| Riccardo Marconi  7151651  riccardo.marconi@edu.unifi.it | Second Author  E-mail address  secondauthor@i2.org |
| --- | --- |

**Abstract**

In this paper, I present the formulation and the study of a C++ algorithm for the construction of a maze solver that employs the random motion of particles. The purpose is to locate the exit of a maze with different algorithms mimicking the random movement of particles inside the maze with a serial and parallel computational model. The latest is associated with the fact that the serial implementation concerns a single particle that is radically moved at random until a target exit is reached. On the other hand, the parallel implementation with the use of OpenMP allows for simulating numerous particles at once, which exponentially contributes to the efficiency of the given search. The algorithmic approach, issues in implementing parallelism, and means of performance assessment; advantages of parallelism and governing methods for solving computationally intensive applications such as solving mazes are also discussed in the report.

1. Introduction

It is a well-known fact that mazes are extensively used in computer science and robotics as a basis for evaluating the performance of specific search heuristics and optimization strategies. The classical algorithms, namely DFS and BFS, are well researched and spreading widely because its logical approach to solve the problem of maze. However, these methods can be very demanding on the computational resources and time consuming especially when trying to solve large and more complex mazes since these methods require one to explore each possible path in a systematic manner to get to the exit.

In this report, the specific type of other algorithm for solving mazes is examined: the random movement of particles contains a stochastic component that can assist in accessing the exit. Unlike fixed path methods such as DFS and BFS, this method uses stochastic properties of random walks to solve the maze. The crux plan focuses on the initiation of a large population of particles at the starting point and allowing them to wander within the maze blindly until one gets out of the maze. This is done both in the sequential manner

and with one particle at a time and with several particles at a time. The serial implementation starts with one particle located at the maze start, and this particle moves around randomly to get to the exit. To perform backtrack, this algorithm uses a stack to follow the path that a particle was moving through until it finds an exit. The parallel implementation, conversely, uses OpenMP of simulating several particles at the same time to cut the requisite time for identifying the exit even shorter. Each thread corresponds to a particle, and there is a ‘stop found exit’ flag to avoid waiting when some of particles find the exit; parallel processing helps to solve the maze faster.Problem

The specific context of this problem entails moving a particle from a pre-designated ‘start point’ to an ‘exit point’ in a 2D grid that simulates a maze. The maze described here has walls and obstacles that are set on the one hand and an environment filled with space in which the particle can traverse through on the other. The problem is finding a probabilistic path from the start node to the exit avoiding other nodes on the given map as far as possible unlike the other more deterministic search methods like Depth First Search (DFS) or Breadth First Search (BFS).

1. Problem

The specific context of this problem entails moving a particle from a pre-designated ‘start point’ to an ‘exit point’ in a 2D grid that simulates a maze. The maze described here has walls and obstacles that are set on the one hand and an environment filled with space in which the particle can traverse through on the other. The problem is finding a probabilistic path from the start node to the exit avoiding other nodes on the given map as far as possible unlike the other more deterministic search methods like Depth First Search (DFS) or Breadth First Search (BFS).

In this case, the particle is only able to move from the cell into any of the neighboring empty spots, but it cannot move diagonally—it can go only up, down, left or right. The strategy utilized to solve the maze is the stochastic approach in which the particle attempts to navigate through the maze by making a random selection ofthe moves at different steps. It goes about this randomly and could take a really long time to get out by bouncing off the bottom, right and left, before realizing it has hit a wall, go back and start all over.

The basic goal is to prove that with enough randomly selected attempts, one can find a path towards the exit. As we conducted a comparison of one serial computation and multiple parallel computations, this report look into the effectiveness of the random movement strategy with the hope of enhancing our understanding on how parallel processing can have a massive impacts on time taken to solve large structures of mazes.

1. Algorithmic Approach

3.1. Serial Implementation

The serial implementation begins with the placement of the first particle at the origin of the maze. The particle moves randomly in one of four possible directions: forward, backward, left or right within 30cm in the X and Y axis and up and down within 15cm along the Z axis. In case the particle crosses into a forbidden path, it turns back to the previous cell it was in as such paths are not acceptable. This random movement will continue in this manner through cycles until the particle is at the exit.

To keep record of the journey made by the particle in quest of the target zone, stack data structure is used. This is conveyed through each position that the particle assumes, is placed onto the stack which helps in the recording of the path. At the end of this journey, the stack holds the exact sequence of the moves accomplished, and thus displays the working path. This method guarantees the recording of the entire record of traversal thus enabling the back track and the path from the entrance to exit once the maze is solved.

3.2. Parallel Implementation

The parallel implementation is accomplished using OpenMP to run multiple particles in parallel while the simulation is in process. Every particle, being the same as in the process described in section 3.1, behaves independently and makes random movements: moves randomly within four directions and changes its direction back when it encounters a wall. The search effort is distributed when simulated many particles at once and the finding of the exit is made more likely in less time.

A common flag is used at the end of the simulation if any of the particle has escaped to return and set the flag. This flag guarantees that all the threads cease their functionalities the moment they come across the exit and thus avoiding performing unnecessary operations after its detection. This is made easy by the parallel processing approach that uses threads to perform computation at very high speeds other than what was experienced under the serial approach. That is why the efficiency shown proves the use of parallel computation applied to search algorithms in complex structures like mazes.

1. Implementation details

4.1. Maze Representation

The maze can be modeled as a two-dimensional integer array using the following conventions, where 0 is empty space and 1 is walls. The positions referred to as st and end are therefore the specific co-ordinates in this array.

const int WIDTH = 10;

const int HEIGHT = 10;

enum Direction { UP, DOWN, LEFT, RIGHT };

int maze [HEIGHT][WIDTH] = {

{1, 1, 1, 1, 1, 1, 1, 1, 1, 1},

{1, 0, 0, 0, 1, 0, 1, 0, 0, 1},

{1, 0, 1, 0, 1, 0, 1, 1, 0, 1},

{1, 0, 1, 0, 0, 0, 0, 1, 0, 1},

{1, 0, 1, 1, 1, 1, 0, 1, 0, 1},

{1, 0, 0, 0, 0, 0, 0, 1, 0, 1},

{1, 1, 1, 1, 1, 1, 0, 1, 0, 1},

{1, 0, 0, 0, 0, 1, 0, 1, 0, 1},

{1, 0, 1, 1, 0, 1, 0, 0, 0, 1},

{1, 1, 1, 1, 1, 1, 1, 1, 1, 1}

};

4.2. Serial Algorithm

Such management is accomplished by using a serial algorithm in a function that positions the particle at the start, moves it randomly in the maze, and determines if the exit has been entered. In order to use backtracking the path is stored by every node in the stack as a look-up list.

struct Position {

int x;

int y;

};

Position start = {1, 1};

Position exitPos = {8, 8};

bool isExit(Position pos) {

return pos.x == exitPos.x && pos.y == exitPos.y;

}

bool isValidMove(Position pos) {

return pos.x >= 0 && pos.x < WIDTH && pos.y >= 0 && pos.y < HEIGHT && maze[pos.y][pos.x] == 0;

}

void randomMove(Position &pos) {

Direction dir = static\_cast<Direction>(rand() % 4);

switch (dir) {

case UP: pos.y--; break;

case DOWN: pos.y++; break;

case LEFT: pos.x--; break;

case RIGHT: pos.x++; break;

}

if (!isValidMove(pos)) {

switch (dir) {

case UP: pos.y++; break;

case DOWN: pos.y--; break;

case LEFT: pos.x++; break;

case RIGHT: pos.x--; break;

}

}

}

void findExit() {

srand(time(0));

Position pos = start;

std::stack<Position> path;

path.push(pos);

while (!isExit(pos)) {

randomMove(pos);

path.push(pos);

}

std::cout << "Exit found at (" << pos.x << ", " << pos.y << ")\n";

std::cout << "Path to exit:\n";

while (!path.empty()) {

Position step = path.top();

path.pop();

std::cout << "(" << step.x << ", " << step.y << ")\n";

}

}

int main() {

auto start = std::chrono::high\_resolution\_clock::now();

findExit();

auto end = std::chrono::high\_resolution\_clock::now();

std::chrono::duration<double> elapsed = end - start;

std::cout << "Serial execution time: " << elapsed.count() << " seconds\n";

return 0;

}

4.2.1. Output

A black screen with white text

Description automatically generated

4.3. Parallel Algorithm

The parallel algorithm developed for this program uses OpenMP to generate several threads where each of them replicates a random moving particle within the maze. The common flag and critical sections are incorporated in order to prompt the end of the simulation as soon as any of the particles arrive at the exit.

struct Position {

int x;

int y;

};

Position start = {1, 1};

Position exitPos = {8, 8};

bool isExit(Position pos) {

return pos.x == exitPos.x && pos.y == exitPos.y;

}

bool isValidMove(Position pos) {

return pos.x >= 0 && pos.x < WIDTH && pos.y >= 0 && pos.y < HEIGHT && maze[pos.y][pos.x] == 0;

}

void randomMove(Position &pos) {

Direction dir = static\_cast<Direction>(rand() % 4);

switch (dir) {

case UP: pos.y--; break;

case DOWN: pos.y++; break;

case LEFT: pos.x--; break;

case RIGHT: pos.x++; break;

}

if (!isValidMove(pos)) {

switch (dir) {

case UP: pos.y++; break;

case DOWN: pos.y--; break;

case LEFT: pos.x++; break;

case RIGHT: pos.x--; break;

}

}

}

void findExitParallel(int numParticles) {

srand(time(0));

bool found = false;

Position finalPos;

#pragma omp parallel num\_threads(numParticles)

{

Position pos = start;

while (!found) {

randomMove(pos);

if (isExit(pos)) {

#pragma omp critical

{

if (!found) {

found = true;

finalPos = pos;

}

}

}

}

}

std::cout << "Exit found at (" << finalPos.x << ", " << finalPos.y << ")\n";

}

int main() {

std::vector<int> particleCounts = {1, 10, 100, 500, 1000}; // Different number of particles for testing

for (int numParticles : particleCounts) {

auto start = std::chrono::high\_resolution\_clock::now();

findExitParallel(numParticles);

auto end = std::chrono::high\_resolution\_clock::now();

std::chrono::duration<double> elapsed = end - start;

std::cout << "Parallel execution time with " << numParticles << " particles: " << elapsed.count() << " seconds\n";

}

return 0;

}

4.3.1 Output

A screenshot of a computer screen

Description automatically generated

4.4. Performance Evaluation

Performance evaluation determines the exact time for execution between the serial and parallel implementations to arrive at the exit in various mazes. The first parameter is the time it takes an agent to complete the simulation until the latter finds the exit. In the serial implementation, only one particle moves through the maze using a random method, which may prove very time consuming, making many slow backtracks.

On the other hand, the parallel implementation can directly search for multiple particles at the same time by dividing the search work among several threads. It is useful for precisely the same reason that parallelism enhances the likelihood of rapidly discovering the exit as more particles operate throughout the Maze concurrently. The plan is to obtain a ’speed-up’ through adding more particles so that less time will be spent in the grander scheme of things to solve the maze.

The findings are expected to reveal that the parallel approach works more effectively when compared to the serial method at higher particle count. This proves the effectiveness of parallel processing if applied in solving search problems in enormous and intricate maze structures.

| Serial Algorithm | Parallel Algorithm |
| --- | --- |
| 0.219318 seconds | Parallel execution time with 1 particle: 0.0013578 seconds  Parallel execution time with 10 particles: 0.0010642 seconds  Parallel execution time with 100 particles: 0.000874 seconds  Parallel execution time with 500 particles: 0.0013041 seconds  Parallel execution time with 1000 particles: 0.0007907 seconds |

A graph with a line graph

Description automatically generated

1. Conclusion

This report shows how random particle movement may be employed in solving mazes, and this is one of the best illustrations of the optimal benefits that parallel processing brings about in lowering computation demands. This type of implementation is quite simplistic and locates the exit gradually because of the sequential approach, single process exploration, and utilization of random strategies. However, the parallel mode takes advantage of many particles and the processing of several particles simultaneously which makes the time taken to solve the maze much smaller.

This result unambiguously demonstrates that PE significantly improves the performance, and that the optimization effectively scales with the incremented particle number, which indicates that the parallel processing approach is well suited for solving large-scale maze problems. It can also be fine-tuned further by modifying the general random movement algorithm or by implementing better backtracking methods. Moreover, the idea used here can be used in more complex form for other search and optimization problems and further research, showing flexibility and potential for further development in computational methods.

1. References

GeeksforGeeks. (2023). Introduction to Parallel Programming with OpenMP in C++. [online] Available at: <https://www.geeksforgeeks.org/introduction-to-parallel-programming-with-openmp-in-cpp/>.

Anon, (2023). Implementing Parallel Algorithms In C++ - Code With C. [online] Available at: https://www.codewithc.com/implementing-parallel-algorithms-in-c/?amp=1 [Accessed 12 Jun. 2024].

Barve, A., Sneha Khomane, Kulkarni, B., Shubhangi Katare and Sonali Ghadage (2016). A serial to parallel C++ code converter for multi-core machines. doi: <https://doi.org/10.1109/ictbig.2016.7892700>.  
  
<https://github.com/R-Marconi01/Maze-Solver>.

‌

‌

‌