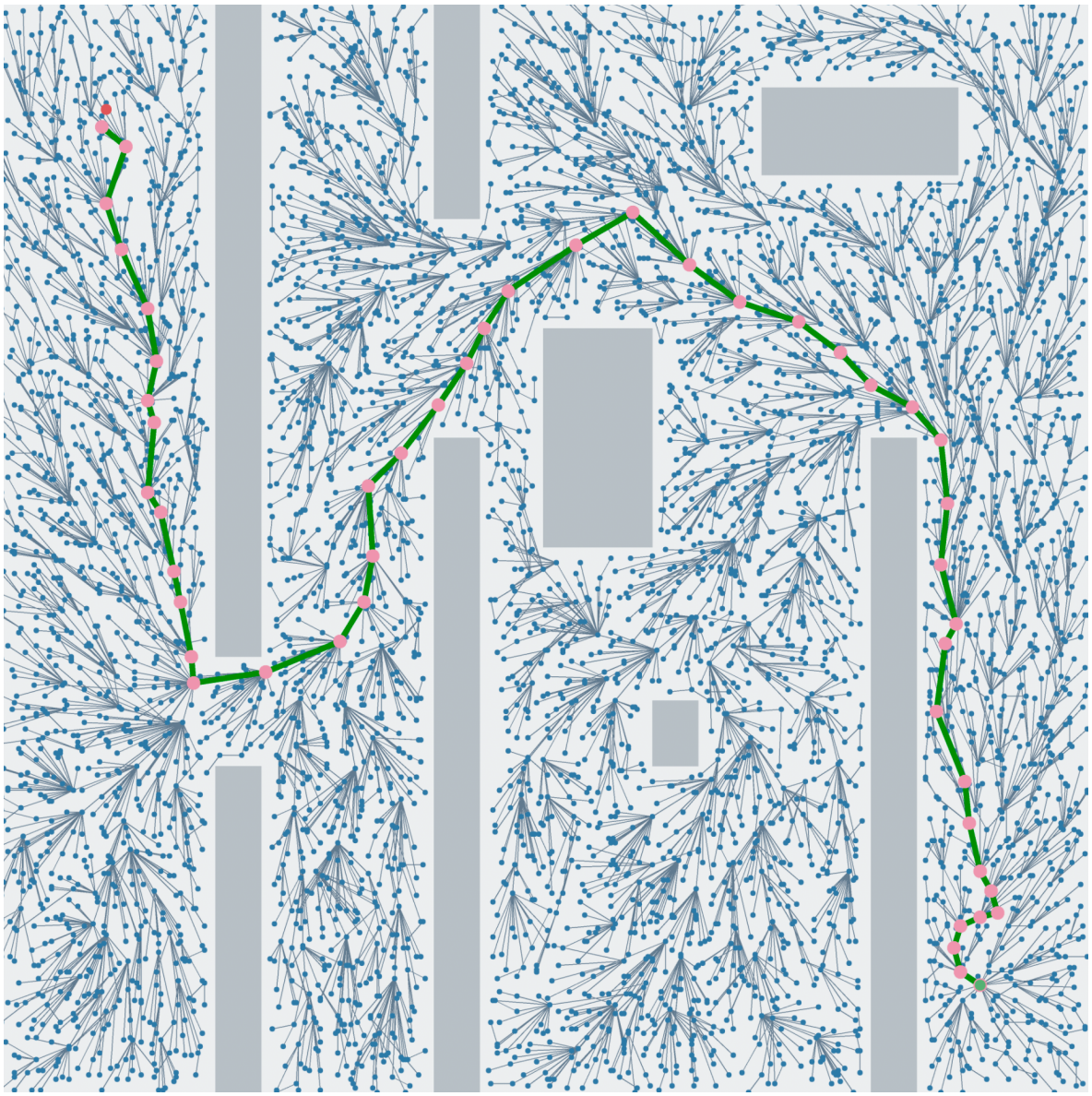
OmpRRT

Summary

We implemented the RRT* path planning / state space-search algorithm in parallel using OpenMP across multiple cores to demonstrate how an online path planning algorithm can be greatly sped-up in parallel. RRT stands for random, rapidly exploring trees. The star part of the algorithm provides an optimization step to each new node generated, greatly improving the path quality/cost but brings with it a high computational overhead. Attached is a photo of what the output of the algorithm looks like with 8000 nodes. (Computed in 0.29 seconds on the PSC machines with 8 workers)



Background

The general pseudocode for RRT* is as follows:

```
Repeat n times:
  i.   Choose a random point
 ii.   Find the closest node to this point
iii.   Create a new node from this node towards the point (by
       some distance)
 iv.   If this node is inside an obstacle, go back to (2)
  v.   Optimize nearby nodes by rearranging paths to lower
       overall path cost
 vi.   If the node is at the goal, traverse the graph backwards
       and record the winning path!
```

- What are the key data structures?
    - The first key data structure here is a directed graph. In this graph, each node is a point in the state-space and is directed towards the parent that the node attaches to. A parent node can have multiple children but can only have one parent. To traverse from a node to the starting point, we simply need to take a node and jump to its parent node until we hit a node whose parent is NULL (origin node). Each node also has a cost associated with it which is the euclidean distance of the current path from that node to the origin node.
    - The next key data structure is the list of obstacles. The obstacles are simply stored in a text file and imported into an array. Each obstacle is rectangular and its 2 diagonal corner x,y points are stored in the obstacles array.
- What are the key operations on these data structures?
    - For the graph, the key operations are
        i. finding the closest node to a random point: Allows the ‘random’ part of exploring a state space
        ii. Growing from that closest node towards the random point: Allows for the graph to be connected
        iii. Optimizing the locality of the graph (*part of RRT*). This involves looking at a physical radius of nodes around a target node to figure out if attaching a node to a different node than the closest one will lead to an overall lower cost path. Once this rewiring is done, another pass of the rewiring happens to see if any other nodes can benefit from this node being rewired. This is what gives the nodes above a ‘fan’ like distribution.
- What are the algorithms’ inputs and outputs?
    - The inputs to the algorithm are the size of the map, the number of nodes to generate, a list of the obstacles and their corner points, the distance each node grows, and the radius each node explores

    - The algorithm outputs 2 files. The first file is a list of all the nodes and an index to their parent (output solely for visualization). The second file is the path file which has the coordinates of the path from goal to start and also contains the index to the node file for that path's node (the index is also purely for visualization)
- What is the part that is computationally expensive and could benefit from parallelization?
    - The * part of RRT* is computationally expensive and takes 93% of the computation time. It is expensive since looking at a radius around a certain node means needing to search/loop through the entire set of nodes (there is no order or locality in the graph data structure). Doing this twice is quite computationally expensive but also affords massive headroom for parallel search algorithms.
- Break down the workload. Where are the dependencies in the program? How much parallelism is there? Is it data-parallel? Where is the locality? Is it amenable to SIMD execution?
    - RRT* is inherently sequential, since the algorithm must add each node one-at-a-time to the graph. However, there is a non-negligible amount of work that must be done for each node. Much of this comes from parts of the algorithm that loop through all the existing nodes, usually to find the minimum/maximum of some function applied to the nodes.
        i. Most of the computation time goes towards the RRT*-specific optimization, as mentioned earlier. This alludes to a data-parallel approach, since we must run the same checks/processing on each node in the array. OpenMP made it simple to allocate chunks of this array to each worker.
        ii. We also have to find the closest node to an arbitrarily chosen point on every iteration. A parallel reduce operation fits this problem very well.
    - Despite all these data-parallel tasks, locality is poor. Because of the nature of the graph, nodes that are close to each other in the workspace are not necessarily close in memory. And, regardless, these operations are accessing a large amount of memory, which ends up being the major bottleneck. We attempted using SIMD support built into OpenMP; however, we saw no noticeable improvement.

APPROACH

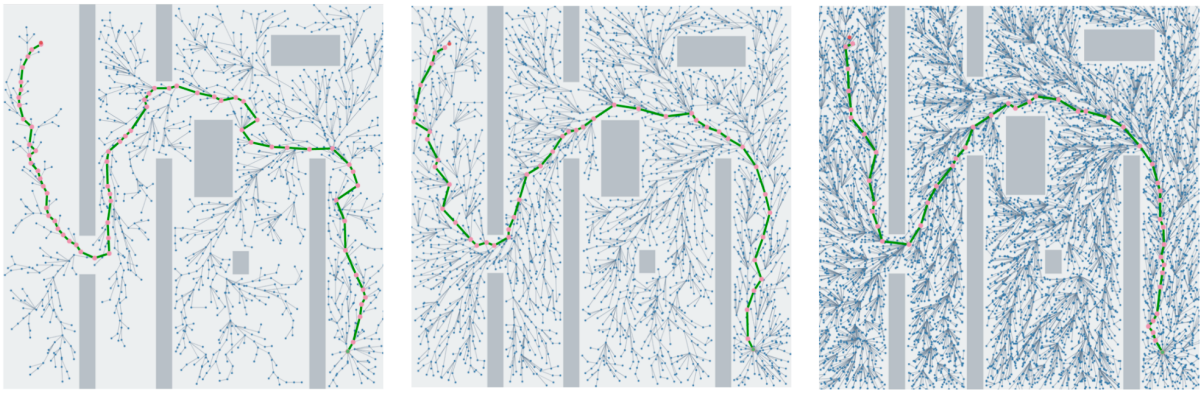
Tell us how your implementation works. Your description should be sufficiently detailed to provide the course staff a basic understanding of your approach. Again, it might be very useful to include a figure here illustrating components of the system and/or their mapping to parallel hardware.

- Describe the technologies used. What language/APIs? What machines did you target?
    - We used OpenMP with C++ as it provided a frictionless interface for us to spin parallel sections of code easily while not needing to worry about the complexity of setting up simd, or different types of scheduling etc. We targeted CPU based machines. Specifically the Pittsburgh super-computing nodes and the GHC machines. The reason for this is that often RRT* is used in an online-on-robot setting. Online robots often use ROS and Linux as their backbone. ROS plays especially nicely with C++ code making our implementation representative of how this algorithm might be used on a real robot.
    - ROS stands for Robot Operating System. More information here:
    - ‘Online robot’ means a robot that performs all its computations locally rather than sending data to some remote server for processing
- 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.)
    - As mentioned before, most of the operations we’re performing map over a collection of graph nodes. Thus, we used OpenMP to partition these collections into batches. Each worker (one worker per processor), then, works on its own batch of nodes and communicates its result by either editing a structure in place or placing its result in a new list to be reduced later on. We found through experimentation that a static scheduling algorithm with blocks of size 32 worked best.
- Did you change the original serial algorithm to enable better mapping to a parallel machine?
    - Yes we slightly changed the serial algorithm but not greatly. Some of the changes we did involve splitting up generating a new random node into multiple functions. This way, profiling and parallelizing each function appropriately was a lot easier. We also broke the original algorithm’s guarantee of finding the closest node by not locking the search process. This hurt our best path cost by 0.1-0.4% but gave us a speedup of 4%. More on this in the speedup discussion. Overall however, the core serial algorithm is largely unchanged.

- 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.
    - We tried many different approaches to parallelizing RRT*. Some strategies were more complicated than others; however, we learned that the largest bottleneck to parallelizing our code was the numerous memory accesses required.
    - We first parallelized all blocks that iterated over the node array. This gave us a significant speedup over the sequential version; however, we were far from the ideal speedup number, especially with higher core counts.
    - We tried to improve this by experimenting with different scheduling strategies. OpenMP supports many different techniques including static scheduling with different block sizes, and a dynamic scheduler that attempts to balance the workload more efficiently. Unfortunately, we found that any improvements in performance were negligible.
    - Additionally, we parallelized a larger block of code that contained multiple "for" loops. With this change, our code resembled that of a Cuda program more closely, where each worker runs the same larger block of code. And, within the block, each subsection computes some intermediary result in parallel. We saw an improvement of a couple percent, since we eliminated part of the overhead of switching from sequential to parallel execution in OpenMP.
    - We then turned our attention to improving the performance of smaller sections of the code. Specifically, we have to find the node with the smallest euclidean distance to some arbitrary point. We thought this would be a good opportunity to precompute these distances, store them in an array, and then find the minimum using a parallel reduce operation. We used OpenMP's SIMD support to automatically use vector instructions to compute the squared distances in parallel. However, the overhead of storing the results in memory and loading them back later on proved to be significantly larger than any improvement from using SIMD. This proved to be a similar lesson to Assignment 2, where recomputing values ended up being faster than caching/restoring them.
    - In conclusion, one of our first approaches, of parallelizing major components of the RRT* algorithm using a data-parallel approach over the list of nodes worked the best. It best made use of the available compute on the machine, without exhausting the memory bandwidth.
- If you started with an existing piece of code, please mention it (and where it came from) here.
    - We didn't start from an existing piece of code but the RRT* algorithm isn't one that we invented. We referenced a few sources for pseudocode, which are listed in the references section. However, we wrote all the algorithms ourselves.

RESULTS

Here are some visualizations of our RRT* algorithm over different numbers of nodes. Note how the graph edges are optimized for lowest path cost, and how the path gets better as we add more nodes.



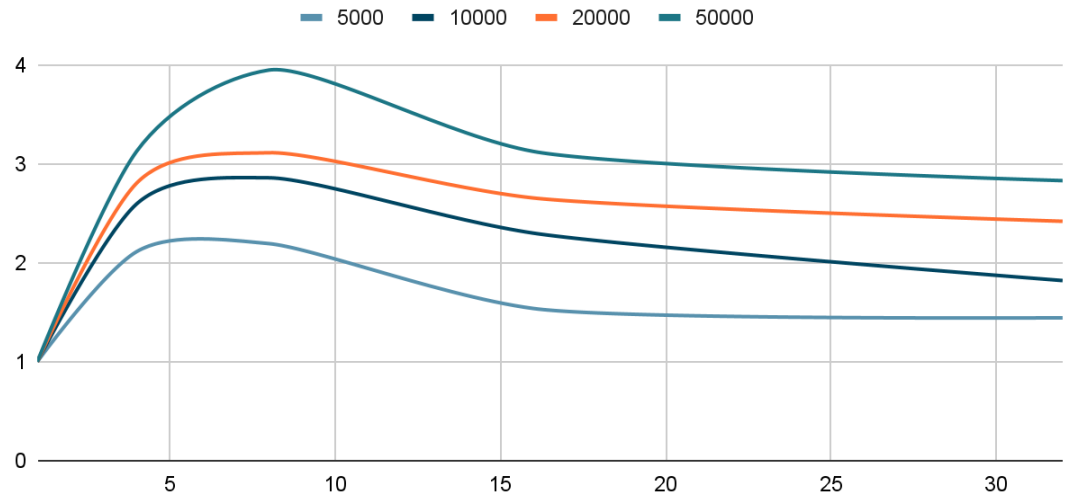
1500 Nodes 3000 Nodes 10000 Nodes

How successful were you at achieving your goals? We expect results sections to differ from project to project, but we expect your evaluation to be very thorough (your project evaluation is a great way to demonstrate you understood topics from this course). Here are a few ideas:

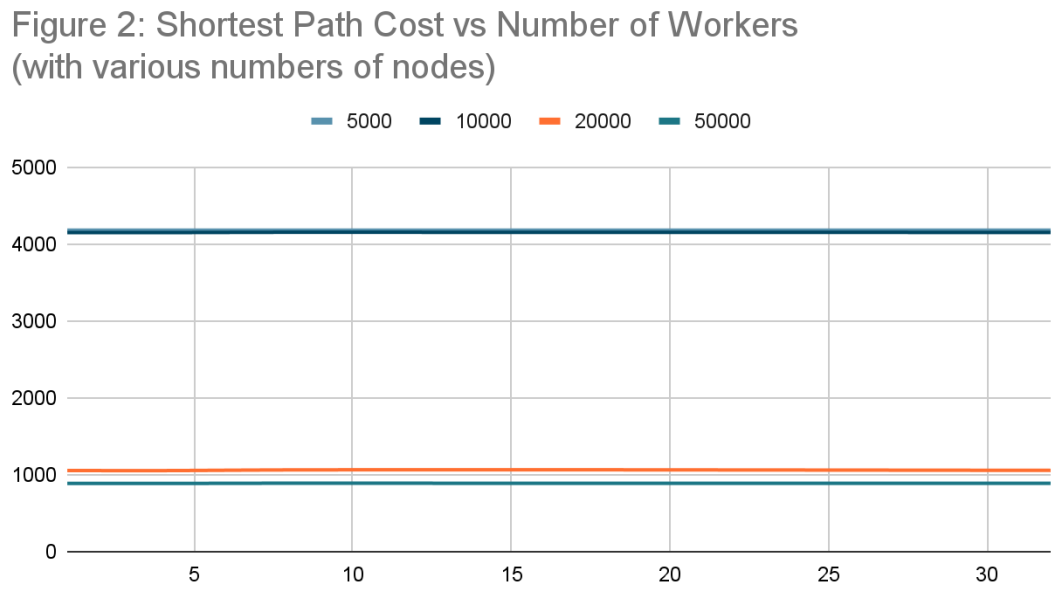
- If your project was optimizing an algorithm, please define how you measured performance. Is it wall-clock time? Speedup? An application specific rate? (e.g., moves per second, images/sec)
    - We measured 2 performance metrics:
        - 1. Given a certain time budget, what's the best path you can generate?
        - 2. What's speedup can one achieve via multiple cores
- Please also describe your experimental setup. What was the size of the inputs? How were requests generated?
    - The RRT* algorithm takes in a map, a start location, and a goal location. The map is represented by a grid (of size 1000x1000 by default, but is configurable). Following the grid size is a list of obstacles, each represented as a rectangle in (x1,y1,x2,y2) format. All this information is packaged into a text file, which we feed into the program. The program then outputs two files: a graph file that contains all the nodes and their edges, and a separate file containing the list of nodes of the final solution (best path). A separate visualizer Python script reads in these files and outputs an image showing the result (an example output can be found at the beginning of this report).

- 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?

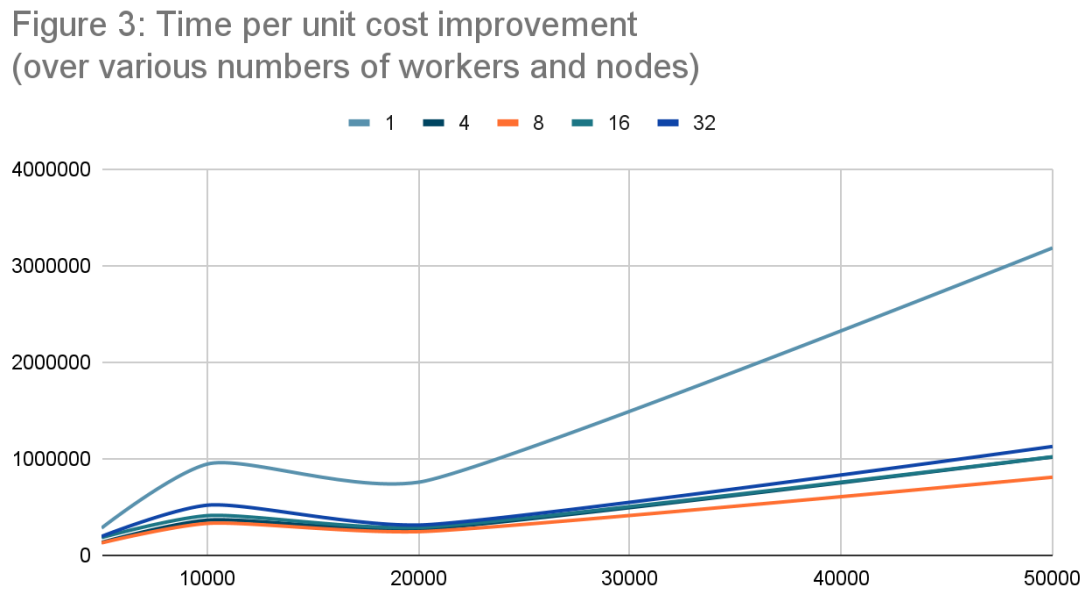
Figure 1: Speedup vs Number of Workers (with various numbers of nodes)



- **Figure 1.** The above graph shows the speedup (Y axis) we achieved as we varied the number of workers (X axis—from the set [1,4,8,16,32]). We can see that the overall shape remains constant regardless of the number of nodes the algorithm creates. Our best speedup occurs around N=8 workers—we believe this results from a balance between improvements in compute time from data-parallelism and extra overhead stemming from concurrent memory accesses/writes. Our baseline is the sequential version of the code (which is exactly the same as running the parallelized code with just one worker).



- **Figure 2.** This graph shows how the shortest path cost (Y axis, how optimal the resulting solution is in terms of euclidean distance) as we vary the number of workers (X axis) over various problem sizes (number of nodes to generate). While generating more nodes does result in a better solution, the level of parallelism has no effect on the quality of the solution.



- **Figure 3.** This last graph shows the wall clock time in microseconds divided by the shortest path cost improvement over 5000 nodes (Y axis) versus the number of nodes being generated (plotted for different numbers of workers). In other words, lower values means that the algorithm spent less time, on average, optimizing its best path, assuming

that this time is scaled proportionally to the cost improvement. We see two minima at 5000 and 20000 nodes, meaning that these node numbers are best “efficiency”-wise.

- 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?
    - Through our testing, we saw that there are 2 primary local-minimas that an engineer can optimize for. One could either optimize for really fast times by keeping a low number of nodes and exiting early (see 5000 nodes in fig 3) or an engineer could optimize for a high quality path but have it take longer (see fig 3 at 20,000) nodes
    - The larger node size (problem size) does lead to more computational time. Looking at Figure 1, we can see that the speedup is actually higher in a high node case. This is due to the higher number of work done for each worker offsetting the fixed cost of spinning up a node and synchronizing it at the end. (eg with 5,000 nodes and 128 workers, each worker will process 19 nodes on avg for the ‘findNearestNode’ function whereas with 50,000 nodes, each worker will analyze 190 nodes on average, diluting the fixed costs of parallelizing.
    - The map size doesn’t impact the parallel speedup of the algorithm. This makes sense given the random nature of exploration not caring about the map size. Of course, the larger a map, the more nodes one needs to explore to find a path from the start to the goal but this doesn’t impact speedup.
- **IMPORTANT:** What limited your speedup? Is it a lack of parallelism? (dependencies) Communication or synchronization overhead? Data transfer (memory-bound or bus transfer bound). Poor SIMD utilization due to divergence? As you try and answer these questions, we strongly prefer that you provide data and measurements to support your conclusions. If you are merely speculating, please state this explicitly. Performing a solid analysis of your implementation is a good way to pick up credit even if your optimization efforts did not yield the performance you were hoping for.
    - Memory access was the biggest bottleneck. Unfortunately there is no real way to get around that given how a graph is physically distributed across memory on the system, how each node can have anywhere from 0 to infinite children, and the rewiring of nodes due to the RRT* algorithm means that the graph gets even messier over time. Due to those reasons, a static memory allocation is simply not possible and we were bottlenecked by NUMA.
    - In fact, we verified the memory bottleneck by running an internal experiment. By simply replacing the memory accesses in the optimizer section of our code with simple constant values, we observed a speedup of 94%
    - To tackle this memory bottleneck, we tried a bunch of different solutions including looking up various suggestions by OpenMp’s team themselves in the following presentation (link 3 in references). However, given the nature of the

reshuffling of nodes, dynamic graphs, and non adjacent memory locations for each node, static allocation into thread space cache or keeping data within each thread's cache was just not possible. This is just an inherent issue with a graph based approach and moving to a different core data-structure would defeat the 'tree' like nature of this algorithm.

- Since memory bottlenecks were something we couldn't directly optimize, what we could do was indirectly optimize it by cutting down the number of calls to memory. To do so, we rearranged some of the order of operations in the original published algorithm's pseudocode. For example, we did a simple check to see if a node was even close enough to a node being optimized before we spent the memory cost of checking if a path between those nodes would collide with an obstacle (involves iterating over all the obstacles). We also saved the indices of all the nodes that were within a certain distance of our original node to prevent needing to iterate through that list a second time. This alone saved us 4% of memory calls and sped up our implementation by 5%. This was the one case where saving data in memory was faster than re-computing since re-computing would have indirectly led to a greater memory search-space.
- We also tried using the SIMD parallizer in performing the collision detector since the way the collision detector check is written, it is very friendly to being sped up by SIMD (many nodes and obstacles, minimal conditional execution). However, the speedup granted by the SIMD execution was massively trumped by the slow bottleneck of needing to save these values in a Lookup Table and dereference them later—i.e. computing on the fly was much faster than pre-computing and storing in memory.
- We briefly toyed with shared memory vs private-memory since the memory being iterated over was not being modified and therefore private-memory would avoid contention on the memory bus and we would not need to do a reduce operation at the end since the memory being worked on was not modified. However, the very act of copying shared memory to private memory was slow enough that it was greater than the speedup from using private memory. Therefore, we stuck to shared memory instead.
- We also experimented with using shared memory and non-shared memory with a ring-reduce (both tested separately) across cores while searching for the closest node. Shared memory was faster likely due to not needing to do a reduce operation. (separate computation + ring reduction might have been faster if the graph's nodes had cache locality). We also experimented with using shared memory with and without locks. Not using locks means a slightly suboptimal nearest node might be found if there's a memory contention. However, in practice, we found the race conditions leading up to a slightly non-optimial nearest node only affected the best path's max cost by ~0.1-0.6% whereas it sped up the

algorithm by ~4% (on 8 cores). These results are likely due to the high overhead of locking-and-unlocking combined with the sparse update of a best node leads to few cases where a race condition overrides data. Given the results mentioned above, a lock-free approach was the design we went with.
    - Next up, we tried dynamic vs static scheduling with various types of static schedules and batch sizes. Given how the memory was nearly randomly distributed in the system, we didn't see a massive difference switching around the schedule sizes and batch sizes. We did notice that static scheduling with a batch size of 32 was marginally faster than the rest but the exact reason for this is something we're not 100% sure about. We think it's due to the lower runtime overhead of static scheduling and 32 being small enough of a batch size for the thread to keep everything in fast cache while computing.
- Deeper analysis: Can you break execution time of your algorithm into a number of distinct components. What percentage of time is spent in each region? Where is there room to improve?
    - Via profiling we discovered that 93% of the time is spent on the star (*) portion of the RRT* algorithm. This is the portion where each new node is uniquely compared against all the other nodes, this is where the memory access bottleneck comes from as mentioned in the last section. This is also where the bulk of the parallelization happens.
    - 6% of the time was consumed by the algorithm finding a 'nearest node' to grow from after selecting a random point. Again this suffers from the same memory bottleneck created by looking through all the nodes. This too was parallelized
    - The rest of the functions took ~1% of the total execution time and therefore we didn't pay much attention to speeding it up since the benefit would be miniscule.
    - Profiling was our first step and helped us target our efforts in the places that needed the most.
- Was your choice of machine target sound? (If you chose a GPU, would a CPU have been a better choice? Or vice versa.)
    - We believe a CPU was the best choice for RRT*. As we uncovered through writing the sequential code, parallelizing it, trying different approaches to optimize our solution, and profiling it over different problem configurations, RRT* is inherently sequential, and relies heavily on memory accesses over a common data structure. A GPU would not help here, since it would just run into the same memory-based bottleneck that we faced with the CPU, just on a much more significant scale. The best way to improve performance would be to design a custom data structure that better respects locality in the memory hierarchy.

REFERENCES

- Both these references were only used for helping us ideate and understand the algorithms. The actual implementation was all done by us.
    - Link 1: https://en.wikipedia.org/wiki/Rapidly-exploring_random_tree#Algorithm: Used for understanding the pseudocode behind RRT and RRT*
    - Link 2: https://stackoverflow.com/questions/5514366/how-to-know-if-a-line-intersects-a-rectangle Used for pseudo code for how to implement a collision detector
- Link 3: Mastering OpenMP: https://www.openmp.org/wp-content/uploads/openmp-webinar-vanderPas-20210318.pdf Used for tips and tricks for making the most out of OpenMp

LIST OF WORK BY EACH STUDENT, AND DISTRIBUTION OF TOTAL CREDIT: Please list the work performed by each partner. Given that you worked in a group, how should the total credit for the project be distributed amongst the participants? (e.g., 50%-50%, 60%-40%, etc.) If you do not feel comfortable placing this information on a public web page, it is okay to include it only on the version that you submit via Gradescope.

Most of our work was done in multiple 4-5 hour long meetings over the course of the last few weeks. We have listed the work we did together as a group, and individually below:

Group:

- Implemented base RRT algorithm together in group sessions
- Trying various parallel optimization pathways/techniques
- Compiled/analyzed results and profiled performance
- Debugging! (lots of it)

Saral:

- Wrote up RRT*
- Wrote up collision checker (node w/ obstacle)

Ravi:

- Wrote up visualizer script

Distribution: 50% – 50%