Parallel Maze Generation and Particle Filtering

Justin Hsu
Carnegie Mellon University
Pittsburgh, PA 15213
justinhsu@andrew.cmu.edu

Utkarsh Murarka
Carnegie Mellon University
Pittsburgh, PA 15213
umurarka@andrew.cmu.edu

1 Summary

In this project, we explore a parallelized application of maze-generation and particle filtering. Specifically, we implemented maze-generation using MPI and used openMP and CUDA for particle filtering. For maze-generation, we achieved a speedup **higher than linear** by carefully modifying the algorithm to accomodate parallelism and advantages/limitations of OpenMPI. Moreover, we believe our algorithm could be adapted for many tree-based workloads. For particle filtering, we achieved significantly higher speedup than single-threaded CPU on the GHC machines. We achieved a speedup of 40x with the CUDA version.

Throughout this journey we learned to integrate sequential and parallel algorithms when the granularity of work changes. Specifically for particle filtering, we learned to structure code to fit the problem. The end result taught us how to format our code to be adapted for GPU-accelerated games, as designing code to fit a data-parallel model is very functionally specific to a particular problem.

2 Background

Maze-generation should be pretty self-explanatory. There are many ways to generate randomized mazes. We chose to use Prim's algorithm, which uses a tree-based depth-based algorithm. This approach assumes that the maze consists of just walls and iteratively removes walls to create passages. We start from the top-left of the grid and remove the wall. Once the wall is removed, we add all the horizontally and vertically adjacent walls to a 'wall-list.' This list is our potential targets for removal in the next iteration. To ensure we don't keep adding and removing the same walls from the list, we also maintain a 'visited' grid. On each wall removal or discovering a wall that we cannot change, we add the wall position to the visited list. If we find a wall that we have previously visited, we ignore it, and we don't convert it to a passage or add to our 'wall-list.'

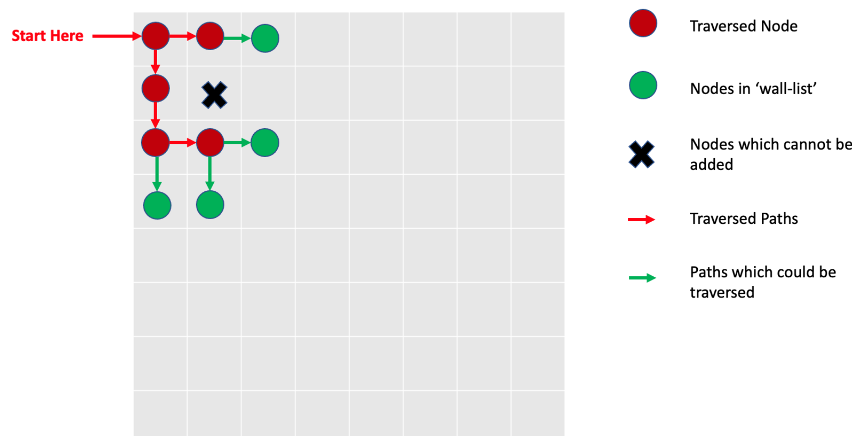

Figure 1: Maze Generation in 2D grid

We can then use our grid generated in *Particle Filtering*. Particle Filtering is a Monte-Carlo algorithm for Bayesian statistical inference. In the context of the filtering problem, we are trying to learn an unseen dynamical state given a set of partial observations. The objective is to compute the posterior, ie, the unseen state given the observations in a Markov process. To solve this problem, we use a set of particles to estimate the posterior distribution. This has an important application in robot localization.

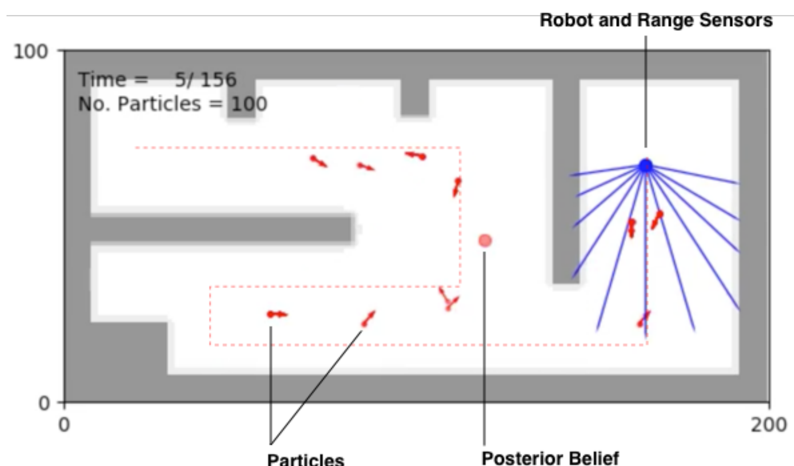

Figure 2: Particle Filtering Example in 2D Grid

Here, the blue dot is the location the robot. The blue rays are local observations of the environment. The red dots are the particles that the robot is simulating. Finally, large clumps of particles represent the particles posterior belief, ie, large clumps imply that it is very likely the robot is in that location in 2D space. **To be clear:** The robot does not know it's location. It is using a random set of particles to simulate it's estimated position (the posterior) through the rays (partial observations).

Randomized maze generation, in different forms, is beneficial for any sort of mapping and measurement of route-finding algorithms. One ubiquitous example is Google Maps. Before rolling out features that can find the shortest path between two locations, Maps has to generate complicated graphs to ensure that the algorithm can work and optimize itself for unseen cases. Particle filtering has many essential applications in robot localization. The roombas that self-clean your house are programmed on this principle. They must learn it's local location in your house given 3D-partial observations, and then figure out the most efficient usage of the space. In this report, we explore maze and particle filtering through 3 different parallel programming models: message-passing (MPI), shared-memory (openMP), and data-parallel (CUDA).

2.1 Inputs, Outputs, and Data Structures

Maze-Generation Maze generation is designed with simplicity in mind. As input, it requires just the dimensions of the grid, the height, and the width. It should write the final 2D grid to the output file as output. It requires three primary data structures:

1) Output-grid (dimensions: height x width)
2) Wall-list: A list of walls that have to be visited.
3) Visited-grid: (dimensions: height x width)

As we traverse the grid, we come across grid positions where we could potentially plot a passage (i.e., remove the wall). The wall-list keeps track of all these walls, where we append and remove walls from this list. Now, it is possible that one grid position has already been traversed and is found to be unsuitable (perhaps it creates a grid block rather than a passage). Moreover, we could just add and remove walls from the wall-list infinitely. We keep track of the visited grid positions using a separate 2D array to prevent this. If we find a wall position from our wall list that we have visited before, we just continue finding another wall that we can convert into a passage.

Particle-Filtering There are two main inputs. The first is the environment state. In figure 2, this would be the map of the walls and open space, excluding the location of the robot and particles. Let S_{ij} be environment that the robot is placed in. $S_{ij} = 1$ if there exists a wall in that location, otherwise it is 0. The second input is the likelihood observations of the particular location in the environment. In this world, we represent our likelihood using a Gaussian distribution. Let O represent a particular partial observation.

$$P(O = o|S_{ij}) \propto exp\{-\left(\frac{x_o - \mu_r}{\sigma}\right)^2\}$$

Here x_o is the observation of a single particle, and μ_r is the true observation of the robot. We can interpret the equation as follows: If the robot is 20ft from a wall, and there are 2 particles: one is 10ft from the wall and the other 15ft, then the belief that we are the 2nd particle is higher than the 1st since we are farther from the wall. The variance σ was found empirically through testing as to prevent the numerator from exploding (due to the fact that the GHC machines cannot represent infinitesimally small numbers).

The output is the posterior belief of the robots location. This can be found using bayes rule.

$$P(S_{ij}|O = o) \propto P(O = o|S_{ij}) * P(S_{ij})$$

Note that the prior $P(S_{ij})$ is simply the number of particles in that location. More particles means we have a higher prior belief that the robot is in that location.

The main data structures used are contiguous row-major arrays. We store the locations and (angle) orientations of each particle, as well as the (ray lengths) observations of the robot. We avoided an array of pointers because that involves multiple layers of array accesses, and is hard to integrate in the data-parallel CUDA model.

2.2 Algorithmic Observations

Maze-Generation There were two significant observations as the grid size was doubled. Though small grids were generated in tiny fractions of a second, larger grids took longer than linear time to generate. This strange behavior led to the belief that larger grids were likely crossing cache lines. One more distinct problem arose. At each iteration, we find a random wall, ignore or convert it into a passage, and then remove this wall from the wall-list. Removing from the wall-list is a problem since all the items after this wall's index have to be moved to the left. There is an error rate of approximately 47 percent in the wall-list, i.e., 47 percent of the walls are pre-visited. For large grids and subsequently large wall-lists, coupled with the 47 percent "miss-rate," this causes a worse than a linear slowdown. We could improve this approach by using parallelism to divide our tasks correctly.

CUDA (Abandoned): Though CUDA exhibitions excellent parallelism, there are not enough tasks for thousands of threads here. It could be beneficial when we have obscenely large grids, where each thread is working on separate parts of the overall tree.

OpenMP (Abandoned): Using OpenMP would be too simple. If we could figure out how to divide the grid, then each thread could do separate parts and use shared memory to build a global grid. This approach seemed too similar to Assignment 3.

OpenMPI: Using MPI would be an exciting challenge. Again, if we can divide the grid, we might be able to figure out a way to minimize communication between the different branches of the tree and generate our grid. Moreover, each processor would have its cache and its own copy of the wall-list, which may solve many of our performance bottlenecks.

In summary, we try to divide the tasks, and divide them evenly to each core. The GHC machines have 8 cores, so we strive to measure sequential performance by doubling by cores from 1 to 8.

Particle-Filtering Sequential versions usually involve 3 phases: 1) *transition* (move the particles according to the robots orientation and speed), 2) *reweight* (calculate the posterior belief), and finally *resample* (sampling new particle locations and orientations according to step 2). First we notice that step 2) is computationally expensive because it involves many line segment intersection calculations. Second, we observe that for a single particle, step 2 must occur after step 1). This lead to the realization that steps 1 and 2 can actually be pipelined for all particles! For example particle 0 can work on steps 1 and 2 before particle 1 finishes step 1. Finally, step 3) can only occur once all particles have finished step 2). This means there must be a barrier between steps 2 and 3.

OpenMPI (Abandoned): MPI does not fit this task due to the fact that we are also simulating the rendering function and robot movement. Furtheremore, we wanted to apply knowledge from assignments 2,3, and 4 to the best extent possible, so MPI was chosen for maze generation and openMP and CUDA were chosen for particle filtering.

OpenMP: We can transition and reweight the robot's posterior belief. However, step 3 can only be completed once steps 1 and 2 have *fully* completed. This leads us to multiple threads for steps 1 and 2), a thread barrier, and then a continuation onto step 3).

CUDA: In the data-parallel model, we attempt to give each particle a thread to calculate each observation (in hindsight we should have been giving a workload of a single ray to each thread). This will involve scan operations to gather the observations and another scan to sample from a distribution in step 3) (for a more detailed explanation take a look at the *approach* section). In addition, we must try to avoid many *memcpy* back and forth from device to host, as this can serve as a bottleneck for $10k+$ particles (we solve this issue by creating a global device struct in constant memory simliar to the idea in Assignment 2).

Note that cache locality isn't too much of an issue in particle filtering. We mostly access each particle in the order of the array, regardless of where it is in the location of the grid. The only locality we need to take care of is during the renderering portion in openGL. When we access each particle, we want to render the particles whose locations are similar to each other.

3 Approach

Both maze generation and particle filtering were implemented and tested on GHC machines.

Maze-Generation We will begin by explaining the sequential approach and then delve into the final approach with MPI. MPI approach will also talk about what challenges were there to achieve a solution and how we overcame them. We will then go over the abandoned ideas and the reasoning.

Sequential

We created the Output-grid (full of walls or '1') and 'Visited-grid' (empty or '0'). We take the top-left cell of the output grid, convert it into a passage (turn into '0'), and visit this node (visited-grid into '1'). All the horizontal and vertical cells from this initial position are added to our wall list.

While we have items in our wall-list, we generate a random index within the wall-list range and take the grid position. We ignore this position if we have visited it before or if the position has more than one passage around it. If not, we make this position a passage, update the output-grid and the visited-grid, and add all the surrounding non-visited walls to the wall-list. We then remove this item from the wall-list, iterate, and find the next wall.

OpenMPI - Evolution and Final Approach

Evolution - Multiple origins: We started the OpenMPI implementation by thinking about each node in the graph. Would it be possible to start from multiple points in the grid and generate the grid. This seems like a simple idea and behaves similar to algorithms like Kruskal's work (which we also abandoned). However, to achieve a correct solution, we need to communicate every time we remove a wall with every other thread. This would lead to a very high communication-to-computation ratio. Moreover, if someone had already written to this location, we couldn't do anything about it. OpenMP would have been a good candidate here, where we could have added each 'contentious' node in a critical section.

Evolution - Multiple-Nodes communication: The logical next step was to consider all the items in the wall-list of a processor as belonging to only that processor. No other processor could work on those items. However, some processors would finish much later than others, especially those with larger wall-lists. It also does not solve the problem of communicating which wall-lists we are working with. Let's also assume that we find a grid position that we cannot convert into a passage because one of the adjacent walls belongs to another processor. How do we make the other processor guarantee, there's at least one path out of the maze? This was too complicated, and striving for simplicity, we moved ahead.

Evolution - Single initialization, branch-based A very different approach, we thought about starting at the only initial position and then providing each processor some part of the tree to work with. However, this again suffered from the same single-node contention that the previous approaches suffered from. However, this idea was intriguing, and we used it in the final approach.

Evolution - Block-based What if we could divide our grid into equal square blocks, where each processor works on each block, and after each processor is done, they can communicate their boundaries to the neighboring blocks. It solved the problem of the communication-to-computation ratio, but then we would have to re-adjust the block to find a path by re-introducing walls. Moreover, each square block would need to know its position in the grid and then communicate accordingly to only its neighboring blocks. Not impossible, but complicated.

Evolution - Block-based with ghost rows We could generate the ghost rows first, send them over to the neighboring blocks, and then build our grid with ghost row information. However, it didn't solve our 'which block to communicate with'. Also, for grids with a larger width, we could potentially be running over cache lines. Idea!

Final Approach - Height-based blocks with ghost rows We divide the grid into equal blocks, but only by height (so we can preserve the cache lines). Like the previous approach, each block randomly generates the top-most row and updates the output-grid to '0' and visited-grid to '1'. This is the ghost row for the block above, and it's sent by point-by-point communication (asynchronous send and synchronous receive). Each block can only continue once it's received the ghost row. Now, each block updates the visited-grid to '1' for the ghost row and the top-most row to '1'. We cannot modify our own or others' ghost rows.

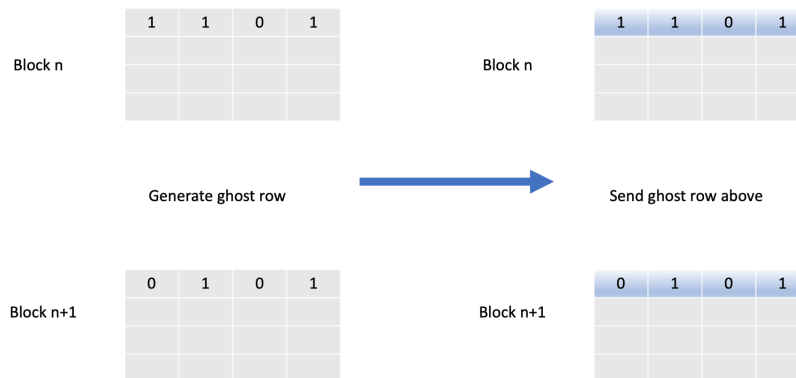

Figure 3: Creating ghost row

Now, for each block, we take the left-most passage from the top, use that as the 'root' and generate a grid typically using the Prim's algorithm. However, we want to connect the top grid to the next grid. So, when we find any wall in the 2nd most bottom row, we change the algorithm slightly. We connect this position, to the first 'left-most' passage of the ghost row. The left-most passage is guaranteed to be the root for the next block, so we can connect them and receive a maze with a solution.

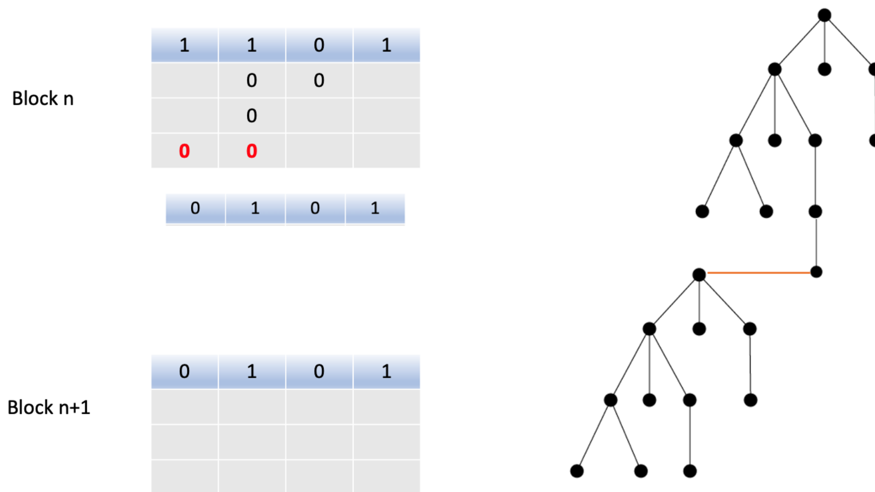

Figure 4: Connecting grids

Finally, we write to our output file sequentially, processor by processor. The first processor creates the file while the subsequent processor waits. Once the processor finishes, it informs the next processor by sending a status over MPI, and the following processors appends its local grid.

Particle-Filtering First we will explain the sequential approach and then the natural extension to openMP. We then describe the CUDA implementation. (Note, we did not use existing code as existing resources are scarce. Most of the code was built off the references on the last page)

Sequential

1) *Transition:* First, we use a for loop to move each particle according to the robot. One movement is defined for one frame in openGL. The particles move at most 1 pixel (they could hit a wall and not move at all) according to their orientation. For example, suppose the robot is facing north and it moves according to the best particle (highest weight). If the robot is facing 90 degrees, and moves forward, then it will move north 1 pixel (unless it hits a wall). If another random particle(doesn't have the highest weight) is facing -90 degrees, then it's "forward" is actually going south 1 pixel. It's important to note this important distinction which means we must store the locations *and* orientations of each particle. (remember that the location can be stored as a single integer and the orientation (angle) can be stored as a single float).

2) *Reweight:* After the particle has moved 1 step, every particle, including the robot, fires 16 rays (can be changed with cmdline arguments). Each ray is evenly spread around 360 degrees (unlike in figure 2 where the field of view is limited). Each ray is then numbered from 1 to 16 according to it's orientation. Let $R_r^{(i)}$ be the ith ray of the robot and let $R_p^{(i)}$ be the ith ray of the a random particle p. Then the observation is calculated with $\sum_{i=1}^{16}\left(R_r^{(i)} - R_p^{(i)}\right)^2$. We do this for all particles.

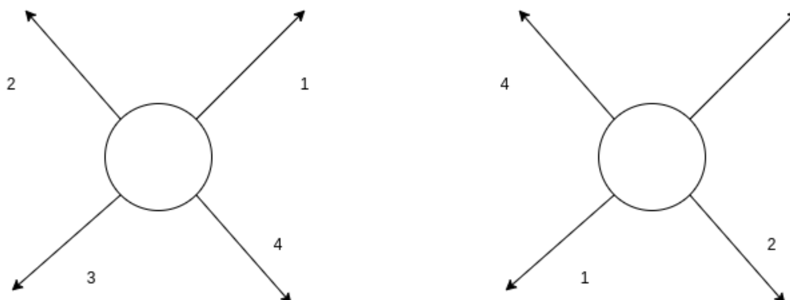

Figure 5: Simplified with 4 rays example particle ray calculation. WLOG assume that the robot is on the left and the particle is on the right.

Note that in the figure, the rays extend to hit each wall. The distance from each particle to the wall is calculated from each ray fired. The numbering of the rays depicted describes the direction in which the robot (on the left) and the particle (on the right) are facing. The 1st ray is the angle they are facing. So the robot is actually facing 45 degrees from horizontal while the particle is facing 225 degrees from normal. What this implies is that a particle could have the same location as the robot but have a low weight due to the fact that it is facing 180 degrees opposite of the robot.

Finally we compute the gaussian distribution (according to the equations in section 1) and store this in an array of *weights*. The higher the weight for a particle the higher the probability that the robot lies in that location

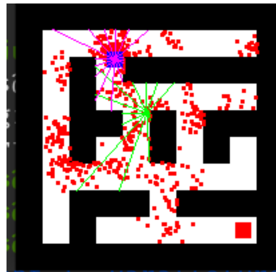

Figure 6: Example particle filtering reweight step with 500 particles

The blue square is the robot and the rays shooting out are it's observations. The green rays are the observations of the **best** particle (highest weight). The robot moves according to this particle. The big red square is the goal. You can imagine the goal as a dirty spot in a kitchen that the roomba needs to clean. Because it believes it is in greens location, it wants to move down. However, because the best particle is facing down and the blue robot is facing right, the robot moves into the wall and the green particle moves down. **Note** that even though the robot makes an incorrect guess, there is a huge clump of particles near the blue robot. The robot is converging to the true location just by moving around and collecting observations.

3) *Resample:* First we calculate a *confidence score*. The confidence score evaluates how confident the robot believes in it's location. (In reality we don't do this because a real robot has better sensors than firing rays. In this simulation, we calculate a confidence score so that robot can confirm it's location. If it is correct, the simulation is over and robot knows it's location. If it is not, we uniformly sample new particles all over the grid). We can calculate a confidence score based on the highest weighted particle. Let w^* be the max weight of all the particles. Then we calculate the total fraction of particles near the best particle p^*. Using a set threshold 50%, if more than half of the particles are in a single grid location, and the robot guesses correctly, then the robot knows the exact location of the robot.

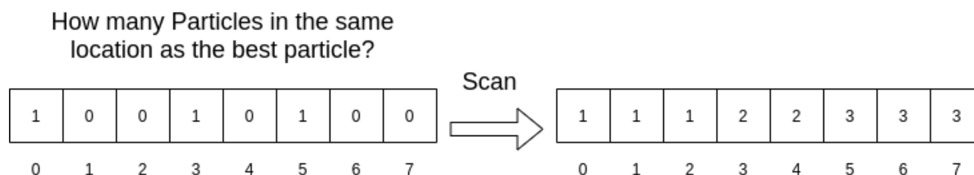

Figure 7.1: example confidence calculation. Assume that the particle with the highest weight is in particle 0. Then the confidence score is 3/8, which does not exceed the 50 percent

Otherwise, we calculate a running value of the weights to resample. How do we sample from a distribution of weights? Well we can perform another prefix sum scan and then generate a random number from 0 to the total sum of the weights. We then loop through each particle's prefix sum and the *first* time the prefix sum weight is *smaller* than the number randomly generated, we sample that particles location and orientation.

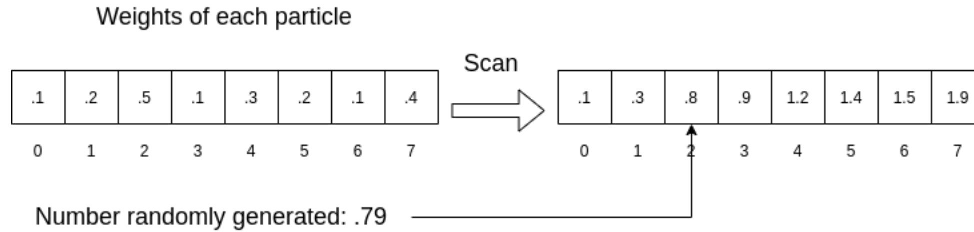

Figure 7.2: First we perform a prefix-sum operation over the weights. Then we randomly generate a number, in this it turns out to be .79. We find the first index in which the prefix sum weight is greater than the randomly generated number. This happens to be the particle 2, which makes sense because it had the highest weight(so it has a higher chance of being resampled)

The sampling has some gaussian noise because in real life the robot observations experience some noise and aren't perfect. Due to this, we resample the particle if the it lies inside a wall.

Extension with openMP: In the shared-address space model, we can simply add threads to step 1) and 2) and access the memory locations for the particle locations, orientations, weights, and ray distances. We experience good spatial locality through our representations of row-majoring arrays rather than an array of pointers which would require two memory accesses. By making granularity of work for each thread a single particle, we don't need to use locks for calculating the gaussian belief. However, when we calculate the confidence score, we must use a lock to avoid the ABA problem(each one of the threads will calculate a partial sum reduction on the confidence score).

There are 8 processors on the GHC machines. This means we should be able to achieve speedup up to 8 threads (where each thread get it's own execution context) and for more than 8 there shouldn't be great speedup.

We will try a *static* assignment of the work per thread. If this is infeasible we may try a dynamic assignment. If this has too much overhead in workload assignment, we will try to implement a *semi-static* assignment where we will try to predict the workload according to the locations of the particles. For example, we speculated that particles in open locations may require less work than particles in more confined (more walls around) locations because we would need to calculate the wall intersection and store it in an array vs. the case where we know the rays do not intersect the wall. This is similar to assignment 2 where we used the circle in box to test to find out whether we a circle covered a pixel. The semi-static assignment is something we would like to try if static and dynamic do not work out.

CUDA: Before we dive into the CUDA implemention, we want to preface with something we discovered: while implementing with CUDA, we had to be careful in file structuring, device constant and global memory, and shared memory in threads. We put pointers to the particle weights, particle actions and locations, and the grid map in *constant device memory*. We did NOT put the robot in cuda memory because there was only one robot. This meant that we needed to be smart about memory transfers from and to device and host. We tried to share memory as much as possible rather than global memory. Furthermore, we chose to have a block of threads on a GPU core to be have 256 threads. This is a traditional amount of threads so that each GPU core is unburdened with the memory overload. This was many hours of debugging.

1) The transition was fairly straightforward. We simply gave each thread a particle to move in one step. This does not have any data dependencies except for checking whether a particle hit a wall.

2) The reweight step was more complicated. First we tried giving each thread a particle. However, this was too much work per thread because we had to calculate a 16-ray intersections per particle. This also had the downside of allocating more memory per particle which didn't give good speedup. We then decided to use a *ray* per thread. This meant that when we launched a kernel, we had to carefully map each thread to a specific ray with it's orientation. This was done by launching 16 times the number of particles (First 16 threads are the 16 rays of the first particle, etc...). Then to calculate the Gaussian belief we performed, you guessed it, and exclusive scan to calculate the running weight(code was copied from assignment 2). This meant that we had an array in shared memory where each warp calculated a portion of the scan quickly. Finally we had to carefully find

the observation per particle. To be clear, the block size was 256 threads, so we had 16 particles per thread. The scan was done for all 16 particles, so we must subtract prefix sums to find the local observation per particle. This gave better speedup than the prior approach, and reduced the size of shared memory. The only limitation that this had was that the number of rays per particle had to be a **factor of 256**, since the rays for a particular particle might be spread out between blocks of threads.

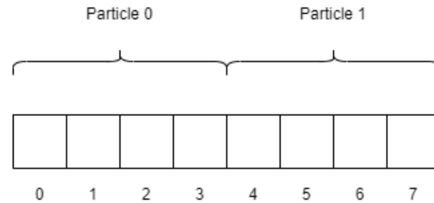

Figure 8: New workload distribution per CUDA thread. Each thread works on a different ray angle. In this example, there are 4 rays per particle.

3) The sample step used a thread per particle. For this we used thrusts max operation to find the index of the particle with the max weight. Then we performed a scan operation in device memory. This was then passed to a kernel for resampling. This whole operation involved two device to host memory operations. With 2 10000-lengthed arrays of ints, we were clearly not limited by memory-bounded since the Nvidia RTX 2080 could support gigabits per second.

4 Results

4.1 Evaluation Metric

Maze-Generation We measured the maze generation through time to generate the grid. Since our implementation deals with writing to the output file, processor by processor, we take the longest processor time for our measurements. This will be the last processor with the bottom-most grid block.

Particle-Filtering We measured the evaluation metric through the milliseconds in *transition*, *reweight*, and *resampling*. This consists of the *animation* time. We also have the *clear* time to clear the environment grid and also the rendering time in openGL, but these were not considered because our project focuses on particle filteirng which is part of the animation time.

4.2 Sizes of Inputs

Maze-Generation We tested the grid out with different grid sizes, first sequentially and then in parallel with different processors. We took a rather small grid (100x100), and then a much larger grid (1000x1000). For the remaining experiments, we increased the size of the grid by four-fold to demonstrate our execution time and speedup.

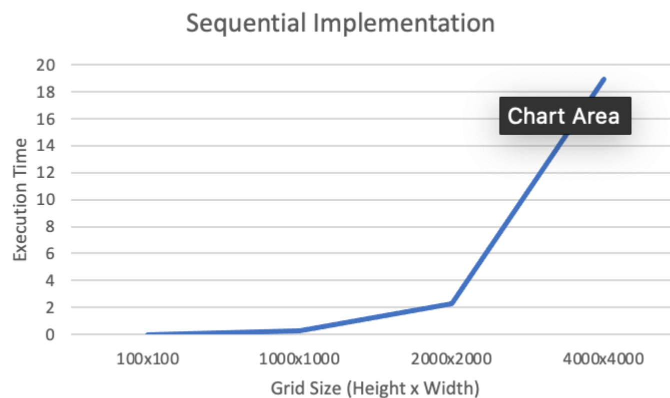

Figure 9: Sequential running with different grid sizes in seconds

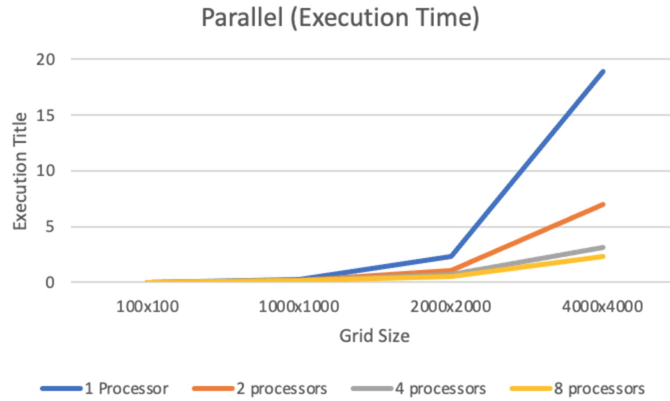

Figure 10: Parallel running with different grid sizes

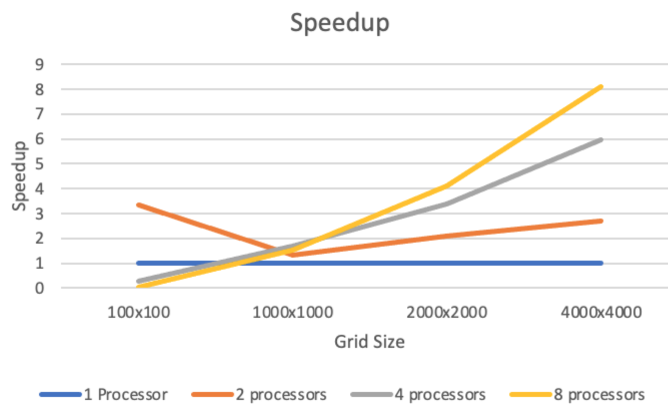

Figure 11: Parallel Speedup running with different grid sizes

We notice that the speedup increases as the number of processors increase. The miss rate of the wall-list remains around 47 percent, but it's now divided between the size of the local grid of each processor. We also notice a non-linear increase in the speedup graph, as we increase the grid size. This means, that we have seen significant performance improvements in the cache block and removing items from the wall-list (which is an O(of total walls)).

Moreover, there is overhead that is associated with created each thread in OpenMPI, as we can see with the results of the grid of size 100x100. The speedup is less than sequential version for the 4 and 8 processors. This graph demonstrates that the communication-to-computation ratio is not favorable for grid of such small sizes. However, when we increase to 1000x1000 grids and above, we see a much better performance result.

Particle-Filtering We tested using 3 different types of grids, easy, mediunm, and hard. As expected, as the grid sizes get larger, starting from 200x200, 420x420, and 1060x420. We also tested with different amounts of particles. We tested with 100, 1000, and 10000 particles. Finally we tested with different amounts of rays. We tested with 4, 16, and 64 rays.

Table 1: Sequential time for different size grids and particles

num particles	100	1000	10000
easy.txt	6ms	30ms	320ms
medium.txt	14ms	130ms	930ms
hard.txt	25ms	220ms	2120ms

Table 2: openMP time for different size grids and particles

num particles	100	1000	10000
easy.txt	4ms	25ms	63ms
medium.txt	9ms	20ms	142ms
hard.txt	10ms	35ms	370ms

Table 3: CUDA time for different size grids and particles

num particles	100	1000	10000
easy.txt	1ms	2ms	9ms
medium.txt	2ms	5ms	30ms
hard.txt	5ms	10ms	50ms

Table 4: Sequential time for different rays with particles=1000 and hard grid

num rays	4	16	64
hard.txt	56ms	215ms	844ms

Table 5: openMP time for different rays with particles=1000 and hard grid

num rays	4	16	64
hard.txt	50ms	140ms	220ms

Table 6: Sequential time for different rays with particles=1000 and hard grid

num rays	4	16	64
hard.txt	5ms	10ms	30ms

Here are some things we noticed. First, as the number of particles increases the time increases as expected. This is the same relation for the number of rays.

NOTE: this is not shown, but when computing the workload distribution per thread using a timer, we found that all threads in openMP finished within a millisecond of each other. This suggests that giving a *static* assignment of particles per thread was a good choice. Moreover, we tried dynamic assignment and got simliar results. If these had very uneven workload times or more overhead in assigning work, we were going to implement a semi-static amount of threads, where the chunksize would be determined empirically and exponentially decay the granularity, similar to knapsack problem for distributed work queues.

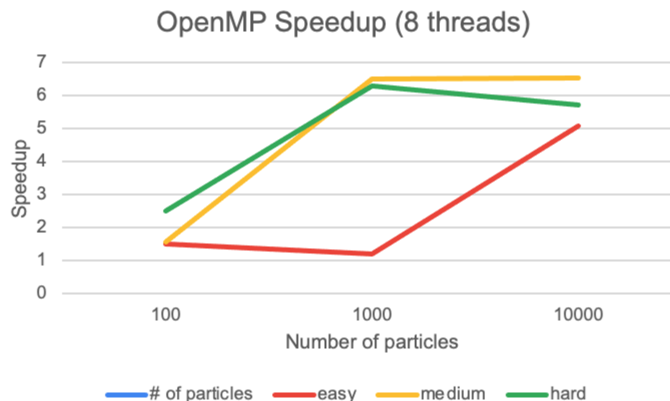

Figure 12: openMP speedup with 8 threads based on the number of particles

We noticed with 8 threads that we achieve speedup of about 7x which is pretty descent. On GHC machines, if we launched more threads, some threads would be swapped out of context and for other threads since there are only 8 cores, so we didn't test for more than 8 threads.

For the easy graph(in red), you'll notice that we don't get good speedup with 100 particles. This could be due to the fact that with easy the grid size is only 200x200 pixels, so the overhead in launching threads could account for the easy maze.

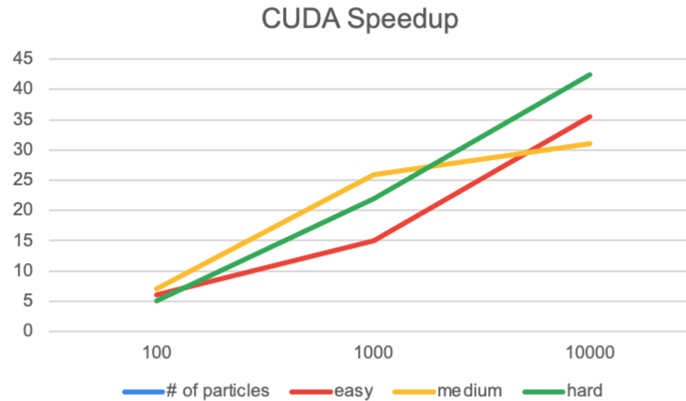

Figure 13: cuda speedup based on the number of particles

CUDA achieved the best speedup. For the hard grid with 10000 particles, we achieved a speedup 40x faster than the sequential version on a single-threaded CPU. The workload distribution for a single warp working on 2 particles (1 thread per ray) worked well.

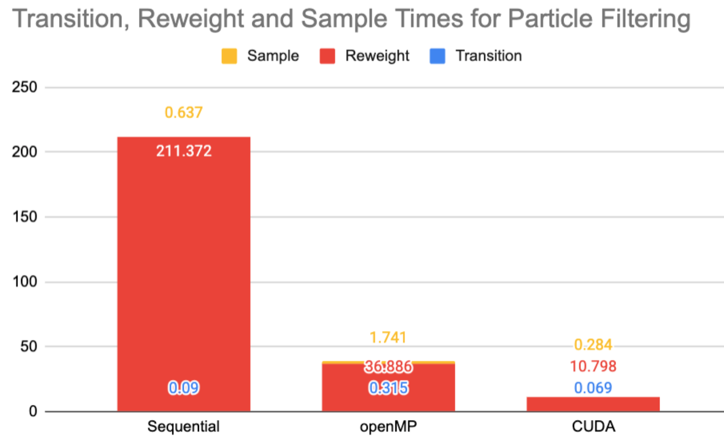

Figure 14: Breakdown of execution time in Particle Filtering (times are in ms) for the hard grid with 1000 particles

You'll notice most of the time is dominated by the reweight portion where we can calcuate 16-ray intersections per particle. The transition and sample time are menial compared to that. In general the CUDA implementation was faster than the openMP and sequential. However, openMP actually has a higher time for both sample and transition. This is most definitely due to the overhead of forking and joining threads. However, the effect is barely noticeable because the reweight makes the openMP version look way faster.

4.3 Deeper Analysis

perf-stat For the particle filtering, We used the handy perf-stat tool to analyze the cache misses. For the grid-generation in for openMP and sequential versions, we had cache-miss rate of about

13%. Tracking this down we that most of the cache misses were due to the particle size scaling. For instance: we assumed in our analysis that particles are 1 pixel, but for the human eye we scaled them up to about 4 pixels. Therefore, during the *rendering* portion of the simulation, we experienced cache-misses through jump between rows. However, we did not deal with this issue as this project focused on maze generation and particle filtering

Furthermore, we wanted to find the "hottest" instruction (most used lines of code) and reduce their runtime. We found that the to_grid function in util.cpp was most used. This function converted a 1d position to a 2d position, which was the reason why it was most used. Unfortunately this is unavoidable execution time so this couldn't be optimized. There was also a lot of time allocating arrays. When generating new particle locations and orientations, we had to create 2 new arrays for that case to avoid alias. This was also something that couldn't be avoided.

Maze-Generation We tried to instrument the code and measure how long each individual task takes, e.g. instrument and measure every single time an item is evicted. However, each iteration of the code is too fast and time for a single iteration of code is not captured. We can only measure the entire time when the entire piece of code is finished. Similarly, the communication time was also witnessed as being 0, even after instrumentation. However, we can measure the total times we find the same wall in our wall list.

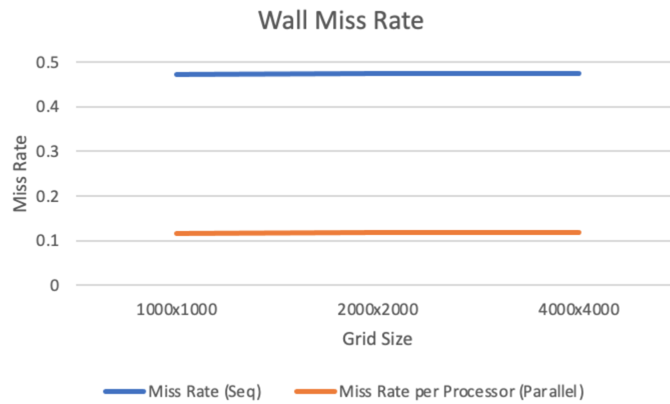

Figure 13: Number of walls which were we had previously visited.

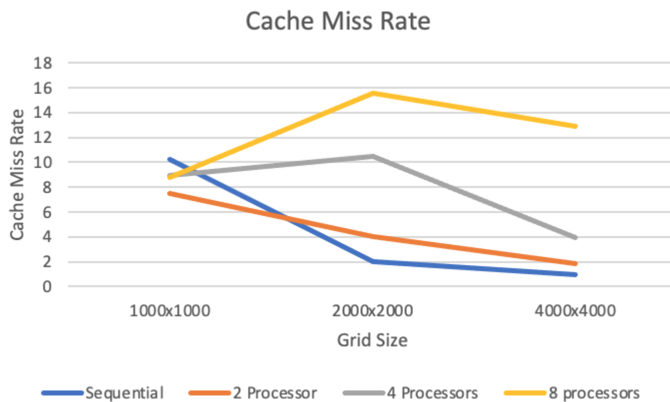

Figure 14: Cache miss rate with increasing number of processors.

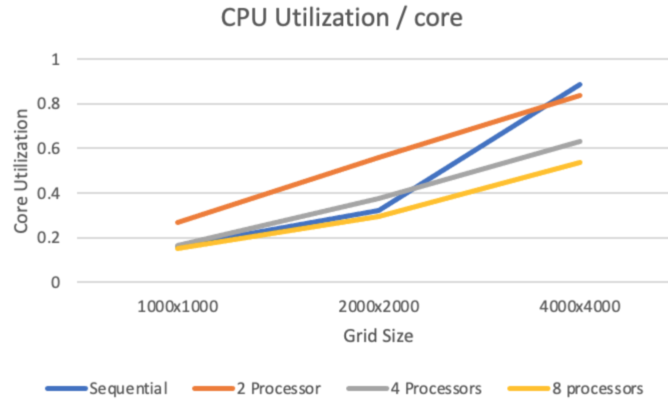

Figure 14: CPU Utilization with increasing number of processors.

We notice that the cache miss rate does not decrease as the number of processors increase. Hence, our assumption about the cache miss rate decreasing was wrong. In fact, the cache rate seems to jump up as we increase the number of processors. This suggests that the parallel implementation has worse cache references. This is very interesting, especially for 4000x4000 grid, since only 4000 Integers can be stored on an L1 cache. We would expect the DFS to perform badly, but because they take grid positions will potential and store them in local memory, they are not evicted and we can continue picking these location from L1 cache. The likely reason is that each processors is fetching from a shared cache (like L3) and that causes multiple evictions. In each iteration, we check the grid to see if a particular location is a wall or not. This might be getting evicted from the shared L3 cache. We tried to do further analysis, but GHC clusters do not provide permissions to access data about specific L1 caches through perf-stat.

On the other hand, we notice that CPU utilization increase sequentially as the size of the grid increases. This suggests that each processor does a lot of operations as the grid size increases, and almost reaches 90 percent by 4000x4000 grid. This seems to imply that our program is largely computation-bound as the size of the grid increases. Since the most computationally intensive task is adding and removing from the wall-list, our assumption holds true, and we can say as that the size of the wall-list is our bottleneck with larger grids.

5 Some Final Thoughts

Maze-Generation Our assumptions on the unexpected cache miss rate depend largely on the multiple processors evicting from the L3 cache. However, due to perf-stat not having permissions, we were not able to measure it. It would have given an interesting insight into using other shared caches even in MPI. We also came to the epitome of a hybrid approach, where we thought about using MPI for communication and openMP to reduce cache-misses in L3 caches.

Particle Filtering We actually tried to calculate a line intersection using SIMD vector instructions. However, we realized that the avx instructions support 8-wide on the GHC machines, and that calculating a line segment intersection involved multiple gather and load instructions. See the 4th reference below for particle filtering in which we used the formula to calculate the line segment. Line segment intersection using vector SIMD was impractical.

We also figured out that just because something is parallelizable it shouldn't be parallelized. Moving the robot on the CPU was better than moving the robot on the GPU. We learned to structure our code to reduce memory transfers from CPU to GPU and vice-versa. The choice to place data structures in constant device or shared memory was carefully chosen. Overall, this project gave us great insight in parallel programming and the architecture.

6 Workload Section

Justin: 55%, main focus on particle filtering and website building

Utkarsh: 45%, main focus on maze generation

References

Maze-Generation

1. https://en.wikipedia.org/wiki/Maze_generation_algorithm
2. https://ieeexplore.ieee.org/document/6773228
3. http://ipsitransactions.org/journals/papers/tir/2019jan/p5.pdf

Particle-Filtering

1. https://en.wikipedia.org/wiki/Particle_filter
2. https://www.youtube.com/watch?v=NrzmH_yerBU&ab_channel=MATLAB
3. https://www.youtube.com/watch?v=uYIjB93oAUo&t=4128s&ab_channel=CyrillStachniss
4. https://en.wikipedia.org/wiki/Line%E2%80%93line_intersection