PC-2022/23 Mid-Term Project
Benefits of parallelism for random mazes generations and solutions

Mattia Bennati
7122582
mattia.bennati@stud.unifi.it

Abstract

This report consists of a comparison between the sequential and parallel implementations of a C++ algorithm in order to highlight the benefits of parallelism in programming. The project has been developed by using the OpenMP multi-platform API to the extent of making use of the multithreading paradigm. The random maze generation and solution have been taken into account as an example.
The source code will be publicly available.

Future Distribution Permission

1. Introduction

One of the aim of this project was also to generate perfect square mazes. A maze is considered "perfect" if it has exactly one solution, it contains no unreachable areas and there is a unique path that connects any pair of points.
The random maze generation can be implemented by using various algorithms, in this case a depth-first search approach has been used (recursive backtracking). Even if this approach is not well suited for parallelization as it follows a sequential logic, it's efficient and optimal for the generation of perfect mazes.
The solution's steps are based onto the random movements of a substantial number of particles and the first exited particle outlines the path for the remaining ones.
In order to make the most correct comparison between the two implemented versions of the program, random seeds are shared between the executions. This means that both the sequential and the parallel version of the project follow the same exact steps in generating and solving the mazes.

2. Maze generation

2.1. Recursive backtracking

The first step in order to generate a maze is to set and evaluate a seed for the random values generation, and a size related to each of its sides. As stated earlier a recursive backtracking strategy has been used. It has been implemented as follows:

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. 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. 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.

Mazes generated with the above approach would look as follows:

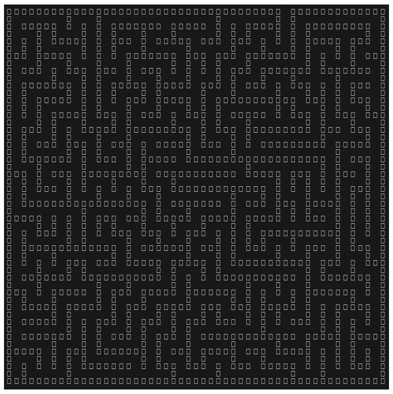

Figure 1. Sample square maze of size 51x51

2.2. Implementing parallelism

Since the algorithm used for the generation of the mazes follows a procedural logic, better performances can be achieved by implementing parallelism during the initial maze creation.
Parallelism does not always result into better performances, some factors must be considered first. The creation and deletion of threads are operations with a certain overhead that can't be ignored. Sometimes it's just better to not use them as the sequential execution of the program could be faster and more efficient.
In order to take that into account, a prior evaluation has been implemented as follows:

```
#ifdef _OPENMP
    if(!omp_get_nested())
        omp_set_nested(true);
    if(omp_get_max_active_levels() < 2)
        omp_set_max_active_levels(2);

    if(size * size /
↪   static_cast<int>(omp_get_max_threads()) > 100) {
        parallelize = true;
        std::cout << "Parallelizing the generation!" <<
↪   std::endl;
    }
#endif
```

The following snippet shows the usage of some OpenMP directives that allows to execute the instructions by using multithreading.

```
// Resizes the matrix to avoid sigsev errors
#pragma omp parallel if(parallelize)
{
    #pragma omp for
    for (int i = 0; i < maze.size(); ++i) {
        maze[i].resize(size);
    }
    // Initializes the matrix
    #pragma omp for
    for (int row = 0; row < size; row++) {
        for (int col = 0; col < size; col++) {
            // Place walls on even rows and columns to
↪   create the grid
            if (row % 2 == 0 || col % 2 == 0) {
                #pragma omp critical
                maze[row][col] = MAZE_PATH::WALL;
            }
                // Set the walkable path
            else {
                #pragma omp critical
                maze[row][col] = MAZE_PATH::EMPTY;
            }
        }
    }
}
```

The block of code enclosed in the ”parallel” section is going to be executed by multiple threads. Since it's not always suitable to implement parallelism, a flag is going to be evaluated here.
The OpenMP ”for” directive tells the compiler to execute each loop iteration in a different thread. This can be safely done here as there are no code dependencies inside the loop.
The ”critical” directive instead is used in order to ensure that only one thread at a time can execute the instruction located below, otherwise there would eventually be data races.
OpenMP has also been used for the creation of some vectors and matrix's that will be used later in the code while generating the maze's path:

```
// Initializes the visited_cells array to false as no
↪   cell has been visited yet
#pragma omp parallel if(parallelize)
{
    #pragma omp for
    for (int i = 0; i < size; ++i) {
        visited_cells[i].resize(size);
    }
    #pragma omp for
    for(int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            #pragma omp critical
            visited_cells[i][j] = false;
        }
    }
}
```

3. Maze solving

3.1. Solving logic

The solution of the maze is based onto two main phases.
The first phase consists of spawning a big number of particles that will start moving randomly. Once a particle reaches the exit, the solution path is outlined for the others and the second phase starts.
The remaining particles can now backtrack their own steps until they are on the right track, and just after that, they start backtracking the exited particle's remaining steps in order to exit the maze.
The previously generated maze's image after solving would look like the following:

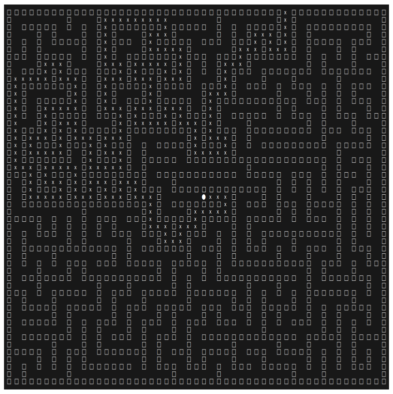

Figure 2. Solved square maze of size 51x51

3.2. Implementing parallelism

Solving the maze can be an expensive memory and computing operation as a huge number of particles could have eventually been spawned in a very big maze.
In this case the benefits of parallelism can be remarkable.
As for the generation, here too there is a simple initial evaluation in order to determine if it's possible to benefit from parallelism:

```
#ifdef _OPENMP
    parallelize = false;
    if(n_particles / omp_get_max_threads() > 100) {
        parallelize = true;
        std::cout << "Parallelizing the solution!" <<
↪   std::endl;
    }
#endif
```

In order to make the code execution faster and more efficient, the particles' data has been handled with a "structure of arrays of structure" (SoAoS) approach, that replaces the "array of structure" (AoS) one used in the sequential version.
In addiction many OpenMP directives have been used in order to achieve better performances.
Here the used directives allows to initialize and add multiple particles at once, as the function used to add the particle has been implemented with a thread safe logic:

```
// SoAoS
Particles particles(n_particles);

#pragma omp parallel for if(parallelize)
for(int i = 0; i < n_particles; i++) {
    particles.addParticle(i, initial_position);
}
```

The random particles movements have been parallelized too in order to handle many of them simultaneously. In particular the directives in the following snippet allows to execute each particle movement (loop iteration) in a different thread.
This can results into an important speedup in relation to the execution.

```
#pragma omp parallel for if(parallelize)
for(int index = 0; index < particles.how_many; index++) {
    if(!exit_reached) {
         std::vector<MOVES> moves = p_get_possible_moves(maze, size,
↪   particles.positions[index]);
        bool same_move_allowed = false;
        for(MOVES move : moves) {
            // Keeps going on if it can go only on opposite directions
            if(moves.size() == 2 && particles.moves[index] == move) {
                same_move_allowed = true;
                particles.update_coordinates(index, move);
                break;
            }
        }
        // The same move wasn't available because of the walls nearby
        if(!same_move_allowed) {
            // Choosing a random move
            std::uniform_int_distribution<int> uniform_dist(0,
↪   static_cast<int>(moves.size()) - 1); // Guaranteed unbiased
            MOVES new_move = moves[uniform_dist(rng)];
            particles.update_coordinates(index, new_move);
        }

        if(show_steps) {
            // Add the particles to the maze_copy

↪   maze_copy[particles.positions[index].row][particles.positions[index].col]
↪   = MAZE_PATH::PARTICLE;
            #pragma omp critical
            // Shows the start everytime
            maze_copy[initial_position.row][initial_position.col] =
↪   MAZE_PATH::START;
        }

        // The particle has reached the exit

↪   if(maze[particles.positions[index].row][particles.positions[index].col]
↪   == MAZE_PATH::EXIT) {
            #pragma omp critical
            {
                exited_particle_index = index;
                exit_reached = true;
            };
        }
    }
}
```

Even if the synchronization is required in order to avoid data races, it's impact is relatively small in this case.
The parallelism logic has also been applied while marking the solution path, when changing the maze's structure representation as follows:

```
#pragma omp parallel for if(exited_particle_path.size()
↪  / omp_get_max_threads() > 100)
// Shows the maze's path that lead to the solution
for(Coordinates coord : exited_particle_path) {
    maze[coord.row][coord.col] = MAZE_PATH::SOLUTION;
}
```

There are other two cases in which the parallelism have been implemented.

1. The first one is related to the initial population of two vectors used to determine the particles' current states.

```
#pragma omp parallel for if(parallelize)
// Initializes the 2 maps elements to false
for(int index = 0; index < n_particles; index++) {
    particles_on_track_map[index]= false;
    exited_particles_map[index] = false;
}
```

2. The second one is related to the particles' backtracking logic. It's the case in which they have to reach the exit by following the first exited particle's path.

```
#pragma omp parallel for if(parallelize)
// Backtracking the particles movements until they are on the solution
↪  path
// After that they follow the first exited particle's movements
for(int particle_index = 0; particle_index < n_particles;
↪  particle_index++) {
    if(!exited_particles_map[particle_index]) {
        // If the particle wasn't on the solution path in the previous
↪  iteration checks if it is now
        if(!particles_on_track_map[particle_index]) {
            for(int path_index = 0; path_index <
↪  exited_particle_path.size(); path_index++) {
                Coordinates coord = exited_particle_path[path_index];
                // The particle is now onto the right track
                if(particles.positions[particle_index].row ==
↪  coord.row && particles.positions[particle_index].col == coord.col)
↪  {
                    particles_on_track_map[particle_index] = true;
                    // Since indexes start from 0 an index of 5 means
↪  that we need to skip 6 elements, so + 1
                    // + 2 in the end as we need to skip the current
↪  position (already equal to the path index's one)
                    particles.paths[particle_index].clear();
                    int remaining_path_size =
↪  static_cast<int>(exited_particle_path.size());

↪  particles.paths[particle_index].reserve(remaining_path_size);

                    // Assigns the remaining correct steps to the
↪  current particle, in reverse order (useful for
                    // using the "update_coordinates" function as it
↪  is)
                    for(int index = remaining_path_size - 1; index >
↪  path_index; index--) {
                        particles.paths[particle_index]
↪  .push_back(exited_particle_path[index]);
                    }
                    break;
                }
            }
        }

        // Following the particle's steps back
        if(!particles.paths[particle_index].empty()) {
            Coordinates next_coords =
↪  particles.paths[particle_index].back();
            if(next_coords.row ==
↪  particles.positions[particle_index].row && next_coords.col ==
↪  particles.positions[particle_index].col) {
                particles.paths[particle_index].pop_back();
                next_coords = particles.paths[particle_index].back();
            }
            particles.update_coordinates(particle_index,
↪  p_get_next_move_from_path(particles.positions[particle_index],
↪  next_coords), true);

            // Displays the particle's steps
            if(show_steps && !maze_copy.empty()) {
                maze_copy[particles.positions[particle_index].row]
↪  [particles.positions[particle_index].col] = MAZE_PATH::PARTICLE;
                maze_copy[initial_position.row][initial_position.col]
↪  = MAZE_PATH::START;
                display_ascii_maze(maze_copy, size);
            }
        } else if (!exited_particles_map[particle_index]){
            #pragma omp critical
            n_exited_particles += 1;
            exited_particles_map[particle_index] = true;
        }
    }
}
```

There is only one critical section here, set to ensure that both the exited' particle's index and path variables are updated consistently. It doesn't actually matter which is the exited particle as there is only one solution path for the maze.

4. Performance analysis

As stated before both the sequential and the parallel version of the program share the same exact seeds for the random values generations. This has been done on purpose in order to consistently compare the executions' performance changes.
The two versions share the same exact generation and solution steps for each execution.
The testing phase has been focused on two hundred executions with different mazes' sizes and different numbers of particles.
The results in terms of execution speedup are shown in the following section.

4.1. Speedup

The speedup has been calculated with this formula:

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

where t_s represents the elapsed time to execute the sequential version in milliseconds and t_p the time elapsed for the parallel version.
In the following list each execution consists of

running both the sequential and the parallel version of the program.
The testing phase has been carried out by fixing the maze's sizes and the number of spawned particles as follows:

1. 50 executions with size fixed to 51 and 10,000 particles.
2. 50 executions with size fixed to 101 and 10,000 particles.
3. 50 executions with size fixed to 51 and 100,000 particles.
4. 50 executions with size fixed to 51 and 1,000,000 particles.

The following table shows the results of the timing measurements performed during the testing phase in order to evaluate the performance benefits obtained thanks to the parallelism.

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.

5. Conclusion

In this report i've shown the advantages of parallelism over the sequential execution of the code when the number of iterations is far bigger than the number of available threads.
The machine used to test the program and calculate the results has 16 physical threads.
An average speedup of 10.259 times has been achieved by implementing simple parallelism paradigms and by specifying compiler directives used to share the code execution across multiple threads, when necessary.
The source code and the results of the various executions are publicly available on github:
https://github.com/Scrayil/RandomMazeSolver