Comparison between sequential and parallel programming on Random Maze Solver with OpenMP

Alessio Chen
E-mail address
alessio.chen@stud.unifi.it

Abstract

This report aims to compare the performance of sequential and parallel programs to solve a randomly generated Maze.

1. Introduction

This report details a comparison between sequential and parallel implementations of a C++ algorithm, demonstrating the power of parallel programming. Leveraging the OpenMP multi-threading API, the project showcases parallelization's impact on performance through the example of random maze generation and solution.

1.1. Setup

The development was conducted on a Mac Mini M2 (2023) with Apple Silicon M2, an 8-core (4 performance + 4 efficiency) chip, running macOS Ventura 14.2.1.

1.2. Maze generation

This project implements a maze generation algorithm using a Depth-First Search (DFS) approach.

Represented as a grid of cells, the maze utilizes *#* for walls, *" "* for empty spaces, *S* for the starting point, and *E* for the exit.

The algorithm begins by initializing the grid and randomly selecting a starting cell. It then iteratively explores neighboring cells, carving paths by removing walls between adjacent cells until all cells are visited. This ensures a connected maze without isolated regions. Employing a stack to track visited cells and randomly selecting neighboring cells for exploration, the algorithm efficiently generates mazes of various sizes and complexities.

2. Code Explanation

In this section we will see all code involved into this comparison. The goal is to find the exit from a maze using the random movement of a particle. To represent a Particle, we utilize the struct as depicted in Listing 1. This struct comprises two main components Poistion that represents the current position of the particle within the maze and path that tracks the movements of the particle as it navigates through the maze.

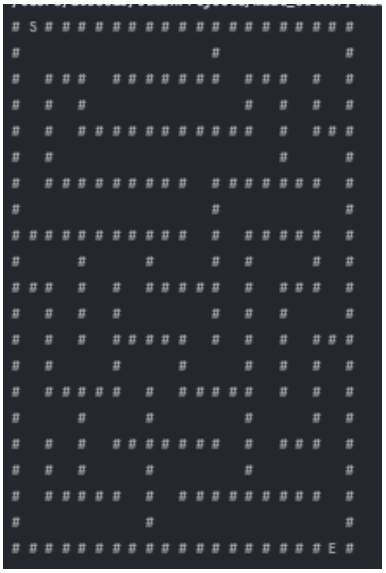

Figure 1: Example of a randomly generated 21x21 maze

The snippet in Listing 2 enables the simulation of random movements for a particle within the maze while ensuring that the moves are valid and consistent with the maze's constraints.

Listing 1: Particle representation

```
struct Point {
    int x, y;
};

struct Particle {
    Point pos;
    vector<Point> path;
};
```

Listing 2: Particle random movement

```
 bool isValidMove(int x, int y) {
    return x >= 0 && x < 2*M+1 &&
    y >= 0 && y < 2*N+1 &&
    maze[x][y] != '#' &&
    maze[x][y] != 'S';
}

vector<Point> getValidMoves(Particle &p) {
    vector<Point> moves ;
    int x = p.position.x;
    int y = p.position.y;

    if(isValidMove(x+1, y)) {
        moves.push_back({1,0});
    }

    if(isValidMove(x-1, y)) {
        moves.push_back({-1,0});
    }

    if(isValidMove(x, y+1)) {
        moves.push_back({0,1});
    }

    if(isValidMove(x, y-1)) {
        moves.push_back({0,-1});
    }

    return moves;

}

void randomMove(Particle &p) {

    vector<Point> moves = getValidMoves(p);
    int i = rand() % moves.size();
    int newX = p.position.x + moves[i].x;
    int newY = p.position.y + moves[i].y;

    p.position.x = newX;
    p.position.y = newY;

    p.path.push_back(p.position);

}
```

2.1. Constants

- **M** and **N**: These constants determine the dimensions of the maze grid. Specifically, the actual size of the maze is calculated as M x 2 + 1, representing the number of rows, and N x 2 + 1, representing the number of columns. Adjusting these constants allows for the creation of mazes with varying sizes, thereby influencing the complexity of the solving algorithms.
- **N_TESTS**: This constant determines the number of iterations for testing each algorithm. Multiple iterations ensure reliable performance measurements and account for variations in execution time.

2.2. Sequential implementation

The sequential solver function iteratively moves particles within the maze until at least one particle reaches the exit point. It simulates particle movements using the *randomMove* function until an exit is found for at least one particle. Once an exit is found, the function updates the solution by marking the path taken by the first particle to reach the exit.

Listing 3: Sequential implementation

```
bool isExitFound(Point &pos, Point &exit) {
    return pos.x == exit.x && pos.y == exit.y;
}


char **sequentialSolver(vector<Particle>
    particles,
    Maze maze, Point exit) {

    Particle firstToFindExit;
    bool solutionFound = false;
    char **solution = maze.maze;

    while (!solutionFound) {
        for (Particle &p: particles) {
            if (!solutionFound) {

                maze.randomMove(p);

                if (maze.isExitFound(p.
                    position, exit)) {
                    firstToFindExit = p;
                    solutionFound = true;
                }
            }
        }
    }

    for (Point &p: firstToFindExit.path) {
        solution[p.x][p.y] = '.';
    }

    return solution;
}
```

2.3. Parallel implementation

Within the parallel region defined by *#pragma omp parallel*, each thread is tasked with iterating over a designed subset of particles. During execution, each thread simulates particle movements using the *randomMove* function untill an exit is successfully located.

Upon discovering an exit for a particle, a thread enters a critical section to update the firstToFindExit variable and set *exitFound* to true. This critical section prevents race conditions and ensures that only one thread records the first particle to reach the exit. After completion of all thread operations, the function proceeds to update the solution by marking the path taken by the first particle to find the exit. This is achieved by iterating through the recorded path coordinates and updating the corresponding maze cells to indicate the traversed path. Finally, the function returns the updated maze solution.

Listing 4: Parallel implementation

```
char** parallelSolver(vector<Particle>
    particles,
Maze maze, Point exit, int threadNums) {

    char** solution = maze.maze;
    bool exitFound = false;
    Particle firstToFindExit;
    int pSize = particles.size() / threadNums;

#pragma omp parallel num_threads(threadNums)
    shared(particles, maze, exit, pSize)
    {
        int threadId = omp_get_thread_num();
        int start = pSize * threadId;
        int end = (threadId == threadNums - 1)
                    ? particles.size()
                    : start + pSize;

        while (!exitFound) {
            for (int i = start; i < end; i++)
                {
                if (!exitFound) {
                    maze.randomMove(particles[
                        i]);
                    if (maze.isExitFound(
                        particles[i].position,
                         exit)) {
#pragma omp critical
                        {
                            if (!exitFound) {

                                firstToFindExit
                                    =
                                    particles[
                                        i];

                                exitFound =
                                    true;
                            }
                        }
                    }
                }
            }
        }


        for (const Point& p : firstToFindExit.path
            ) {
            solution[p.x][p.y] = '.';
        }

        return solution;
    }
```

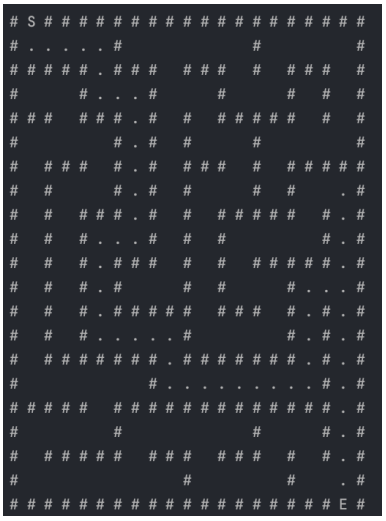

Figure 2: Example of a solution in 21x21 Maze

3. Tests

In the evaluation of the parallel solver, two distinct tests were conducted to analyze performance variations under different scenarios:

- **Thread Scalability Evaluation**: this test aimed to assess the impact of increasing the number of threads while maintaining a fixed number of particles.
- **Particle Scalability Evaluation**: in this test, the focus shifted to evaluating performance variations by altering the number of particles navigating the maze.

Both tests were conducted on a maze of size **51x51**, and each test was repeated **20 times** to ensure statistical robustness and reliability in the results.

3.1. Thread Scalability Evaluation

The test is conducted with 10,000 particles. As expected, the execution time decreases from 98,933 ms in the sequential version down to 20,561 ms in the parallel version with

8 threads (*Figure 3*). This performance improvement aligns with expectations, as parallel execution allows for concurrent processing of particles, leading to significant time savings compared to sequential execution.

Figure 4 illustrates the related speedups. Based on the measured times, the achieved speedups are promising, reaching up to 4.81168 on 8 threads.

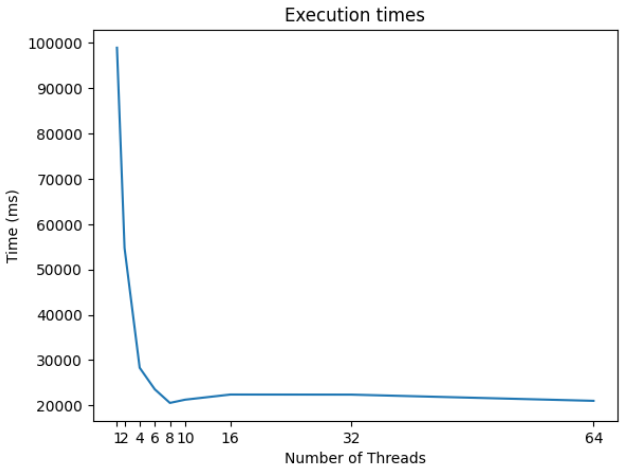

Figure 3: Time solving 51x51 maze with 10000 particles

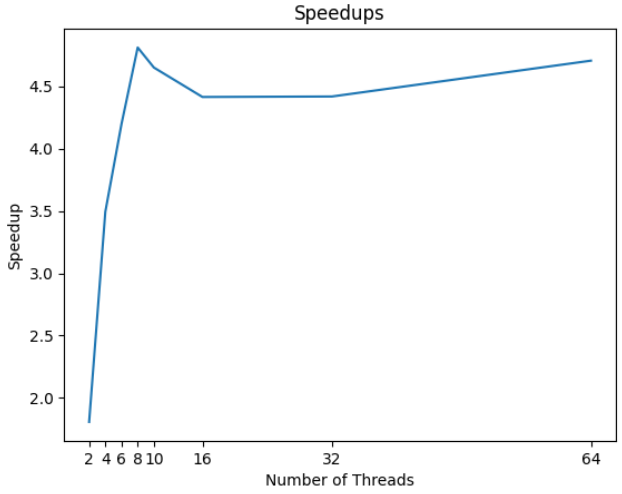

Figure 4: Speedups solving 51x51 maze with 10000 particles

3.2. Particle Scalability Evaluation

In the second test, the number of particles varied from 500 to 10,000, while the number of threads ranged from 1 to 32. As expected, the execution time increases from 8,135 ms to 98,933 ms in the sequential version as the number of particles increases (*Figure 5*).

In contrast, in the 8-thread parallel version, the execution time ranges from 1,829 ms to 20,561 ms, demonstrating a performance increase despite the variation in the number of particles. This confirms the effectiveness of parallelization in improving performance, even with varying workloads and thread counts.

Then we can see related speedups in *Figure 6*, as the number of threads increases, the speedup values generally rise, indicating that parallel execution leads to significant reductions in execution time compared to the sequential version. For instance, with 8 threads, the speedup values range from approximately 4.4 to 4.8, illustrating a substantial enhancement in efficiency.

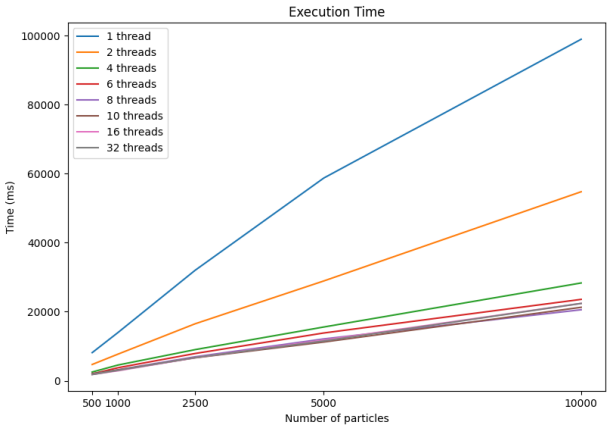

Figure 5: Time solving N Particles, 51x51 maze, increasing threads

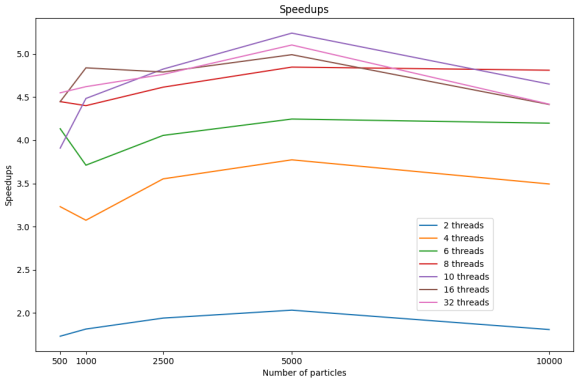

Figure 6: Speedups solving N particles, 51x51 maze, increasing threads

4. Conclusion

In conclusion, the comparison between sequential and parallel implementations of the Random Maze Solver using OpenMP highlights the significant performance improvements achieved through parallelization. The evaluation of both thread scalability and particle scalability demonstrates the effectiveness of parallel execution in reducing execution time and enhancing overall efficiency. In particular, the speedup analysis reveals promising results, with speedup values generally increasing with the number of threads, indicating efficient utilization of computational resources. However, it's essential to acknowledge the potential for diminishing returns at higher thread counts due to factors such as overhead and resource contention. Nonetheless, the overall trend underscores the benefits of parallelization in optimizing performance and scalability for maze-solving algorithms.