum distance.

Approach

The underlying data structure of NaiveTree and the RRT algorithm is implemented completely in C++. The RRT implementation is translated from a python version that was implemented by one of our team members as a part of Robot Autonomy (18-662) coursework in Spring 2020. The NaiveTree class is implemented from scratch along with all the required data structures. RRT is a probabilistic algorithm that samples random points to add to the tree. This might result in significantly different execution times and path length for different runs with the same setup. To avoid and minimize the effects by random number generation, we set the random number generator seed to 0 before starting each run.

OpenMP

There were 2 approaches to parallelize the serial version using openMP. First approach was to parallelize just the function that computes the nearest node in the tree. This sped up the execution but there are still a lot of computations that had to be done by a single thread. This mainly included the random sampling and finding the node to be inserted. Also checking if the node was in collision or not and finally appending the node to the tree data structure. All these operations were still carried out by a single thread.

The second approach is to sample multiple random nodes in parallel, one by each thread. All the computation related to that node is done by the assigned thread. The section when this node is added to the tree data structure is serialized to avoid multiple nodes with the same UID. This approach changes the algorithm a bit as the nearest node returned by a thread may not be the nearest node as multiple nodes are sampled in parallel. In the worst case it will be the nth nearest node when running with n threads in parallel. However this does not change the outcome of the algorithm as RRT is probabilistically complete, hence it will still find a path but it will need to sample more nodes in order to do so.

CUDA

So as we increase the number of obstacles we notice that the time taken to check if the points collide with the obstacles in space. As the obstacles increase, the time taken also increases. So in CUDA we tackle this problem by parallelizing the obstacle checking. So each thread checks if the current point is colliding with each of the obstacles instead of sequentially checking these conditions.

Results

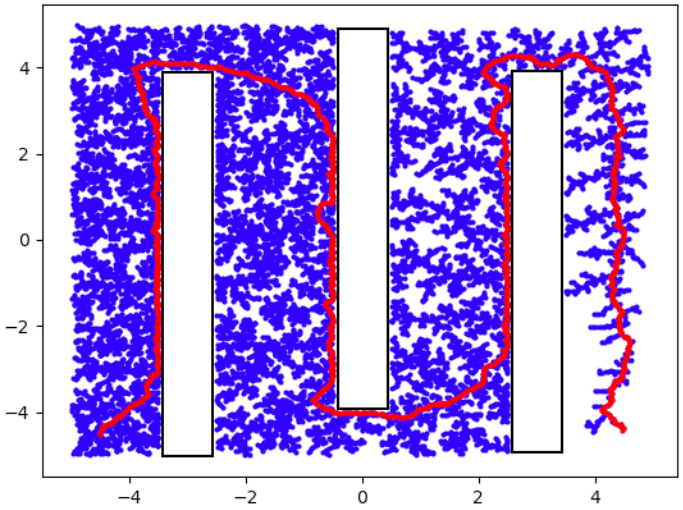
The experimental setup has defined 2 maps with appropriately placed obstacles to maximize the difficulty in finding the path. We perform a detailed analysis when the starting and target points are taken to be the extremes in the map. Then we take 6 more trails of randomly generated starting and target points to get a generalized idea of the performance of the parallelized version.

For OpenMP we used ghc48 machine which has Intel(R) Core(TM) i7-9700 CPU @ 3.00GHz and for the CUDA implementation we used ghc50 machine which has Nvidia GeForce RTX 2080.

OpenMP

Map 1 - We ran RRT with extreme points. The starting point is (-4.5, -4.5) while the target is

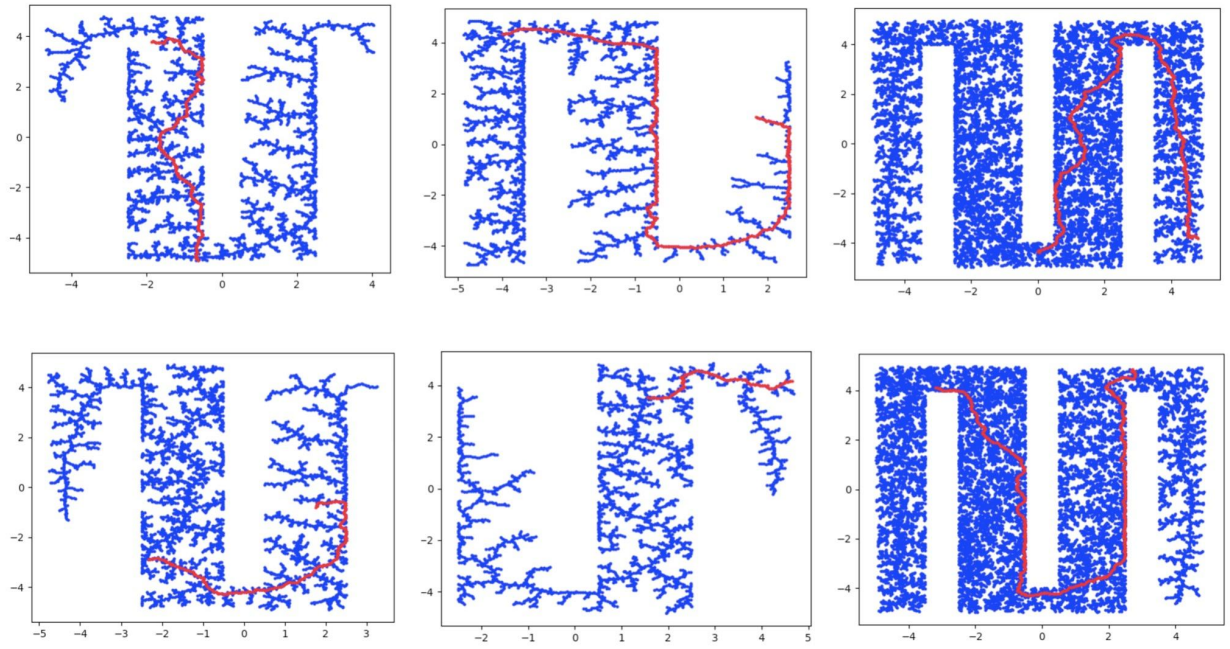
(4.5, -4.5). These points force the algorithm to find a zigzag path through the obstacles.



Speedups for 1,2,4,8 cores

Cores	Time (secs)	Nodes Sampled	In Collision	Path Length	Speed up
1	9.765	46,966	21,099	1630	1.00x
2	5.248	48,554	22,088	1553	1.86x
4	2.574	47,836	23,759	1560	3.79x
8	1.855	57,256	25,169	1541	5.26x

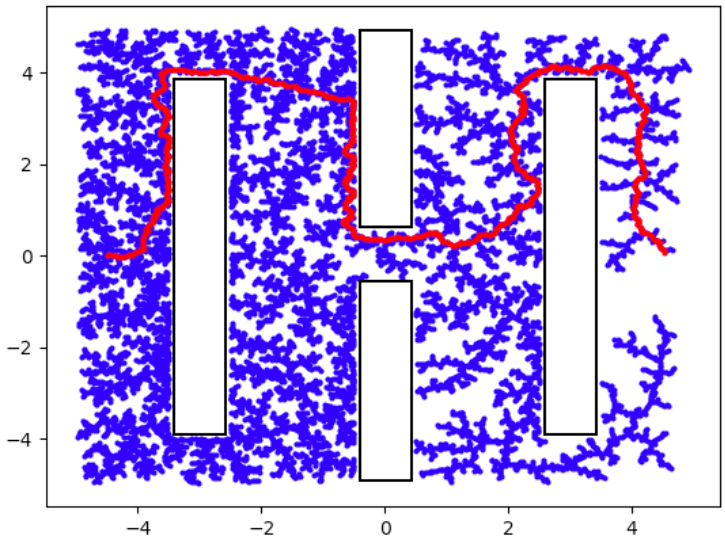
6 Random starting and target points



Speedups for 1 and 8 cores for average speedups

Starting Point	Target Point	Time (secs)		Speed up
		1 core	8 cores	
(-0.625, -4.995)	(-1.994, 3.663)	0.467	0.067	6.97x
(-3.987, 4.332)	(1.663, 0.998)	0.925	0.185	5.00x
(0.046, -4.387)	(4.998, -3.785)	47.809	6.567	7.28x
(-2.340, -2.876)	(1.567, -0.860)	2.191	0.330	6.63x
(1.567, 3.521)	(4.832, 4.198)	2.249	0.351	6.40x
(-3.197, 4.112)	(2.776, 4.910)	21.609	5.217	4.14x

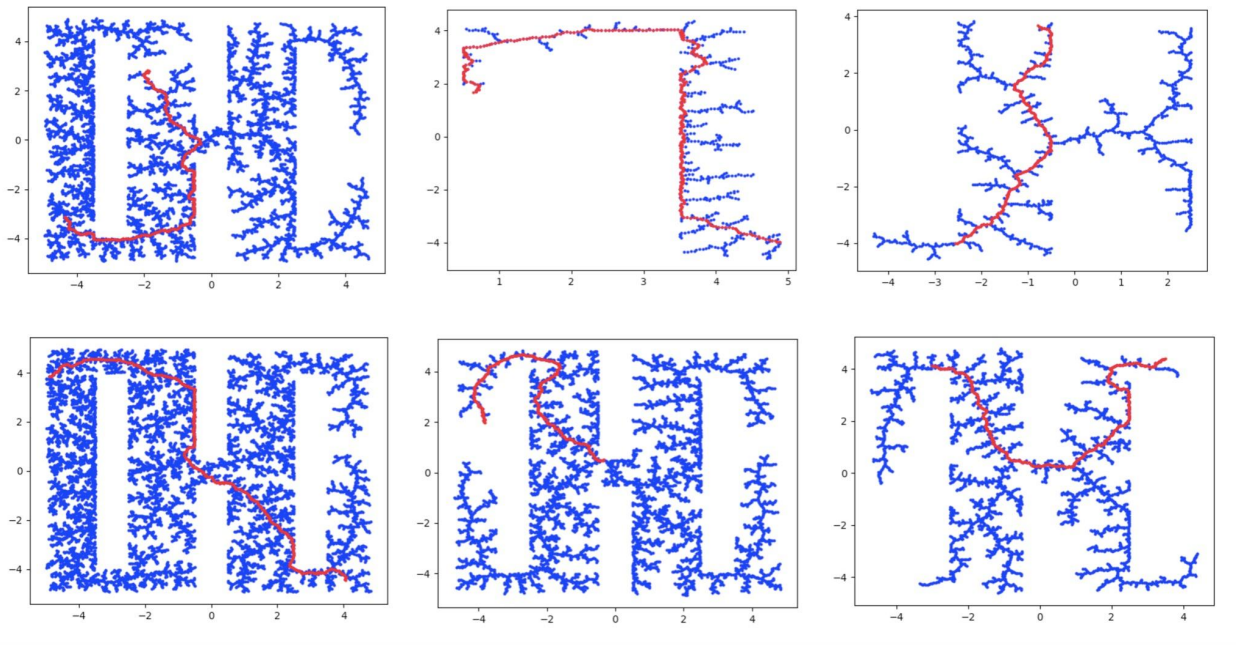
Map 2 - We ran RRT with extreme points. The starting point is (-4.5, 0) while the target point is (4.5, 0). These points force the algorithm to find a zigzag path through the obstacles while passing through a small opening between the obstacles.



Speedups for 1,2,4,8 cores

Cores	Time (secs)	Nodes Sampled	In Collision	Path Length	Speed up
1	9.441	39,240	12,280	884	1.00x
2	4.844	39,242	13,590	880	1.94x
4	2.973	39,160	9,240	815	3.17x
8	1.477	47,608	16,376	819	6.52x

6 Random starting and target points



Speedups for 1, 8 cores for average speedups

Starting Point	Target Point	Time (secs)		Speed up
		1 core	8 cores	
(4.886, -3.991)	(0.562, 1.532)	11.141	1.754	6.35x
(-4.368, -3.117)	(-1.987, 2.952)	0.780	0.107	7.2x
(-2.553, -4.013)	(-0.942, 3.776)	0.133	0.096	1.38x
(0.332, 0.514)	(-3.998, 1.871)	13.237	1.889	7.00x
(-4.883, 3.861)	(4.011, -4.818)	2.410	0.357	6.91x
(-2.998, 4.117)	(3.675, 4.491)	4.368	0.623	7.01x

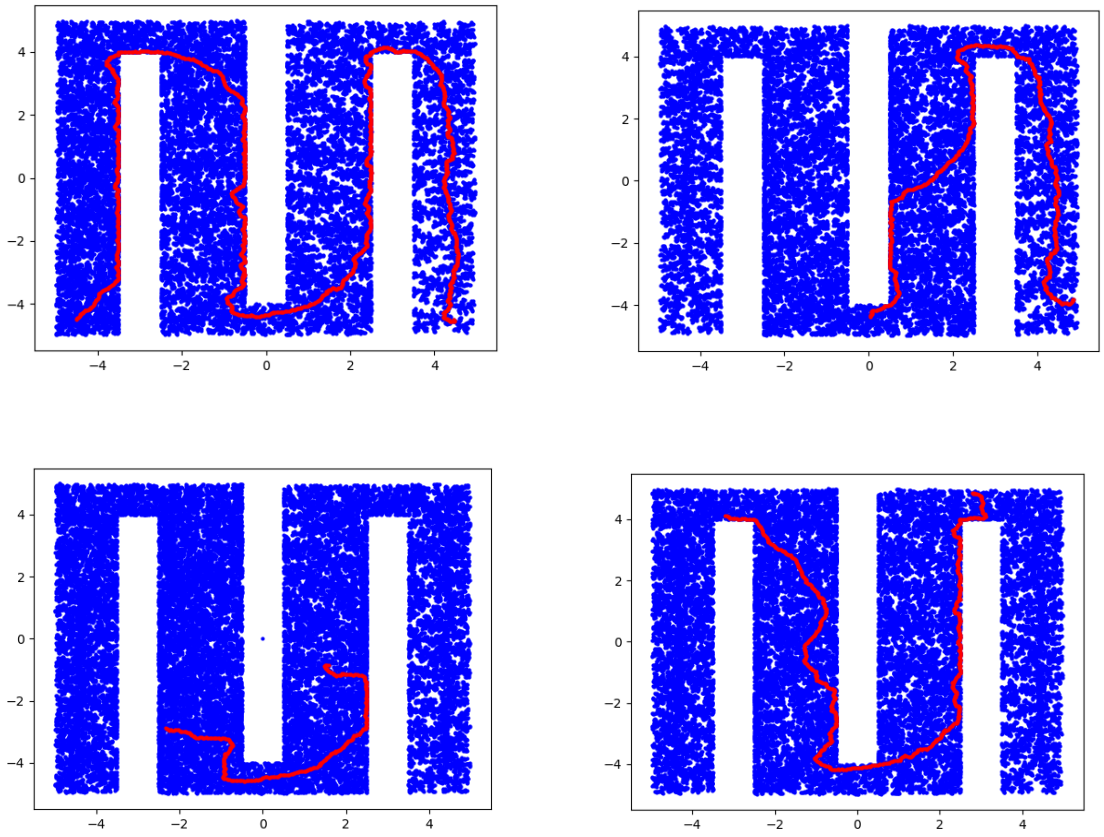
Key Observations (OpenMP)

We observe a sublinear speedup as we increase the number of cores. The average speedup for 8 cores from our experiments is approximately 6x. This is due to the fact that there is some serialisation when we modify the tree data structure. Along with it we observe that as the number of cores increases, the number of nodes sampled increases. This happens as the nearest node can be the nth nearest node when n threads are running in parallel. This makes it

necessary to sample more nodes to reach the target. Hence even from the same problem, as the number of threads increases, the total work done to find a path also increases.

The second prominent observation is that as the problem size increases, the parallel version performs better. This is because for small problem sizes, the number of nodes sampled is less and the ratio of incorrectly computed nearest nodes to the total nodes is high thus resulting in extra work and giving less speedup. As the problem size increases, this ratio goes down considerably thus giving a close resemblance to the linear speedup. There are some outliers observed in this trend as RRT depends on random sampling and in some cases it might find a valid path in less time if the algorithm gets lucky.

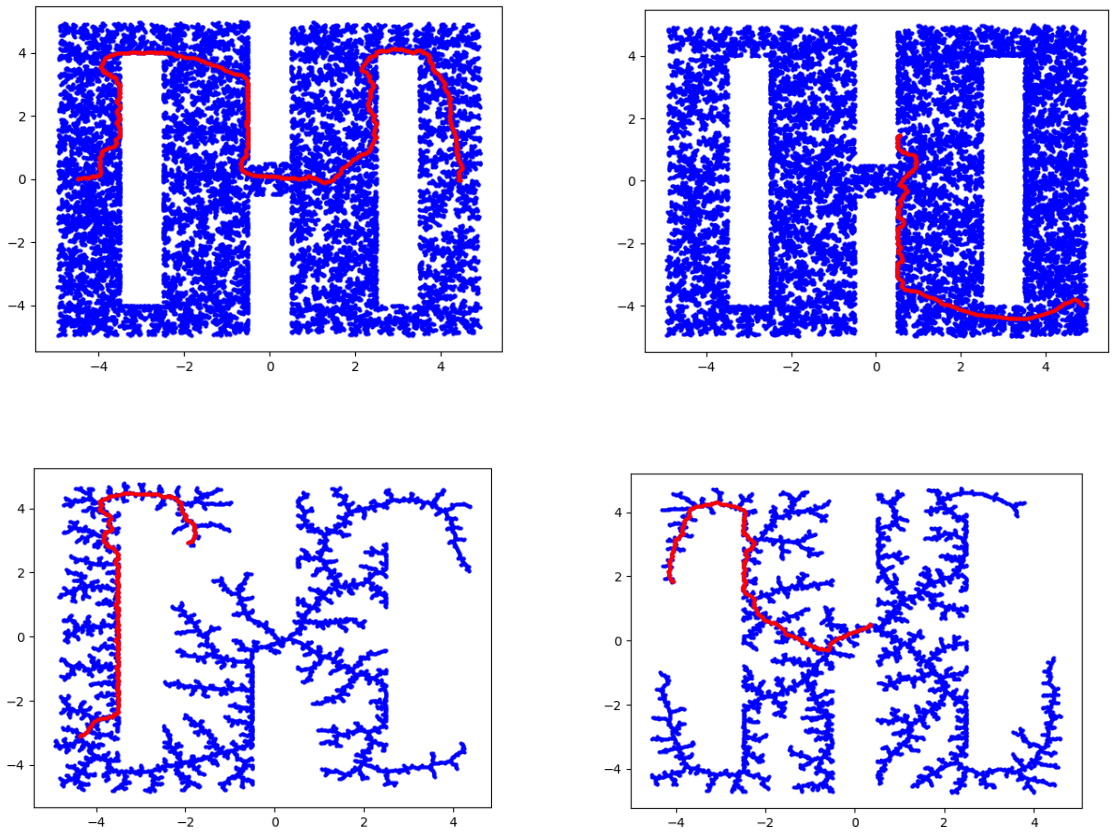
CUDA



Map 1

Starting Point	Target Point	Time (secs)	Speedup (vs 8 cores)
(-4.5, -4.5)	(4.5, -4.5)	2.411	0.76x
(0.046, -4.387)	(4.998, -3.785)	5.299	1.233x

(-2.340, -2.876)	(1.567, -0.860)	0.72	0.45x
(-3.197, 4.112)	(2.776, 4.910)	2.746	1.899x



Map 2

Starting Point	Target Point	Time (seconds)	Speedup (vs 8 cores)
(-4.5, 0)	(4.5,0)	0.62	2.38x
(4.886, -3.991)	(0.562, 1.532)	0.186	9.43x
(0.332, 0.514)	(-3.998, 1.871)	1.135	1.6643x
(-2.998, 4.117)	(3.675, 4.491)	0.507	1.229x

Key Observations (Cuda)

From the above results we can see that as we introduce more obstacles (Map 2 has more obstacles than Map 1) we can see a significant speedup in the performance compared to 8 core OpenMP. In Map 1 we can see that in case 1 and case 3 we see negative speedup this is

because more time is spent in sending memory to and fro from the GPU than calculating if the point is colliding with the obstacles. Whereas in Map 2 we increase the obstacles in the 2D space after which we can see significant speedup compared to the 8 core OpenMP solution. This shows that our interpretation of speedup with respect to collision checking proves. In the future we would like to explore in parallel the process of choosing the next point from an initial point using CUDA instead of OpenMP as we can initiate more threads with the GPU.

References

1. http://joonlecture.blogspot.com/2011/02/improving-optimality-of-rrt-rrt.html
2. 16-662 Robot Autonomy course material

Division of work

The work was divided up equally.

Code

https://github.com/athens2000/14-618_rrt