Proceedings of the final project of ID5130
Parallel Scientific Computing
3 May, 2024, Chennai, India

PARALLEL IMPLEMENTATIONS OF RRT* CONNECT

Prithvi Raj B
Senior Undergraduate
Department of Mechanical Engineering
Indian Institute of Technology, Madras
Chennai, 600036
Email: me20b137@smail.iitm.ac.in

Chinmay Giridhar
Senior Undergraduate
Department of Mechanical Engineering
Indian Institute of Technology, Madras
Chennai, 600036
Email: me20b051@smail.iitm.ac.in

ABSTRACT

In this project, we construct and examine two parallel implementations of (a modified version of) RRT Connect, a serial sampling-based motion planning algorithm. All results are presented in the case of two-dimensional cartesian coordinates, for which collision checking is straightforward. The parallelization is carried out using MPI (Message Passing Interface) and OpenMP hybrid model. The performance of both implementations are benchmarked with the original serial algorithm, using appropriate metrics.

INTRODUCTION

Over the past few decades, robot manipulators have been extensively used in various applications like manufacturing, pick and place, remote maintenance and even surgical procedures. For applications like these, it becomes essential to plan 'on the fly' to account for the varied scenarios one can face. However, one major issue is that most robot manipulators have 6 or even 7 degrees of freedom. Given a joint resolution of 1°, it results in $(360)^6$, i.e $2.1767823e15$ states for a 6 DOF arm. In such cases, traditional search-based algorithms like A* and Dijkstra's algorithm struggle to find a solution due to the high dimensionality of the problem. Also, obstacles in 3-dimensional space cannot be easily transformed into higher-dimensional configuration spaces.

In the wake of this, a new class of algorithms called sampling-based planners became popular. One of the most popular ones is the Rapidly Exploring Random Tree (RRT) algorithm [1]. It utilizes random sampling to arrive at a collision-free path from the start position to the end position. The algorithm is illustrated below:

1. Designate a node as the start position:
2. Repeat the following till goal state is achieved:
   (a) Sample a random point in the configuration space.
   Verify that the point is a valid configuration by using a collision detection algorithm.
   (b) Find the nearest neighbor node in the space by considering Euler distances.
   (c) Move a fixed step of Δq toward the sampled point from the nearest neighbor node.
   (d) Add this node to the tree as a child of the nearest neighbor node.

While RRT is probabilistically complete, i.e it will give a solution given enough time, it does not give an optimal path. Giving more time will not give a more optimal solution. Here RRT* [2] comes into the picture, rewiring nodes in a fixed radius around the new node so that costs reduce over time. RRT* is probabilistically complete as well as asymptotically optimal, meaning it will give more optimal solutions as time goes on (at the cost of expensive rewiring computations).

Even though these algorithms are faster at providing a solution than A* or Dijkstra, it still takes minutes to hours to find a solution, depending on the complexity of the problem. Hence, it can benefit from parallelization. Specifically, it is known that collision checking and finding nearest neighbour tend to be the bottlenecks [1]. We plan to implement a variation of RRT called RRT-Connect [3] wherein a tree is grown from both the start and goal. This can benefit from a distributed memory parallel archi-

tecture. We plan to test the performance by utilizing MPI and OpenMP.

For the nearest neighbour problem, we have also implemented a K-Dimensional Tree. This data structure aims to speed up nearest neighbor search by storing the locations in a binary tree format which can be traversed much faster (Time complexity: $O(log(n))$) than doing a linear search.

Overview of RRT*-Connect

RRT*-Connect [4] seeks to grow two graphs, one from the start and the other from the goal. An element of greediness is introduced, which serves to draw the points on each graph closer together, while direction is still dictated by random sampling, which maintains exploration of free space. The aim is to form a connection between both graphs when they are sufficiently close. This connection should ideally be made far from the start *and* end points, for the algorithm to be a practical improvement on conventional RRT*. We will refer to the tree growing from the start at `graph_start` and that from the end as `graph_finish`.

1. A new node is appended to `graph_start`, á la RRT.
2. The closest node of `graph_start` to the newest node `node_finish` of `graph_finish` is identified.
3. `graph_start` grows deterministically toward `node_finish`, until it reaches this node or is faced with an obstacle. If it reaches the node, exit.
4. Steps 1 through 3 are repeated, after reversing the roles of `graph_start` and `graph_finish`.

Parallelization Strategy

RRT Connect cannot be parallelized in the aforementioned form, as the identification of nearest nodes and 'rapid growth' steps are serial, by construction. An alternate, parallel algorithm is proposed which the authors consider to be a parallel interpretation of RRT-Connect. Further, the trees are grown according to RRT*. MPI is used to run this program on 2 cores. Process `start`/`finish` stores `graph_start`/`graph_finish` respectively.

1. A new node is added to both `graph_start` and `graph_finish`, as in RRT*.
2. The coordinates of the new nodes are sent to the *other* process, by message passing. They are stored in `new_node_start_pts`, `new_node_finish_pts` in the respective processes.
3. The closest nodes of `graph_start`/`graph_finish` to `new_node_finish_pts`/`new_node_start_pts` is determined in each process.
4. Both graphs grow toward this respective point, until they meet an obstacle or reach the point. If a process reaches it's point, a flag is returned and passed to both processes by a reduction. Both processes exit after checking this flag.

There are possible optimizations to this approach. The datatype used to store the nodes can be either an ordinary C++ vector or a custom K-D tree datatype. The K-D tree is very well suited for search operations, with dramatic speedups in comparison to an ordinary elementwise search. Further, to increase the rate of nodes spawned per second, the process of randomly spawning nodes - `extend()` and 'seeking' the other graph - `connect()` are locally parallelized using OpenMP in each process. We have thus written two parallel implementations using MPI -

1. Incremental Strategy - Vector to store graph, parallelize $O(n)$ search in each process using OpenMP.
2. Improved Strategy - K-Dimensional Tree to store graph, parallelize `extend()` and `connect()` (see Fig (5)) via OpenMP.

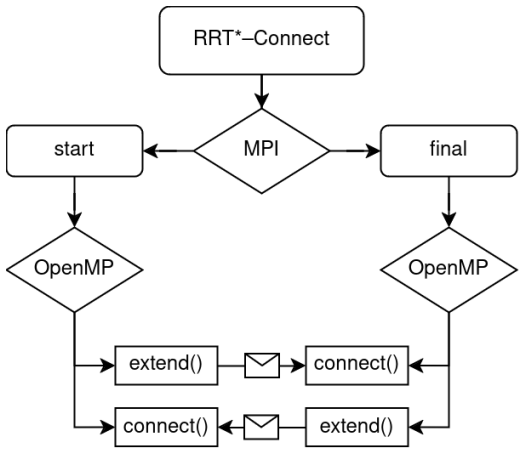

FIGURE 1: Improved Parallelization Strategy

Results

To compare performance, each of the serial, incremental, and improved approaches are run for 10 instances, and the following information is recorded: time taken (t_i), total number of nodes spawned (n_i) .

$$\text{Nodes spawned per second} \quad : \quad u_i = \frac{n_i}{t_i} \tag{1}$$

The efficiency metric is the harmonic mean of these empirical 'velocities'

$$\eta := \frac{10}{\sum_{i=1}^{10} \frac{1}{u_i}} \tag{2}$$

TABLE 1: Serial Data

Time (sec)	Nodes spawned
16.8679	7552
17.2897	7694
47.3684	12246
11.1872	5593
33.4201	9556
44.6958	12280
16.9814	7646
13.9204	6335
37.6085	10304
22.5944	8834

TABLE 2: Incremental Parallelization Data

Time (sec)	Nodes spawned
9.88411	5920
31.7973	9446
12.7739	6742
15.3664	7586
13.2573	7228
26.108	9620
15.8373	8204
22.7039	9350
10.9458	6570
16.8645	7548

TABLE 3: Improved Parallelization Data

Time (sec)	Nodes spawned
0.969845	13981
0.652448	9971
0.623218	10161
0.500802	8877
0.888065	13367
0.573156	10543
0.477484	8780
0.351732	6775
0.303329	6674
0.555735	10058

TABLE 4: Efficiencies

η	Nodes/sec
Serial	369.19
Incremental	464.15
Improved	17219

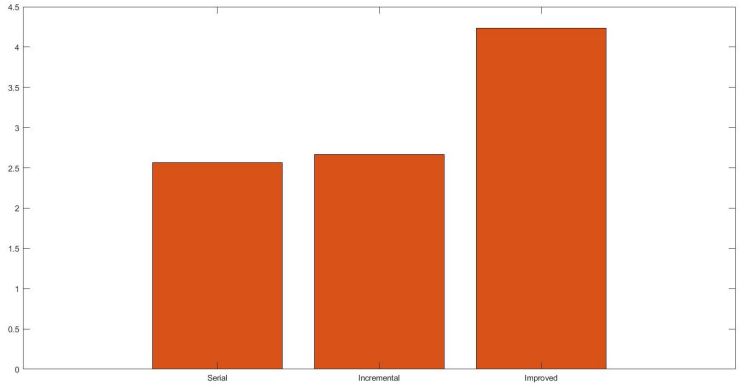

FIGURE 2: $log_{10}(\eta)$ visualized for Serial, Incremental and Improved approaches (left to right)

Challenges

Blocking Nature of MPI Functions

Functions like MPI_Recv(), MPI_Send(), MPI_Allreduce() etc. are **blocking**, meaning the program is paused till the data is received or sent. When one process reached the end state it could

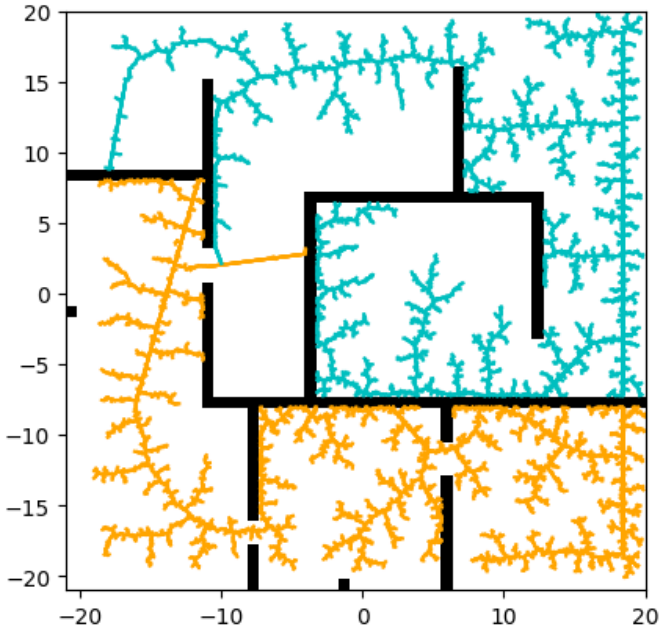

FIGURE 3: Serial output

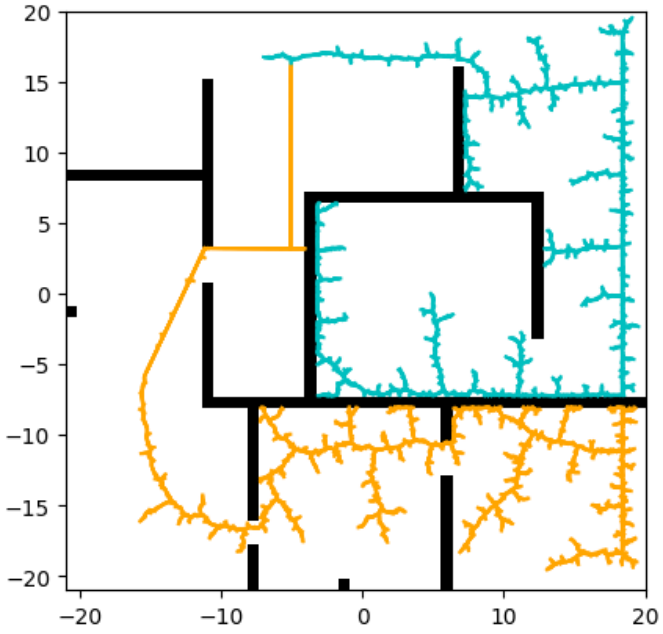

FIGURE 4: Incremental output

not communicate the same to the other process as it was waiting on a different type of communication, say, an MPI_Allreduce().

Fix: The solution was to have a **non-blocking MPI_Irecv()** running in the background to catch the potential ending signal from the other process. **MPI_Iprobe()** is used to check if there is a message to receive before committing to receiving it. Now if the other process reaches end state, this process won't block itself waiting for a communication that is not coming.

Racing condition in K-D Tree Insertion

Since each MPI process had 2 OpenMP processes running underneath, it was possible that they inserted points into the tree at the same time. This is problematic as the K-D Tree uses comparisons to decide where to append the leaf node.

Fix: Implemented an OMP lock using **omp_set_lock()** function so that the other OpenMP process couldn't append to it at the same time.

Making extend() and connect() Independent

The connect() OpenMP process takes longer than the extend() process to complete as there is more than one node insertion. This results in wastage of time by the extend() OpenMP process as it tries to send every extended point to the other MPI process.

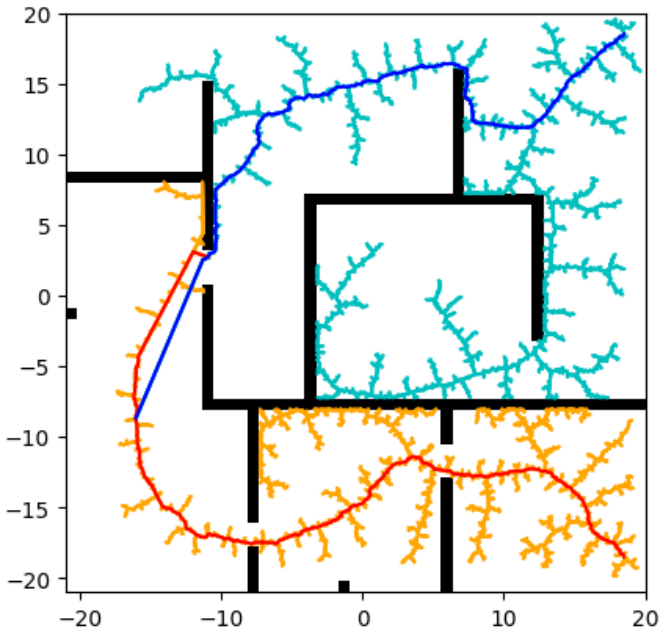

FIGURE 5: Improved output

Fix: A non-blocking MPI_Isend() is used in the extend() so that it can continue expanding nodes till it finishes sending. MPI_Test() function is used to check if it has finished sending at each iteration. If it has finished, the most recently expanded node is queued up to be sent.

Vector Manipulation SegFaults

Upon adding an element to a vector, if adequate adjacent space is not available, the vector moves the entire array to another space in memory to accommodate the new element. This is not adequately communicated to other processes and hence a segmentation fault is encountered when illegal access is attempted, even if the vector is shared between multiple processes.

Fix: Use `std::vector` member functions `emplace_back()` and `erase()` instead of `push_back()` and `pop_back()`.

Inferences

The K-D tree dramatically outperforms both serial and the incremental approaches. Table (3) also indicated that far more nodes are spawned per unit time as a result of paralellizing the `extend()` and `connect()` steps.

REFERENCES

[1] LaValle, S., 1998. "Rapidly-exploring random trees: A new tool for path planning". *Research Report 9811*.

[2] Karaman, S., and Frazzoli, E., 2011. "Sampling-based algorithms for optimal motion planning". *The International Journal of Robotics Research,* ***30***(7), pp. 846–894.

[3] Kuffner, J., and LaValle, S., 2000. "Rrt-connect: An efficient approach to single-query path planning". In Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065), Vol. **2** of *Series name*, pp. 995–1001 vol.2.

[4] Klemm, S., Oberländer, J., Hermann, A., Roennau, A., Schamm, T., Zollner, J. M., and Dillmann, R., 2015. "Rrt*-connect: Faster, asymptotically optimal motion planning". In 2015 IEEE International Conference on Robotics and Biomimetics (ROBIO), pp. 1670–1677.