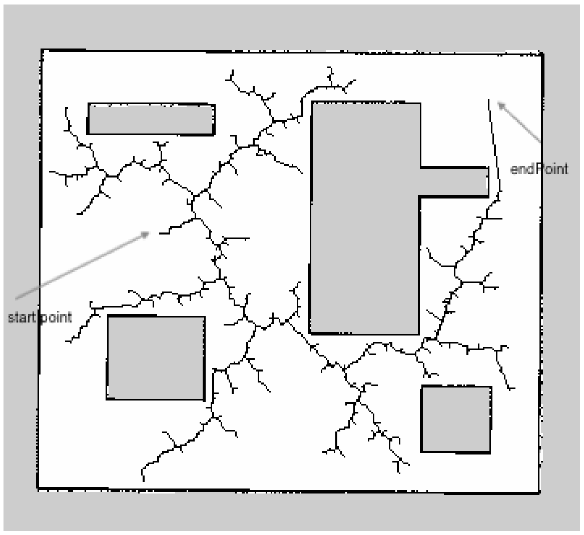
Final Project Report

Zixin Shen(zixins)

Haoran Zhang(haoranz4)

Summary

We implemented a parallel version of rapidly-exploring random tree(RRT) using OpenMP and achieved 4.2x speedup against single thread serial version, 2.5x speedup against an optimized parallel implementation for a single CPU under GHC machine 4 cores. RRT is an algorithm designed to efficiently search nonconvex, high-dimensional spaces by randomly building a space-filling tree [1]. It can be used to search a path from a given source to a given destination within a space filled with obstacles. We analyzed and identified a series of bottlenecks inside the algorithm and improved their performance respectively. Below are some RRT graphs we generated (Fig 1, Fig 2, Fig 3).



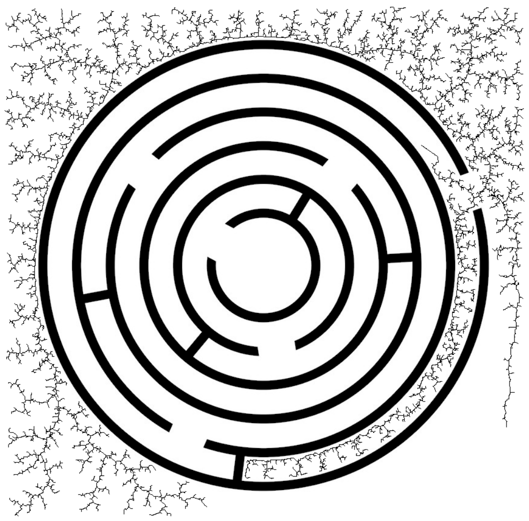


Fig 1: map 0 Fig 2: map 2

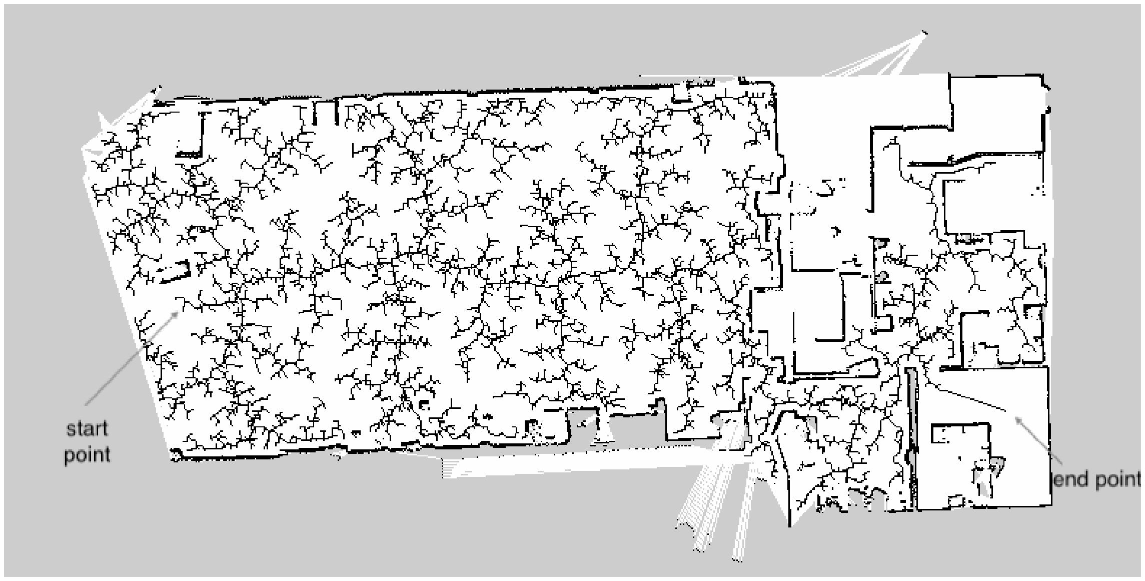
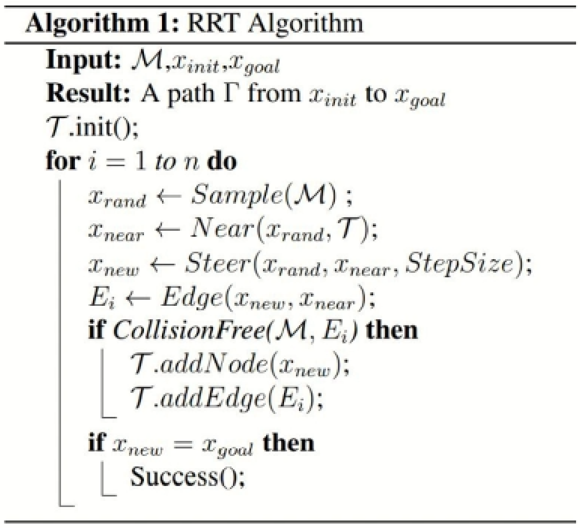


Fig 3: map 1

Background



```
Algorithm 1: RRT Algorithm
  Input: $\mathcal{M}$,$x_{init}$,$x_{goal}$
  Result: A path $\Gamma$ from $x_{init}$ to $x_{goal}$
  $\mathcal{T}$.init();
  for $i = 1$ to $n$ do
      $x_{rand} \leftarrow Sample(\mathcal{M})$ ;
      $x_{near} \leftarrow Near(x_{rand}, \mathcal{T})$;
      $x_{new} \leftarrow Steer(x_{rand}, x_{near}, StepSize)$;
      $E_i \leftarrow Edge(x_{new}, x_{near})$;
      if $CollisionFree(\mathcal{M}, E_i)$ then
          $\mathcal{T}.addNode(x_{new})$;
          $\mathcal{T}.addEdge(E_i)$;
      if $x_{new} = x_{goal}$ then
          Success();
```

Fig 4: pseudocode

What are the key data structures

1) Map with original pgm information, which is the input of our solution, including file information and raster (is a flattened 1d array) showing the

information of obstacles and freespace position in pgm file, and the resolution.

2) Undirected graph.Since it’s a graph algorithm, undirected graph is to represent the connection in the graph.
    a) The graph is composed of a nodeList of Coordinates and the parent nodeList of Coordinates, which is used to search for the nearest node and to draw the line between the current node and its parent node in visualization.

What are the key operations on these data structures?

1) For Map structure, main operations are
    a) load_pgm, save_pgm, draw_line.
    b) **Inflate_obstacles** is to change the points within certain distance of obstacle to obstacles that helps simply the process of detecting obstacles
    c) **detect_obstacles** whether there is any point in or between the potential point and its nearest node already in the map.
2) As for the nodelList, main operations are
    a) **Find_nearest_node** is to find the node whose distance between itself and the current node is the smallest among all the existing nodes.

b) Insert current node and its parent node (which is the most adjacent node in the graph). Basically the parent node is only used to trace the position on the graph for visualization.

What are the algorithm's inputs and outputs

The input of the algorithm is the start point, the end point and the map to search on (in .pgm format) for our search algorithm.

The output of the algorithm is a feasible path from start point to the end point and the pgm path of all the possible nodes in the nodeList.

What is the part that is computationally expensive and could benefit from parallelization?

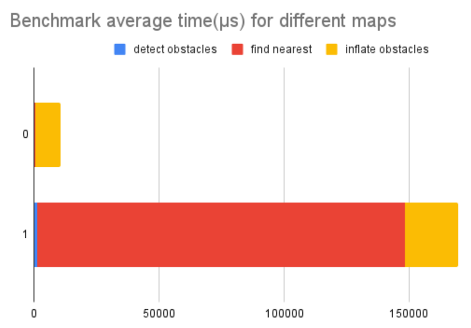
The first part that could benefit from parallelization is inflating the obstacles. What it does is to go over the raster(bitmap) data and find the points that belong to the obstacles. Then we “inflate” these points, basically turning the points within a certain radius around this point into obstacles as well. This operation is necessary because it provides a cushion so that our path stops growing automatically when it reaches within a certain distance to the obstacles, which simplifies our logic of detecting obstacles.

The second part we have found is to find nearest points. This part is a part of the algorithm where upon a random point is generated we need to find its nearest neighbor among the already fixed points inside the tree. This part will slowly become computationally heavy as more points are added to the tree. Since we need to go through all the points to find the nearest one. And we need to do this operation for every random generated point.

The third part is to detect the obstacles. For every point between every random generated point and its nearest neighbor points, we need to see if it steps on any obstacles. This is computational heavy because we need to traverse every point between the line segments.

Break down the workload

RRT is a sequential algorithm with strong data dependency between generating different points in the RRT tree, because each generation needs to find the nearest node of the existing tree. Therefore we can not do anything about the search loop. We are focusing on different steps, mainly the three steps including inflating obstacles, detecting obstacles and finding the nearest node in RRT. Firstly we did two analyses on different maps, map 0 is a rather simple map, it takes roughly 571 times of exploration to get to the destination. Map 1 is a rather complicated map, it takes 16072 times of exploration. Since they have different number of steps and possible nodes on the graph, the bottleneck for both maps varies a lot. Not only for different map size, they takes different time in total (Fig 5)



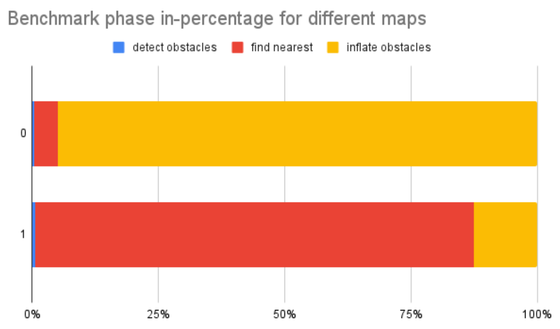


Fig 5: benchmark average time   Fig 6: benchmark phase in-percentage

Fig 6 shows the distribution difference on the same phases of serialization version. It displays that in the simple map, the step of inflating the map takes most of the time, while in the complicated map, the step of finding the nearest node weighs most. To be more specific, that's because the step of inflating needs to transverse the whole picture to inflate the pixel value of the obstacle and its surroundings. It doesn't depend on the size of the map. Therefore there is no difference between Map 0 and Map 1. However, if you look at finding the nearest node , you will find that every time the node tries to find the existing neighbor node, if there is a more complicated map, the finding step takes more time. Based on the two bottlenecks in the two scenarios, we are focusing on paralleling the code in both inflating obstacles and detecting obstacles. And we also noticed that with the increasing of the average steps as map 0, 1, 2, the time of detecting obstacles also increases(Fig 7, Fig 8). Therefore, we also tried to improve detecting obstacles in our problem. Thus, optimization of inflating the map, finding nearest nodes and detecting obstacles are our three optimization goals.

map	average steps
0	662
1	16072
2	20181.15

Fig 7: step count

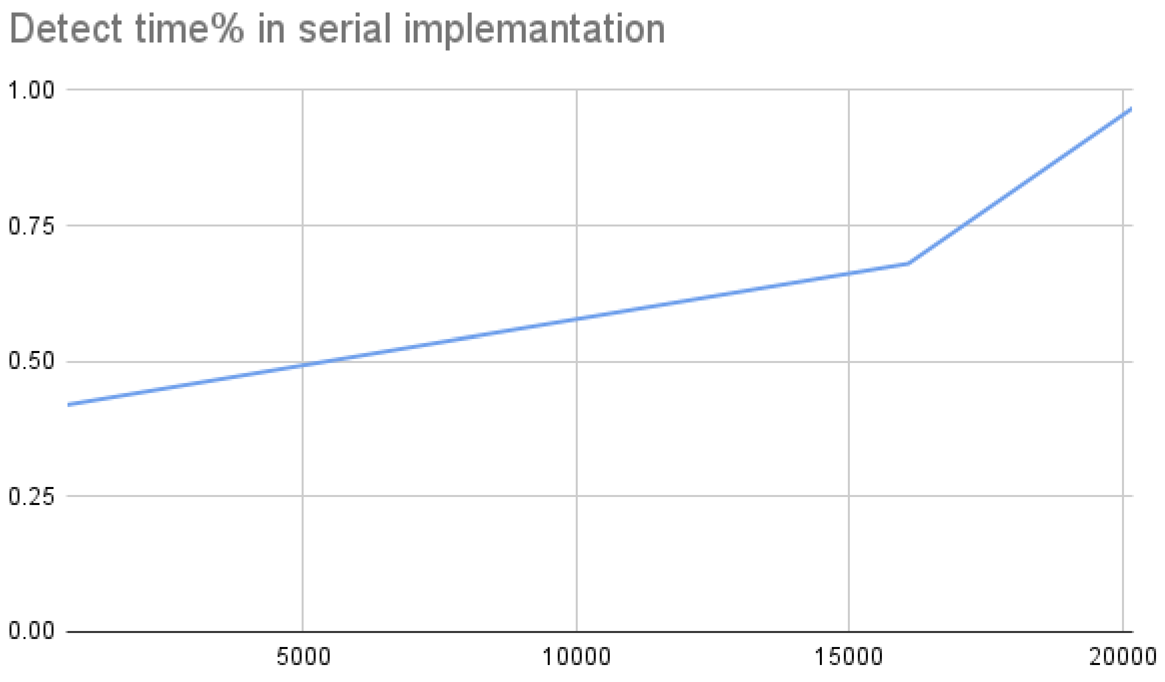


Fig 8: detect obstacle % in total time

Approach

Describe the technologies used. What language/APIs? What machines did you target?

We used OpenMP and C++ to do the parallelization which targets at using cpu machines doing high-efficiency real-time path computation.

Describe how you mapped the problem to your target parallel machine(s). IMPORTANT: How do the data structures and operations you described in part 2 map to machine concepts like cores and threads. (or warps, thread blocks, gangs, etc.)

We basically used the strategy of data parallelism which partitions node data into different batches and makes it run on different cores in OpenMP. For the three optimization goals, we have different strategies because of different logics. For the inflating map, since there is a if-condition which causes the load imbalance, we found that doing a dynamic schedule can maximize the performance. For finding the nearest neighbors, we adopt the explicit barrier and the strategy of parallelizing the local finding minimum part with nowait. For detecting obstacles, we choose to adopt the static schedule.

• Did you change the original serial algorithm to enable better mapping to a parallel machine?

Yes, we changed slightly the logics in finding nearest node and detecting obstacles to maximum the performance and.

• If your project involved many iterations of optimization, please describe this process as well. What did you try that did not work? How did you arrive at your solution? The notes you’ve been writing throughout your project should be helpful here. Convince us you worked hard to arrive at a good solution.

The first optimization happens in inflating obstacles, which takes in the image information and then tries to inflate the point with the distance of radius to the obstacle as a point of obstacle to avoid some corner case. We noticed that there might be some load balancing issues if we only simply use omp parallel because for the points already the obstacle, they don’t need to be inflated. Therefore, we use dynamic schedule rather than the static schedule, and we also tried different chunks like 8, 16, 32, and know that the chunk of 16 can get the maximum speedup against the one core.

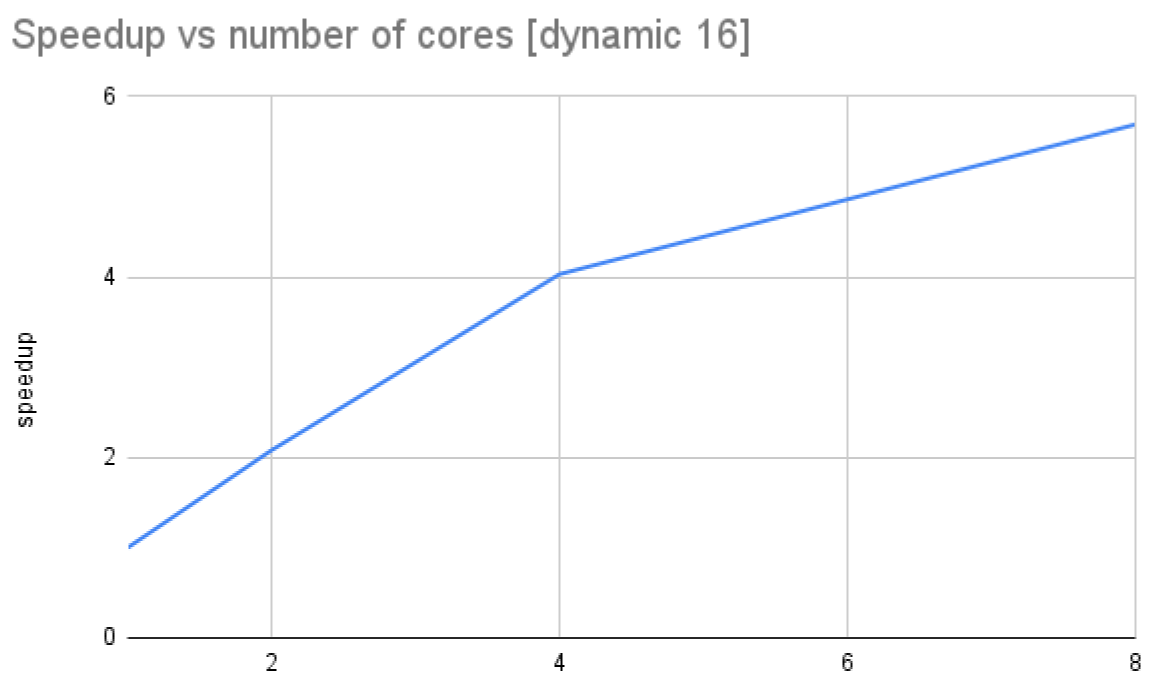


Fig 9: speedup of inflating obstacles on 8 cors

As the fig 9 shows, it almost achieves the best performance on core 1 to 4. And also get an ideal speedup on core 8.

The second optimization we are focusing on is the second important part: finding the nearest node. It's tricky part. We used to plan to use the k-d tree to optimize the time cost of finding the nearest node. We did part of the work, but we noticed that the k-d tree does improve the performance but it’s because of its data structure, it’s hard to implement parallelization for the k-d tree. Therefore we tried several ways to optimize the performance in the openmpi domain rather than thinking of a better sequential data structure.

1) The first and most obvious optimization is that we can use two loops, one loop trying to note down all the results, and a second loop to find the minimum value among all the results.
2) The second optimization is that we use reduction methods in one loop to find the node with minimum path
3) The third optimization and the optimization we finally adopted is that we choose to eliminate the implicit barrier for local optimal, and update the global optimal in omp critical. It achieves 2.4 times speedup over an optimized parallel implementation for a single CPU and 3.6 times speedup over the baseline sing-thread cpu code.

The third optimization is on detecting obstacles. We found that the serial version has the advantage of early stop, which means when they transverse every point between two points, if there is a small point it won't go into the loop again. However openmp cannot have such function, we learn that they have omp cancellation, however it only works after openmp 4.0. We first tried the same trick as the last one setting the local optical and canceling the implicit barrier. And set an atomic operation for the global variable, however the time we found that is not even not the similar order of magnitude as fig 10 shows.

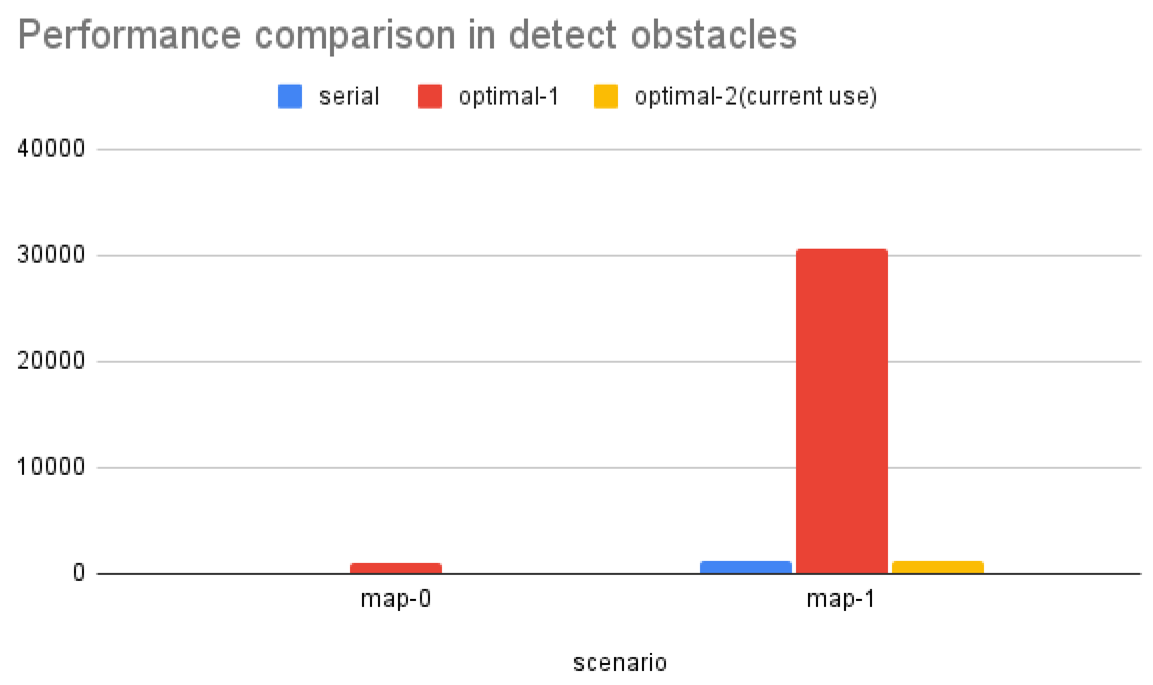


Fig 10: performance comparison in detecting obstacles

Therefore our strategy is that we are gonna make all the threads use the shared variable obstacle, if it equals 1 then other threads don't need to wait, we also not care who or how many threads can set the obstacle as true, we only care whether when there exist a obstacle, it set the obstacle variable. It is proven to have the similar magnitude as serial one. However it doesn't improve through the increment of cores as shown in fig 11, we found that that's normally because the loop size is too small, which most of them are only under 10, it won't benefit from the parallelization a lot.

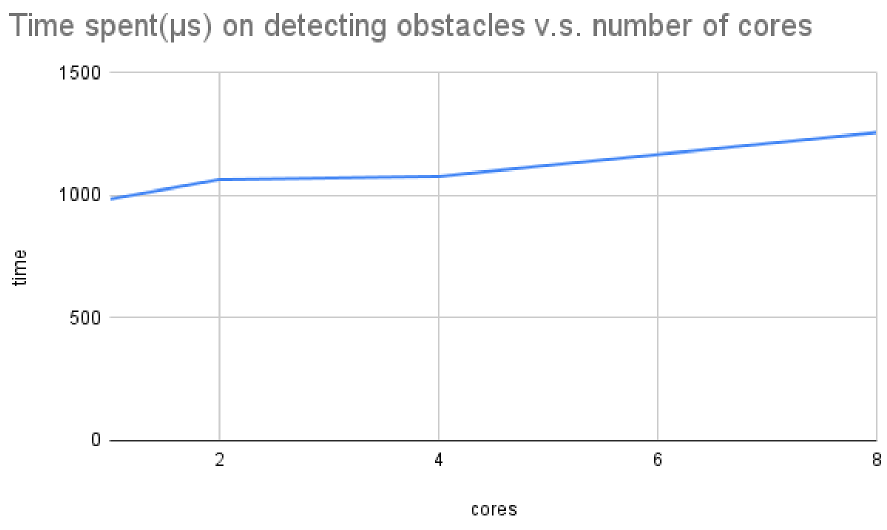


Fig 11: speedup of detecting obstacles

If you started with an existing piece of code, please mention it (and where it came from) here.

Our start code: https://github.com/Ignacio-Perez/openmprrt, it has somewhat limited optimization on parallelizing the RRT. We referred to it when implementing our basic serialized version. We do the comparison in our experiment, which is shown in the following chapter.

Results

Here are the visual results of our RRT algorithm. We mainly test our program in 2 maps, where map0 as shown in fig1 represents a simple scenario with fewer obstacles and therefore fewer number of iterations and points generated. While map1 shown in fig3 represents a much more complicated scenario which requires many steps and iterations to finally reach the destination.

If your project was optimizing an algorithm, please define how you measured performance. Is it wall-clock time? Speedup? An application specific rate? (e.g., moves per second, images/sec)

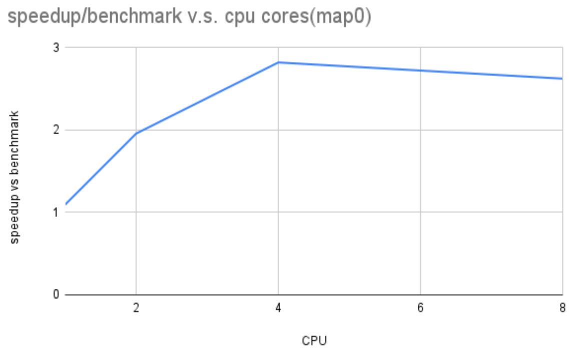
We measure our algorithm performance mainly by speedup, and we measure the speedup by getting the wall-clock time between the whole program and each bottle-neck phase starts and ends. And we measure the speedup and wall-clock time under fixed start point and end point, with a fixed seed so that we can have the same list of randomly generated points every time.

Please also describe your experimental setup. What was the size of the inputs? How were requests generated?

Our RRT algorithm takes in a start point, an end point and a map in .pgm format. The map is read into the program and transformed into a bitmap array, each element representing a pixel with the value of its gray scale. We treat every pixel with a gray scale larger than 250 as a part of the obstacles. When the program finishes, it outputs a pgm file of the original map with all the paths our RRT program has traversed. We use a shell script to benchmark the program under different numbers of CPU cores multiple times. When it’s done, the program also generates a series of log files recording the time spent running the whole program as well as the bottleneck phases.

Provide graphs of speedup or execution time. Please precisely define the configurations being compared. Is your baseline single-threaded CPU code? Is it an optimized parallel implementation for a single CPU?

Using a fixed random seed of 5000, we measured the speedup as the number of CPU cores increases under map0 and map1 as shown below. The graph on the left uses the serialized version as the baseline while the graph on the right uses the parallelized version with 1 core. We can see overall the speedup on the right is slightly lower, which is caused by the extra overhead brought by parallelizing the program. We can see that our speedup peaks at 4 CPU cores at around 4× and then drops slightly, which is when the overhead of scheduling and synchronization, as well as accessing shared data, finally caught up with the benefit brought by multithreading.



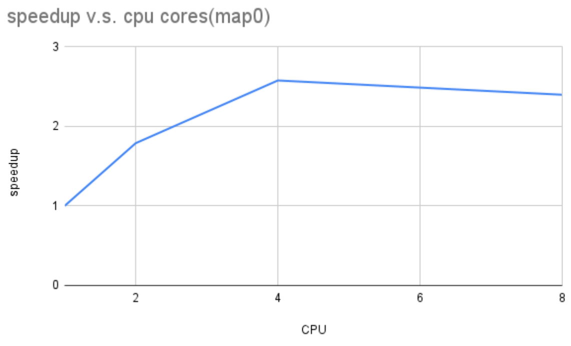
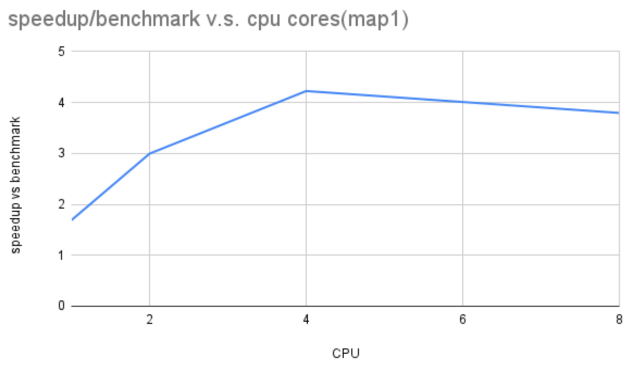


Fig 12: serialized version (map 0) speedup

Fig 13: parallel version with one core (map 0) speedup



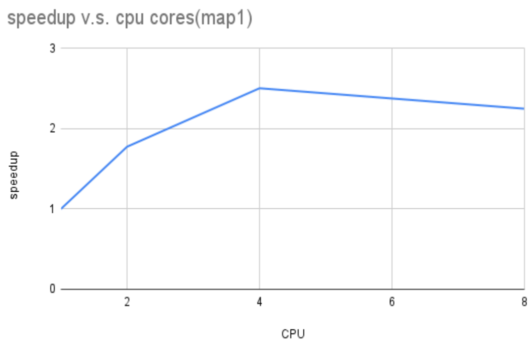


Fig 14: serialized version (map 0) speedup

Fig 15: parallel version with one core (map 0) speedup

We also measured the execution time of each bottleneck phase before and after parallelizing the program using map0 and map1, we stack the time each phase spends and put them in one graph for comparison as shown below. We use the 8 core parallel version for comparison. As the fig 16 shows each bottleneck, especially finding nearest point and inflating obstacles has achieved substantial speedup.

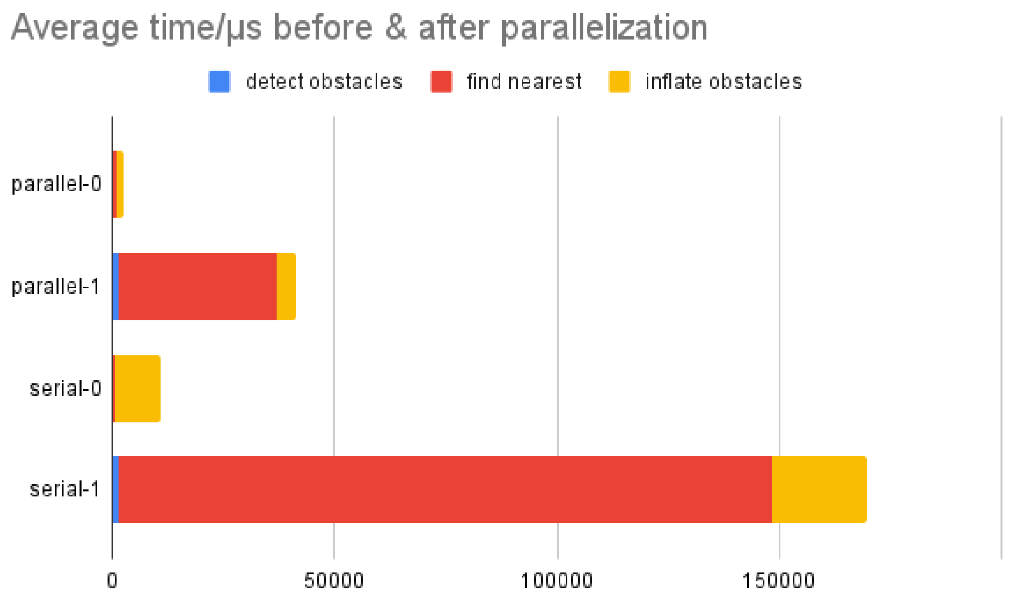


Fig 16: average time before after the serialization

Recall the importance of problem size. Is it important to report results for different problem sizes for your project? Do different workloads exhibit different execution behavior?

Yes, it is important to report results for different problem sizes. We have three maps, mainly we focused on map 0 (fig 1) and map 1 (fig 2). We found that for different problems, as I have mentioned, the time spent on different phases varies. Here is the distribution of main phases in the benchmark version of RRT.

We noticed that for simple questions like map 0, inflating obstacles takes most of the time. For tough questions like map 1, the time on finding nearest dramatically increases and detecting obstacles increases slightly. After we implement the parallel version, the percentage distribution is shown as fig 17

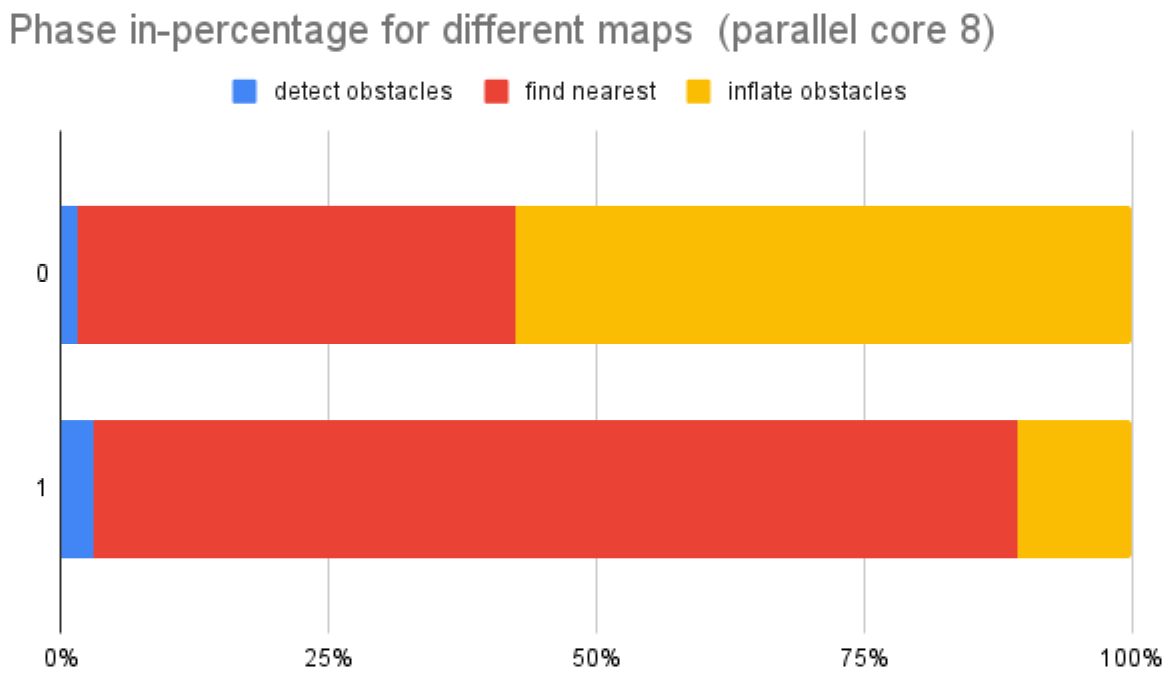


Fig 17: time different phase takes in percentage for different maps

Different workloads do exhibit different behavior as well. For the tasks of light workload like detecting obstacles, the computing resource spent on the scheduling spent more time on the time that saves the . And also we have critical operations in finding the nearest, so a lot of resources are wasted on request for entering the critical region, which decreases our performance as well.

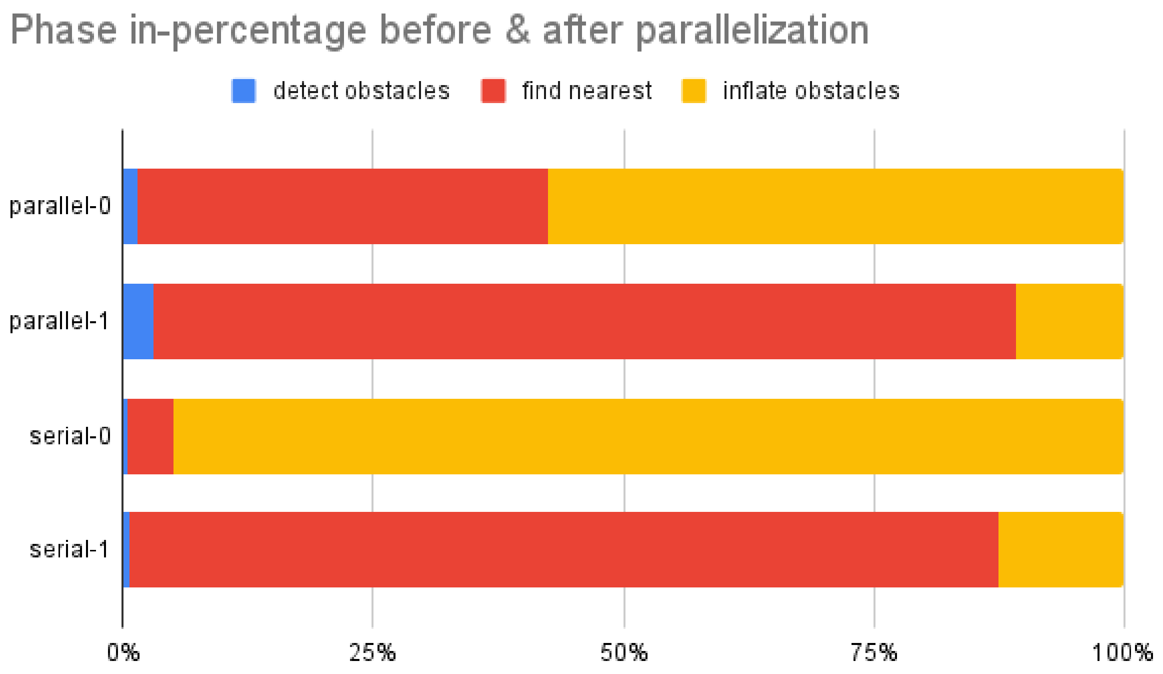


Fig 18: Comparison of the time different phase takes up in percentage before and after parallelization

Work and Credit Distribution

Group: we take around 5 hours a week during the project weeks. We did most of our optimization work together as a group. We have weekly meetings to implement the base algorithm, think of various parallel optimization, analyze the performance

Zixin Shen:

1. Did visualization pipeline

Haoran Zhang:

1. Tried different parameter combinations.

Distribution: 50-50

Reference:

[1] https://en.wikipedia.org/wiki/Rapidly-exploring_random_tree]