Parallel Programming 2023
Mid-term assignment

Lucia Giorgi

1. Goal of the assignment

Find the exit from a maze using the random movement of a particle.

- Start a large number of particles, move them randomly bouncing on the walls
- Backtrack the first particle to get out of the maze to find the exit

2. Implementation

The idea is:

- Take an image of a maze from the Internet
- Extract information about the maze geometry and load it into an appropriate data structure
- Move a particle (or multiple particles for the parallel version) randomly from the starting point of the maze until it reaches the exit.

In this assignment the parallelization is achieved using OpenMP.

2.1. Classes

The maze is represented by the class `Maze` that contains a collection of cells (class `Cell`).
From https://www.astrolog.org/labyrnth/glossary.htm: a cell is point in a Maze, or more technically the passage units which are linked in a grid or network to form the Maze. Each cell is a vertex, and has zero or more passage edges leading away from it to other cells.
More practically, if we divide the maze into a grid, the cells are the white squares (or rectangles) in which the particle can move and the walls or borders are the black lines that separate the cells (fig. 1).
We can then represent the position of the particle with the coordinates (x, y) of the cell that contains it. From a cell, the particle can move in at most four direction (left, up, right, down) depending on the walls adjacent to the cell.
The class `Maze` contains a method that returns the cell adjacent to another cell in a specific direction, while the class `Cell` provides the admissible directions for the following movement (the directions where there isn't a wall).
At each step, the direction in which the particle will move is chosen randomly from the admissible ones, so that the movement is a random walk. Also, the position of the visited cell is stored in a list in the main function of the program.
In addition, `Maze` has of course methods for loading the maze from an image and saving the image with the solution path. (figures 2 3))).

2.2. Main function

There are four loops in the main function of this program.

- The most inner loop contains the steps for the particle to get to the next random cell and to store the current position in the list, and it repeats as long as a boolean variable `solution_found` remains false.

Figure 1: A cell with two possible directions and two walls

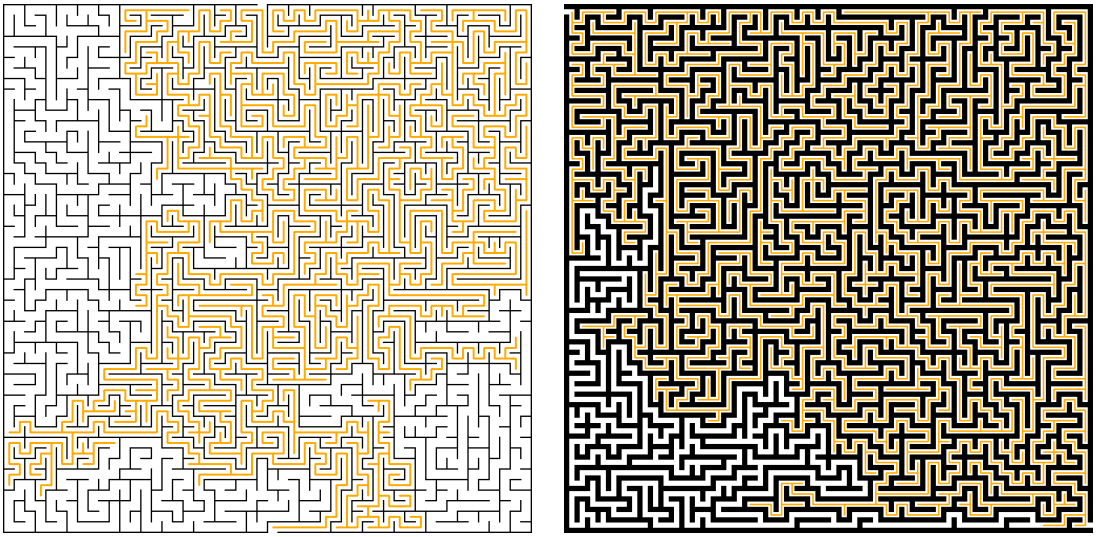


Figure 2: Two 50×50 mazes with the solution found by the particle

- The second one is a for loop that cycles over all the particles that are launched from the start cell to find the exit (this is the loop the parallelization is applied to). When a particle finds the exit cell and decides to move towards the exit direction, a `OutOfMazeException` is threw, `solution_found` is set to true and the loop is exited. The other particles will immediately know that someone has found the solution and will also exit the loop.
- Then we have another for-loop that has the function to repeat the experiment multiple times to obtain an average over all the runs
- And finally the outer loop that cycles over all the maze image in a specific folder. The mazes have different sizes so that we can obtain the average execution times over different magnitudes of data.

The program can receive command line arguments for setting the number of particles, the number of threads and the number of runs.

3. Parallelization

In the sequential version of this program a single particle at time is launched and therefore the total number of particles is irrelevant because the particles that come after the first one will found `solution_found = true` and will exit the loop right away.
In the same way, in the parallel version it's irrelevant how many particles are launched as long as this number is at least equal to the number of threads (otherwise there would be less particles than the number of threads and some threads would be idle).
The parallel version is obtained by adding a `#pragma omp parallel for` directive before the particles loop, with the idea that having more particles running in parallel improves the chance of finding the exit sooner than with a single particle.
The variable `solution_found` is passed as a `shared` variable between all threads. The value of this variable is checked at every step of the particles and is set to true when a particle finds the solution. It's not therefore protected by a lock because it could slow down the execution considerably.

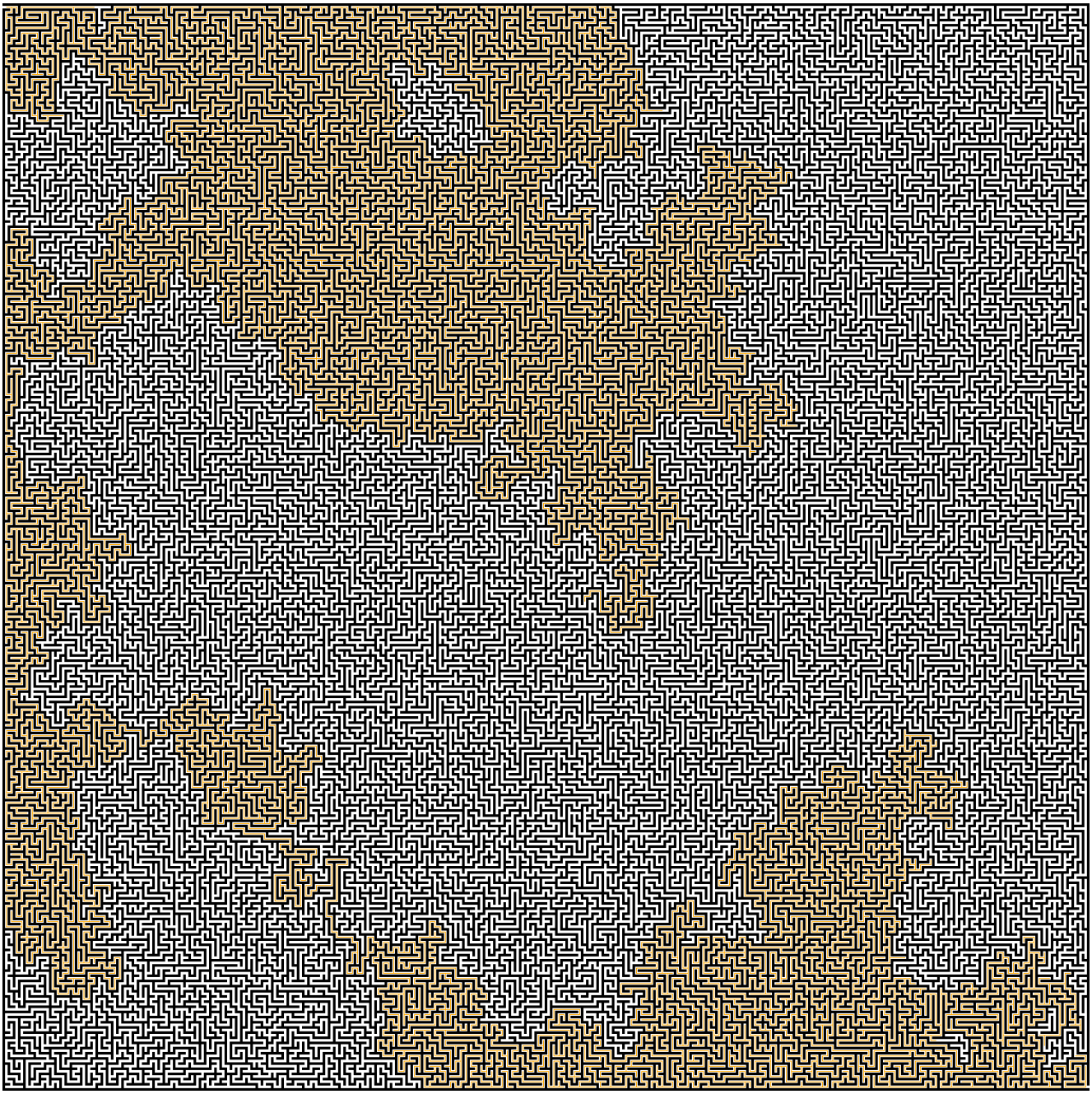


Figure 3: A 210×210 maze with the solution found by the particle

This means that there is the possibility that two threads find the solution: the first one sets `solution_found = true` but the second one reads the old value where it's set to false and it also finds the exit right after. In this case we would want only one thread to contribute to the execution time count, therefore another boolean variable `solution_found_locked` is necessary. This variable is protected by a lock and every thread that exits the maze read its value and proceeds with writing the execution time in the logs and setting it to true only if the variable is false (this means that they are the first thread to find the solution).

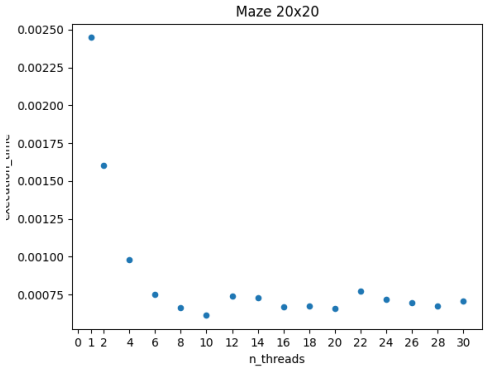
```
while (!solution_found) {
// steps for moving the particle
// ...
     catch (OutOfMazeException &e) {
        solution_found = true;
        endTime = std::chrono::high_resolution_clock::now();
        bool is_first;
        #ifdef _OPENMP
        omp_set_lock(&solution_found_write);
        #endif
        if (!solution_found_locked) {
            solution_found_locked = true;
            is_first = true;
        }
        #ifdef _OPENMP
        omp_unset_lock(&solution_found_write);
        #endif
        if (is_first) {
            // log the execution time
            // ...
        }
    }
}
```

4. Experiments and results

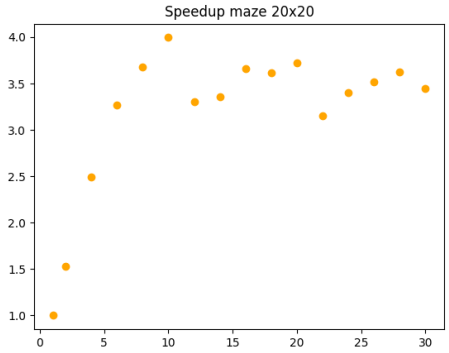
I ran the sequential version of the program (compiled without OpenMP) and the parallel version with 2 to 30 threads by increments of 2. The number of particles was set to 100 (just to have a number greater than the threads number). There were 10 mazes of different sizes and for every maze the number of run was set to 100 in order to have an accurate average over the execution times.
For the experiments I had to set to false the flag for saving the solution, because with large mazes the memory needed to store the path of the visited cells was too big for my machine and the process was automatically killed, especially with many threads.
The execution times were measured with the start time taken just before the parallel loop and the end time right after setting `solution_found = true`.
Even for the small mazes the execution time is greatly improved by using just 2 threads, as we can see in 4a, with a maximum speedup of 4 with 10 threads (4b).
The maximum speedups for the different mazes range between 2.9 and 6.4, and it occurs for a threads number between 8 and 16, with 10 as the most recurring value. After the maximum speedup is reached, the speedup usually decreases with more threads (fig 5).

I used Google perftools to profile the program, and from the output it looks like the majority of the time is spent generating a random number for moving the particles in random directions, as we can see in the following partial output from the profiler.

```
Total: 81970 samples
   16134  19.7%  19.7%    46927  57.2% std::generate_canonical
   14829  18.1%  37.8%    29644  36.2% std::mersenne_twister_engine::operator
   14815  18.1%  55.8%    14815  18.1% std::mersenne_twister_engine::_M_gen_rand
    7671   9.4%  65.2%    81929  99.9% main._omp_fn.0
    6443   7.9%  73.1%    58316  71.1% std::uniform_real_distribution::operator
    5338   6.5%  79.6%    14167  17.3% Maze::move
    3895   4.8%  84.3%     4888   6.0% Maze::getCell
    2127   2.6%  86.9%     2127   2.6% Cell::getY
    ... other lines
```

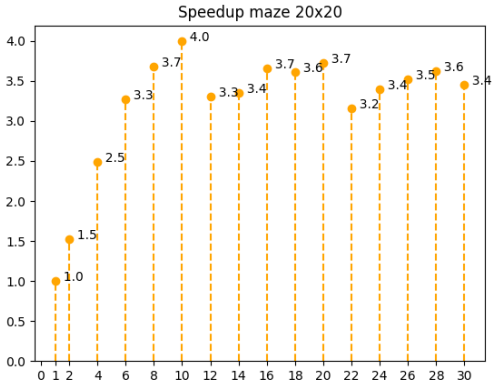


(a) Execution times for a 20×20 maze (s)

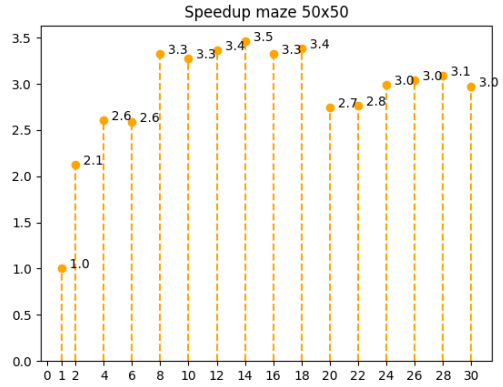


(b) Speedup for a 20×20 maze

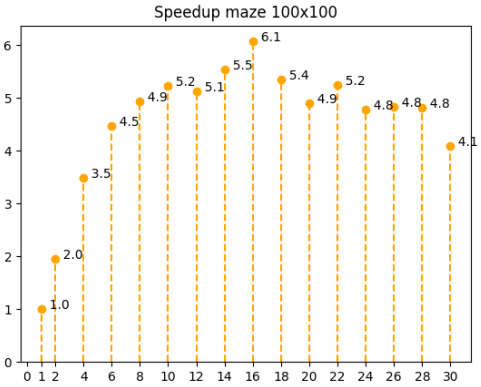
Figure 4: Execution time and speedup for a 20×20 maze, averaged on 100 runs



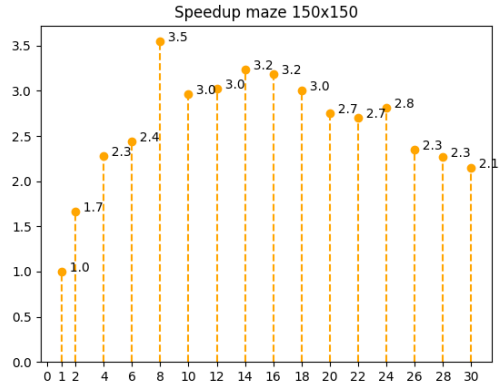
(a) Speedup for a 20×20 maze



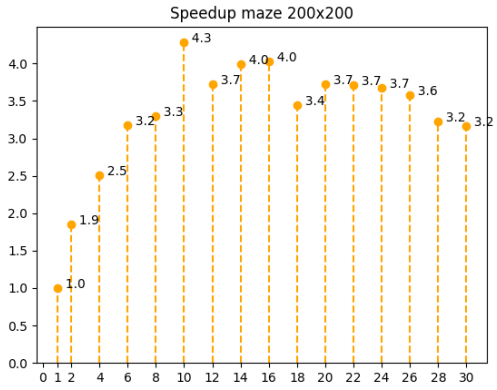
(b) Speedup for a 50×50 maze



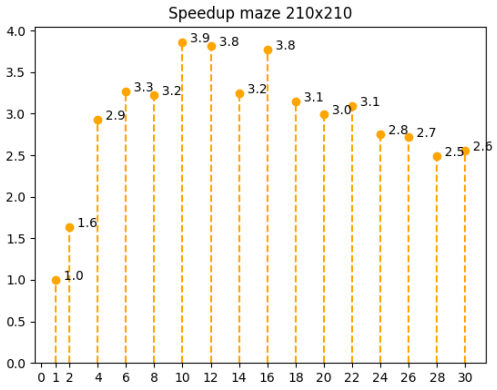
(c) Speedup for a 100×100 maze



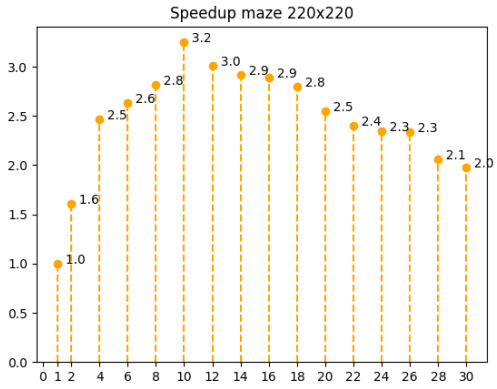
(d) Speedup for a 150×150 maze



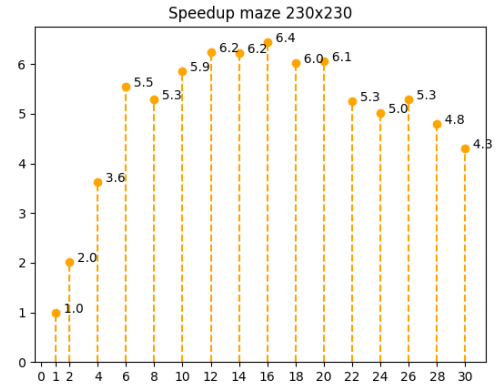
(e) Speedup for a 200 × 200 maze



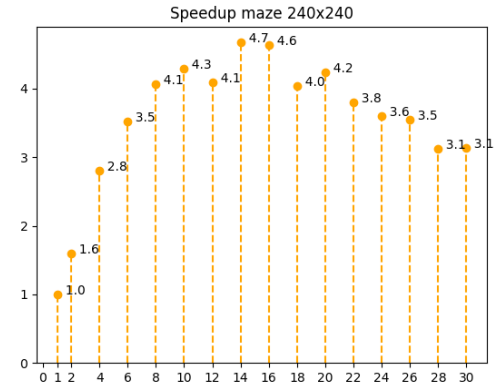
(f) Speedup for a 210 × 210 maze



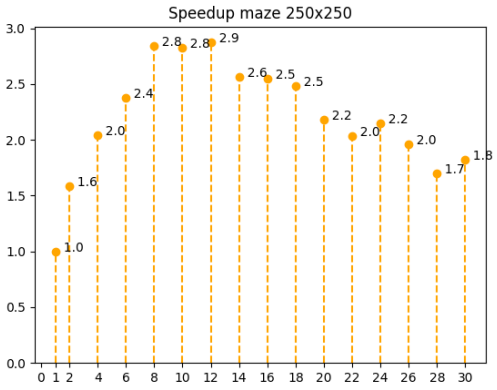
(g) Speedup for a 220 × 220 maze



(h) Speedup for a 230 × 230 maze



(i) Speedup for a 240 × 240 maze



(j) Speedup for a 250 × 250 maze

Figure 5: Speedup for mazes of various sizes, averaged on 100 runs