Parallelization of RRT* Using OpenMP

2024/12/23

蔡其均 313553008
廖昶竣 313551125
葉冠宜 313551150

Abstract

Path planning is a crucial problem in robotics, autonomous vehicles, and drones, aiming to find optimal paths while avoiding obstacles and respecting constraints. The Rapidly-exploring Random Tree (RRT*) algorithm is widely used for generating asymptotically optimal paths but faces challenges with increased computation time as environment complexity grows. This work addresses these challenges by parallelizing the RRT* algorithm using OpenMP to reduce execution time. By leveraging multi-core processors, we enhance RRT* efficiency, enabling real-time decision-making. This approach contributes to more scalable and efficient motion planning for applications in robotics, autonomous driving, and drone navigation.

1 Introduction

Path planning is a fundamental problem in robotics, autonomous vehicles, drones, and other fields that involve navigating through environments with obstacles. The goal of path planning is to find an optimal, feasible, or safe path from a start point to a goal while avoiding obstacles. As real-world applications become more complex, with dynamic environments and high-dimensional state spaces, the need for efficient algorithms has grown significantly. One algorithm that has gained considerable attention for its ability to generate asymptotically optimal paths is the Rapidly-exploring Random Tree (RRT*).

RRT* enhances the basic RRT algorithm by incorporating cost optimization and a rewiring process. While RRT focuses on rapidly exploring the search space to find a feasible path, RRT* refines this approach by iteratively improving the quality of the path. It optimizes the tree structure by considering path costs, such as distance or energy consumption, and re-evaluating connections to minimize these costs. The rewiring step ensures that suboptimal paths are replaced by more efficient ones, leading to the discovery of not just feasible, but also cost-efficient and asymptotically optimal paths over time.

Figure 1 shows the pseudocode of the RRT* algorithm. The Rapidly-exploring Random Tree (RRT*) algorithm constructs a tree incrementally through a series of well-defined steps with the goal of finding an optimal path by sampling random nodes, connecting them intelligently, and refining the tree structure over time. The complexity of the RRT* algorithm depends on the number of iterations, denoted as K. The algorithm proceeds as follows:

Algorithm 1 RRT* Algorithm

```
Initialize tree T with initial node N_init
for k = 1 to K do                          ▷ Number of iterations or until goal is reache
    Sample random node N_rand from the space
    Find nearest node N_nearest in T to N_rand
    Steer from N_nearest towards N_rand to generate new node N_new
    if Obstacle-free path from N_nearest to N_new then
        Find all nodes N_near within a radius from N_new
        N_min ← N_nearest
        c_min ← cost(N_nearest) + cost(N_min, N_new)
        for each node n_near in N_near do
            if Obstacle-free path from n_near to N_new then
                c_new ← cost(n_near) + cost(n_near, N_new)
                if c_new < c_min then
                    N_min ← n_near
                    c_min ← c_new
                end if
            end if
        end for
        Set parent of N_new to N_min in T
        for each node n_near in N_near do
            if N_new is a better parent for n_near then
                Rewire n_near to have N_new as parent in T
            end if
        end for
        Add N_new to T
    end if
end for
return Best path found in T
```

Figure 1. baseline pseudocode for RRT* algorithm

1. **Generate a Random Node Position**: The process begins by generating a random node within the defined space. This randomness facilitates effective exploration of the space, enabling the algorithm to identify solutions in complex, high-dimensional environments.
2. **Find the Closest Node to the Random Position**: Upon generating a random node, the algorithm identifies the closest node in the tree to the random position. This step is crucial as it determines the direction in which the tree will expand.
3. **Steer Towards the Random Node**: The next step involves steering from the closest node towards the random node. Rather than connecting directly, the algorithm ensures that the expansion remains feasible, considering constraints such as obstacles and distance limitations.
4. **Add a New Node to the Tree**: The resulting point from the steering process is added to the tree as a new node, thereby expanding the tree towards previously unexplored regions of the space.
5. **Find Neighbor Nodes**: After incorporating the new node, the algorithm identifies neighboring nodes within a

specified radius. This step allows the algorithm to explore multiple potential connections and build a more refined tree structure.

6. **Optimize the Parent Connection**: Among the neighboring nodes, the algorithm selects the parent node that minimizes the cost of reaching the new node. This decision is based on optimization criteria, such as path length, to ensure the algorithm progressively converges to a more efficient solution.
7. **Rewire the Tree**: A key feature of RRT* is its rewiring mechanism, which improves the tree by revisiting and potentially modifying connections between the new node and its neighbors. If a different connection results in a lower cost, the tree is updated accordingly. This step ensures that the tree remains not only feasible but also as optimal as possible.
8. **Update Children's Cost**: Finally, the algorithm updates the cost of all child nodes affected by the rewiring process. This cost propagation ensures that the improvements achieved through rewiring are reflected throughout the tree.

In step 1, a random node is generated, which is a constant-time operation, $O(1)$. Step 2 involves finding the closest node in the existing tree. This requires a search through the tree's nodes, which results in a time complexity of $O(k)$, where k is the current number of nodes in the tree. Step 3 is the steering phase, where the algorithm computes a trajectory towards the random node. This operation is straightforward and takes constant time, $O(1)$. In step 4, a new node is added to the tree, which again is an operation that takes constant time, $O(1)$. Step 5 involves finding the neighboring nodes within a specified radius of the new node. This requires a search through the tree's nodes to identify the neighbors, resulting in a time complexity of $O(k)$. Step 6 optimizes the parent connection of the new node by evaluating the cost of reattaching it to neighboring nodes. The number of neighboring nodes to consider is denoted by m, so the complexity of this step is $O(m)$. In the worst case, where every node in the tree might be a neighbor, the complexity becomes $O(k)$. Step 7 consists of the rewiring process, where the tree is updated by revisiting neighboring nodes and potentially reassigning parents to improve the tree's structure. This operation also takes $O(m)$ time, and in the worst case, it can involve checking all k nodes in the tree, resulting in a worst-case complexity of $O(k)$. Additionally, the process of updating the costs for all affected child nodes, which may require traversing through affected nodes and recursively propagating changes, also takes $O(k)$. Thus, the overall complexity per iteration is dominated by the search for the closest node in the tree and the search for neighboring nodes of the newly added node. Therefore, the time complexity for each iteration is $O(k)$, where the linear term accounts for the closest node search and neighbor search. Over K iterations, where K represents the total number of iterations, the total complexity of the RRT* algorithm becomes $O(K^2)$.

However, as the complexity of the environment increases or the number of obstacles grows, the computation time required to find a path also increases. This can be a significant challenge in real-time applications, where quick response times are crucial for autonomous systems to navigate safely and efficiently. The motivation for this work arises from the need to address these computational limitations. Real-time path planning requires algorithms that not only guarantee optimal solutions but also operate within time constraints. While RRT* is known for its optimality, it often becomes computationally expensive as the problem size grows, leading to slower decision-making in critical scenarios. To mitigate this issue, we focus on parallelizing the RRT* algorithm using OpenMP, a widely-used tool for parallel computing. By leveraging multi-core processors, we aim to reduce the execution time of the algorithm, making it feasible for applications that require rapid response times.

2 Proposed Solution

In the previous section, we've mentioned that the total time complexity of RRT* algorithm is $O(k^2)$, where k means the total iterations to be run, the high time complexity of this algorithm is potentially to be parallelized.

In this project, we use [6] as our template, which implements a serial version of RRT* using C++ language. We focus on evaluating the baseline serial version of the RRT* algorithm. This initial step is essential to understand the computational demands and performance characteristics of the algorithm in its unoptimized form. By measuring the computational time requirements, we aim to identify the bottlenecks and key processes that contribute to the overall runtime. These insights will provide a baseline for comparison with subsequent parallelized implementations. The serial version operates sequentially, processing one step at a time, which allows us to clearly observe the time complexity associated with each phase of the algorithm. This understanding is critical as it sets the stage for justifying the need for parallelization and highlights the potential areas where significant performance improvements can be achieved. Through this evaluation, we aim to gain a clear and quantifiable perspective on the computational efficiency of the serial RRT* algorithm. The result of using perf as our profiling tool on RRT* is shown in Table 1. The result matches the discussion in the previous section, which is step 2: $O(k)$, step 5: $O(k)$ and step 6: $O(m)$. Based on the profiling result, we propose two solutions.

Table 1. execution time distribution in baseline RRT* algorithm

Find the closest node to the random position	63.3%
Find the neighbor nodes of the new node	31.5%
Find the parent of the new node	2.6%
Others	2.6%

2.1 version 1

We first find that the time complexity of three functions in Table 1 is high which is $O(k)$, and these three functions are serially executed in the baseline, so we decided to parallelize these functions and merge them when finished, as shown in Figure 2.

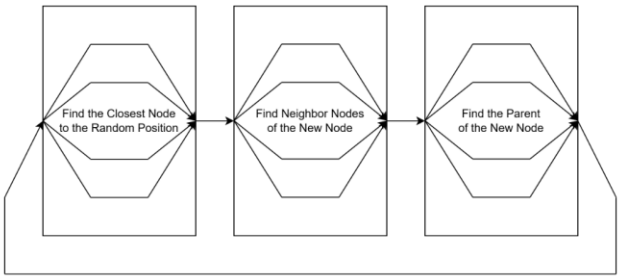

Figure 2. version one of parallelized RRT* algorithm, which parallelizes each function and execute them in serial.

We can observe that in each iteration, there are lots of threads created, and those threads need to be synchronized and merge together at the end of each function, the overhead will be discussed in the following section.

2.2 version 2

In addition to parallelizing these functions, we also found that the outer loop of this algorithm has highly potential to be parallelized, which is we sample random nodes in parallel and merge the new node to the solution base if a new node were found, as shown in Figure 3 .

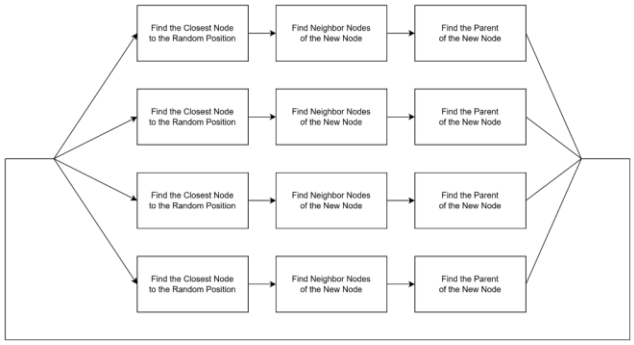

Figure 3. version two of parallelized RRT* algorithm, which samples many nodes parallelly.

There is a problem encountered while implementing version two, which is in the baseline code, if a better path is found, the algorithm will return the better path to main body, and the main body will decide whether to run the remaining iterations, as shown in Figure 4.

```
while(iter < MAXITER) {
  iter++;
  // do something
  if (/* found better path */) {
    return PATH;
  }
}
```

Figure 4. pseudocode of the outer loop of baseline RRT*

We propose to use #omp parallel, #omp single and #omp task to create tasks for many threads to compute, and use a flag variable to notice main thread when to stop create tasks if better path is found, as shown in Figure 5. Our proposed solution has furthermore problem, which is the main thread stop creating new task only if the flag is set true, which is set by the task a few steps before, for example, in Figure 6, task 200 will find a better path, but before task 200 compute the path, there are some tasks are created by the main thread. It doesn't influence our algorithm at all, because the algorithm is to approximate the shortest path in a short time, if a path is found after task 200 is better then path found in task 200, the path is sure to be a newly optimized path from now.

```
// global path
PATH_glo = NULL;
bool flag = true;

#pragma omp parallel
#pragma omp single
while(iter < MAXITER && flag) {
  iter++;
  #pragma omp task
  {
    // do something
    if (/* found better path */) {
      flag = false;
      PATH_glo = PATH;
    }
  }
}

if (PATH_glo != NULL)
return PATH_glo;
```

Figure 5. pseudocode of our proposed solution of parallelized RRT* algorithm

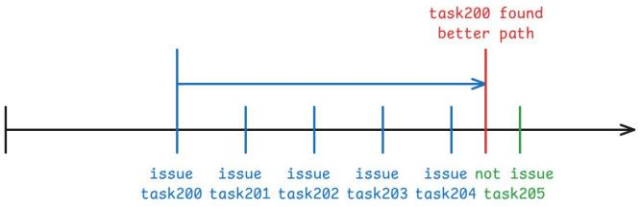

Figure 6. toy example of our proposed solution

3 Experimental Methodology

We propose using OpenMP for parallel programming on this project. OpenMP is suitable for multi-threaded applications and provides a simple interface for parallelizing tasks across multiple cores. Its ease of integration with existing C/C++ codebases makes it a practical choice for implementing the parallel RRT* algorithm.

3.1 Experimental Environment

1. Operating System: Debian 12.6.0
2. **Processors**: Intel® Core™ i5-10500 @ 3.10GHz and Intel® Core™ i5-7500 @ 3.40GHz
3. **Software**: C++ with OpenMP for parallelization

3.2 Input Data

The experiment utilizes two maps of different sizes:
Figure: Dimensions of **1200 × 1000**
Figure 77: Dimensions of **1000 × 1000**

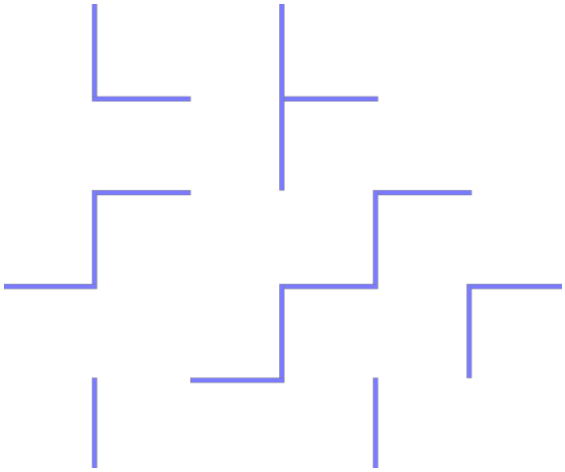

Figure 6. test map 1 for RRT*

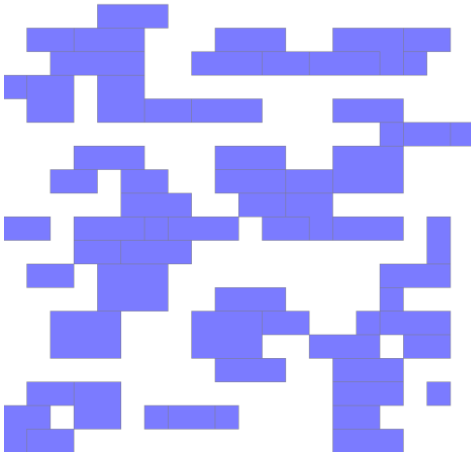

Figure 77. test map 2 with dense obstacles for RRT*

4 Experimental Results

The experiment results of test map 1 and test map 2 over solution version one is shown in Table 2 and Table 3. And the comparison of two methods is shown in Table 4 and Table 5. We can observe that the second method runs faster than the first method, this is because in the first method, each iteration creates lots of threads, which is 3T, where T is the number of threads a function will create. The total number of threads created in first method will be 3kT, and the number of threads in second method will be at most k which has a smaller overhead then first method.

Table 2. speedup of our solution version one comparing with baseline using test map 1.

# Thread	Find closest	Find neighbor	Find parent	total
1	1.00x	1.00x	1.00x	1.00x
2	2.00x	1.98x	1.76x	1.93x
4	3.75x	3.66x	2.70x	3.40x
6	5.20x	5.01x	3.33x	4.48x

Table 3. speedup of our solution version one comparing with baseline using test map 2.

# Thread	Find closest	Find neighbor	Find parent	total
1	1.00x	1.00x	1.00x	1.00x
2	2.06x	1.99x	1.84x	1.77x
4	3.88x	3.66x	3.11x	2.70x
6	5.20x	5.10x	4.06x	3.21x

Table 4. speed up between two versions using test map 1.

Thread	Method 1	Method 2
1	1x	1x
2	1.93x	1.91x
4	3.40x	3.85x
6	4.48x	5.71x

Table 5. speed up between two versions using test map 2.

Thread	Method 1	Method 2
1	1x	1x
2	1.77x	1.83x
4	2.70x	3.42x
6	3.21x	5.02x

5 Related Work

Path planning has long been a fundamental aspect of robotics, and many approaches have been proposed to tackle this problem. Sampling-based algorithms like the Rapidly-exploring Random Tree (RRT) and its enhanced variant, RRT*, have garnered attention due to their ability to efficiently explore high-dimensional spaces and ensure asymptotic optimality under certain conditions. Karaman and Frazzoli (2011) presented RRT* as an optimal path planning algorithm, demonstrating its effectiveness in achieving lower path costs compared to RRT, while maintaining probabilistic completeness [1].

To improve upon the limitations of RRT* in terms of computational efficiency, several studies have explored parallelization strategies. Bialkowski et al. (2011) investigated the feasibility of massively parallelizing the RRT and RRT* algorithms on Graphics Processing Units (GPUs), showing that parallel collision-checking can significantly speed up pathfinding in high-dimensional spaces [2]. However, our proposed approach uses OpenMP to leverage the full computational resources of multi-core processors, which are more widely available in modern computing environments, making our approach both more accessible and scalable for a variety of applications.

Recent developments in UAV path planning have incorporated various optimization techniques into RRT*. For instance, Fan et al. (2023) developed a bidirectional APF-RRT* algorithm that integrates an artificial potential field (APF) and goal-bias strategies to improve convergence rates and path quality [3]. Similarly, Chen et al. (2023) proposed the GB-PQ-RRT* algorithm, which combines target bias and gravitational potential field adjustments with a quick RRT* variant to optimize path generation in 3D environments, achieving faster convergence and smoother paths [4].

[5] Introduces a potential field approach to path planning, which has significantly influenced the development of various algorithms, including RRT*.

Recent developments in UAV path planning have incorporated various optimization techniques into RRT*. For instance, Fan et al. (2023) developed a bidirectional APF-RRT* algorithm that integrates an artificial potential field (APF) and goal-bias strategies to improve convergence rates and path quality [3]. Similarly, Chen et al. (2023) proposed the GB-PQ-RRT* algorithm, which combines target bias and gravitational potential field adjustments with a quick RRT* variant to optimize path generation in 3D environments, achieving faster convergence and smoother paths [4]. Unlike these works, which focus on algorithmic enhancements through domain-specific strategies, our implementation is centered on improving computational performance by parallelizing the standard RRT* algorithm using OpenMP. While these previous studies integrate advanced heuristics and problem-specific strategies to improve path quality or convergence in specific scenarios, our approach preserves the original algorithm's simplicity and structure. This ensures that our implementation can be applied broadly across different domains without requiring adaptations for specific scenarios.

6 Conclusions

In this work, we addressed the computational challenges of the Rapidly-exploring Random Tree (RRT*) algorithm by parallelizing it using OpenMP to leverage the full capabilities of multi-core processors. By focusing on key functions and loops within the algorithm, we were able to significantly reduce execution time while maintaining the same algorithm's optimality and quality of the generated paths. Our approach offers a practical solution for real-time motion planning in applications such as robotics, autonomous vehicles, and drone navigation, where rapid response times are essential.

The results from parallelizing the RRT* algorithm show that multi-core processing can provide substantial performance improvements, particularly for large-scale problems with complex environments. Moreover, the simplicity and flexibility of our approach ensure its applicability across a variety of domains without the need for domain-specific adaptations.

For future work, we could investigate the real-time implementation of the parallelized RRT* algorithm in dynamic environments, where obstacles or goals may change during execution. Expanding in these areas would allow us to further push the boundaries of efficient and scalable path planning for real-world autonomous systems.

REFERENCES

[1] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *The International Journal of Robotics Research*, vol. 30, no. 7, pp. 846–894, 2011.

[2] J. Bialkowski, S. Karaman, and E. Frazzoli, "Massively parallelizing the RRT and the RRT*," in *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*, San Francisco, CA, USA, 2011, pp. 3513–3518.

[3] J. Fan, X. Chen, and X. Liang, "UAV trajectory planning based on bi-directional APF-RRT* algorithm with goal-biased," *Expert Systems With Applications*, vol. 213, 2023.

[4] X. Chen, Y. Zhao, J. Fan, and H. Liu, "Three-dimensional UAV track planning based on the GB-PQ-RRT* algorithm," in *Proceedings of the 42nd Chinese Control Conference*, Tianjin, China, 2023, pp. 4639-4640.

[5] Hwang, Y., & Ahuja, N., A Potential Field Approach to Path Planning. *IEEE Transactions on Robotics and Automation*, 8(1), 23-32.

[6] https://github.com/Ali-tp/RRTSTAR.