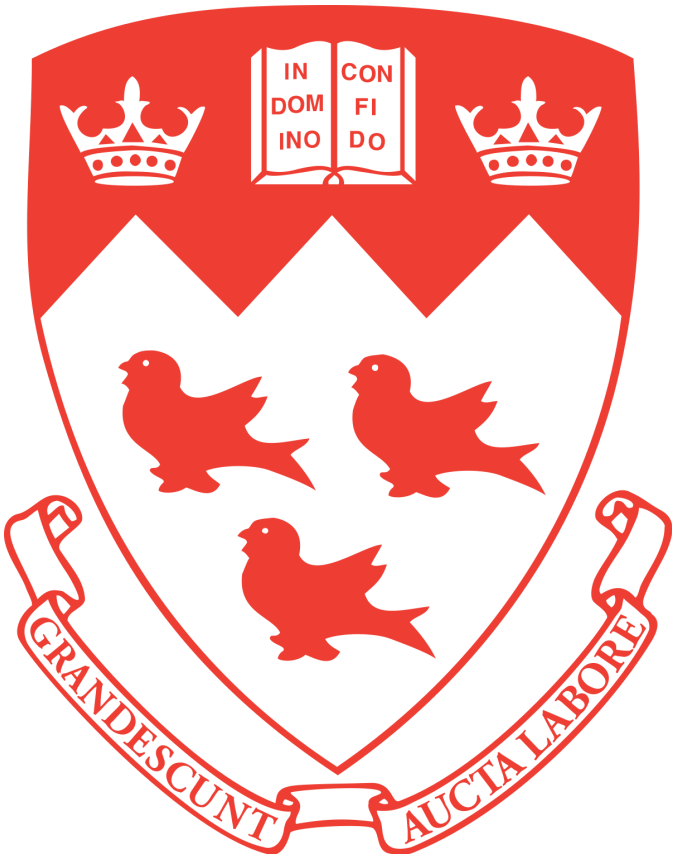


ECSE 420: Parallel Computing
Project 23
Fall 2019

Project Report
Parallel Maze Solver

Stefano Commodari 260742659
Anudruth Manjunath 260710646
Pierre Robert-Michon 260712449
Romain Couperier 260724748

Setup Instructions

How to Run the Sequential Maze Generator and Solver

This sequential maze generation and solving program is an adaptation of the code found on https://rosettacode.org/wiki/Maze_solving. It is a C++ program created in Microsoft Visual Studio and does not utilize the GPU. Our adaptation of the code can be found at https://github.com/anudruth/SequentialMaze.

To run the program, upon program startup, simply enter the required size of the maze. The program will output the total runtime as well as two mazes stored in BMP format in a folder named Maze. The first maze, named maze<size>.bmp, is unsolved while the second one, named maze<size>_s.bmp, contains the solution generated by the sequential code. Please note that the images generated by this program contains a 1 pixel wide border around the entire maze. Due to this, if 51 is entered as the required maze size, the actual maze area will only be 49x49 pixels.

We also made a helper program to convert the unsolved maze created by the maze generator into a usable maze for our parallel solver. In order to run this program, give it the unsolved maze from the generator and the size of the final maze as arguments. Make sure to open the image outputted by the maze generator and save it as a png or else the program will be unable to open it. Furthermore the size of the final maze should be given as 49 from the example above and not 51. This program will invert the colors to create black walls and white paths and also crop the image to the desired size. This step must be done before using the parallel maze solver. This program can be found at https://github.com/RomCoup/InvertImage.

How to Run the Parallel Maze Solver Program

This project was created as a CUDA project using Microsoft Visual Studio as the IDE. The source code can be found in its GitHub repository page which can be found here: https://github.com/pierrerm/ParallelMazeSolver.

In order to run this project you will need to input an image path as an argument. All of the mazes we tested were placed in the Maze folder, and all mazes were named mazeSize_inv.png. From this, the program will output the maze solution to a file with the name mazeSize_p.png. In order to make sure this project runs you will need to change the size of the maze in the kernel.cu. Manually input the rows and columns of the maze into the variables M and N respectively on lines 11 and 12. If these variables do not match the size of the image then the program will give you an error.

If you want to run the program with the optimal number of threads then do not input a second argument. Otherwise, input a number indicating the total number of threads to use.

Project Description

Problem/Motivation

The purpose of this project is to navigate through a maze with the help of parallel computing. The program starts at the top left corner of a given maze and attempts to find a path to the bottom right corner. The simplest method for finding this path is using recursion to backtrack from the end to the start, similar to depth first search. However, recursion consumes a lot of time and space when performed.

For example, when going from position (0,0) to (r,c), where r is the number of rows in the maze and c is the number of columns, the brute force approach would be to find every possible path to its sub problem, ie. to (r-1, c) or (r-1, c-1) or (r, c-1). We can repeatedly do this until we reach the origin (0,0). This approach has a run time complexity of $O(2^{r+c})$ or $O(3^{r+c})$ based on the exact specification of moves the maze solver is allowed to make.

One simple optimization is to use memoization to ensure that we check each grid in the maze only once. This would help us reduce our runtime complexity to O(rc). Even with this optimization, we would have multiple recursive calls and we would have to wait for the previous branches of the recursive tree to return from its deepest node before we can try a different path. Therefore, a parallel approach is needed to explore multiple possible paths at the same time.

Solution Structure

Initial Approach

The problem which we are trying to parallelize - finding a path through a maze - has a major constraint which must be taken into account: the solution cannot be fully parallelized and must still be partially treated sequentially. Indeed, for traversing the maze and building the final path, the reachability of each point (or pixel) along with the order in which it has been discovered, both have to be accounted for and maintained for the entirety of the program execution for it to be valid. We cannot treat this problem in the same way as the image processing problems seen in the labs throughout the semester, where only one call to a parallel function, with one thread for each pixel (or group of pixels), is used.

However we can still partially parallelize the traversal of the maze by using threads to concurrently navigate through the different possible paths. To do so, we initially wanted to traverse the maze as seen in Figure 1, where each colour corresponds to the points being checked in parallel for one iteration. Each thread would check if the points directly above and to the left of it contain any walls. If these points are accessible, they are written to the array of accessible points to be checked at the next iteration. Since a point can be checked multiple

times, we also check if the point is already in the array. This process will repeat until the starting point is reached, where we then backtrack from start to finish to construct the path.

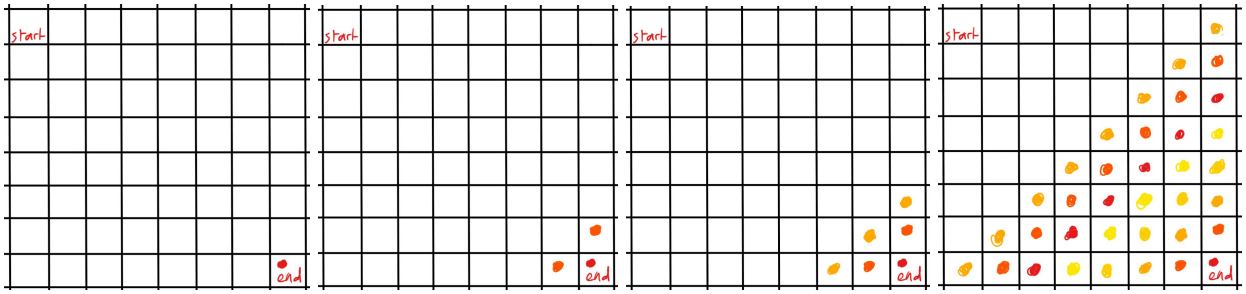


Figure 1 - *Illustration of the iterative, diagonal traversal of the maze.*

Using the method described above, the following remarks can be noted:

- The maximum number of points to be checked at each iteration is along the largest diagonal, depicted in the last frame of Figure 1. For an M (rows) by N (columns) maze, the size of the largest diagonal will be the smallest between M and N. Therefore, if we want to maximize speedup by having one thread per point to be checked, then we will need at most Min(M, N) threads.
- The number of iterations required to traverse the maze from end to start is the number of diagonals. For an M (rows) by N (columns) maze, the number of diagonals will be the largest between M and N. Therefore, with this approach, the time complexity can theoretically be brought down to O(Max(M, N)).

Refined Approach

One problem quickly became apparent with the above method: by limiting the accessibility of the current point to only those directly above and to the left of it, there are situations such as the one depicted in Figure 2 where a path exists but isn't being detected by the maze solver. Indeed, in this situation, the point at row 4, column 4 (start is 0,0) would need to be able to check the point directly below it. The situation where a point needs to check the point directly to the right of it can also occur. We therefore need to take all four directions into account, so that each point checks the points directly above, below, to the left and to the right of it. This will ensure that we will always find the path if it exists.

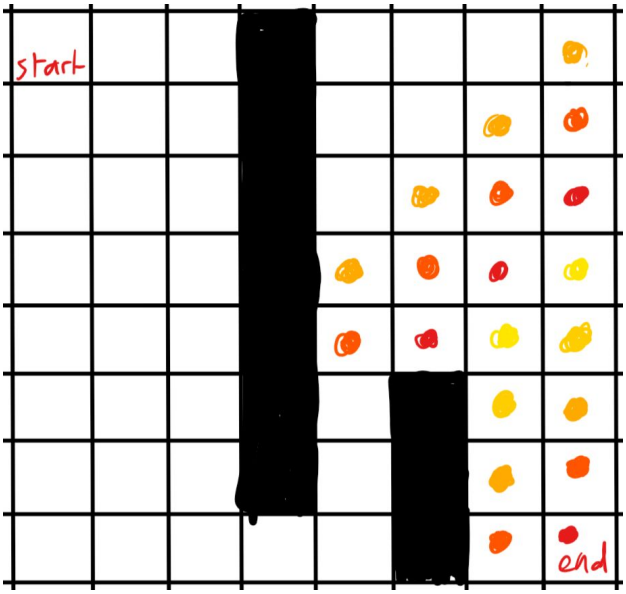


Figure 2 - *Illustration of a problematic maze.*

Using the above refinements, the following remark can be noted:

- A point has to have been accessed from a previous point for it to be valid. Therefore, although it consequently checks 4 directions, it can at most discover 3 new points to be checked, since one of the 4 points surrounding it will be its predecessor.
- Since two extra directions are being checked for each point, the maximum number of threads now becomes 2 * Min(M, N).
- The total number of iterations is now determined by the longest possible path. For an M by N maze, the longest path would be a zigzag path from start to end. In this case, N/2 columns would correspond to walls while the remaining N/2 columns would be paths. Thus the length of this path would be M * (N/2).

Implementation

Our code is comprised of a main method, a device method, two helper device methods and a post-processing host method. The main method loads the maze contained in the input image filename, initializes the required data structures and iterates through the diagonals of the maze, calling the device method with threads for each point to be checked at every iteration. The device method checks all four directions surrounding a given point to see if they are valid, and calls the helper device method to write the valid points to the shared array containing the points to be checked at the next iteration. The helper device methods ensure that the given point isn't already in the shared array to avoid duplicates, and initialize the shared array respectively. The post-processing method backtracks and constructs the path.

We have implemented a Point data structure, which stores the row and column of a given point. This simplifies the storage, retrieval and usage of coordinates for points in the maze that have to be checked. All points that need to be checked at each iteration are stored in a shared array that gets copied to the device memory. Within the device method, we can retrieve the coordinates of the points to be checked and retrieve the color of the corresponding pixel in the maze image, to test whether the given point is a wall or not.

We also used a shared array of Points in the device memory to store the points that are accessible and need to be checked at the next iteration. We need a shared array as opposed to giving each thread its own array to emulate memoization. In other words, we ensure that points are only being written once to prevent duplicates. We therefore need all threads to have access to an up-to-date version of the array containing the points having already been written. However, an issue with this array was discovered through testing. The problem was that multiple threads were writing to the same position in the array at the same time, resulting in certain points being overwritten. As a result, certain paths would no longer be pursued as a point, and its neighbours, would never be checked again.

To fix the above problem, we used an integer to represent the first free index in the shared array. Whenever a thread writes to the shared array, it uses an atomic increment function which blocks the other threads from accessing the index. The index is then incremented so that other threads can write to the next position. This index is also helpful because it ensures that we are always writing to the start of the shared array, without leaving any empty spaces. This allows for

much simpler indexing and memory management as it is impossible to predict how many points will be discovered at each iteration and how many indexes will be needed to access them.

Finally we had to make sure that we were traversing the maze without revisiting points. In order to do this we decided to use the image as our memory, whenever a pixel is visited, we change the color to red. When a thread is looking at the surrounding pixels all it has to do is make sure that the color of the index in the image is not red or black. This serves as an additional means of memoization, ensuring that points are only checked once.

This memory also allowed us to find the path within the maze. In order to do this, at each iteration of the parallel check we give the threads a run number. This increments by one after each iteration and allows the threads to color the next pixels with a shade of red that is 1 less than the current pixel. This then allows us to run some post processing to find the path through the maze. We start in the top left corner and find the pixel that is one shade off its current red color. We continually do this until we reach the bottom right corner. Because there are only 255 shades of red this technique would not allow us to construct a path longer than 255. Thus, the run number resets to 0 when it reaches 255 and the same logic is applied to the post processing. However, we decided to change this logic so that it resets when it reaches 10, so the change in color is imperceivable to the naked eye.

Parallel Performance

Test Results

As seen from the figures below, the green path is the first successful path from pixel (0,0) to pixel (r,c). The red paths indicate the areas that were simultaneously explored by multiple threads while determining the correct path. As seen in the figures below, only a small subset of the paths in a given maze are checked before a solution is found.
The correctness of the paths obtained using the parallel maze solver can be verified by comparing them to the paths in appendix A which were obtained using the sequential maze solver. All times were recorded using the <sys\timeb.h> library.

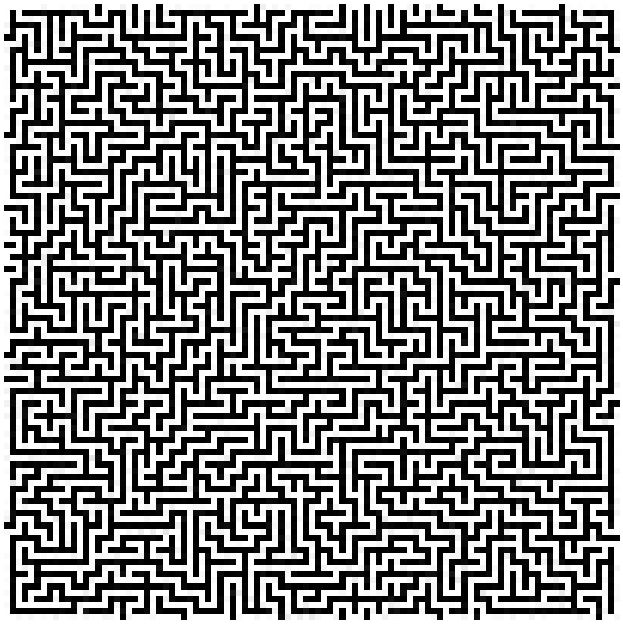


Figure 3 - *101x101 maze*

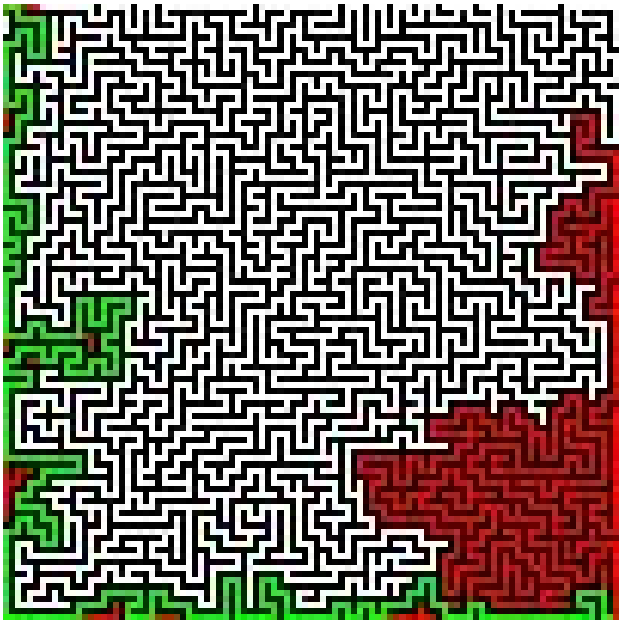


Figure 4 - *101x101 maze solved in parallel*

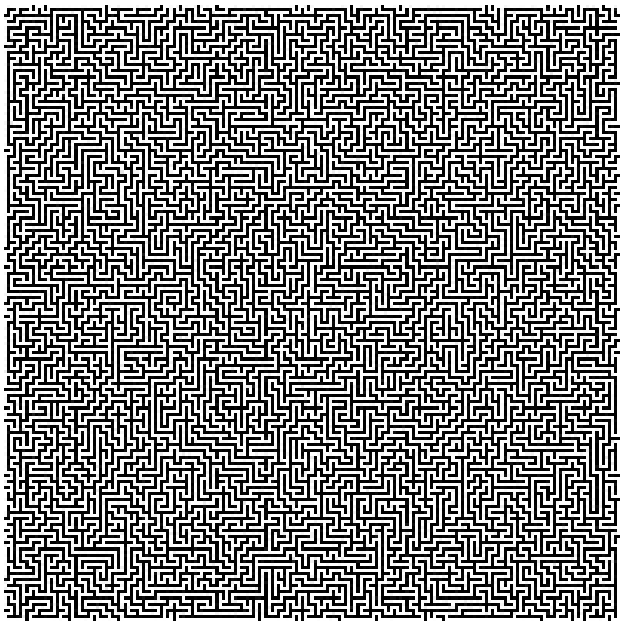


Figure 5 - *201x201 maze*

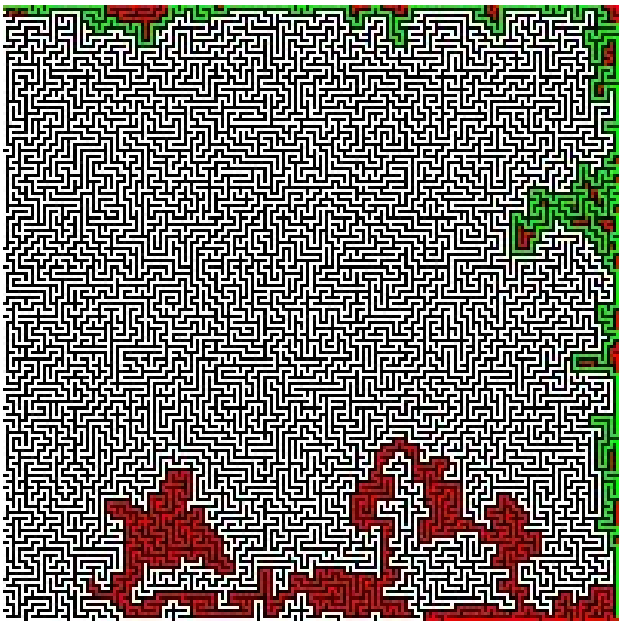


Figure 6 - *201x201 maze solved in parallel*

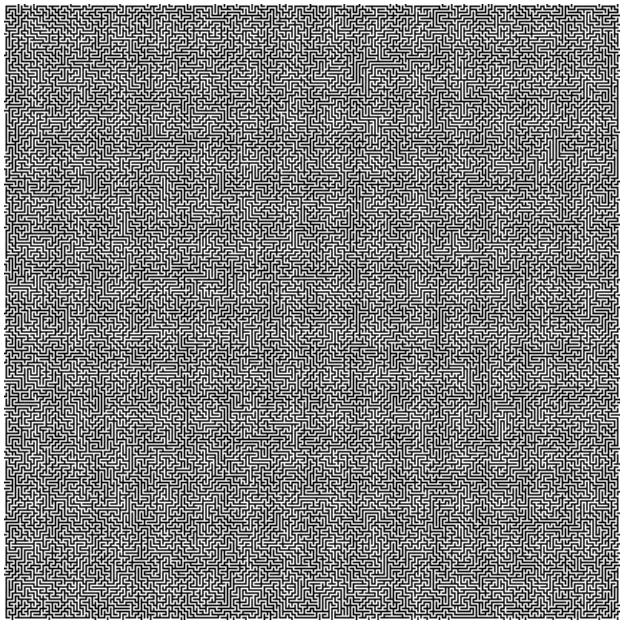


Figure 7 - *401x401 maze*

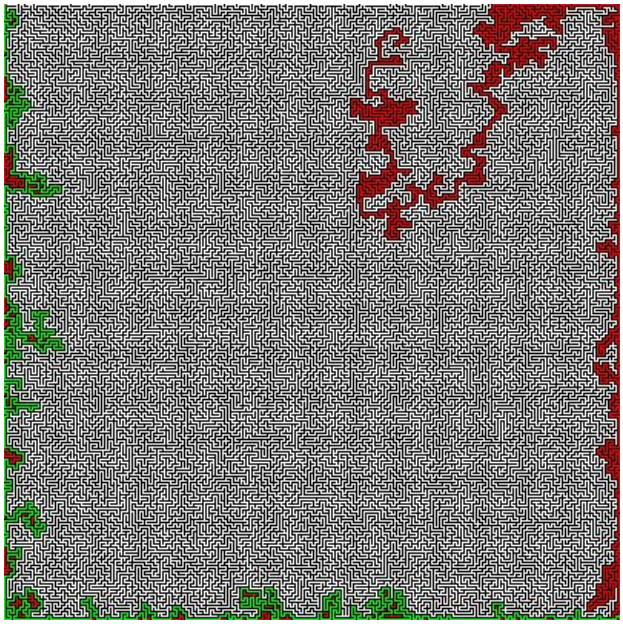


Figure 8 - *401x401 maze solved in parallel*

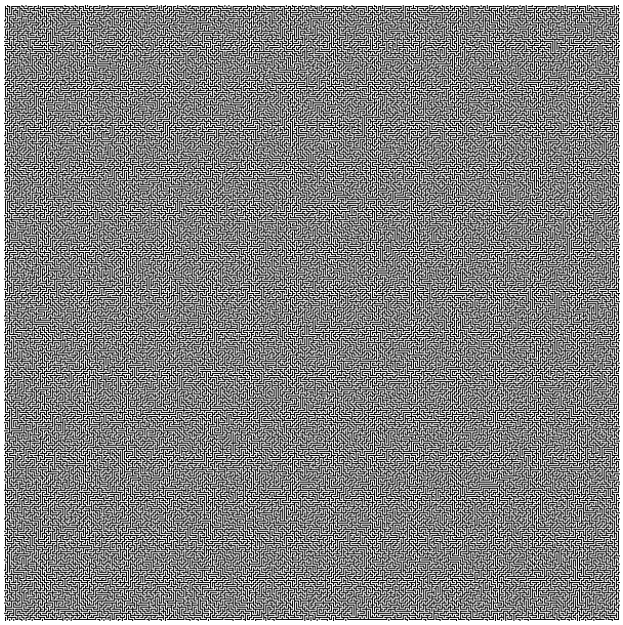


Figure 9 - *601x601 maze*

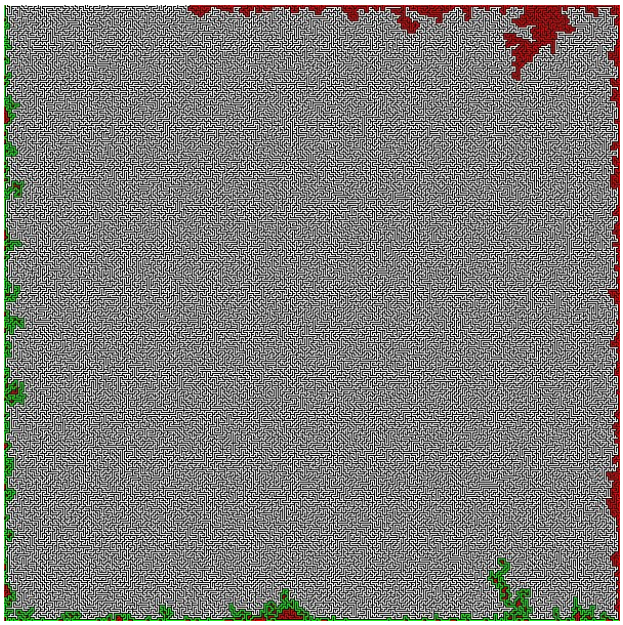


Figure 10 - *601x601 maze solved in parallel*

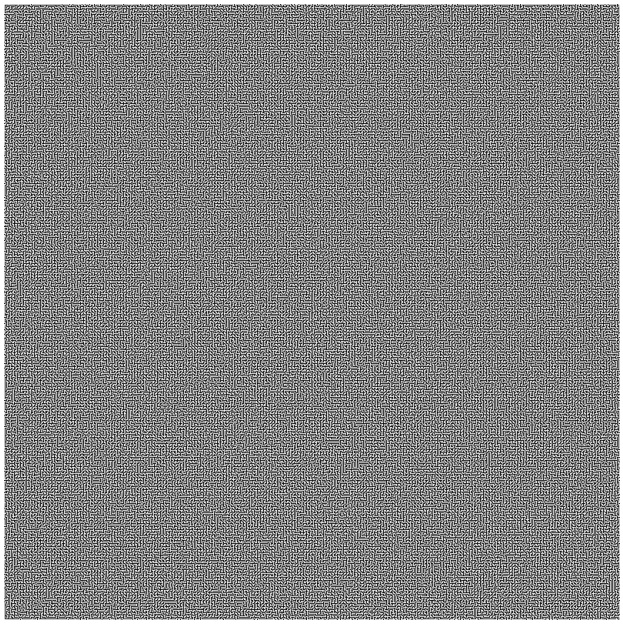


Figure 11 - *801x801 maze*

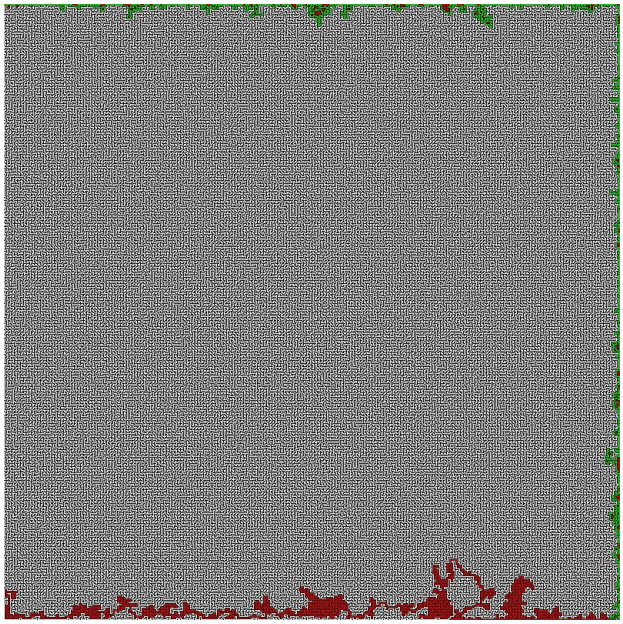


Figure 12 - *801x801 maze solved in parallel*

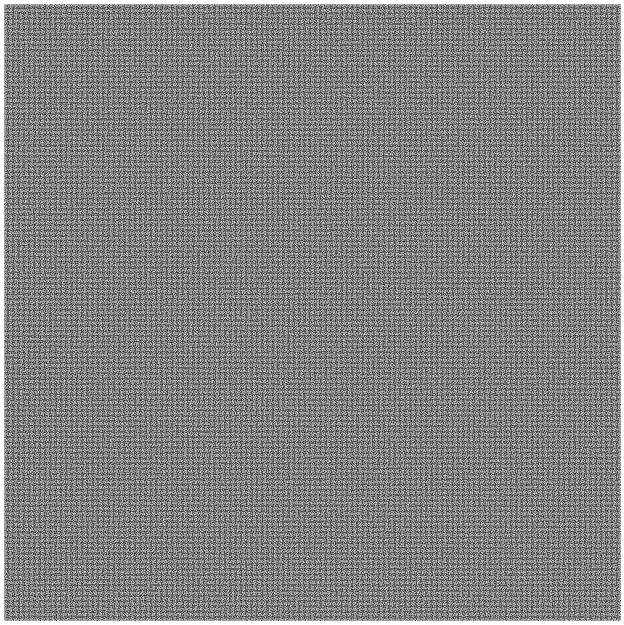


Figure 13 - *1001x1001 maze*

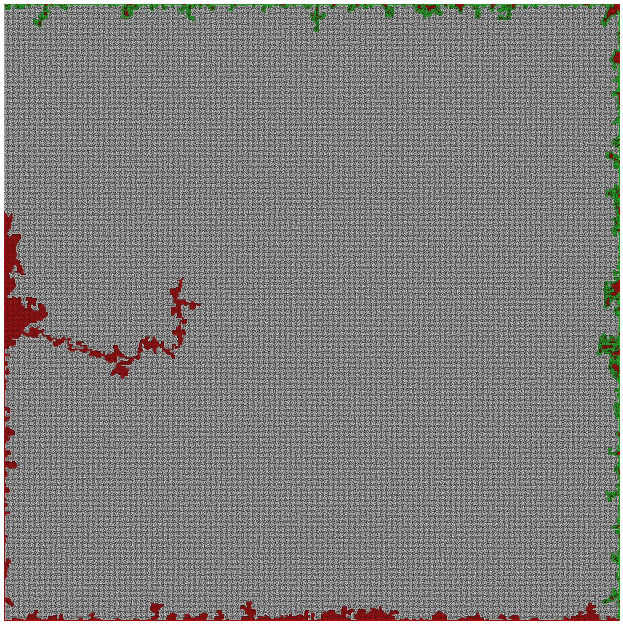
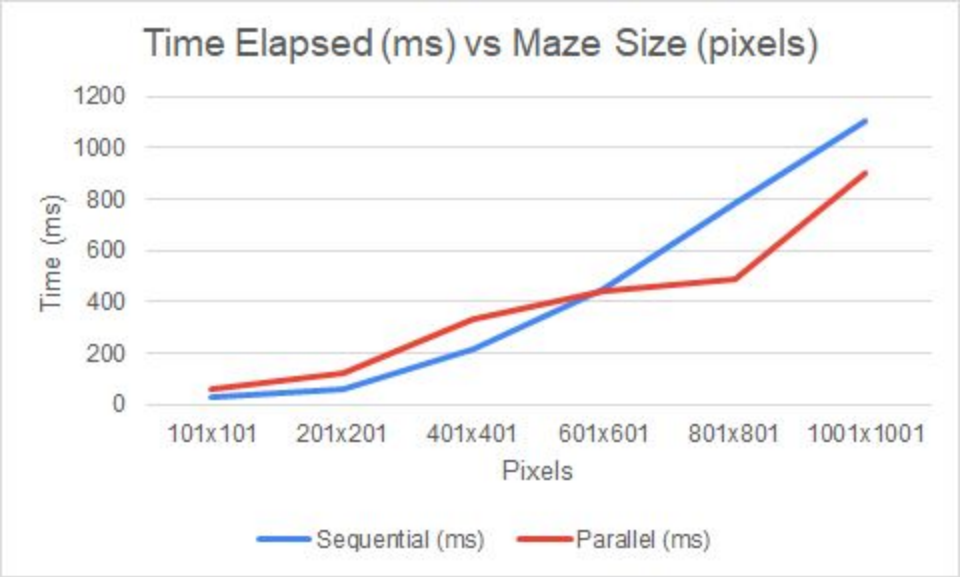
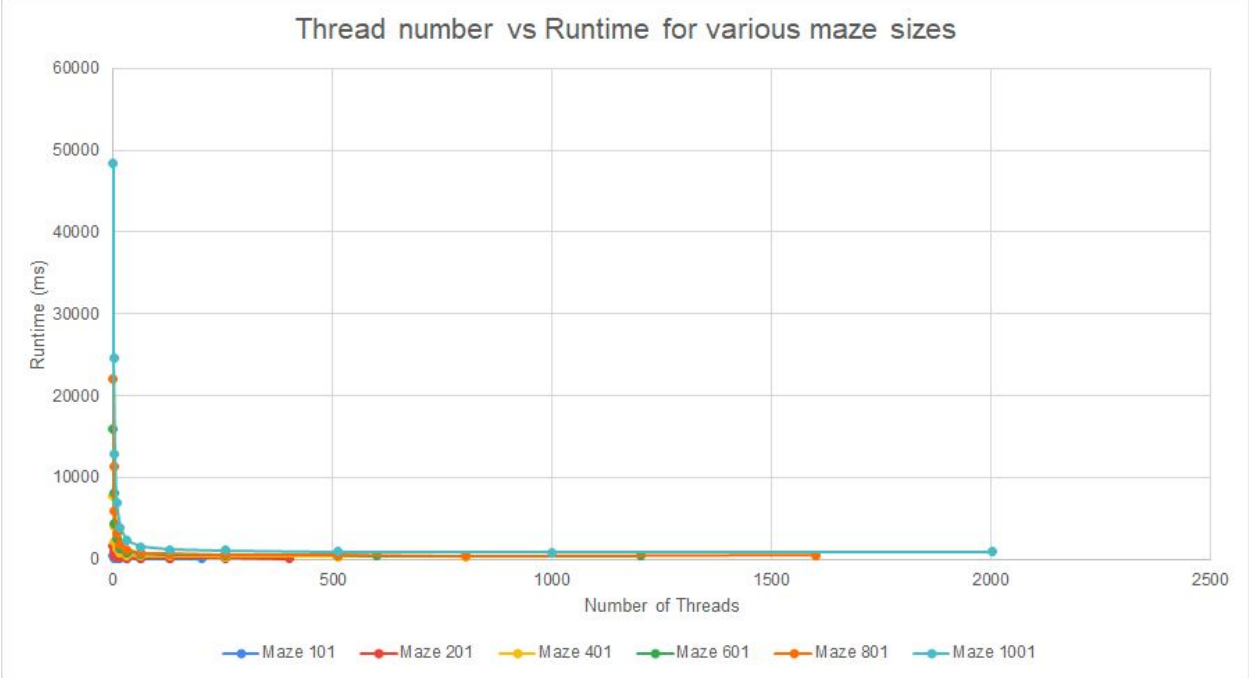


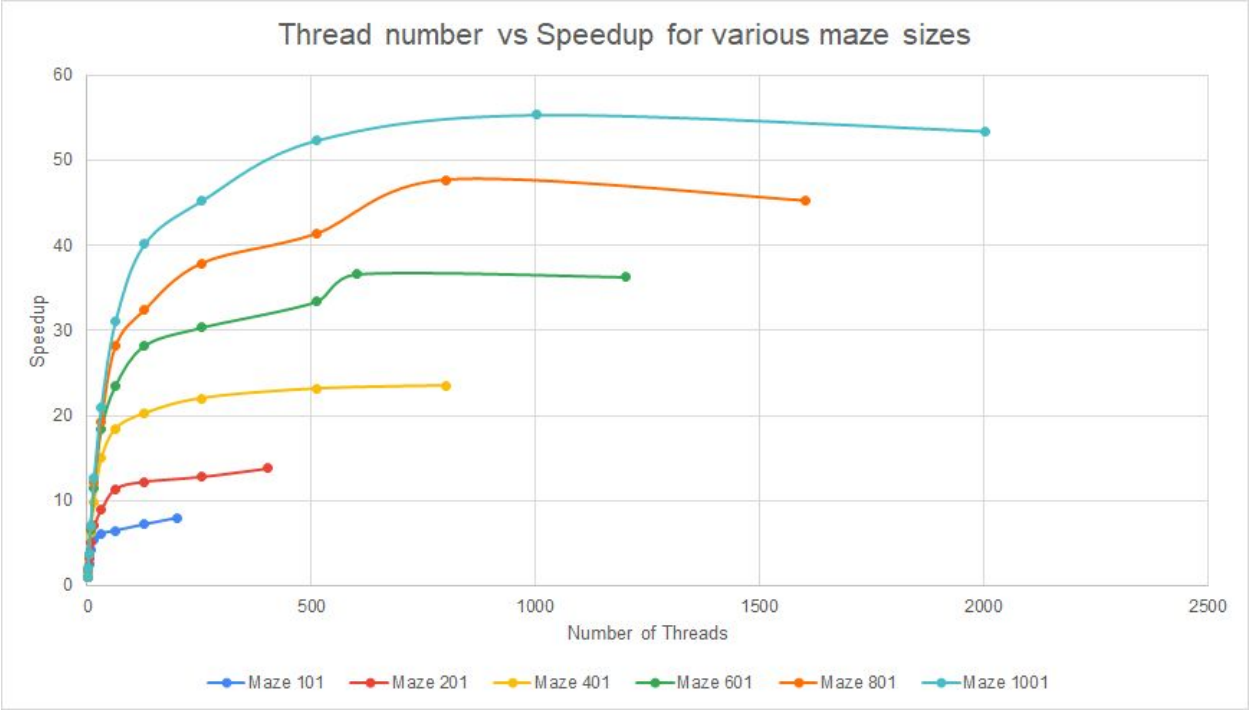
Figure 14 - *1001x1001 maze solved in parallel*



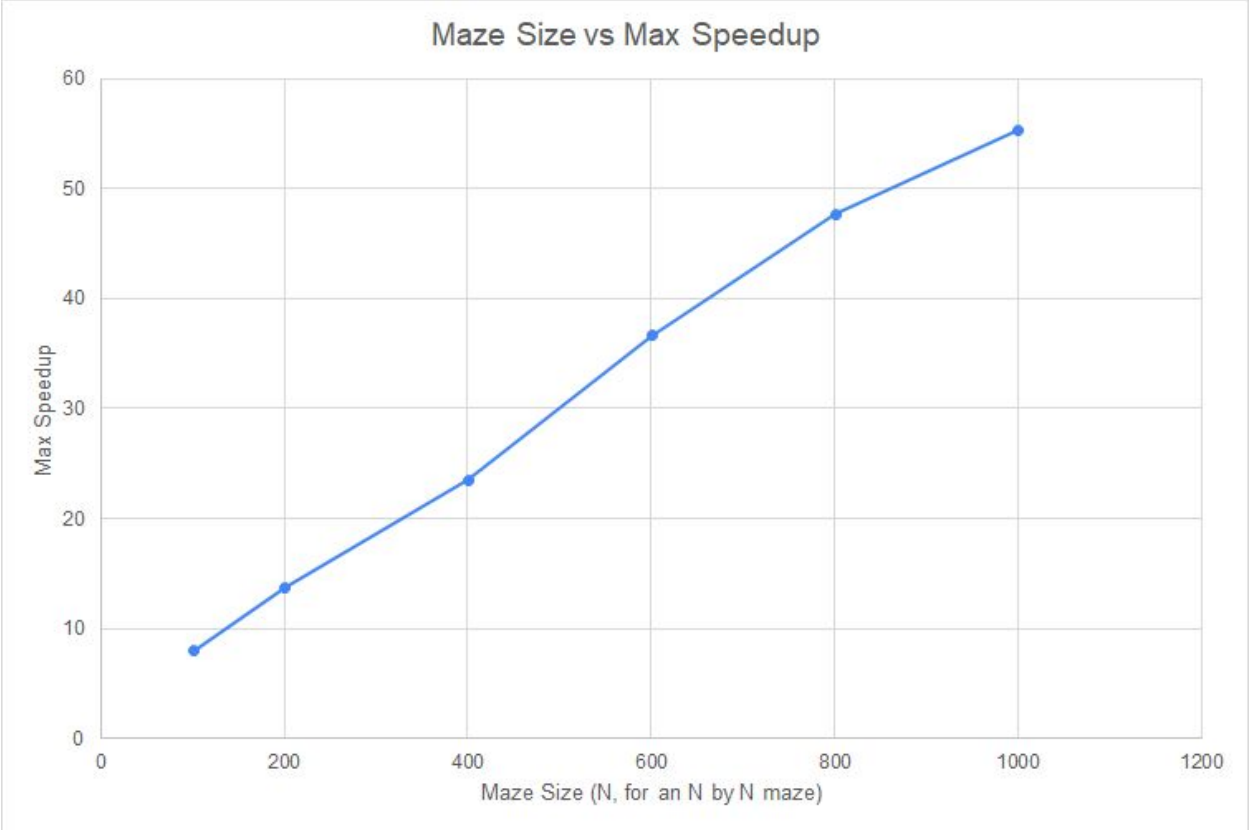
Graph 1 - *Sequential and parallel runtimes for above maze sizes. For these timing measurements maximum possible threads were used, ie. two times the width or height of the maze in pixels.*



Graph 2 - *Thread number vs Runtime (for above maze sizes)*



Graph 3 - *Thread number vs Speedup (for above maze sizes)*



Graph 4 - *Maze size vs Max speedup (for above maze sizes)*

Analysis

As seen in graph 1, for mazes below the size of 601x601 pixels, the sequential maze solver tends to be faster than the parallel maze solver. However, for larger mazes we notice that the runtime of the parallel maze solver is significantly lower than the sequential solver. We attribute this to the fact that there are considerable overheads in the parallel code such as copying the input matrix to the GPU, maintaining a shared array to keep track of the next points to visit, and spawning new threads during execution. Due to these overheads, the benefits of parallelization are only noticeable for larger mazes.

In graph 2, we can see that the runtimes for the parallel maze solver decrease exponentially and consequently in graph 3 we can observe a logarithmic increase in the speedup achieved as the number of threads is increased.

We also note in graph 4, that the maximum achievable speedup increases with the size of the maze. This is due to the fact in our parallel code, the maximum number of threads that can be used to deduce a path through a maze is two times the width or height of the maze in pixels. Therefore, as we use larger mazes we can spawn a larger number of threads to solve the maze. In regards to decomposition techniques, our program does not contain anything elaborate or complex. As mentioned above, if only one argument is passed to the program, then we simply multiply the diagonal of the maze by 2 to get the total number of threads. If the number of threads exceeds 1024, we simply increase the number of blocks by 1 (the default number of blocks is 1).

For maximum possible parallelism, we need to know when maximum speedup is achieved. Fortunately, Graph 3 tells us what the maximum speedup is for each maze size based on the number of threads used. When a maze is less than 601x601 pixels, the maximum speedup occurs when the number of threads is double the maze size. After that point, the maximum speedup only occurs when the number of threads is equal to the maze size.

When the maximum possible parallelism is achieved, then the remaining portion of the program cannot be parallelized. In other words, the remaining portion must correspond to the critical path. In the case of the maze solver, there are two critical paths. The first is the atomic increment function, as every other thread has to wait for the index to increase before being assigned to it. The second involves updating the array of new points to visit in the next iteration. In particular, copying the array from the device to the host and vice versa. This array needs to be setup and initialized before any parallel functions can occur.

Appendix A: Solutions from sequential maze solver

In the images below the path from the origin (0,0) to the destination (r,c) is colored in red.

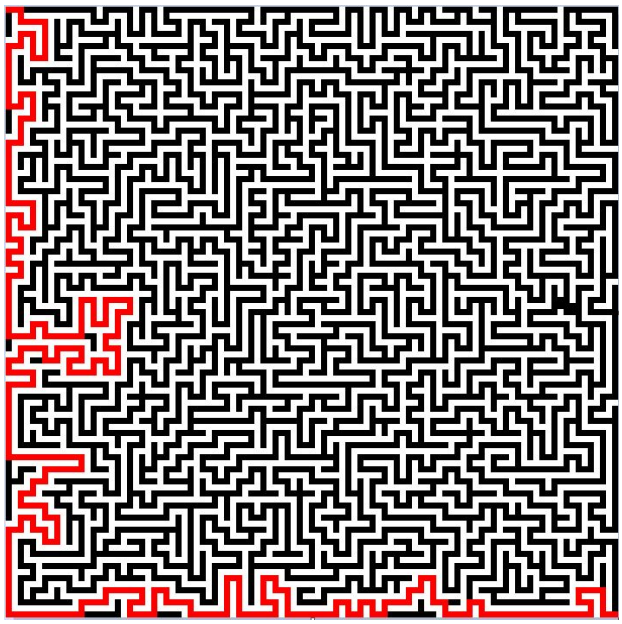


Fig 15: *101x101 maze solved in sequential*

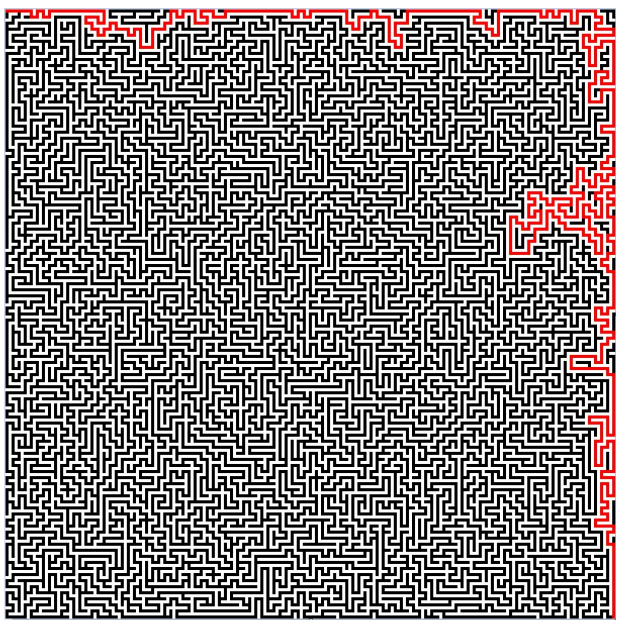


Fig 16: *201x201 maze solved in sequential*

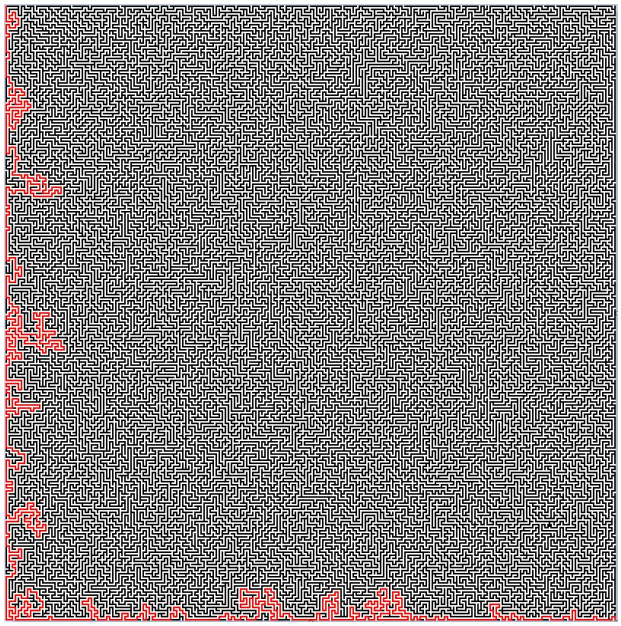


Fig 17: *401x401 maze solved in sequential*

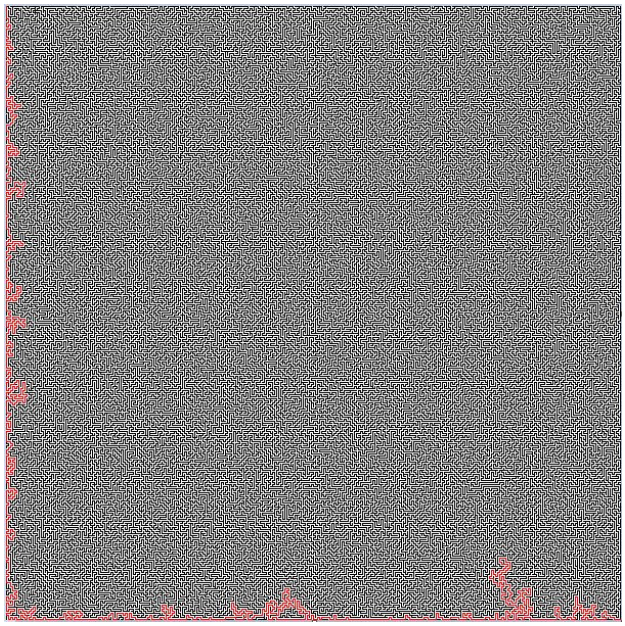


Fig 18: *601x601 maze solved in sequential*

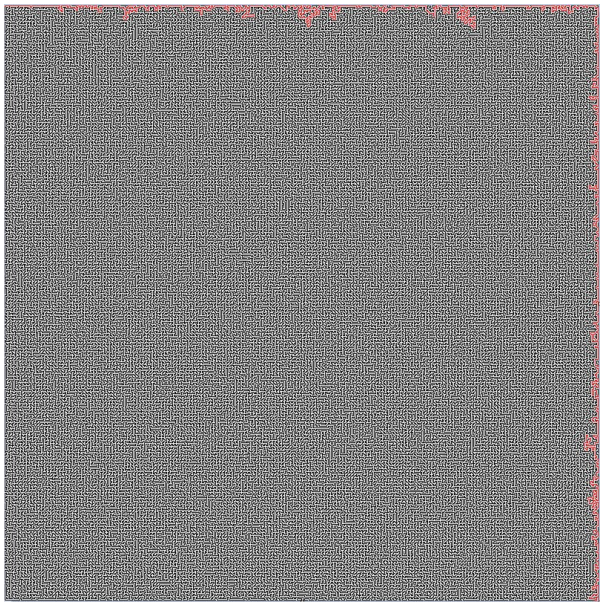


Fig 19: *801x801 maze solved in sequential*

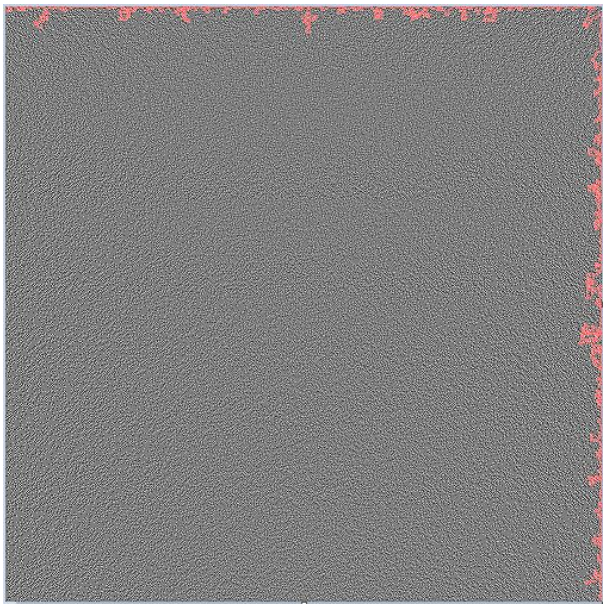


Fig 20: *1001x1001 maze solved in sequential*