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

Plator Rama
plator.rama@edu.unifi.it

Abstract

This report aims to compare the performance of sequential and parallel programs in solving a randomly generated maze. The algorithms implemented do not focus on finding the optimal solution but rather on successfully reaching the exit of the maze. Performance evaluations were conducted using various maze sizes and different numbers of particles, allowing for a comprehensive analysis of both programming paradigms. By examining execution times and scalability, this study reveals the advantages and limitations of sequential versus parallel approaches, ultimately highlighting how parallel programming can enhance efficiency.

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. Hardware Setup Used

The following hardware setup was used to perform the tests aimed at demonstrating the power of parallel programming:

- **Processor**: AMD Ryzen 5 3600X
  - Architecture: Zen 2
  - Number of cores: 6
  - Number of threads: 12 (with Simultaneous Multithreading, SMT)
  - L1 Cache: 64 KB per core
  - L2 Cache: 512 KB per core (3 MB total)
  - L3 Cache: 32 MB shared
- **RAM**: 16 GB DDR4 @ 3200 MHz
- **Operating System**: *Windows 10 22H2*
- **Compiler**: *MinGW*
- **Libraries used**: OpenMP for parallelization

1.2. Maze generator

This project implements a maze generation algorithm using a Depth-First Search (DFS) approach. Represented as a 2D 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. The starting point and exit point have a fixed position, the starting point is at position [1, 1] instead the exit point is at position [mazeSize - 1, mazeSize - 1].

2. Code Explanation

In this section, we will see all the code involved in this comparison. The goal is to find the exit from a maze using the random movement of a particle.

2.1. Point and Particle

In the maze-solving algorithm, we use a structure to represent the particles that explore the maze. Each particle is defined as follows:

- **Point**: A simple structure representing a point in the maze with two coordinates:
  - *x*: The horizontal coordinate of the point.
  - *y*: The vertical coordinate of the point.
- **Particle**: Each particle holds the following information:

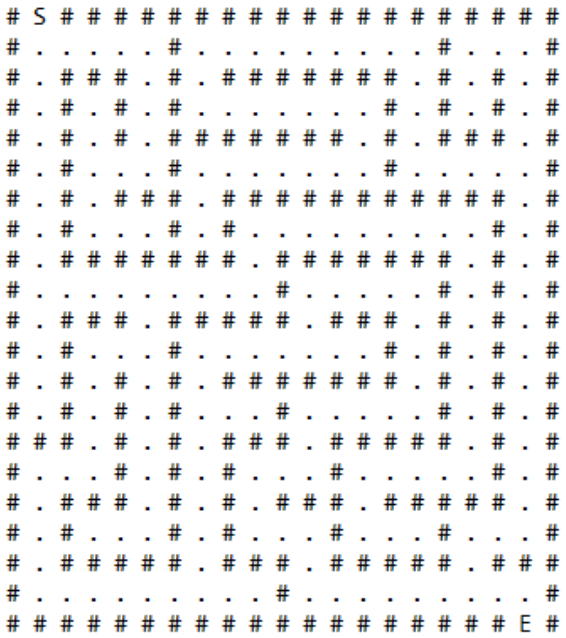

Figure 1. Example of a randomly generated maze with size 11x11.

  - *currentPosition*: A *Point* representing the particle's current location in the maze.
  - *path*: A list of *Point*s representing the path the particle has followed so far.

The particle moves through the maze by updating its *currentPosition* and appending each new position to the *path*.

2.2. Particle Movement

The particle navigation in the maze is governed by two main algorithms: *randomMove* and *getValidMoves*. These algorithms work together to ensure that particles explore the maze efficiently while avoiding invalid positions.

The *randomMove* algorithm (see Algorithm 2) is responsible for selecting a valid random move for the particle. It first retrieves a list of all valid moves by calling the *getValidMoves* function. Then, a move is chosen randomly from this list. The particle's position is updated according to the selected move, and the new position is recorded in the particle's path.

The *getValidMoves* algorithm (see Algorithm 1) generates all the valid moves a particle can make from its current position. It considers four possible directions (right, left, down, and up). For each direction, the *isValidMove* method checks if the move is valid by ensuring that:

- The new position is not outside the boundaries of the maze.
- The new position is not a wall.
- The new position is not the particle's starting position.

Only moves that satisfy all these conditions are considered valid and are added to the list of valid moves.

Algorithm 1 Get Valid Moves for Particle

```
1:  procedure GETVALIDMOVES(Particle particle)
2:      possibleMoves ← [(1, 0), (-1, 0), (0, 1), (0, -1)]   ▷ right, left, down, up
3:      validMoves ← []
4:      x ← particle.currentPosition.x
5:      y ← particle.currentPosition.y
6:      for all move in possibleMoves do
7:          if ISVALIDMOVE(x + move.x, y + move.y) then
8:              Append move to validMoves
9:          end if
10:     end for
11:     return validMoves
12: end procedure
```

Algorithm 2 Random Move for Particle

```
1: procedure RANDOMMOVE(Particle particle)
2:     validMoves ← GETVALIDMOVES(particle)
3:     move ← Randomly select one element from validMoves
4:     newX ← particle.currentPosition.x + move.x
5:     newY ← particle.currentPosition.y + move.y
6:     particle.currentPosition ← (newX, newY)
7:     Append particle.currentPosition to particle.path
8: end procedure
```

2.3. Sequential Solver Algorithm

The `sequential_solver` algorithm is designed to solve a maze by utilizing multiple particles that explore the maze simultaneously.

The `sequential_solver` algorithm begins by initializing a pointer, `winning_particle`, to keep track of the particle that successfully finds the exit. The main loop continues until a winning particle is found. Within this loop, each particle in the vector `particles` is processed in turn.

For each particle, the algorithm executes the following steps:

1. Call the `randomMove` function to update the particle's position based on valid moves within the maze.
2. Check if the current particle has reached the exit using the `isExitFound` method. If so, the pointer `winning_particle` is updated to point to this particle.

Once a winning particle is identified, the algorithm exits the loop. Finally, if a winning particle was found, the

Algorithm 3 Sequential Solver

```
1: procedure SOLVEMAZE(Maze, vector<Particle>)
2:     winning_particle ← nullptr
3:     while winning_particle is nullptr do
4:         for each particle do
5:             if winning_particle is not nullptr then
6:                 continue
7:             end if
8:             if isExitFound(particle.currentPosition) then
9:                 winning_particle ← address of particle
10:            end if
11:        end for
12:    end while
13:    if winning_particle is not nullptr then
14:        maze.drawPath(winning_particle);
15:    end if
16: end procedure
```

`drawPath` function is called to visualize the successful path taken by the particle in the maze.

2.4. Parallel Solver Algorithm

The `parallel_solver` algorithm leverages parallel processing to solve a maze using multiple particles. By distributing the work across several threads, this algorithm aims to find the exit more efficiently.

The `parallel_solver` algorithm begins by calculating the partition size based on the number of threads specified. It initializes a pointer `winning_particle` to track which particle successfully finds the exit and a boolean variable `exit_found` to indicate whether the exit has been reached.

Using OpenMP, the algorithm creates multiple threads that each handle a portion of the particles:

1. Each thread computes its starting and ending indices based on its thread ID and the partition size.

2. The main loop runs until an exit is found. Within this loop, each thread processes its assigned particles:

   - Each particle makes a random move within the maze.
   - The algorithm checks if the particle has reached the exit.
   - If a particle finds the exit, a critical section ensures that only one thread can assign the winning particle and set `exit_found` to true, preventing other threads from continuing further.

Algorithm 4 Parallel Solver

```
1: procedure PARALLEL_SOLVER(Maze, vector<Particle>, int threads_number)
2:     partition_size ← particles.size() / threads_number
3:     winning_particle ← nullptr
4:     exit_found ← false
5:     #pragma omp parallel num_threads(threads_number) shared(particles, maze, partition_size)
6:         threadId ← omp_get_thread_num()
7:         start_partition ← partition_size × threadId
8:         end_partition ← (threadId == threads_number - 1) ? particles.size() : start_partition + partition_size
9:     while !exit_found do
10:        for i = start_partition to end_partition do
11:            if exit_found then continue
12:            end if
13:            maze.randomMove(particles[i])
14:            if maze.isExitFound(particles[i].currentPosition) then
15:                #pragma omp critical
16:                if !exit_found then
17:                    winning_particle ← address of particles[i]
18:                    exit_found ← true
19:                end if
20:            end if
21:        end for
22:    end while
23: end procedure
```

After the threads complete their execution, the algorithm checks if a winning particle was found. If so, it invokes the `drawPath` function to visualize the successful path in the maze.

In Figure 2 a random maze solved.

3. Test

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

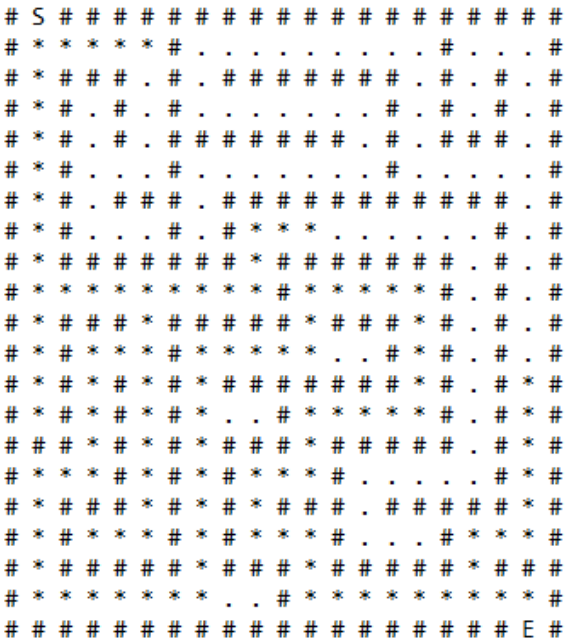

Figure 2. Example of a randomly generated maze with size 11x11, solved.

shifted to evaluating performance variations by altering the number of particles navigating the maze.

Both tests were conducted with different **maze size**((5x5), (5x5), (10x10), (20x20), (30x30), (40x40), (50x50)), and each test was repeated **25 times** to ensure statistical robustness and reliability in the results.

3.1. Thread Scalability Evaluation

The test is conducted with 2,000 particles. As expected, the execution time decreases in the parallel version.

The graph shown in Figure 3 illustrates the scalability of the algorithm with respect to the number of threads in terms of execution time (in milliseconds).

As seen in the graph, the execution time decreases significantly as the number of threads increases, highlighting the notable reduction in computation time due to parallelism. In particular, at the beginning, increasing the number of threads has a substantial impact on performance, with a sharp decrease in execution time.

However, as the number of threads continues to grow, the performance gains start to level off, showing a more gradual decrease in execution time. This behavior is expected, as beyond a certain point, the overhead introduced by thread management and synchronization begins to negatively affect overall performance.

Figure 4 illustrates the related speedups. Based on the measured times, the achieved speedups are promising.

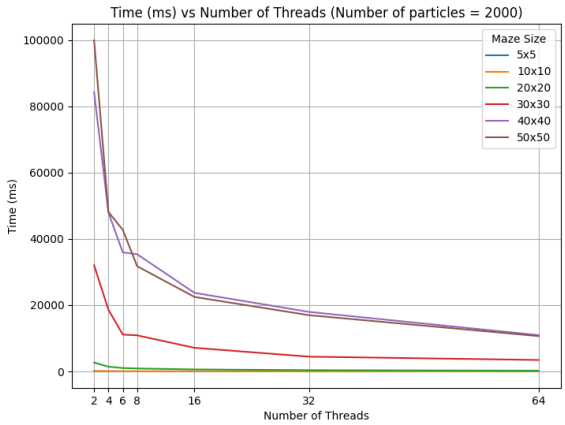

Figure 3. Thread scalability: with 2000 particles

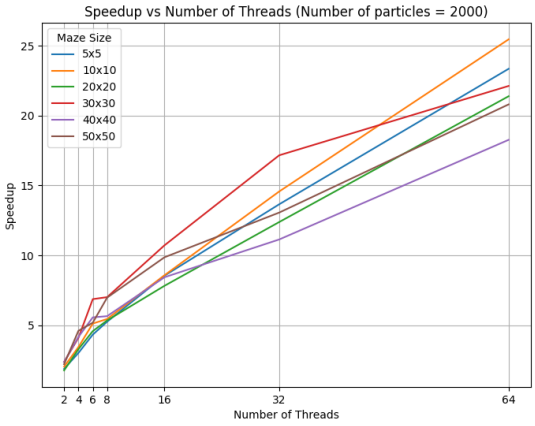

Figure 4. Speedups: with 2000 particles

3.2. Particle Scalability Evaluation

In the second test, the number of particles varied from 100 to 2,000, while the number of threads ranged from 1(sequential algorithm) to 64. As expected, the execution time increases from 18,378 ms to 221,673 ms in the sequential version as the number of particles increases (Figure 5).

In contrast, in parallel version, the execution time ranges from 1,100 ms to 99,923 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.

4. Conclusion

In conclusion, the comparison between sequential and parallel implementations of the Random Maze Solver using OpenMP highlights the significant performance improve-

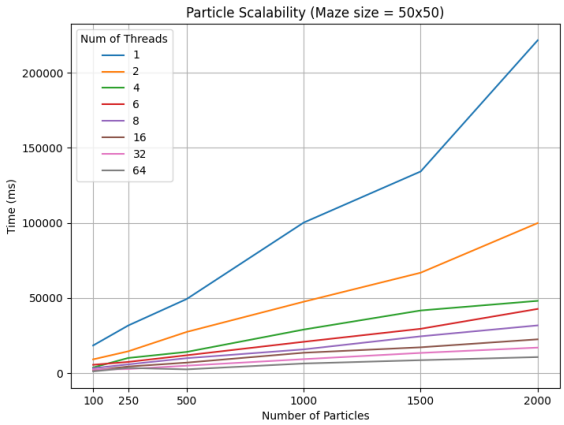

Figure 5. Particle Scalability: with maze size 50x50

ments 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.