Parallel Decisions in Strategic Game

Ping-Kuan Kao
Institute of Network Engineering
310552053

Kuang-Hsiang Kuo
Institute of Computer Science and Engineering
311551104

Cheng-Yuan Liu
Department of Computer Science
0816105

ABSTRACT

2048 is a strategic game popular around the world. It is not easy to win the game due to its randomness. We found the AI-Solver code for 2048 on the Internet as the serial version of implementation, and parallelized it using 3 different methods, including MPI, OpenMP and standard library. We choose to parallelize the critical part of the code, in order to get the best performance. As a result, we can achieve 335%, 324%, 181% improvement on time of solving one game using MPI, OpenMP and standard library respectively.

KEYWORDS

parallel programming, strategic game

1 INTRODUCTION

There are more and more fun and strategic games recently, and people are trying hard to win the game. This kind of game is very addictive, and 2048 is one of them. In 2048, there will be a board with 4x4 tiles displayed on the screen. Initially, there will be 2 tiles with a number on it, and the rest of the tiles are null. All the number displayed will be the power of 2. The player has four moving decisions, which are up, down, left and right. Once the player chooses one of them, all the tiles with number will move to that direction. Tiles with same numbers will be combined into one tile and the combined tile will be the sum of the original two tiles. For each move, there will be one additional tile with number **randomly** appeared on the null tile. Player's goal is to get the number **2048**. If all the tiles are numbered but the player can't make any move, then the player loses. On the other hand, if the player successfully get the number 2048 on the screen, he wins the game.

What we want is to win the game, but how to increase the probability to win the game? If the player can see the future, and moreover the player can take more conditions into consideration, the winning probability can increase. However, taking more conditions into consideration indicates spending more time, which is not efficient.

We can see that the player has only four decisions, up, down, left and right, so we can simply move four directions by turns, and compute the best move we can get. Nevertheless, moving four directions by turns is time-consuming, and moreover, new tile will appear randomly on null space after each move, which make it more difficult to win the game. We can use parallel programming to speed up the time for making the decision. We use the concept of multi-thread to parallelize the computation of the 4 decisions, and simulate the game for that move until it loses. This four moves are independent from each other, so we don't have to worry about the synchronization problem. After the simulation, we can get the current best decision, this helps the player to win the game.

The rest of this paper is organized as follows: Section 2 presents the proposed solution. In Section 3, we introduce our experimental methodology in details. In Section 4, we will show our experimental results. Related work and conclusion are presented in Section 5 and Section 6. References will be in Section 7.

2 PROPOSED SOLUTION

We choose **Monte Carlo** to find the most desirable move from four choices at each round. In one round(move), agent execute Monte Carlo algorithm to determine which direction should choose. It would do random simulation multiple times before moving. In each simulation it would keep playing 2048 game until game ends, which means the board could not move anymore.

For instance, agent randomly choose a direction to move in a simulation initially. After first move, agent would randomly guess next location that tile will spawn. The score of tile (2 or 4) would also be guessed by agent. Then keep randomly move and guess until this simulation game ends. When game ends, agent would calculate final score it get at this simulation and records the score and **first move** it choose. Finally, agent would accumulate four direction total moves and times that each direction be chosen. The average score at each direction is calculated via dividing total score by first chosen counts. After finishing processes above, then choose the direction with highest average score. Agent would keep doing simulation and move until one score of a tile reaches 2048 then terminates the game.

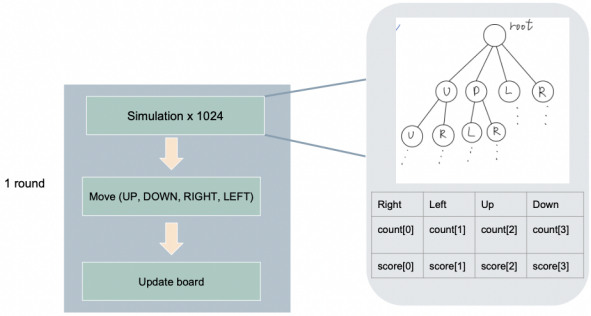


Figure 1: Simulation In Each Round

We focus on **simulation step**. In each round, agent simulates **1024** times. We parallel these 1024 tasks to accelerate the procedure of move decision.

Three methods for parallel simulation

- `MPI`
- `OpenMP`
- `Standard Library`

When encountering new round, 1024 tasks would be distributed to all workers via these three methods. So the program would finish game faster than serial program version.

3 EXPERIMENTAL METHODOLOGY

Cause we only focus on how fast procedure of each round could be improved. In each game, agent ends game when one tile's score reach 2048 or can't move anymore.
We test serial version and three methods version on same computer.

Environment

- `Architecture: x86_64`
- `OS: Ubuntu: 22.04`
- `CPU: 4 core`
- `CPU MHz: 3408`
- `Threads per core: 1`

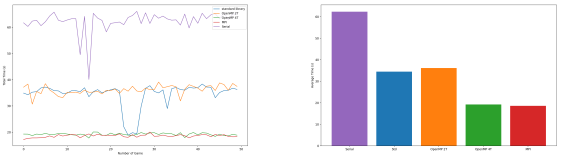
Parameters of program are fixed, so total steps of a game is around one thousand. Each game in experiment starts with clear board and random initial tiles. Steps and spawn of tiles are different in each game. Due to a bit difference in each game.
In first experiment, each method would play 50 times and record total execution time of a game and then get the average time. This design of experiment could lessen the impact of little bit difference due to random mechanism in each game. In our observation, execution time of each game with certain method is stable. Only in game fail situation (does not reach 2048), execution time would dramatically drop. So this experiment design is effective to unveil the performance of each method.
In second experiment, we want to observe the execution time of each round. So we record execution time of each round with different method to observe the relationship of epoch of game and **Monte Carlo** searching space. Even though steps are not same in each game, but it is stable at around one thousand. Cause we stop game when tile reaches 2048, and Monte Carlo method is effective. So almost all games could reach 2048. Therefore, steps would be steady at around one thousand and result is clear to observe.

4 EXPERIMENTAL RESULTS

We use two ways to analyze our performance. Each game will terminate either there is any tile's number reach 2048 or can't make any move.

Execution Time Per Game

We let the agent use different strategies to play 2048 50 times and gather the total execution time per game. The x label represents the number of game and the Y label represent the total time for playing the game.



(a) Execution Time Per Game (b) Average Execution Time Per Game

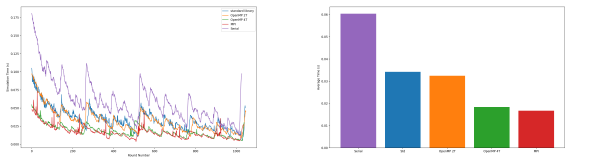
Figure 2: Execution Time Per Game

In the Figure 2, we can see that the standard library and OpenMP with 2 threads only use about half of the time that Serial spend and the standard library's performance is slightly better than OpenMP with 2 threads. For OpenMP with 4 threads and MPI, the time they need is less than 1/3 that the Serial need, and MPI has the best performance.

If we look at the line graph, we can see that some of the games in the Serial and standard libraries spend less time than others. We think the reason causing this situation is that the agent didn't win in that game. In other words, there wasn't any tile that reach 2048 when the game ended. As a result of that, the execution time of that game is shorter than others.

Execution Time Per Round

We let the agent use different strategies to play 2048 1 time and gather the situation time for each round. The X label represents the number of the round and the Y label represents the simulation of that time.



(a) Execution Time Per Round (b) Average Execution Time Per Round

Figure 3: Execution Time Per Round

As we can see in Figure 3, standard library and OpenMP with 2 threads only use half of the time that Serial needs, OpneMP with 4 threads and MPI use 1/3 of the time that Serial needs, just like the previous result. The only difference is that OpenMP 2 threads have a slightly better performance than the Standard library. The strategy with the minimum execution time is MPI.

In the line graph, we can see that the time each round need is slowly decreasing but rebound in some round for every strategy. We think that it is because, in 2048, each move can eliminate more than 1 tile, hence these steps may eliminate a large number of tiles. This kind of movement can increase the probability for the agent to survive and have more options to simulate.

Type	Game	round
OpenMP 2T	1.72x	1.86x
OpenMP 4T	3.24x	3.29x
Standard Library	1.81x	1.77x
MPI	3.35x	3.61x

In conclusion, we can see that the MPI has the best performance in both situations, which is 3.35x faster than Serial in Execution Time Per Game and 3.61x faster than Execution Time Per Game. OpenMP with 4 threads is slightly slower than MPI, but it is still 3.2x faster than serial. Although the Standard Library used 4 threads to speed up, its performance is almost equal to OpenMP with 2 threads. We think that it is because the standard library is a high-level library, so it spends a lot of time calling functions, hence its performance is not satisfactory.

5 RELATED WORK

We found the serial version of AI-Solver for 2048 on the Internet. Its goal is to implement algorithms to achieve as high of scores as possible in the game 2048. The work implemented the solver using three different algorithms, including Monte Carlo, Minimax, and Expectimax algorithms. We chose Monte Carlo and parallelize it. **The following link is what we referenced.**

- `GitHub`: [1]

6 CONCLUSIONS

In this final project, we speed up the 2048 solver by parallel programming, since the original implementation is slow and will lose the game if the simulation process is short. In our work, we use three different methods: OpenMP, standard library, and MPI to implement parallel calculation. By observing our final result, we can see that the performance is improved, and the probability to win the game is much higher, which is what we are expecting. If there is a chance in the future, we want to try more different ways to implement parallelization, like CUDA.

REFERENCES

[1] https://github.com/NathanWine/2048-AI-Solvers