Random mazes generations and solutions

A parallel implementation

Mattia Bennati

Arezzo 26/05/23

Introduction

Project overview

Objective: The purpose of the project is to highlight the benefits of parallelism over sequential programming for generating and solving perfect mazes randomly, by moving a considerable amout of particles inside of them.

Perfect mazes consists of:
• a single solution path
• the absence of unreachable areas
• a unique connecting path between any pair of points

Two versions of the project have been realized, in order to evaluate the performance benefits of the multithreading paradigm.

This project makes use of the OpenMP multiplatform API in order to implement parallelism and this leads to a much faster execution.

In order to make tests and performance evaluations consistent and reliable, both the sequential and the parallel version of the project share the same execution seeds.
This ensures that both versions follow the same exact generation and solution steps.

Note: seeds are randomly generated, if not specfied.

Introduction

Project overview

The project consists of a single executable file that is capable of running both the sequential and the parallel version of the code.

In order to specify which version to execute, a configuration file has been provided.
The configuration allows also to specify other parameters like: the number of executions to perform, the size of the maze, the number of particles to generate and the seeds.
Note: the size must be an odd integer value in between [51;301] as it is necessary to create a grid structure in order to represent the maze correctly.

It is also possible to show all the steps related to the mazes generation and solution, by rendering each maze with ascii characters in the console.
This mode slows the execution down, but it's useful to see and understand how mazes are being handled.

By specifying the generation's and solution's seeds it is possible to get a predictable maze structure and the related solution steps.

The source code of the project can be retrieved at the following location:

https://github.com/Scrayil/RandomMazeSolver

Maze generation

Implemented algorithm

There are many algorithms for generating mazes, in this case a depth-first search (recursive backracking) approach has been used.
This generation strategy follows a sequential logic and it's not possible to parallelize the underlying algorithm, but it is efficient enough for this purpose and it's optimal for perfect mazes.

The implemented algorithm outlines the following steps:
1. Creation of a matrix containing a grid of walls that represents the maze's inner structure.
2. Selection of a random exit cell as starting point for the path generation.
3. Selection of the nearby cell, that does not contain a wall, as the first one to connect with.
4. Selection of a new cell to connect with, chosen randomly in between all the cells located nearby, behind a wall.
5. After selecting the next cell, the wall in between the two is deleted, this procedure actually creates a unique connection path.
    i. Each cell added to the path is marked as visited.
6. The steps are repeated by selecting only unvisited cells until a dead end is found.
7. Once a dead end is reached, the backtracking logic starts.
    i. It proceeds until a cell with nearby unvisited cells is found, so that the path generation can be resumed from there.
8. The generation stops when all the cells have been marked as visited.

Maze solution

Implemented algorithm

The solving algorithm is quite simple, a big number of particles is spawned inside the maze by selecting a random starting point.

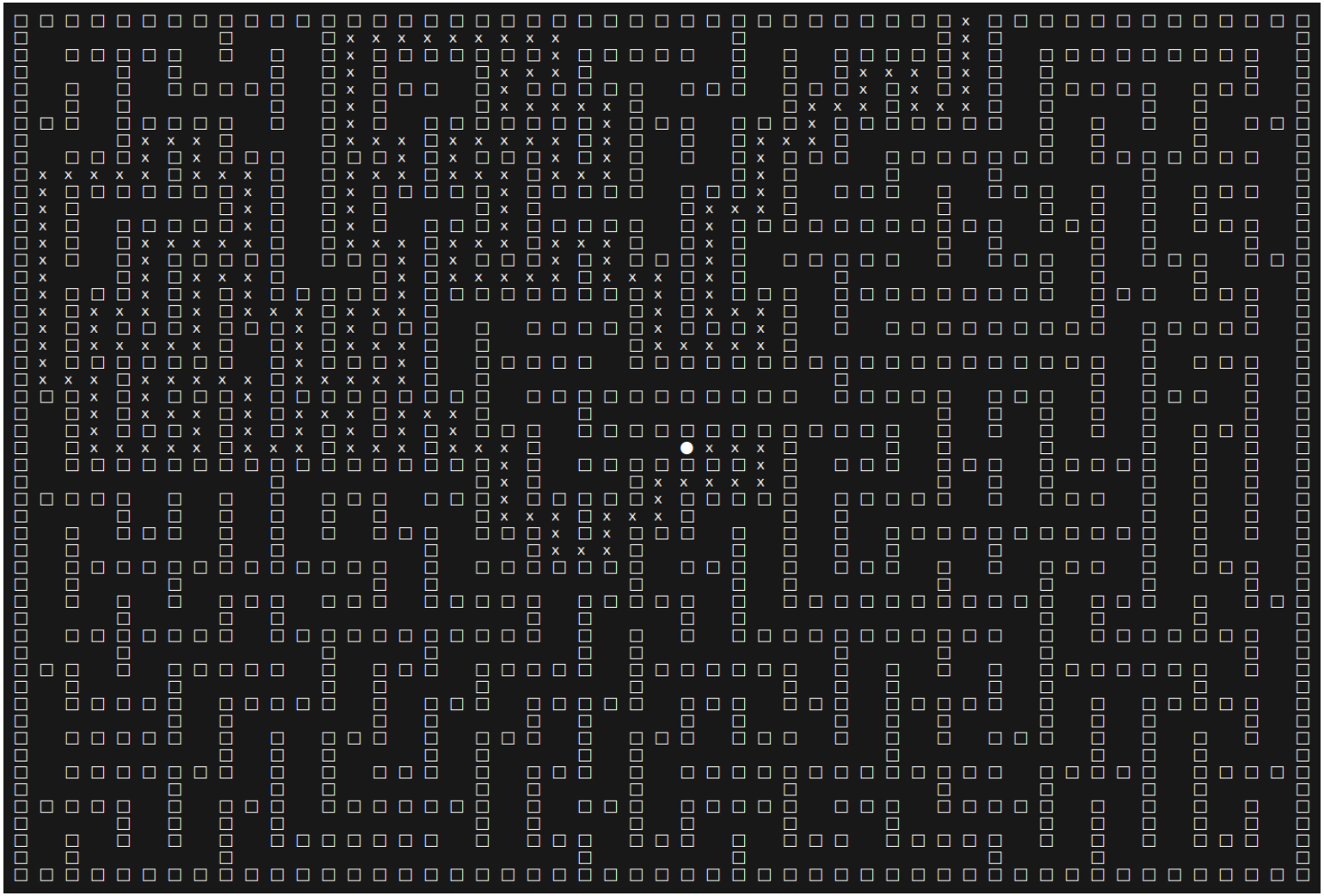
There are two phases:

1. All the particles move around randomly until one of them casually reaches the exit of the maze.
2. Once a particle reaches the exit, the solution path is outlined for all the others.
   i. Particles now start backtracking their own steps until they are onto the solution track
   ii. Aftert that, they backtracks the exited particle's remaining steps in order to reach the exit.

The execution stops when all the particles have managed to exit the maze.

In order to make the generation and solution processes of mazes more clear, an animation has been provided at the following link:

https://github.com/Scrayil/RandomMazeSolver/blob/e159ac2b17e8bf9910b3aa63140936106d15792b/report/media/video/slowed_down_steps.gif



Visual representation of a solved maze with size 51x51 by using ascii characters

Testing machine

Specifications

The computer used for the development and the testing phases of the project has the following characteristics:

CPU: 11th Gen Intel(R) Core(TM) i7-11800H @ 2.30GHz with 8 physical cores

RAM: 2 x 16GiB SODIMM DDR4 Synchronous 3200 Mhz

GPU: GA107M [GeForce RTX 3050 Ti Mobile] with 2560 CUDA cores

Sequential version

Limitations

The generation algorithm allows to quickly generate mazes with sizes up to 701x701.

Bigger sizes couldn't be tested on the current machine as they were occupying too much space in memory to be handled properly.

Note: even if the algorithm is fast, the **maze's structure**, if too big, can be memory consuming.

The solution algorithm instead is the real bottleneck of the sequential version as the more particles are spawned, the more time is required to move them all around.

So the real limit here is that if the **number of particles** is greater than 1,000,000 the program can be very time consuming.

It's important to mention that the overall time taken to execute each version also depends on the starting point location, and on the length of the solution path; not only on the particles' random movement sequences.

Parallel version

Overcoming some limitation

There are some sections in the code in which vectors and matrixes are initialized and populated with default values.

These operations can be performed in a thread safe way.
Thanks to the OpenMP library, it is possible to execute such tasks faster, by running each loop iteration in parallel.

The actual number of instructions that can be executed in parallel, depends on the available number of threads.
The advantage of using OpenMP is that each iteration can be executed by a different thread.

Note that this works when there is no code dependency inside the loop and there is no risk of concurrent access to shared variables (data races).

In order to achieve the parallelism mentioned above, the following OpenMP compiler directive can be specified. OpenMP automatically takes care of spawning the required threads and to parallelize the loop:

#pragma omp parallel for

Parallel version

Overcoming some limitation

In order to better handle the memory usage, for the particles' data structures in the parallel version of the code, the approach has been switched from array of structures (AoS) to Structure of arrays of structures (SoAoS).

The real benefits of parallelism can be seen in relation to the solving phase of the mazes as by executing iterations in parallel, even with a big amount of particles, the achieved speedup is remarkable.

Many sections have been parallelized with OpenMP directives, and thanks to this, many particles' movements can be evaluated and processed simultaneously.

This actually reduces the time required to execute the code and solve the mazes.

By applying the approaches above, there are benefits in terms of performance and memory efficiency.

Performance evaluation

Results

The table below shows the average speedup computed by comparing the execution time of the two versions. Both the size and the number of partcles have been taken into account.
In order to calculate the speedup, the following formula has been used:

$$S_P = \frac{t_s}{t_p}$$

Where t_s represents the execution time of the sequential version and t_p the timing of the parallel one.

Table 1. Average speedup by size and number of particles

	Maze Size	N° Particles	Avg. Speedup
Overall	X	X	10.259
Fixed Size	51	X	8.609
Fixed sizes and particles	101	10,000	15.212
	51	10,000	6.519
	51	100,000	11.679
	51	1,000,000	7.987

* The "X" is just a placeholder indicating that for the calculus the field has not been fixed to a specific value. All related values have been taken into consideration.

Performance evaluation

Conclusions

This project outlines the advantages of parallelism over the sequential execution of the code, in order to handle multiple iterations at once.
The performance benefits are evident, especially for bigger mazes.
Using a particle count between 10^3 and 10^6 leads to the better results.

The testing phase was conducted on a machine with 16 physical threads.
By implementing parallelism and using compiler directives, an average speedup of 10.259 times has been achieved.

Parallel computing offers significant performance improvements with the implemented algorithm and in this case this can lead to a faster and more efficient code execution.