15418 Project RRT Proposal

Bob Gao, Xintong Qi

April 2023

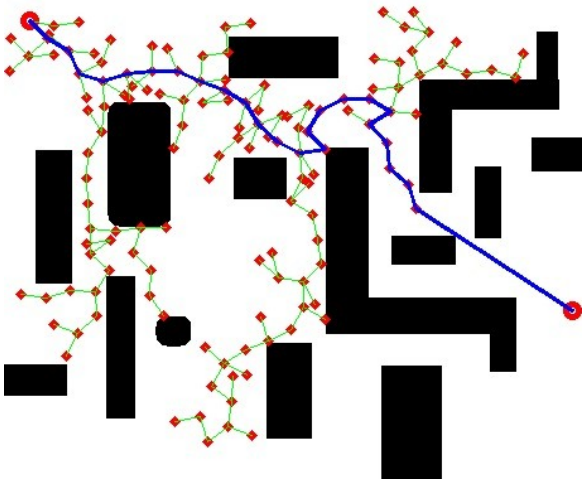
Project Website

1 Summary

For our 15418 final project, we are going to create an accelerator of rapidly exploring random tree (RRT), a commonly used algorithm in robotics on path planning given a known environment.
There will be two main challenges in our project. One is to implement task stealing between threads for a single agent, and the other is to achieve high speedup when planning paths for multiple agents using any parallel programming frameworks.
For our final deliverables, we will create a visualization of the map, agents, and paths taken. There will be several benchmark files to test our implementation with different maps, start and goal position pairs. The time taken to plan these paths will be collected to analyze our implementation's performance.

2 Background

Path-planning is a critical task in robotics that involves finding a path from an initial state to a goal state while avoiding obstacles. Rapidly exploring random trees (RRT) is an algorithm commonly used for this purpose. RRTs work by randomly exploring nodes and checking for collisions in a massive state space, making them efficient for solving complex path-planning problems.
Since RRTs are often used in real-time robotics applications, their runtime speed optimization is critical. Fortunately, there is massive potential for parallelization in exploring this state space. To parallelize RRTs, there are several outlets that can be taken.
One approach is to add and process multiple nodes to the tree at the same time, which can significantly speed up the exploration process. Another approach is to parallelize individual steps within the loop, such as finding the closest node/nodes in the tree to a certain point in the workspace when 'pruning' the tree. A combination of both approaches can also be used to achieve further performance gains.
Overall, parallelization is an effective technique to optimize the runtime speed of RRTs, making them more suitable for real-time robotics applications.



A visualization of RRT

3 Challenges

To make sure that RRT can guarantee a solution given there is a path exists in the environment, RRT is notorious for being computational expensive and unstable. Since it must expand in all different directions, the single core execution time can vary drastically, from finding the goal in the first try to hitting the wall all the time.
This brings us to our first challenge, which is to implement RRT to approach a linear speedup with the number of cores used in a single agent planning context. Due to the nature that some nodes will hit the wall and terminate early, that means we will likely to observe load imbalance issue if we just naively assign subtrees to each core. Therefore, our current idea is to implement task stealing, so that when some other threads terminate early due to hitting dead ends, they can take the expanding subtrees from others to finish the work faster. We need to choose carefully when to steal and what data is stolen, so that:

- **Correctness** the processor can steal and continue on the correct branches.
- **Communication** the overhead would be as minimal as possible.

After we have a working implementation for the first challenge, we want to expand the topic to multi-agent planning. On top of the first challenge, this time we need to consider the case when multiple agents have overlapped paths. We need to check the following in order to have a better multi-agent planning performance:

- When two agents travel to the same spot but at different times, such plans should be allowed.
- When collision (same spot at same time) occurs, which thread(s) should rewind, and how much to rewind?
- What message should be passed between all agents so that they can know each other's path at a minimal communication cost?
- Will memory accesses become a bottle neck as the number of agents increases, leading to more data required?

As we hit issues on our way, we will consult with course instructors on possible solutions or optimizations.

4 Resources

For the first challenge mentioned above, we are going to use GHC machines, since 8 cores are enough to check the implementation of task stealing and the performance.
For the second challenge, we will use PSC machines, where each agent can take a few cores to implement task stealing, and we can observe our code performance when there are a lot of agents (meaning that we need more cores).
To start, we can build on top of a RRT simulator found on GitHub here. It uses cpp and we can easily edit Makefile to include tools for performance measures. Plus, it works for multiple platforms we learned and used in this semester (OpenMP, Cilk, OpenMPI, etc.)

5 Goals and Deliverables

Our goals in this project is straightforward: to bump the speedup of RRT close to the number of processor used **as much as we can**. We have a set of measures of our goals as listed below:

1. Challenge 1: task stealing on a single agent
    - PLAN TO ACHIEVE: **correct** and at least **7 times speedup** on a 8-core GHC machine. With task stealing implemented with minimal communication required, we can solve both overhead and workload imbalance issues, achieving at least 7 times speedup using 8 cores.
    - HOPE TO ACHIEVE: that speedup is very close to the number of processors used (n$\pm$0.5 given n processors).

2. Challenge 2: multithreading on multiple agents

    - PLAN TO ACHIEVE: **correct** and at least $\frac{n}{2}$ **times speedup** using n cores on PSC machine. More communication is needed to check between agents. We still don't know if there are any hidden challenges.
    - HOPE TO ACHIEVE: $\frac{3n}{4}$ times speedup using n cores on PSC machine.

We plan to deliver our final project both in analysis and interact demo forms. In our analysis, we will process statistics collected from various benchmark files and show the graphs to our audience, talking about the challenges we encountered and advantages/drawbacks of our implementation. Next to our poster presenting the statistics, we will launch our implementations on GHC machines, letting people create their own maps and define start and goal positions. Then we will run both the sequential and parallel implementations to show the time difference to give people a better understanding of our project.

6 Platform Choices

We are planning to use cpp with the following reasons:

- We have used it throughout this semester, so the time spent to implement it would be less compared to other programming languages.
- There are multiple frameworks with task stealing in cpp, such as Cilk, OpenMP. If we scale the problem to the multi-agent one, we could also use OpenMPI.
- For the final demo purpose, we want to wrap up everything into executable files, so that we can show the program to other people.

The machines we will use are GHC and PSC machines. We will reach out to instructors if we ever need to use frameworks that are not available on these machines.

7 Schedule

Here is a tentative schedule for our final project:

Tasks	Deadlines
Implement sequential version	April 7th
Implement parallel version	April 12th
Improve challenge 1 and collect statistics	April 16th
Project milestone report	April 19th
Implement multi-agent version	April 21st
Improve challenge 2 and collect statistics	April 28th
Final report and demo preparation	May 4th
Project poster session	May 5th