Performance comparison of a maze solver with sequential and parallel implementations

Federico Chiesa
7108176

Giovanni Casini
7108181

Abstract

The aim of this paper is to compare the performance of a simple maze-solving algorithm to show the benefits of parallelization of code.
We are also comparing the performance of different parallelization methods, in particular the native C++ Threads and OpenMP.

1. Introduction

The algorithm we have developed aims to solve a maze using particles by taking the maze image as input, along with coordinates that indicate from where the solver should start. The movement of these particles is random, and in our test is a discrete uniform random variable over the range of $[-3, 3]$ in both X and Y axes. The particles are moved sequentially in a `for` loop, so for every loop all the particles will have made a move. Once the first particle reaches the end of the maze the algorithm is stopped, we mark that particle as the first finisher, and we output the time it took to solve, and an image that shows which path the particle took to solve the maze.

We wanted our code not to use external libraries to make it easier to run for others and to prevent compatibility issues with certain systems. To achieve this, we decided to use images in the PGM format[1]. We used the ASCII P2 format so that we could work with the image as a matrix of strings that represent the level of gray of every pixel, making the code easier to write and read, and avoiding external libraries.

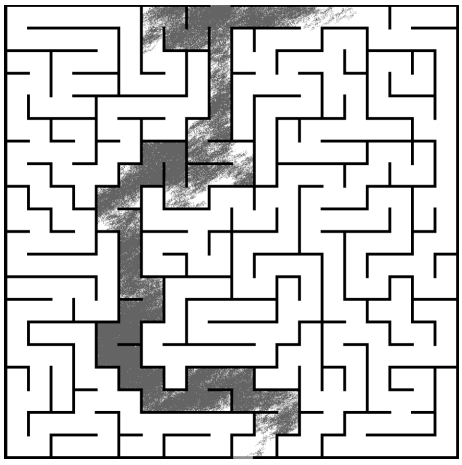


Figure 1. Result

2. Implementation

We began implementing the algorithm with a basic data structure for each particle that consists of a pair of `std::vectors` that contain the x and y coordinates that the particle traveled through during the execution of the code.

We then focused on how to load the image in a way that we could transform it into a matrix of pixels. To do this, we stripped the PGM headers, and then we got each line of data from the file using the `getLine()` method and concatenating each line to a `std::string`. We then split the string by spaces, obtaining an array of strings with each value representing the gray value of the pixel. At this point we can treat the array as a matrix by indexing it as is shown in the code listing

[1] `https://en.wikipedia.org/wiki/Netpbm`

below.

```
std::string maze;
std::string textLine;
std::ifstream fileReader((filename + ".pgm"));

std::string word;
std::vector<std::string> mazeVector;

while (getline(fileReader, textLine))
        maze = maze + textLine + " ";
fileReader.close();
if (maze.length() != 0)
{
    std::stringstream ss(maze);
    while (ss >> word)
        mazeVector.push_back(word);
}
```

Listing 1. Image loading code

At the beginning of both the sequential and parallelized algorithms we create and initialize an array of particles of the amount specified in the parameters.

Once we were able to do these preliminary operations, we were able to concentrate on the main loop that actually runs the algorithm, which is slightly different in each implementation due to the how parallelized code works.

2.1. Sequential implementation

For the sequential algorithm we used a `for` loop inside a `while(true)` loop. The `for` loop iterates over the array of particles and for each particle it calculates the next position by summing a random integer between -3 and 3 to the actual position of the particle. If the calculated point is on a wall, the position is recalculated until a valid point is found.

When the first particle reaches the end of the maze, which is detected based on the fact that it's a shade of gray instead of black or white, we measure the time it took using the `chrono` library and then we write the path that the particle took to a new PGM file.

Below is a listing of the sequential loop:

```
std::chrono::steady_clock::time_point begin = std
    ::chrono::steady_clock::now();
while (true)
{
    for (int i = 0; i < ballArray.size(); i++)
    { // random movement of the balls in both X
    and Y axes
        int nextX = ballArray[i].x.back() + uni(
    rng);
        int nextY = ballArray[i].y.back() + uni(
    rng);
        while (nextX < 0 || nextY < 0 ||
    mazeVector[nextY * (imageWidth - 1) + nextX]
    == "0")
        {
            nextX = ballArray[i].x.back() + uni(
    rng);
            nextY = ballArray[i].y.back() + uni(
    rng);
        }
        ballArray[i].x.push_back(nextX);
        ballArray[i].y.push_back(nextY);
        if (mazeVector[nextY * (imageWidth - 1) +
     nextX] != "0" && mazeVector[nextY * (
    imageWidth - 1) + nextX] != "255")
        { // if the ball is at the end of the
    maze...
            std::chrono::steady_clock::time_point
     end = std::chrono::steady_clock::now();
            long long time = std::chrono::
    duration_cast<std::chrono::milliseconds>(end
    - begin).count();
            std::cout << time << " [ms]" << std::
    endl;
            for (int j = 0; j < ballArray[i].x.
    size(); j++)
                mazeVector[ballArray[i].y[j] * (
    imageWidth - 1) + ballArray[i].x[j]] = "100";
            std::ofstream file("result.pgm");
            if (file.is_open())
            {
                file << mazeVector[0] + "\n" +
    mazeVector[1] + " " + mazeVector[2] + "\n" +
    mazeVector[3] + "\n"; // add headers to new
    file
                std::string result;
                for (int k = 4; k < mazeVector.
    size(); k++)
                    result += mazeVector[k] + " "
    ; // separate pixels with a space
                file << result;
      // then add the modified image data
                file.close();
            }
            return;
        }
    }
}
```

Listing 2. Sequential loop

2.2. Parallel implementations

2.2.1 C++ Threads

For this implementation we used the same `for` loop of the sequential algorithm. The only change we made is the condition of the `while` loop. Since we're running multiple threads if we were to run the same `for` loop as the sequential implementation we wouldn't get the time of the first particle to finish on the fastest thread, but instead

we would get the times for the first particle for every thread, and we don't need this information.

To stop all the other threads once the fastest one finishes, we used a shared boolean variable called `finished`, protected by a mutex to prevent two or more threads from setting the variable at once, and once this variable is set to `true` the while condition of all the remaining active threads, which is `while(!finished)`, stops all other threads, and we get only one time measurement.

Below is the listing of the modifications we made to the sequential loop to make it work with C++ Threads:

```
std::chrono::steady_clock::time_point begin = std
    ::chrono::steady_clock::now();
while (finisher == NULL && finished == false)
{
    for (int i = 0; i < ballArray.size(); i++)
    {
    [...]
        if (mazeVector[nextY * (imageWidth - 1) +
     nextX] != "0" && mazeVector[nextY * (
    imageWidth - 1) + nextX] != "255")
        { // if the ball is at the end of the
    maze...
            mutex.lock();
            finished = true;
            mutex.unlock();
            [...]
            return;
        }
    }
}
```

Listing 3. C++ Threads changes of the sequential loop

2.2.2 OpenMP

The OpenMP implementation works very differently from C++ Threads. It uses pragma marks to indicate which parts of the code should be parallelized and in which way. In our implementation we made the following changes to parallelize the code:

```
    volatile bool abort = false;
    while (!abort)
    {
#pragma omp parallel for num_threads(4) schedule(
    static)
        for (int i = 0; i < ballArray.size(); i
    ++)
        { // random movement of the balls in both
     X and Y axes
            if (!abort)
            {
                [...]
                if (mazeVector[nextY * (
    imageWidth - 1) + nextX] != "0" && mazeVector
    [nextY * (imageWidth - 1) + nextX] != "255")
                { // if the ball is at the end of
     the maze...
                    abort = true;
                    [...]
                }
            }
            else
                continue;
        }
    }
```

Listing 4. OpenMP changes to the sequential loop

We first made a boolean variable called `abort` to make all the other threads quit when the first particle reaches the end. We could't use the return method we used in the C++ Threads because due to how OpenMP works, `return` and `break` are not allowed in parallelized blocks. We then realized that sometimes we'd still get more than one time reading from the algorithm, and we realized that that was due to the fact that despite the fact that we set the abort variable, so all threads would quit, that didn't stop the `for` loop and if during the remaining cycles of the `for` loop a particle from another thread would reach the end it would print more than one result.

To avoid this behavior without using `break`, which as we said isn't allowed, we put an `if` statement inside the `for` loop that would simply `continue`, so in practice it would do nothing, until the end of the `for` loop, and then would quit due to the `while` condition.

In the pragma mark, `omp parallel for` means that the `for` loop is parallelized, `num_threads(4)` means that instead of automatically determining the number of threads to use, we want to use 4, and `schedule(static)` means that OpenMP deals out an equal number of `for` iterations to each thread. It's possible to manually specify the size of the chunks in which the iterations get split into, but we decided to leave it to OpenMP to decide automatically so even if we change the number of particles it would automatically split into equal parts.

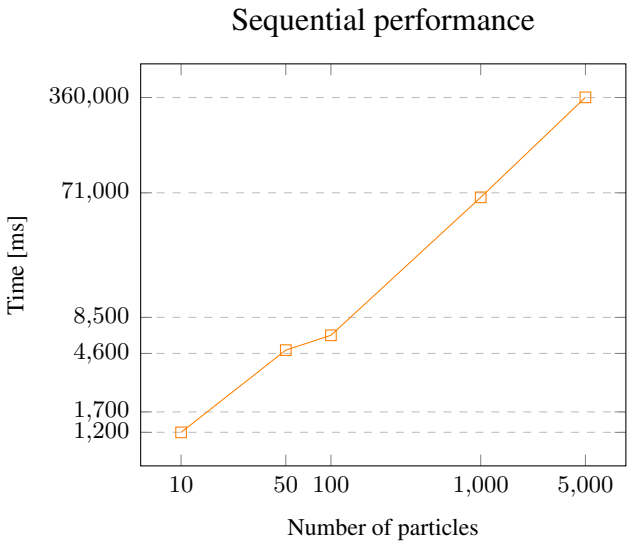
3. Tests

To test the speed of the different implementations of the algorithm, we ran them multiple times with a different number of particles and then made an average of the times to make the tests as precise as possible. Since the algorithm is random by nature, if we didn't do this we would risk to get inaccurate results due to the particles taking wrong paths or getting stuck in dead ends for a long time.

In the parallel implementations we kept the overall number of particles the same as in the sequential implementation e.g. if we make a run with 16 particles in the sequential implementation, in the C++ Thread implementation we'd run 4 threads with 4 particles each.

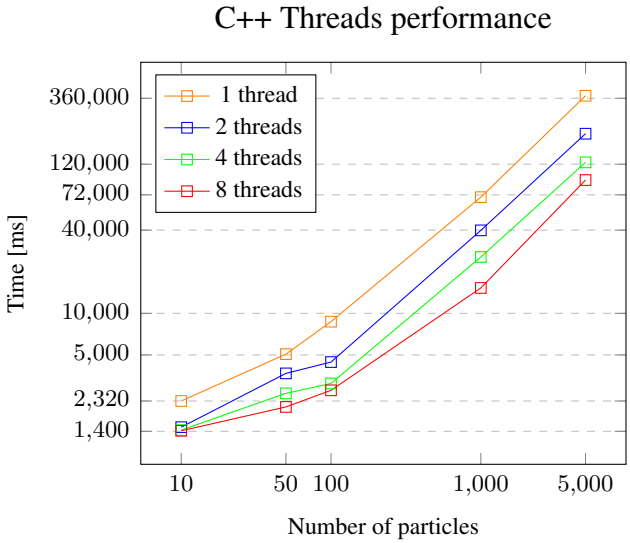
We ran all the tests on a Ubuntu 22.04 VM using VMWare Player with 16GB RAM and a maximum of 8 cores from the host machine, which has an Intel Core i7-1255U CPU.

3.1. Sequential Implementation

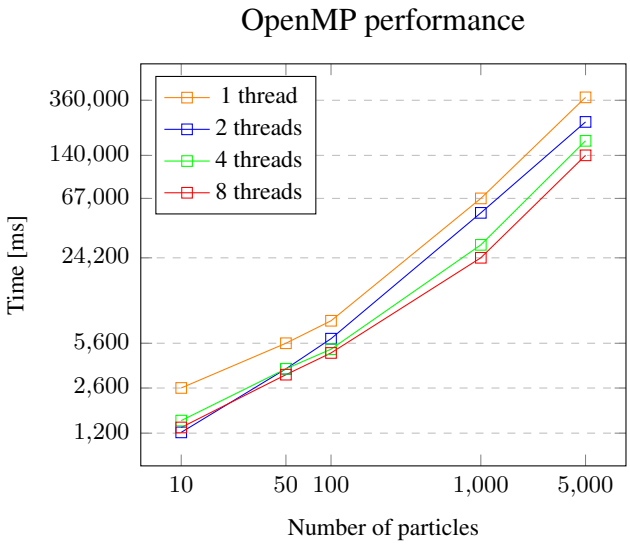


3.2. Parallel Implementations

3.2.1 C++ Threads

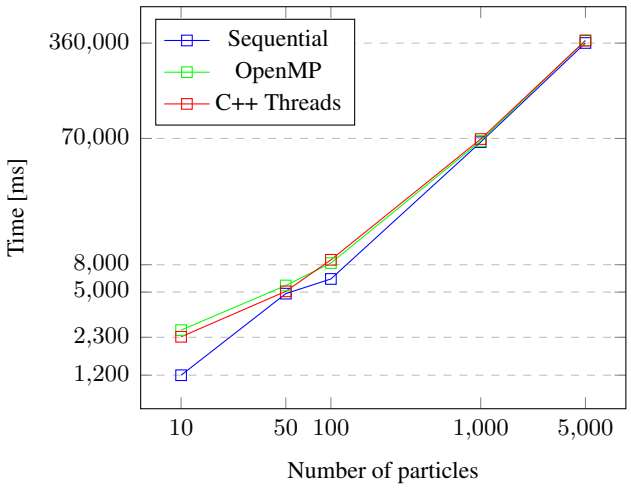


3.2.2 OpenMP

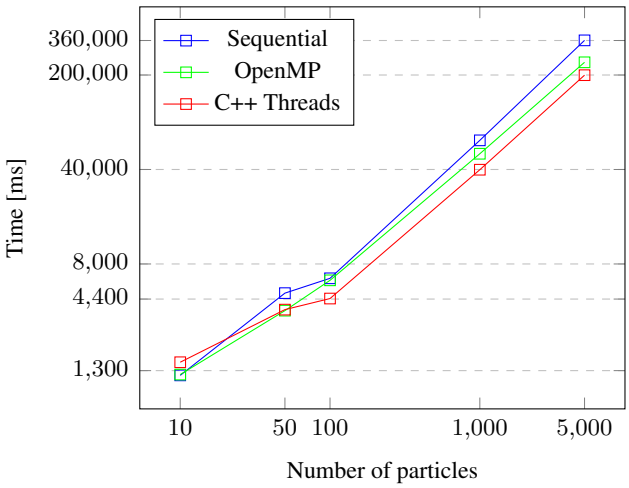


4. Conclusions

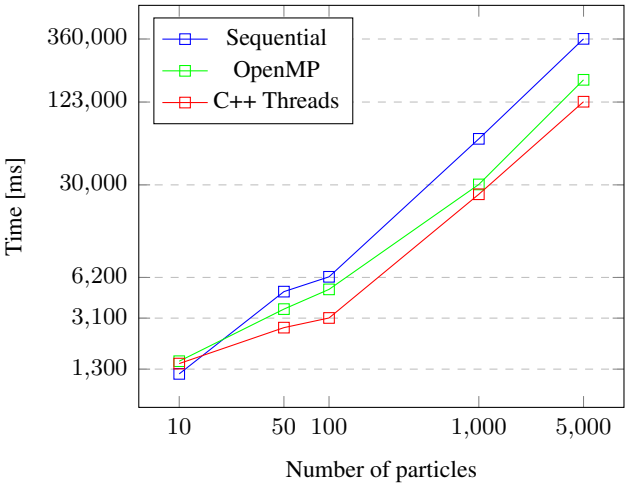
Comparison 1 thread



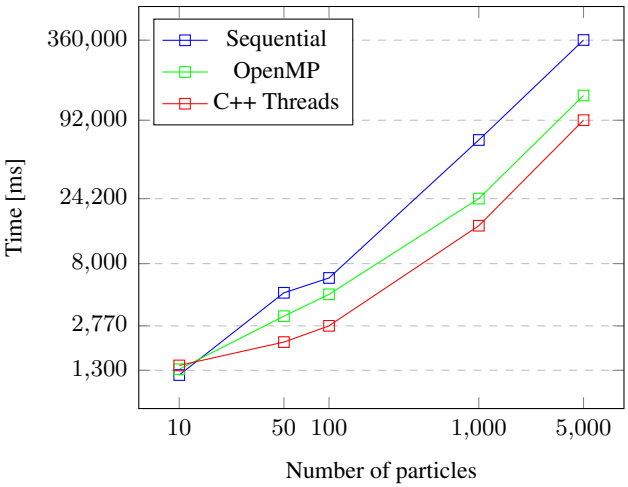
Comparison 2 threads



Comparison 4 threads

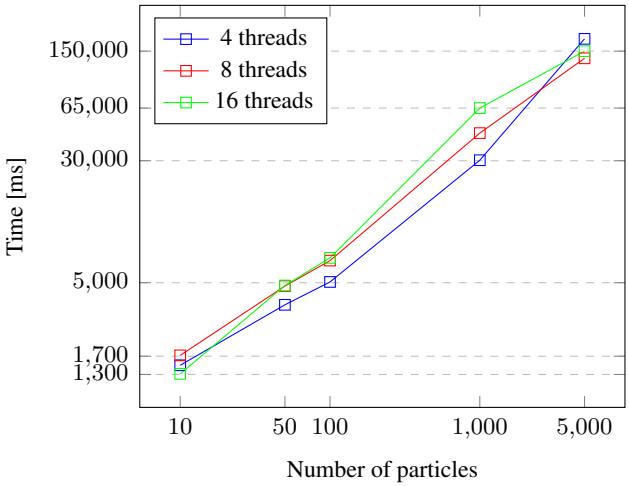


Comparison 8 threads



From what we've been able to observe during testing, the sequential implementation is, as expected, the worst performing. The C++ Threads implementation is the one that scaled the best out of the two parallel implementations by far, with a maximum speedup of 2.6 with 4 threads, compared to the slightly worse results we got using OpenMP with a maximum speedup of 2.2 with 4 threads.

OpenMP 4 cores



C++ Threads 4 cores