Parallelizing the Rapidly-Exploring Random Tree Algorithm on the CPU and the GPU

Moises Mata *mm6155* | Ines Khouider *ik2512* | Faustina Cheng *fc2694* | Mahdi Ali-Raihan *mma2268* | Mario Carrillo-Bello *mc5132*

I. Abstract

The Rapidly-exploring Random Tree (RRT) algorithm, a widely used path planning algorithm in robotics, suffers from significant computational complexity due to its intensive sampling and collision checking, making it difficult to use in real-time, zero-latency, robotics applications. To address these performance issues, we introduce a parallel approach to the RRT algorithm that accelerates RRT through the implementation of a multithreaded Python version of the RRT algorithm and a CUDA GPU version. The Python implementation enables concurrent sampling, while both Python and CUDA implementations optimize matrix multiplication, LU decomposition, and sample validation processes. Experimental results demonstrate that our parallel approaches achieve some speedup compared to their serial implementations across various obstacle scenarios for matrix and sample validation, though the opposite is true for sampling.

II. Introduction

In robotics, path planning, also known as motion planning, is a fundamental challenge requiring algorithms to enable robots to navigate simple or complex environments while avoiding obstacles. The Rapidly-exploring Random Tree (RRT) algorithm is a solution for this challenge. RRT works by incrementally building a tree that explores the robot's configuration space through random sampling, connecting new configurations to the nearest existing nodes whenever possible. While RRT is effective in finding a path, it is computationally expensive on a average CPU running serially.

The computational bottlenecks in RRT root from several intensive operations such as random sampling in the configuration space, collision checking between the robot and an obstacle, and the linear algebra operations required for these calculations. These operations have become pretty demanding as the complexity of an environment increases or when a robot's degrees of freedom increase. Traditional serial implementations of RRT, on a regular CPU, struggle to meet the real-time requirements of modern robotics applications, where path planning decisions often need to be made within milliseconds to enable smooth and responsive robot movement. [3]

While research efforts have produced variants of RRT to improve its performance, such as RRT* and Informed RRT, some research has been done to leverage modern hardware architectures, such as the NVIDIA GPU. This is important because GPUs excel at processing large amounts of calculations simultaneously.

In this paper, we present two parallelization approaches that accelerate RRT by utilizing both CPU multithreading in Python and GPU parallelization using CUDA. In the Python approach, we parallelize sampling, sample validation, LU decomposition, and matrix multiplication. In our CUDA approach, we parallelize LU decomposition, matrix multiplication, inverse matrix, and sample validation.

We evaluate our parallel implementations against the traditional serial RRT across various scenarios with different obstacle configurations and complexity. Our results demonstrate a significant speedup while maintaining the algorithms ability to find valid paths.

The remainder of this paper is organized as follows. Section III is background information on inverse kinematics, sampling, and sample validation. Section IV reviews related work, again for, inverse kinematics, sampling, and sample validation. Section V is our methodology, which includes our experimental setup and evaluation methodology. Section VI presents and discusses our results, and Section VII concludes with future work directions.

III. Background

Given a space, a start point and end point, the RRT algorithm generates a path from the start to the goal, avoiding any obstacles. Our experiments concern three parts of the RRT pipeline. The first is matrix parallelization for inverse kinematics, the second is sampling and the third is sample validation.

A. Matrix Parallelization for Inverse Kinematics

The core of Inverse Kinematics (IK) in RRT is to solve for the joint angles of the robot arm to get from the starting position to the ending position. Our IK is linear and follows the form of

$$\mathbf{J}\dot{\mathbf{q}} = \dot{\mathbf{x}}$$

where $\mathbf{J}$ is the Jacobian matrix, $\dot{\mathbf{q}}$ is the vector of joint velocities and $\dot{\mathbf{x}}$ is the desired end-effector velocity. In essence, we are solving a linear system of equations for IK.

In our experiments, we applied two different approaches to solving for $\dot{\mathbf{q}}$.

GitHub Repository: https://github.com/moisesmata/POfR-Final-Project

1) Solving Systems of Equation through the Inverse: Since the Jacobian matrix and desired end-effector velocity are provided as inputs, the first approach we applied was to calculate the inverse of the Jacobian matrix and perform matrix multiplication on both sides to solve for $\dot{\mathbf{q}}$:

$$\mathbf{J}\dot{\mathbf{q}} = \dot{\mathbf{x}}$$

$$\mathbf{J}^{-1}\mathbf{J}\dot{\mathbf{q}} = \mathbf{J}^{-1}\dot{\mathbf{x}}$$

$$\mathbf{I}\dot{\mathbf{q}} = \mathbf{J}^{-1}\dot{\mathbf{x}}$$

$$\dot{\mathbf{q}} = \mathbf{J}^{-1}\dot{\mathbf{x}}$$

2) Solving Systems of Equation through LU Decomposition: In our second approach, we decided to break down the Jacobian Matrix into a lower triangular matrix, $\mathbf{L}$, and an upper triangular matrix, $\mathbf{U}$ and solve for $\dot{\mathbf{q}}$ using forward substitution with $\mathbf{L}$ and $\dot{\mathbf{x}}$ matrices to solve for $\mathbf{y}$ and then backward substitution with $\mathbf{U}$ and $\mathbf{y}$ to solve for $\dot{\mathbf{q}}$:

$$\mathbf{J}\dot{\mathbf{q}} = \dot{\mathbf{x}}$$

$$\mathbf{J} = \mathbf{L}\mathbf{U}$$

$$\mathbf{L} = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ \mathbf{L}_{21} & 1 & 0 & \cdots & 0 \\ \mathbf{L}_{31} & \mathbf{L}_{32} & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \mathbf{L}_{n1} & \mathbf{L}_{n2} & \mathbf{L}_{n3} & \cdots & 1 \end{bmatrix}$$

$$\mathbf{U} = \begin{bmatrix} \mathbf{U}_{11} & \mathbf{U}_{12} & \mathbf{U}_{13} & \cdots & \mathbf{U}_{1n} \\ 0 & \mathbf{U}_{22} & \mathbf{U}_{23} & \cdots & \mathbf{U}_{2n} \\ 0 & 0 & \mathbf{U}_{33} & \cdots & \mathbf{U}_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & \mathbf{U}_{nn} \end{bmatrix}$$

$\mathbf{L}\mathbf{y} = \dot{\mathbf{x}}$ forward substitution solving for y

$\mathbf{U}\dot{\mathbf{q}} = \mathbf{y}$ backward substitution solving for $\dot{\mathbf{q}}$

B. Sampling

Sampling is how the RRT algorithm generates the tree that represents the movement from start position to the end position. There are many ways sampling is classically done in RRT, but in this approach we are going to use the randomly generated trees that traditional RRT uses.

During the sampling phase, points are randomly generated. These points can correspond to many things, but in the case of our sampling algorithm, these points represent the locations of all six of the joints of the robot, using a tuple. Once these random points are generated, the sampling algorithm calls the sample validation algorithm to check if they are valid points. If the point is confirmed as valid, it is added onto the tree, attached to its closet neighbor. The tree continues to grow until it reaches a point that is very close to the end point. Then the algorithm returns the completed path, often making it better through resampling and short-cutting.

For these experiments, we explored two approaches to parallelizing the RRT sampling process. The first approach, which we called "One Tree," addressed the inefficiency of selecting invalid points, especially when the algorithm encountered obstacles. To reduce the time spent on this, we developed an algorithm that sampled multiple points in parallel and selected the best one. The second parallelization method, called "Multiple trees" involved expanding multiple trees simultaneously, with the expectation that one of the random trees would find a solution more quickly.

C. Sample Validation

Sample validation determines whether a newly sampled configuration can be added to the tree. This involves checking if the configuration is within the allowable bounds of the configuration space and if the proposed connection to the existing tree is collision-free and feasible with respect to the system's constraints (e.g., dynamic limits). This step often involves computationally intensive operations such as collision detection against complex geometric objects in the environment, checking for kinematic or dynamic constraints, or evaluating trajectory feasibility, which have to be repeated for many points along the proposed connection. Consequently, sample validation constitutes a significant bottleneck in the runtime performance of the RRT algorithm, especially in complex scenarios and for robotic systems with many degrees of freedom.

To mitigate this bottleneck, several strategies have been proposed, some of which are detailed in the related works section. Our approach focused on parallelizing sample validation, where multiple samples are generated and validated concurrently, reducing the time spent in this phase. If one sample is found to be invalid, the other samples will return without completing the validation process and the whole configuration will be deemed invalid. We explored parallelization through CPU multithreading and launching multiple threads on a GPU, and have compared their performance.

IV. Related Work

Because RRT is a random algorithm, it does not always generate the optional path, but any path. A lot of the recent research on RRT works on searching for optimal paths, or different methodologies to make the algorithm better than simply identifying the optimal path. We decided to keep the related work in this section related to the three parts of the direct RRT algorithm we wanted to parallelize, but there is a lot of future work possible based on the variation of the types of the RRT approaches.

A. Matrix Parallelization for Inverse Kinematics

Prior work in parallelizing robotics algorithms have demonstrated both the challenges and opportunities in leveraging parallel architectures. [5] Harish et al. presented a parallel

implementation of inverse kinematics using both CPU multithreading and GPU acceleration, achieving 10-150x speedup compared to serial implementations. Their approach exploited different levels of parallelism - using CPU threads for high level operations while leveraging GPU cores for computationally intensive matrix operations.

The parallelization of matrix operations in the context of robotics has been extensively researched due to their computational intensity. As shown by Kucuk & Bingul [6], even basic robot control operations like inverse kinematics involves intensive matrix computations. This includes multiplication, decomposition, and inversion. Their work highlights how these operations become bottlenecks in real-time applications, similar to the computational challenges faced in RRT.

B. Sampling

We know that sampling, as well as sample validation should be major portions of the search algorithm parallelization to improve performance, especially when samples are very large. [2]

In "Parallel Sampling-Based Motion Planning with Superlinear Speedup", the outer graph structure, which is what sampling generates is parallelized. A key tenet of this paper is the idea partition-based sampling which allows each thread to look at a different part of the algorithm. This reduces the size of the data each thread has to work through, and as this paper deals with modifying a single tree, means no thread will modify the same data structure. [2] Though we considered a similar approach for our algorithm, we ultimately decided against it because we wanted to lean into how parallelization can enhance the randomness of the RRT algorithm. There were also discussions on the differences between GPU and CPU parallelization. Because the GPU is better at doing the same thing over and over, and sampling can be difficult when its includes go though a partitioned tree. Because we wanted to do experiments on the GPU we wanted to make each part of the algorithm as distinct as possible [2]

Our experiments are similar to the approach by Carpin et al, which as one of their approaches ran several RRT computations in parallel. [3]

There are also algorithms that used the RRT sampling methods in different contexts. For example, Plaku et al. introduced planner that splits up the sampling portion into several portions. It also leverages distributed systems and a large amount of processors. [4]

We also reviewed data on sampling and the number of joints. One paper compared robots with 2 joints and 9 joints, showing that the 9-jointed robot experienced a significantly more pronounced speedup. This finding reinforces that our 6-jointed robot, being closer to the larger end of the spectrum, would provide relevant and meaningful data. [2]

C. Sample Validation

Research on parallelizing Rapidly-exploring Random Trees (RRT) often focuses on sample validation due to its high computational cost. In most cases, sample validation constitutes the largest computational expense in these algorithms unless an extraordinarily large number of samples is required. Even in such scenarios, sample validation remains a significant portion of the algorithm's runtime, particularly for robotic manipulators, which is the application we are testing with RRT [2]. For RRT*, graph operations grow logarithmically over time, but collision-checking time remains substantially higher. The number of collision checks per iteration scales as O(log n) [1].

Sample validation can also be adapted for parallelization in non-traditional contexts. For instance, Bialkowski et al. utilize sample validation to verify straight-line paths when the algorithm is close to the goal state, while Zhang et al. focus on limiting the search space [1, 7].

Sampling and sample validation are further enhanced through a documented approach called the parallel process splitting paradigm. This method involves running the same algorithm across multiple threads and sending a termination signal to all threads once one thread completes. This approach aligns closely with the method we chose for sample validation [8].

V. Methodology

Each method had 2-3 algorithms to compare from the serial case, and most had a Python and a CUDA implementation. The Python implementation was tested in the simulation, while the CUDA implementation was tested locally with generated data.

A. Matrix Parallelization for Inverse Kinematics

Listing 1. Solving Linear Systems of Equations Through Matrix Inversion

```
import numpy as np
import time
import os
from multiprocessing import Pool

def row_multiply(args):
    row, matrix2 = args
    return np.dot(row, matrix2)

def multithreaded_matrix_multiply(matrix1,
    matrix2):
    with Pool() as pool:
        results = pool.map(row_multiply, [(
            row, matrix2) for row in
            matrix1])
    return np.array(results)

def invert_matrix(matrix):
    """Invert a single matrix."""
    return np.linalg.inv(matrix)
```

In this first approach, the inverse matrix of the provided Jacobian matrix is calculated using $np.linalg.inv$ in $invert\_matrix()$ function, which handles the parallelization of Gaussian elimination for us. Then, matrix multiplication is done between the inverse Jacobian matrix and the desired end-effector velocity $\dot{\mathbf{x}}$ matrix using $multithreaded\_matrix\_multiply(matrix1, matrix2)$ and

$row\_multiply(args)$. Each row of the inverse Jacobian matrix can be taken up by a thread in its own process to perform dot product with each column of the end-effector $\dot{\mathbf{x}}$ matrix. Not only does it speed up computation of the matrix multiplication using multiple threads, spatial locality of the cache is also optimized by performing the matrix multiplication in row major order.

In the second approach the linear system of equations is solved using LU decomposition by using scipy's $lu\_factor()$ and $lu\_solve()$ routines. The parallelization of LU decomposition is handled by setting OMP_NUM_THREADS and MKL_NUM_THREADS environment variables to '8', enabling parallel computation through OpenMP and Intel's Math Kernel Library. $lu\_factor()$ and $lu\_solve()$ take advantage of the highly optimized BLAS/LAPACK, which stands for Basic Linear Algebra Subprograms and Linear Algebra Package, which are highly optimized libraries for matrix operations.

Listing 2. Solving Linear Systems of Equations Through LU Decomposition

```
import os
import numpy as np
from scipy.linalg import lu_factor,
    lu_solve
os.environ['OMP_NUM_THREADS'] = '8'
os.environ['MKL_NUM_THREADS'] = '8'

def solve_linear_system_lu(A, b):
    A = np.asarray(A, dtype=float)
    b = np.asarray(b, dtype=float)
    lu, piv = lu_factor(A)
    x = lu_solve((lu, piv), b)
    return x
```

B. Sampling

Sampling was tested as both algorithms were inputted into the greater serial RRT algorithm one at a time, to see its impacts on the overall algorithm.

The MultiTree algorithm is parallelized by launching multiple threads, each independently running the find_path function. This allows different threads to explore different parts of the configuration space simultaneously, increasing the likelihood of finding a valid path quickly. The results from all threads are collected through a shared queue, and after the threads finish, the paths are returned. This parallel approach accelerates the process of motion planning by leveraging concurrency to explore the space in parallel, ultimately producing multiple potential paths from different threads.

The Single Tree algorithm parallelizes by creating multiple threads that simultaneously explore different parts of the configuration space. Each thread executes the worker function, which calls the `parallel_stepping` function to randomly sample configurations, find the nearest point in the tree, step toward the random sample, and check if the motion is valid. Valid results are placed in a shared queue, and the main thread processes these results, extending the tree toward the goal. This parallelization allows for concurrent exploration, speeding up the pathfinding process by evaluating multiple potential paths at the same time. Once all threads complete their tasks, the main thread synchronizes the results by waiting for all threads to finish before continuing with the pathfinding process.

C. Sample Validation

Our sample validation code parallelizes checking between the start and end position if the path is currently valid. It does this by dividing the path into smaller segments and validating each segment in parallel using multiple threads. The `is_segment_valid` function calculates the number of segments based on the distance between the start and end point. The number of segments also depends on the type of obstacle in the scenario: the more complex the obstacle, the more segments are calculated. It then creates a thread for each segment (our implementation uses the `threading` module in Python), and checks whether the corresponding part of the path is valid using the `validation_worker` function. If any thread detects an invalid segment, it signals the main function to stop further checks using a shared `stop_event`. This multi-threaded approach improves the efficiency of path validation by concurrently processing multiple segments, allowing for faster detection of invalid paths, especially when the sample validation is done via a call to a simulation service. In the aforementioned case, the task is I/O bound, which is sped up greatly through multi-threading since the scheduler can effectively utilize the waiting time incurred during I/O operations, allowing other threads to execute and perform validation checks on their respective segments. This overlapping of I/O waits and computation significantly reduces the overall validation time compared to a sequential approach, where each segment would be validated one after another, accumulating the I/O latency for each.

The CUDA implementation of parallel sample validation on the GPU was carried out in a similar manner as the CPU multi-threaded implementation. The `PyCuda` library was used to launch the CUDA kernel from Python code. The number of segments is calculated in the same way but their validation is divided among the threads in a thread block of a kernel that is launched on the GPU. `PyCuda` allocates memory and copies over the specified input/output arrays to the GPU by specifying `cuda.In` for an input array and `cuda.Out` for an output array. It also manages the cleanup of resources. Instead of using a `threading.Event` as we did in the CPU multi-threaded case, shared memory is used in the GPU to signal the other threads to return when a segment is found to be invalid. Only the first thread initializes the shared memory and sets the return value in the case that all segments were valid. This is to prevent duplicate work. The threads stride by the number of threads in the kernel to ensure all segments are checked, unless a segment is invalid, in which case the thread responsible sets the shared memory and the return result to invalid. The threads are synced after shared memory initialization so that no thread reads the uninitialized value of the shared memory and mistakenly assumes a segment was found to be invalid in another thread. Threads are also synced before returning the result to ensure all segment validations are accounted for.

D. Data Gathering

It was really important to the team that we gather data that was relevant to the RRT algorithm and were able to test it reliably. As a result, we decided to gather data on the RRT algorithm though a simulation powered by ROS2. This simulation centered a robot arm with six degrees of freedom and six joints (UR5 Robot Arm), and the problem that we were simulating was that of the robot arm end-effector going from one point to another, with varying complexities of generated obstacles in the way. We used the MoveIt! library to test cases with no obstacles, a simple obstacle, a hard obstacle, and a super hard obstacle. The simple, hard, and super hard obstacles were structures made of one box, two boxes, and four boxes respectively. The end-effector goal position was the same in each test case, as to ensure our benchmarking was consistent.

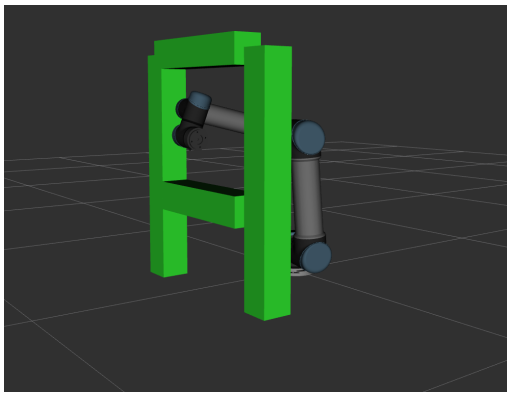

Fig. 1. The UR5 Robot in RVIZ2 with the "Super Hard Obstacle"

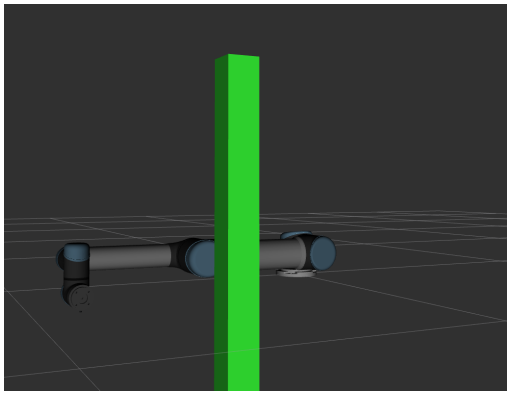

Fig. 2. The UR5 Robot in RVIZ2 with the "Simple Obstacle"

Each part of the algorithm had data gathered from separate runs. The matrix inverse/LU algorithms are run with the position data of the arm. The sampling and sample validation are run independently with their serial counterparts. The total time of these algorithms includes the time from the start of sampling to the end of sample validation.

At first, when we were running tests manually, we realized there was a large variation from the results of the tests, due to the variation of the obstacles as well as the variation in RRT due to the random nature of sampling. As a result we wrote a script to automate everything. This script launches and terminates different ROS2 (Robot Operating System 2) nodes for testing our different algorithms. It tests the serial case, the case with One Tree parallel sampling, the case with Multiple Tree parallel sampling and finally the case with serial sampling and parallel sample validation. The script waits 6 minutes and 15 seconds for each iteration of the algorithm to run, and registers the time it takes. It then writes this to a CSV file for us to process the data. we ran this script overnight and were able to get over 200 data points per algorithm variant.

In addition to the testing we did using ROS2, we also benchmarked individual phases of the RRT algorithm within the experiment to explore the specific differences in performance that could be seen as a result of our parallelization scenarios.

VI. Results & Discussion

A. RRT CPU Code

Our experimental results both confirmed and challenged what we initially expected. After our over 200 tests, it seems that the implementation of parallelized sample validation through collision checking came out as our best performing RRT implementation. This made sense to us, as previous work done in the field has shown that the collision checking procedure is the biggest bottleneck.

What is surprising however, was the good performance of the serial case, especially once paired with the relatively poor performance of our other parallelized algorithms. Even then, the parallelized sample validation only had a speed up of around 4 seconds at most on average, as seen in figure 3. This remains an interesting outcome and is marked as an interesting avenue for further research. We go into further detail on subsections of the algorithm in the subsequent section.

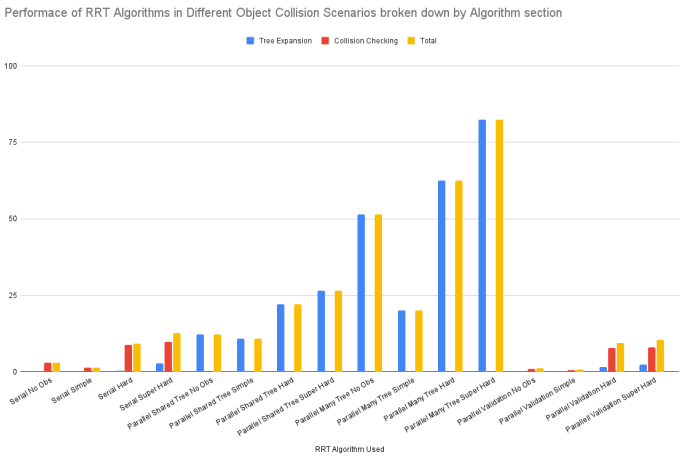

Fig. 3. Comparison of Parallel and Serial RRT Performance Across Different Obstacle Scenarios, vertical axis in seconds

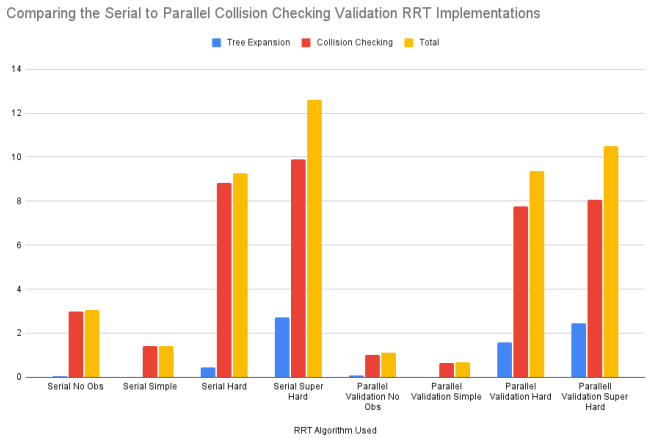

Fig. 4. Two specific implementations compared against each other, vertical axis in seconds

1) Matrix Parallelization for Inverse Kinematics: Because of the different parts of the algorithm these tests were run on the CPU separately from the general tests. The total time shown here is just the time for the matrix algorithm.

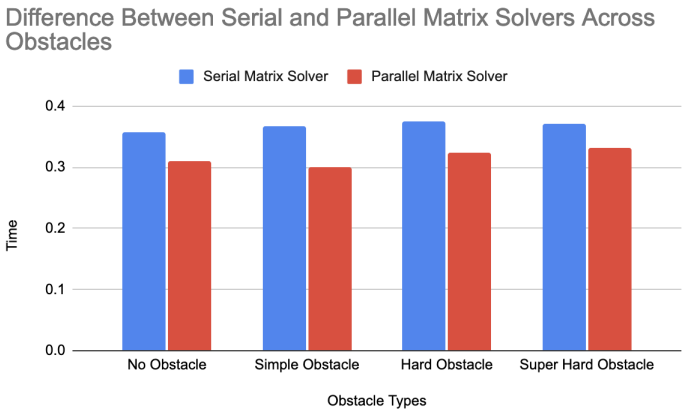

Fig. 5. Difference between serial and parallel Matrix Solver

We found that the parallel matrix solver is consistently better than our serial solver across obstacles. This is surprising because these matrices are 6x6 (6 joints by 6 degrees of freedom). We would not expect as consistent of a speedup. However, the speedup itself was minimal, in average adding up to 0.05 seconds, and so it is possible that this is not conclusive enough data, or that the speedup for the small matrix was not very significant.

We also noticed that the matrix values were pretty consistent regardless of the obstacle and thus the complexity of the path that the robot would have had to go through. This also makes sense as the inverse simply computes the kinematics irrespective of how much the math it has to compute has to avoid obstacles.

2) Sampling: Our parallel sampling algorithms did not have any improvement from the serial case, and in fact in many examples did worse than the serial case. As seen in Figure 6 below, the One Tree algorithm did worse than the serial in all cases expect the hard obstacle, and the Many Tree algorithm did much worse than both in both cases.

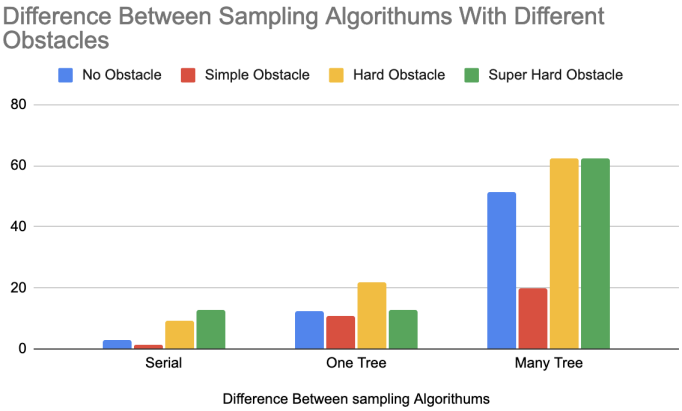

Fig. 6. Fig 4. Sampling Data and different tree algorithms being compared against each other

Initially, we suspected that the Many Tree implementation would output better performance, as it makes sense that basically having n copies of RRT running and then continuing when any delivers a good path would work better than the One Tree method, which required locking every time a thread accessed/added to the tree. Though we did not expect these parallel tree expansion methods to be faster than the parallel sample validation method, we did expect a speedup when compared to the serial version. Current hypotheses that explain this result are the fact that our tree always expanded by a constant amount, so perhaps the sample validation slow down is not that big of a bottleneck in our specific implementation, and the fact that we launched these tree expansion implementations with 10 threads. Another explanation could be our approaches. For example, our ”many trees” implementation creates multiple trees that do not build on each other. If all these trees are equally ineffective at exploring the search space, this method incurs additional overhead due to the cost of creating and managing threads. On the other hand, the ”one tree” algorithm can generate many points; if most of them are redundant, these extra calculations only increase the runtime without improving performance.

Another factor could be the randomization of the obstacles and the inherently stochastic nature of the sampling process. We observed significant outliers in the parallel cases, which might be more sensitive to such outliers for the reasons outlined above.

Something else to note is that the serial algorithm tends to go up as the obstacles increase in complexity but our serial algorithms do not. This suggests that the parallel algorithms are effective at validating points in a way that offsets the increasing complexity of the search space. This is one of the benefits we would imagine from the sampling algorithms.

We can also note that across all examples, the simple obstacle case did the best. This suggests that having simple constraints can help the random sampling algorithm get to a solution faster. One possible reason for this is that simpler obstacles result in fewer invalid regions in the configuration space. With fewer constraints, the rejection rate for sampled points decreases, meaning the algorithm spends less time discarding points and more time extending the tree toward the goal. However, few constraints did even better than none. This should be looked into more, but suggests that a simple obstacle can help guide the RRT algorithm toward the target more efficiently than none at all.

3) Sample Validation: An interesting topic of research for us when coming into this project was breaking down the time taken in RRT that is spent in the tree expansion section versus the collision checking/sample validation section. We noticed that the ratios of Tree Expansion to Collision Checking did change between these two implementations of RRT. We note that the purely serial RRT had a Tree Expansion duration to Sample Validation Time ratio of about 1:4, while our parallelized version had a ratio of 1:3. We note that the Tree expansion seems pretty similar between the two implementations.

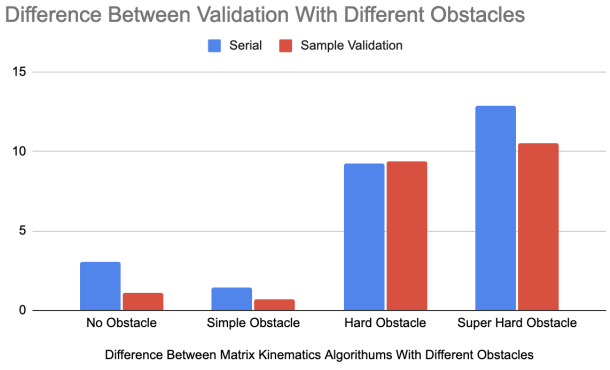

Fig. 7. Parallel and Serial Sample Validation Over Different Obstacles

We also noted that sample validation did constantly better than the serial implementation on the real data. The differences were most observed during no obstacle, which makes sense, as the more points that are valid, the more the validation checker needs to go through all the points, and the more work the validation checker needs to do, the better our parallel implementation should be.

B. CUDA Code

Though we were not able to run the CUDA code on the actual simulation, we were able to run some of it locally.

1) Matrix Parallelization for Inverse Kinematics: The performance of LU decomposition shows very surprising results between CUDA and the Python implementations. The CUDA implementation ended up being slower for matrices of size 10, 100, 1000, and 10000. For the 10,000 x 10,000 matrix, it took our CUDA implementation 35 seconds to calculate while our Python implementation took 25 seconds. This difference can be alluded to the overhead of data transfer between the host and kernel, and the syncing of the threads. In theory, CUDA should be faster than Python, however, this was not the case. In this scenario, since we are not handling matrices that are greater than 100,000 x 100,000, multithreaded Python would end up being faster than CUDA.

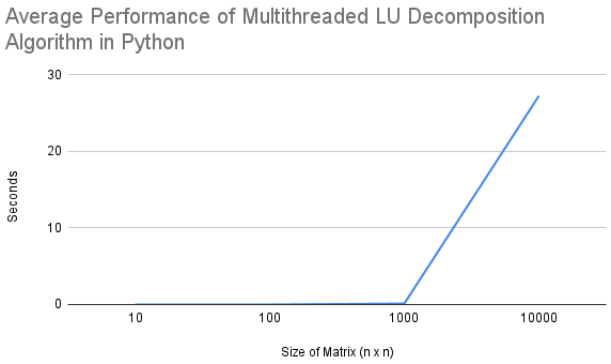

Fig. 8. Average Performance of Multithreaded LU Decomposition Algorithm in Python

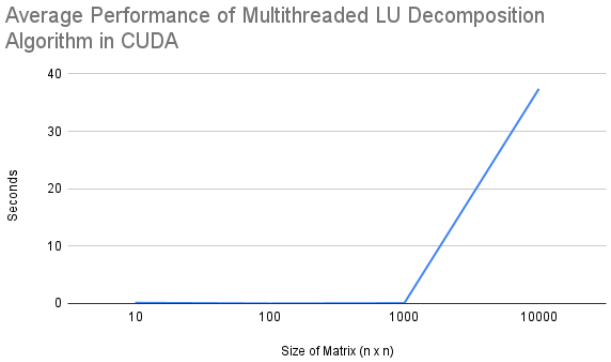

Fig. 9. Average Performance of Multithreaded LU Decomposition Algorithm in CUDA

2) Sample Validation: The execution time was compared between the serial, CPU multi-threaded, and CUDA implementations (on an Nvidia T4 GPU) of sample validation. For each implementation, the different types of obstacles (none, simple, hard, and super) were tested. To simulate a state validation service, a function that would return False 1.5% of the time and True for the rest of the time was used across all implementations and types of obstacles. Seeding mechanisms were used to generate the start and end configurations in order to standardize them across runs. By using different fixed seeds, the random configurations generated for both the start and end configurations are the same for each run with the same obstacle type across different implementations. However, due to the randomness of the state validation function, we will still be able to see how each case performs under different number of segments checked per run.

Implementation	Obstacle Type	Time Taken to Find Valid Sample (s)	Number of Samples Checked	Time per Sample (s)
CUDA	HARD	0.063186	289	0.000219
	NONE	0.001080	3	0.000332
	SIMPLE	0.005934	27	0.000225
	SUPER	3.351393	15162	0.000212
Serial	HARD	0.037209	432	0.000085
	NONE	0.000336	4	0.000085
	SIMPLE	0.002218	26	0.000086
	SUPER	1.647016	18866	0.000084
CPU Multi-Threaded	HARD	0.098769	369	0.000264
	NONE	0.001614	4	0.000395
	SIMPLE	0.007384	26	0.000281
	SUPER	4.734123	18290	0.000258

TABLE I
MEDIANS OF SAMPLE VALIDATION BENCHMARK

The benchmark was run 500 times, each time until a valid sample was found, and the median for each case was taken for comparison. We also look at the execution time/number of segments checked ratio as another metric for comparison (Table 1).

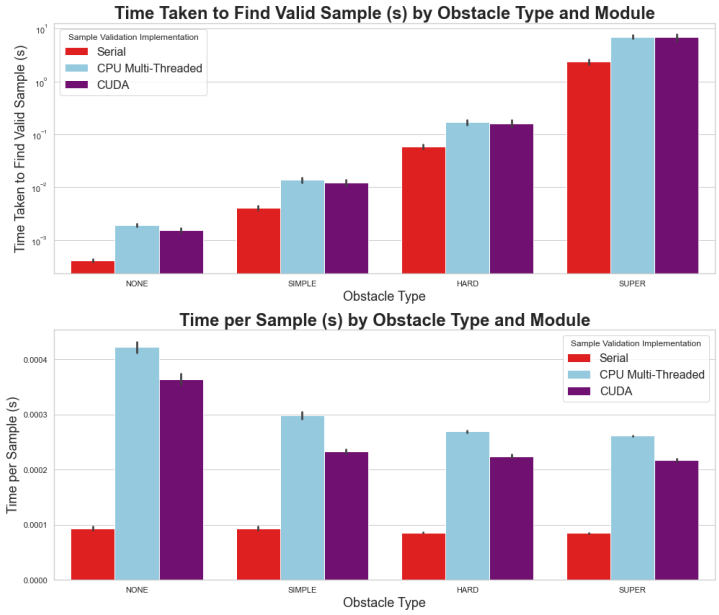

Fig. 10. Median Execution Time and Time Per Segment Checked for Sample Validation

Because the simulated state validation function doesn't make a call to a state validation service, it is not I/O bound since it doesn't need to wait on external services. Therefore, we don't see the benefits of overlapping the I/O latencies with parallelization and the decrease in execution time due to that. On the contrary, the overhead associated with parallel execution seems to have outweighed the benefits of parallelization, as can be seen in the longer time taken for parallelization in both the CPU multi-threaded and CUDA cases (Fig. 5). However, comparing the time per sample, the CUDA implementation ran faster than the CPU multi-threaded implementation. This was expected, as the number of threads launched on the GPU (128) was more than the number of threads launched on the CPU (8). In addition, we expected threading on the GPU to be more performant than threading in Python because Python employs a Global Interpreter Lock (GIL), which only lets one thread execute Python bytecode at a time within a single Python process. This means that even on multi-core processors, only one thread can be actively executing Python code at any given moment. For I/O bound tasks, this still led to a significant speed up in execution time because of the overlapping I/O latencies, as seen when we ran the RRT algorithm with ROS. As expected, the execution time increased the higher the complexity of the obstacle. This is due to the increased number of segments that need to be validated when the complexity of the obstacle is increased.

C. Challenges

It was hard to get parallel tree expansion with parallel validation because that involves multi-threading multi-threads. As a result, we decided to run the experiments with the portions of the program parallelized in order to compare specifically what one algorithm changed as opposed to another

Some challenges were reached were in running the CUDA code on the GPU and potentially running a C++ version of the code. We had initially wanted to rewrite everything in C++, but changing the entire library to run into C++ seemed difficult so we settled on Python CUDA code.

We also had challenges with moving the CUDA pieces of the algorithm to our simulation. In many of the algorithms, we needed to call dependent functions from ROS that could not be easily changed to CUDA. For instance, for sample validation we needed to make a function called self.is_state_valid CUDA-fied. This function was too complex to parallelize, and calling it repeatedly would not succeed in running things on GPU as it would be instead repeatedly called a CPU function. To resolve this, we attempted to grab the obstacle data and current state of the robot from the simulation, transform the data into our configuration space, and implement our own algorithm for collision checking in 6D space on the GPU so we could get this validation to happen on the GPU. This was not successful. Our CUDA implementation of sample validation using a simulated state validity check function had the limitation of not accurately reflecting the I/O bound nature

of the function, and as such presented different results than we saw when we ran the RRT algorithm with ROS.

Testing the CUDA code on colab was not the best, as there are hard memory limits you can reach in colab. For example, testing matrices that were 100,000 x 100,000 were not feasible as the GPU 'ran out of memory' according to colab.

VII. Conclusion & Future Work

In conclusion, we saw interesting improvements in our parallelized CPU-Python-based RRT algorithm. Our best result by far was from parallelizing the sample validation, which our research suggested that would be optimal as this is the biggest bottleneck identified for RRT. We were surprised by how poorly our parallelized sampling did, but believe this had to due with algorithmic design and size and data collection. Finally, on the CPU, our parallelized matrix algorithms did better that the serialized ones, by a little.

On the GPU side we were able to test exciting algorithms that we learned about in class —primarily parallel matrix inverse and LU decomposition —and see their behavior on the GPU. It is a similar story for our parallelized sample validation/collision checker —though we struggled to integrate these into our larger ROS2 experiments, we were able to get individual pieces running as a proof of concept. The Python multi-threaded implementation performed better than the CUDA implementation for matrix solver but the reverse was true for sample validation when the execution time was averaged out over the number of samples validated. Figuring out how to do only one memory transfer in a larger GPU-enabled RRT would be very beneficial, as we were faced with costly memory transfer overhead times.

Some future work would include quantifying obstacle approaches, as our obstacle generation was random. Where the obstacle is can greatly influence the time that an RRT algorithm takes. We bypassed this by running experiments many times but in the future it would be good to have consistent experiments with obstacles in the same space.

We would also want to explore different RRT approaches such as RRT*, Informed RRT*, and LQR RRT*, as much of the literature references this wide variety of methods. It would be interesting to compare and contrast the similarities and differences of regular RRT, and determine the most optimal aspects to parallelize for these variations.

Finally, although we were able to write most of our code in CUDA, we were not able to run it on the actual simulation. Therefore, we would need to solve our compatibility issues and compare CUDA performance using simulation data. A fascinating area of research would be figuring out how to load all needed data (Obstacle in Joint Space, Inverse Kinematics, Forward Kinematics) into the GPU at once, so that the costly GPU memory transfer overhead we faced would be minimized.

References

[1] Bialkowski, Joshua, Sertac Karaman, and Emilio Frazzoli. "Massively parallelizing the RRT and the RRT." In 2011 IEEE/RSJ International Conference on Intelligent Robots and Systems, 3513-3518. Institute of Electrical and Electronics Engineers, 2011.

[2] Jeffrey Ichnowski and Ron Alterovitz "Parallel Sampling-Based Motion Planning with Superlinear Speedup"

[3] Stefano Carpin, Enrico Pagello "On Parallel RRTs for Multi-robot Systems" Department of Electronics and Informatics The University of Padova - ITALY

[4] E. Plaku and L. E. Kavraki, "Distributed Sampling-Based Roadmap of Trees for Large-Scale Motion Planning," Proceedings of the 2005 IEEE International Conference on Robotics and Automation, Barcelona, Spain, 2005, pp. 3868-3873, doi: 10.1109/ROBOT.2005.1570711.

[5] Pawan Harish, Mentar Mahmudi, Benoît Le Callennec, and Ronan Boulic. 2016. Parallel Inverse Kinematics for Multithreaded Architectures. ACM Trans. Graph. 35, 2, Article 19 (May 2016), 13 pages. https://doi.org/10.1145/2887740

[6] Serdar Kucuk, Zafer Bingul, "Robot Kinematics: Forward and Inverse Kinematics," Industrial-Robotics-Theory-Modelling-Control, ISBN 3-86611-285-8, pp. 964, ARS/plV, Germany, December 2006

[7] Zhang Q, Liu Y, Qin J, Duan J. An Informed-Bi-Quick RRT* Algorithm Based on Offline Sampling: Motion Planning Considering Multiple Constraints for a Dual-Arm Cooperative System. Actuators. 2024; 13(2):75. https://doi.org/10.3390/act13020075

[8] Sengupta S. A Parallel Randomized Path Planner for Robot Navigation. International Journal of Advanced Robotic Systems. 2006;3(3). doi:10.5772/5730