| 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.

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 of their 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, a specific type of 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, 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 the particles find the exit; parallel processing helps to solve the maze faster.

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 of the 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 retry..

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 looks into the effectiveness of the random movement strategy with the hope of enhancing our understanding on how parallel processing can have a massive impact on time taken to solve large structures of mazes.

1. Algorithmic Approach

3.1. 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.

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, 0, 0, 1},

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

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

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

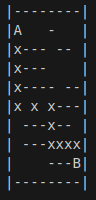
{1, 0, 1, 1, 1, 0, 0, 0, 0, 1},

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

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

};

output:



3.2. Serial Algorithm

The serial implementation begins with the placement of the first particle at the origin of the maze. 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 backtrack and the path from the entrance to exit once the maze is solved.

3.2.1 Outstanding Code: Find-Exit function

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()) {

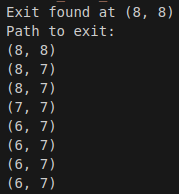
Position step = path.top();

path.pop();

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

}

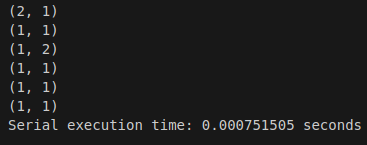
}

3.2.2. Output  


.

.

.



3.3. Parallel Algorithm

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 within four directions and comes back when it encounters a wall. The search effort is distributed when simulating 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 particles has escaped. This flag guarantees that all the threads cease their functionalities the moment one comes across the exit, avoiding to perform unnecessary operations after the 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.

3.3.1 Outstanding Code: Parallel Find-Exit function

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";

}

3.3.2 Output

A screenshot of a computer screen

Description automatically generated

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. 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.

| Particles | Time Taken |
| --- | --- |
| 1 | 0.00045579 |
| 10 | 0.0002734 |
| 50 | 0.00027373 |
| 100 | 0.00016527 |

A graph with a line graph

Description automatically generated

5. 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 parallelizaion 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.

6. 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>.

‌

‌

‌