# Parallel Maze Solver

*Parallel Computing Final Project*

*Omar Khater and Shaamyl Anwar*

*Github Link*

## Introduction

Our project introduces a parallelized algorithm to solve a Maze. A Maze, in our definition, consists of a grid containing a start node, a goal node, and walls (inaccessible locations) as obstacles. The goal is to find a path from the start to the goal.

## Algorithm

The random maze solver focuses on having threads explore the maze randomly to find a specific goal. As the threads move they keep track of their path in a 3D array. The array has 3 dimensions: the thread ID, the x position, the y position; for example, arr[x][y][z] would contain where thread number x came to position [y][z] from. Since CUDA does not support multi-dimensional indexing, the array had to be flattened. The flattened array would have num_threads * num_blocks * num_rows * num_cols elements. A simple representation can be seen in figure #1.

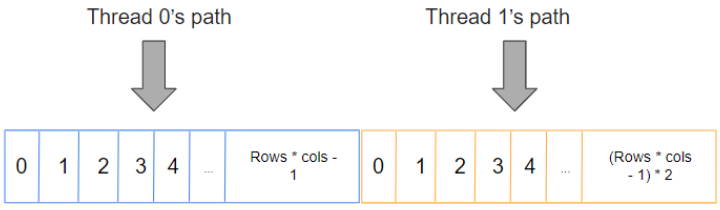



Figure #1, flattened path array repersentation

Given an index I, to find which thread it belonged to you would divide I by num_rows * num_cols. The x_pos would be equal to : I - rows *cols * thread_ID / cols. The y_pos would be

equal to: I - rows * cols * thread_ID % cols. The actual index in the path array that needed to be filled out was given by: path_idx = pos_in_the_maze + rows * cols * thread_ID.

The way the threads move was also very important. Threads had to only move to valid locations in the flattened 2D maze. Meaning we had to make sure of three things: firstly, that they did not go out of bounds, secondly that the move was valid (For example: if you were at the end of a row, you could not add one to that position since that means you are zipping over to the start of the next row, an invalid move), and finally that the position the thread was moving to was not a wall.

When the threads would get eliminated was a big question. Our first iteration of the algorithm had a condition that once the threads got stuck then they would get immediately eliminated. While it did not make a huge difference with small and medium sized mazes, the algorithm failed to solve big and dense mazes since the threads would get eliminated way too quickly. To solve this, we added a simple backtracking trick that utilizes the path array. If a thread gets stuck at position x, instead of returning right away, it would go to path[curr_pos + (rows * cols *tid)] which represents the position that preceded x and would try and explore from there. The threads would be essentially going back and trying to explore other paths. If a thread backtracks all the way back to its start position, then the maze is unsolvable and it is then eliminated.

**CUDA Architecture:**

There are three CUDA kernels present in our code. Init_rand_kernel initializes the random state for each thread. Random_solve_path_kernel solves the maze and outputs the path and random_solve_kernel solves the maze but without outputting the path. Both use the same algorithm with one core difference: random_solve_path keeps track of where the threads moved.

Random_solve, on the other hand, is not limited by trying to keep track of the path which means the threads run infinitely until they find the goal. If the goal is unreachable then they will run in circles forever trying to find the goal.

**Interesting Challenges:**

One of the most interesting challenges was coming up with a way to keep track of a thread's path without the explicit use of a multidimensional array. Indexing through the flattened array and trying to keep communication between threads minimal was very challenging. Another big design hurdle was the question of how the threads should behave if they get stuck. The solution we came up with was that if a thread reaches a position but can't legally move anywhere else, it would backtrack to its previous position using its path array. It would keep backtracking until a legal move is located. If it reaches its start position and there are no legal moves then the maze is unsolvable since the thread has explored every possible value Another big challenge was making sure that each thread only moved using the available legal moves. Again indexing in a flattened two-dimensional array proved to be very difficult.

## Visualizer

In order to evaluate and communicate our solutions, we created a custom Visualizer written in Java. The Visualizer uses the Java awt.Graphics library along with OOP design to efficiently and compactly visualize paths taken in parallel by many threads.

**Drawing The Maze:**

We design a Visualizer Class which holds a Maze 2-D Array of MazeNode objects. Each MazeNode object is self-sufficient in terms of containing the information needed to draw the node (square) on the screen. The graphics library's paint method cycles through the 2-D array every-time a MazeNode is changed and repaints the whole array. Note that a more sophisticated algorithm can be implemented to cycle through only the parts of the array that are changed but this is not worth it for our current project scope.

**Simulating Thread Movement:**

In the simulateThreads method, a paths file is converted to an ArrayList of paths where each item in the ArrayList is an array of a single threads path. A single thread's path is of the row-major form and each item in the path is the location of that thread at the index time-step. The ArrayList is iterated through and each thread's position at the current time-step is painted red simultaneously. Further, a node is painted darker if there are more threads visiting it and lighter (transparent) if it has fewer threads visiting it.

**Interesting Challenges:**

One interesting challenge was coming up with the formula to paint the opaqueness of a node depending on the number of threads visiting it. The formula looks like the following:

```
java.awt.Color(255, 200 - (int)Math.round((double) maze[i][j].totalThreads/totalThreads * 200), 200 - (int)Math.round((double) maze[i][j].totalThreads/totalThreads * 200));
```

The 255 signifies that the base color is red. The fact that we are subtracting the other two color channels from 200 (and not 255) signifies that with increasing number of threads, opaqueness decreases, but doesn't decrease to full transparency (hence 200, not 255).

Another interesting challenge was figuring out a file format to store and read each thread's path through the maze. We settled on a format where a text file stored on each line the row-major path of each thread. The implementation advantage we gained was that the paths file could simulate thread movement by having a global time-step index for all the threads that incremented chronologically.

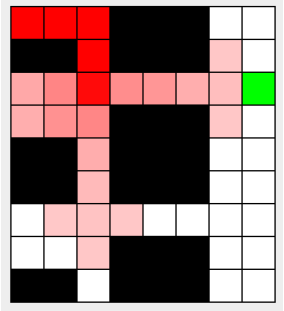
## Evaluation

Note that we had an older version of our algorithm whose results are also shown for comparison. The older algorithm failed because once the threads got stuck, they were eliminated, and for large mazes with a lot of walls, getting stuck was pretty easy. For our final algorithm, once a thread gets stuck, it backtracks to the previous position hoping to explore a different path. So if a thread gets stuck at position x it would go into the path array at path[curr_pos + (rows * cols *tid)] (this represents the position that preceded x) and set that as the current position and it would try to go from there. If the thread keeps getting stuck and backtracks all the way back to its start position then it has explored all possible avenues and then it is eliminated.

*All run with 16 * 1024 threads*

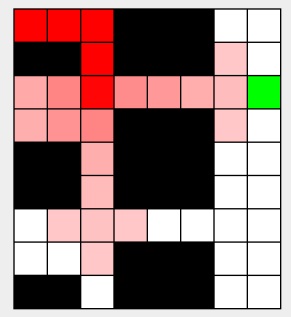
**Maze 1:**

**Random Parallel (Old):**



Our algorithm solved a small maze in parallel - in line with expectations. There are some unfruitful paths but there are enough threads to find the goal.

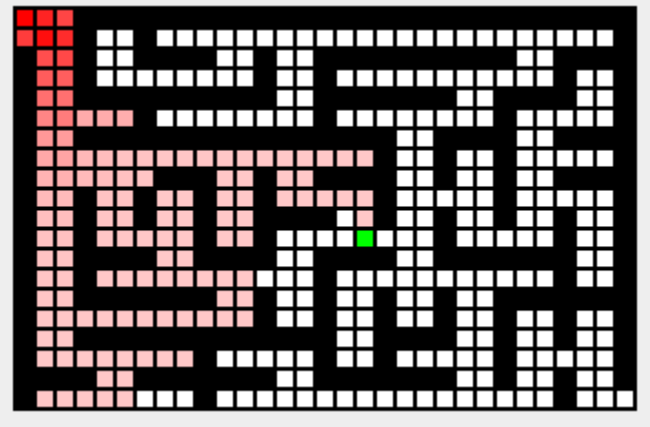
**Random Parallel With Backtracking:**



For such a small maze, adding backtracking makes no difference.

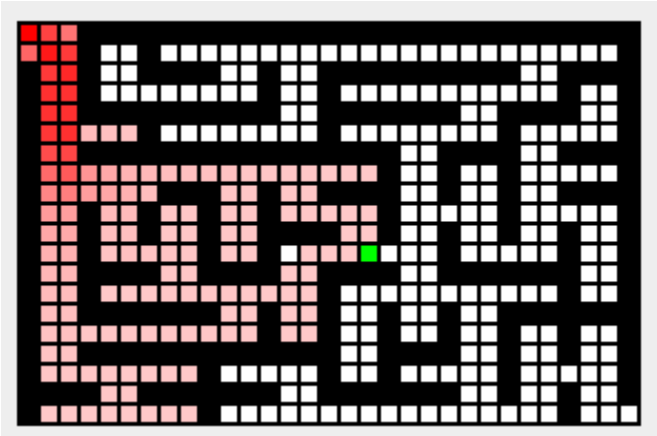
**Maze 2:**

**Random Parallel (Old):**



Again, our solved performs as expected, solving the maze with threads exploring many parallel paths simultaneously.

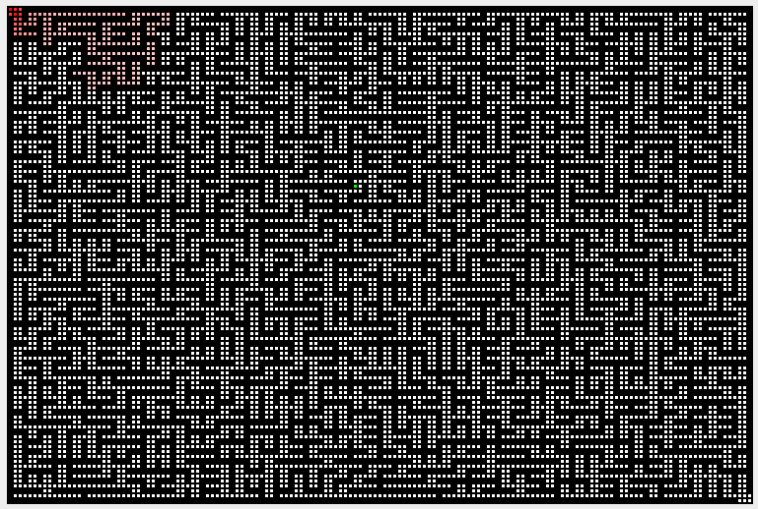
**Random Parallel With Backtracking:**



Adding backtracking doesn't make any major difference. The only difference is that some more nodes are explored in the bottom left of the maze since more threads stay alive and backtrack (and are not eliminated instantaneously).

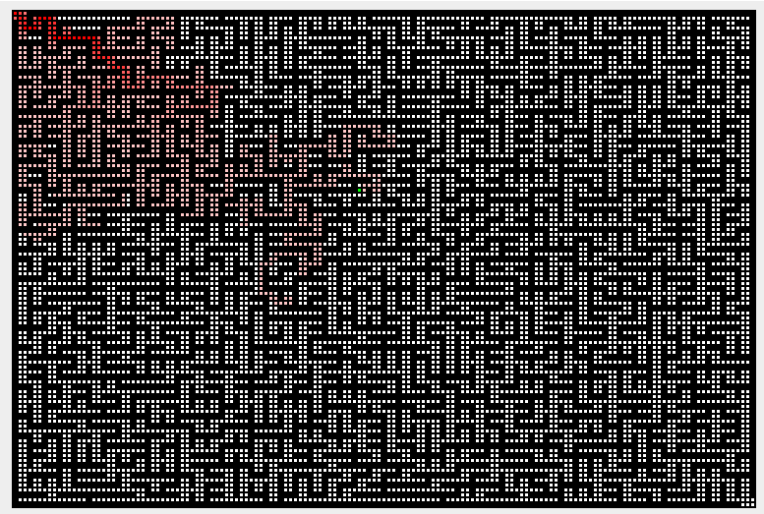
**Maze 3:**

**Random Parallel (Old):**



In a large maze with many walls, the algorithm failed to solve the maze because threads got stuck very early and were subsequently eliminated (when cornered/blocked) before they reached the goal.

**Random Parallel With Backtracking:**



However, when the condition that threads are immediately eliminated if they get stuck is removed from our algorithm, the maze is solved as shown above. So, in such mazes, removing instant thread elimination and adding backtracking makes a large difference.

## Future work

The random_path_solver has one big limitation: it is randomly walking, and thus it is slower than a serial DFS implementation. It is, however, faster than a serial random solver implementation. One interesting avenue of future research would be to try to implement a parallel DFS or BFS solver; it would almost definitely be faster than the serial DFS. Implementing such a structure would come with its own challenges. Figuring out how to partition the search tree while minimizing communication without threads would be one of the biggest challenges.