Solving a maze using OpenMP

Francesca Lanzillotta
E-mail address
francesca.lanzillotta@edu.unifi.it

Abstract

In this project we'll compare the performances of a maze solver using both a sequential and a parallel approach based on OpenMP library.

1. Introduction

The goal of this project is to compare the performances of a maze solver using both a sequential and a parallel approach using the OpenMP library. The maze, generated using a recursive algorithm, is solved by randomly moving a certain amount of particles until one reaches the exit.

1.1. Maze

The maze is generated using the recursive backtracker algorithm, a randomized version of the depth-first search algorithm. This method is one of the simplest way to generate a maze and it's easily implemented with a recursive approach. The maze is firstly initialized as a grid resembling a chess board, alternating walls and free space cells. On this object we manually set a starting point, roughly in the middle of the maze, where all particles will be initially spawned. We also randomly set the exit on one of the sides of the maze. The following recursive algorithm is invoked to create the pathways:

1. Pick a starting cell.
2. Mark the cell as visited.
3. While the cell has unvisited neighbours.
   (a) Randomly pick an unvisited neighbouring cell.
   (b) Remove the wall between the current cell and the chosen cell.
   (c) Repeat recursively steps 2-3 on the chosen cell.

The algorithm stops once it successfully backtracks all the way to the starting cell, ensuring all cells have been visited. Figure 1 depicts the maze before and after the pathways have been generated.

The mazes created with this approach have a single solution i.e. there exist a single path connecting the starting point and the exit. Moreover, the mazes tend to have many long corridors, since the algorithm fully explores every branch before backtracking. All these functionalities are implemented in the `Maze` class.

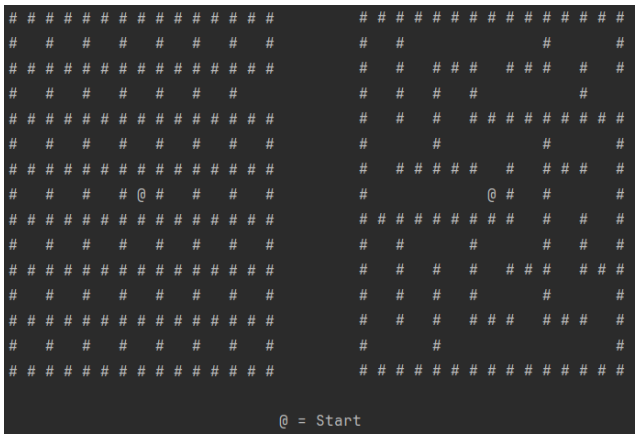

Figure 1. On the left, the maze grid. On the right the maze generated by the recursive backtracker algorithm

1.2. Particle

Particles are objects that move through the maze at random, until some particle reaches the exit: once the solution is found, all particles follow its path to the exit.
Each particle is spawned from the same starting point. At each step, the particles randomly pick a free neighbouring cell. In this context, a cell is free i.e. a particle can move to it, if it's not a wall. This means that more than one particle can reside on a single cell: this ensures that the particles don't clog the pathways and can move freely within the maze. As already mentioned, the mazes used in this project tend to have many long corridors: to make the particles move less erratically, we implemented the function picking the random moves so that the option of going backward is inhibited with a certain probability.

2. Solving the maze

The strategy to solve the maze is divided in two main steps:

1. Move randomly all particles until one finds the exit
2. Have all the particles follow the solution's path until they exit the maze

The goal of this project is to solve the maze using this strategy both with a sequential and a parallel approach. In the following sections we'll illustrate in depth the two methods.

2.1. Sequential solution

The sequential method to solve the maze is defined in the function `s_solveMaze`. In the function, we iterate over all the particles and we move them randomly until one finds the exit: this particle sets the boolean variable `out` to true, thus stopping the for loop. The path's solution is highlighted within the maze and, lastly, we iterate once again over all the particles that haven't reached the exit yet: each particle backtracks until it finds a cell on the solution's path and then it follows the traced path to the exit. It is guaranteed that every particle will encounter a cell on the solution's path, since all particles have the same starting cell. The code is presented in listing 2.1.

```
void s_solveMaze(Maze &m, vector<Particle> &particles,
     int ms= 0, float backProb= 0.7){

    bool out = false;
    vector<pair<int, int>> solution;
    while (!out){
        for(auto& p : particles){
            p.randMove(false, backProb);
            if (p.getPosition() == m.getExit()){
                out = true;
                solution = p.getPath();
                break;
            }
        }
        if (ms > 0){
            delayedCLS(ms);
            cout << m.toString();
        }

    }

    for (auto c : solution){
        m.setCellType(c, PATH);
    }

    auto exit = m.getExit();
    for (auto &p: particles) {
        if (p.getPosition() != exit){
            auto s = p.backtrack(solution, ms);
            p.followPath(s, ms);
        }

    }
```

2.2. Parallel solution

To parallelize the maze solver we used OpenMP, an API for shared-memory parallel programming.
OpenMP is designed for systems in which each thread or process can potentially have access to all available memory. One of the main advantages of using OpenMP is that it does not require restructuring the serial program: compiler directives are used to specify which sections of the code need to be parallelized. This model greatly simplifies parallel computing, hiding details at the expense of expressiveness.
In the function `p_solveMaze`, there are three for loops, all parallelized using

```
#pragma omp parallel for schedule(static)
```

This construct creates a parallel region and divides the iterations of the for loop to be distributed across different threads. Specifying the schedule clause as static means that each thread is assigned a chunk of the iteration space.
In the first for loop, we use the directive

```
#pragma cancel for
```

to stop the for loop execution once a particle has found the exit. The code is presented in listing 2.2.

```
void p_solveMaze(Maze &m, vector<Particle> &particles,
     int nThreads, int ms=0, float backProb=0.7) {

    vector<pair<int, int>> solution;
    bool out = false;
    auto exit = m.getExit();

#ifdef _OPENMP

    omp_set_num_threads(nThreads);

    while (!out) {
#pragma omp parallel for schedule(static)
        for (auto &p: particles) {
            p.randMove(0, backProb);
            if (p.getPosition() == exit) {
                out = true;
                solution = p.getPath();
#pragma cancel for
            }
#pragma omp cancellation point for
        }
    if (ms > 0) {
        delayedCLS(ms);
        cout << m.toString();
    }
}

#pragma omp parallel for schedule(static)
    for (const auto & cell : solution) {
        m.setCellType(cell, PATH);
    }


#pragma omp parallel for schedule(static)
    for (auto p : particles) {
        if (p.getPosition() != exit){
            auto s = p.backtrack(solution, ms);
            p.followPath(s, ms);
        }
    }
#endif
}
```

3. Experimental Results

All experiments were performed on a Lenovo Legion 5 with an AMD Ryzen 7 5800H with 8 cores and 16 threads. Three types of tests were performed:

- The first test aims to compare the performance of the sequential maze solver against the parallel one, using an increasing number of threads.
- With the second test we want to assess the performances of the solvers using an increasing number of particles.
- In the last test, we want to compare the performance of the sequential solver using a single particle against the one of the parallel solver using a fixed number of threads with both a single particle and a particle per thread.

All tests were performed on a 81x81 maze and repeated ten times in order to validate the data measured. All plots and tables contain the averages of these tests.

3.1. Increasing number of threads

The results of the first test, displayed in figure 2, show, as expected, a decrease in execution time using the parallel version of the maze solver. Moreover, the best performance was achieved using 16 threads, as we also expected to find using a CPU with 16 threads.

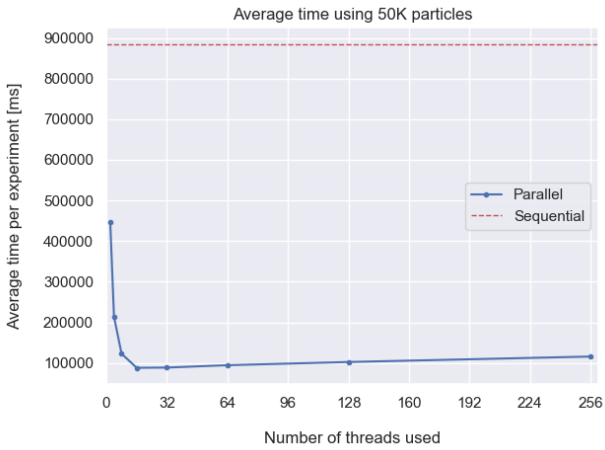

Figure 2. The time steeply decreases using the parallel version of the maze solver, with peak performance using 16 threads

3.2. Increasing number of particles

The results of these tests are shown in table 1 and in figure 3. In the table we can see that the average time increases using more particles, with the parallel version always performing better than the sequential one. In the figure are displayed the speedups, computed as the ratio between sequential and parallel time. As we expected, adding more particles increases the speedup.

Threads	Number of particles		
	10K	25K	50K
Sequential	6.1 s	213.698 s	884.825 s
2	4 s	115.9 s	447.5 s
4	2.2 s	58.8 s	213.8 s
8	1.3 s	37.5 s	122.8 s
16	1.1 s	24.3 s	88.2 s
32	1.2 s	26.7 s	88.8 s
64	1.3 s	26.2 s	94.6 s
128	1.6 s	31 s	102.4 s
256	2.3 s	35.6 s	115.9 s

Table 1. The parallel solver's performances increase along with the number of particles

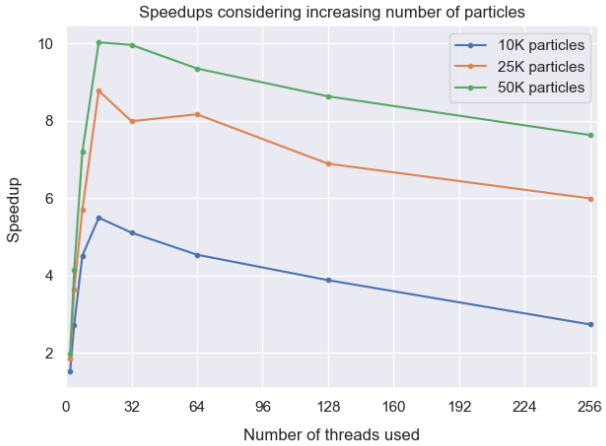

Figure 3. The parallel solver's performances increase along with the number of particles

3.3. Single particle

The results of the last test are shown in figure 4: the sequential version is the fastest while the parallel version with a single particle using 16 threads is the slowest. This was to be expected, since the parallel version adds the overhead for scheduling and managing 16 threads, 15 of which are doing no computation at all, thus increasing completion time. The parallel version with one particle per thread performs significantly better, but not as well as the sequential version. This is probably due to two reasons: first of all, even though the more particles we add, the more likely it is to find the exit, we still need to move all the rest of the par-

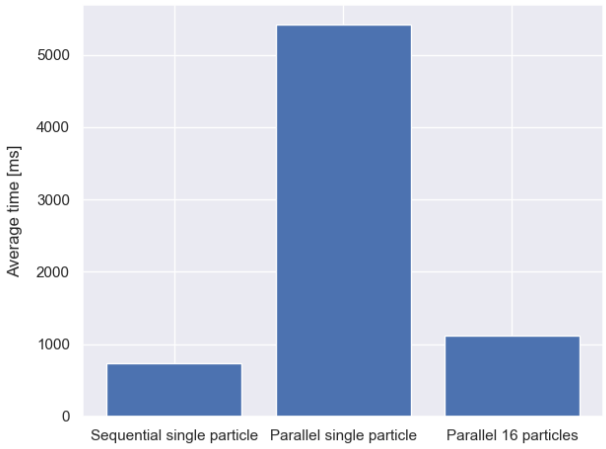

Figure 4. The parallel approach has better results with massive computation, so that overhead is evened out

ticles outside the maze, which means that adding more particles doesn't necessarily mean reducing completion time. The other reason is that by parallelizing code we add unavoidable overhead for managing and scheduling threads: this loss in performance is usually outweighted by the benefits of parallel computation when computing a huge amout of data, as the previous tests have shown.

4. Conclusions

From the test carried out, we can conclude that the parallel approach is usually the best one in almost all scenarios. The sequential version might be useful in the cases where the number of particles is reduced.