The application of parallel programming in accelerating reinforcement learning in 2048

Ming-Hsien Wu
Institute of Computer Science and Engineering
National Yang Ming Chiao Tung University Hsinchu
w2bc8860@gmail.com

Chun-Yi Chang
Institute of Computer Science and Engineering
National Yang Ming Chiao Tung University Hsinchu
zhangjunyi3405@gmail.com

Yu-Hung Hsiao
Institute of Network Engineering
National Yang Ming Chiao Tung University Hsinchu
tom890525.cs11@nycu.edu.tw

Abstract

Reinforcement Learning is a type of machine learning where the program is not explicitly given instructions on what to do. Instead, it learns through interaction with an agent and the environment, aiming to optimize rewards. In this process, the agent observes different outcomes, makes decisions based on these observations to take actions, and seeks to optimize its results.

However, since its training is initiated by the machine itself without human guidance, the direction and experiences during the training phase may not be correct. This can result in longer training times compared to other machine learning models. To address this, we aim to expedite the training process, focusing on two aspects.

Firstly, we plan to parallelize certain processes using Pthread and OpenMP to avoid wasting CPU resources and enhance performance. Secondly, we will implement MPI to leverage multiple hardware resources, increasing our available computing resources and improving overall efficiency. Ultimately, we will observe and compare the performance differences between parallelized and non-parallelized approaches to assess the effectiveness of acceleration.

Introduction

During a certain class, I learned how to implement a program using reinforcement learning to train a machine to play the 2048 game. However, we observed that the training process took a considerable amount of time, prompting the idea of parallelizing the training process to reduce the waiting time.

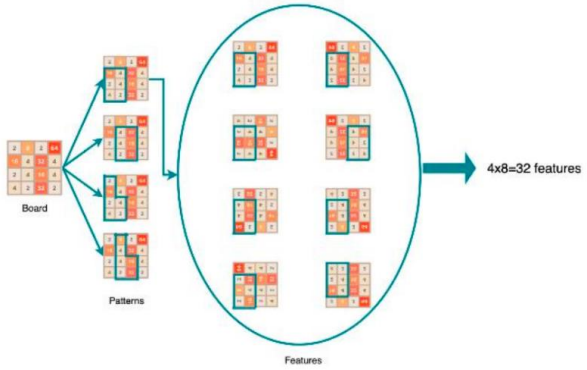
2048 is a well-known game where initially, a 4x4 grid is presented, and a randomly chosen cell is populated with a number 2 or 4. In each turn, the player chooses to move upward, downward, left, or right. During the movement, the numbers on the grid slide in the chosen direction, and if two identical numbers collide, they merge to form a new single number. After the slide, a new 2 or 4 is randomly generated in an empty cell. The player repeats these actions until there are no adjacent identical numbers on the grid, and no empty cells are left, signaling the end of the game.

The training process goes through several stages, broadly categorized into two types: Dependent and Independent. After completing each game, the program gradually updates the training parameters through Back Propagation. This process involves updating parameters from later stages to the current previous grid, creating dependencies between different grid states, hence categorized as Dependent. The Independent part, which is the focus of parallelization in our experiment, occurs during the calculation of the best move (Select best move) at each movement. This involves computing and determining the optimal path (up, down, left, right). Subsequently, we check if the best move is found (Selected?). If not, the game is considered finished, and the experimental parameters are updated (e.g., reducing learning rate), and the total score of the game is recorded before moving on to the next game. Otherwise, we update the reward parameters and proceed to the next movement.

Proposed Solution

The meaning of the ‘Select best move block’ is to ‘choose a direction to move in the current state that, compared to selecting other directions, results in the highest score for the game.’ To determine the optimal move, we consider four scenarios: moving up, moving down, moving left, and moving right in the next step. In each case, we calculate the value using the estimate function, representing the value after moving in that direction.

To avoid excessive memory allocation during the value calculation process, we introduce four patterns to replace the entire 4x4 board for training. Each pattern can have a user-defined tuple size; in our experiment, each pattern occupies a size of six grid cells (tuple=6). Additionally, to utilize information from the entire board, each pattern undergoes rotations (90 degrees each time, four possibilities) and mappings (left-right mapping, up-down mapping, four possibilities). Each pattern produces eight results, resulting in a total of 32 features (as shown in Figure 1). The final estimated value after a move is the sum of the values of these 32 results, averaged.



After the above description, it's evident that estimating the value involves layered calculations considering values in various scenarios, summed up separately and then averaged. Moreover, the calculation processes for each feature are independent until the final sum. Hence, we can parallelize the calculation process using parallel processing in the estimate function to enhance the computational efficiency of the 'Select best move block.'

In this experiment, we implemented the program using C++. Initially, we tested the time required for Reinforcement Learning after a hundred thousand epochs using the serial method. Subsequently, we rewrote the program using Pthreads, OpenMP, and MPI methods taught in class. Finally, we analyzed the issues encountered with each method and compared the time spent with the serial approach.

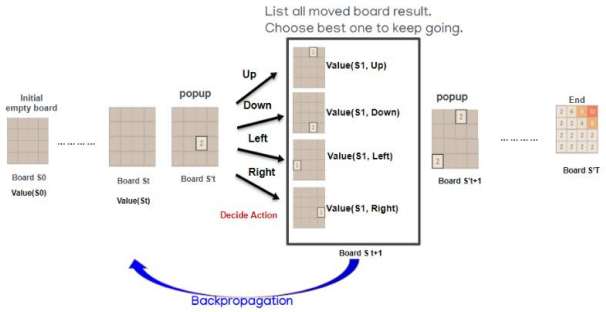
Experimental Methodology

In this experiment, we conducted the research on Ubuntu 22.04.3 LTS, utilizing a computer equipped with an Intel i5-12400 processor. The processor features 6 cores and supports a multithreaded environment with 12 threads.

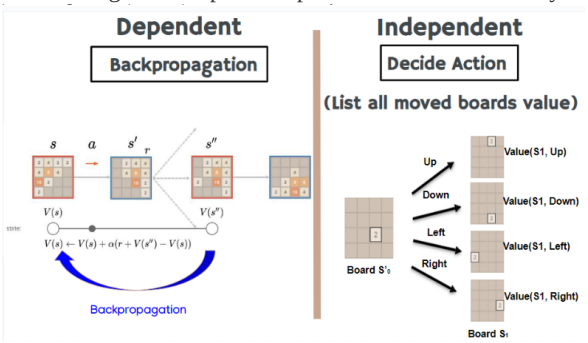
This section discusses the process of playing the 2048 game using Reinforcement Learning (RL). Here, I will briefly introduce it. First, we start with an initialized game board. Next, we have four possible actions to consider: up, down, left, and right. In RL, we rely on different actions to obtain rewards for different board configurations. The choice of our action is determined by maximizing the reward.

It is crucial to note the importance of the reward for each board configuration. Therefore, when we encounter a GAME OVER, our RL algorithm updates the reward values for each board through backpropagation. The update for each board involves utilizing the value of the succeeding board to update the current board's reward. This updating process starts from the end and progresses towards the front of the sequence of boards.

This represents one iteration of the RL process playing the 2048 game.

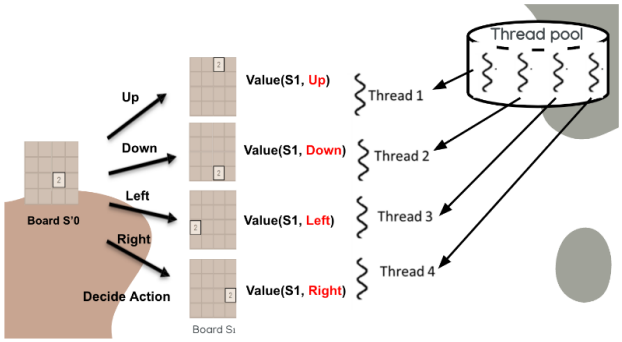


For the Dependent and Independent part.
Based on the recent introduction, we can understand that our entire process can be divided into dependent and independent parts. The dependent part comes into play during the final backpropagation, where, due to the need for updating the current layout value from the subsequent page, it is subdivided and assigned to different threads for specialization. On the other hand, the independent part involves performing various actions separately, as these components operate independently without mutual interference. Therefore, we propose parallelizing the independent part to enhance efficiency!



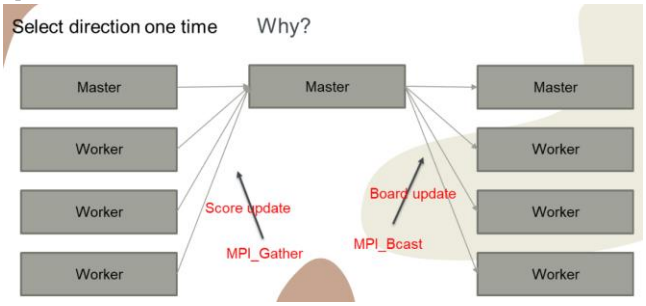
OpenMP

At the beginning, OpenMP initializes a predetermined number of threads and calculates the values after moving up, down, left, and right respectively. The key distinction between OpenMP and Pthread lies in OpenMP's built-in thread pool mechanism, which eliminates the latency incurred by repeatedly creating threads.



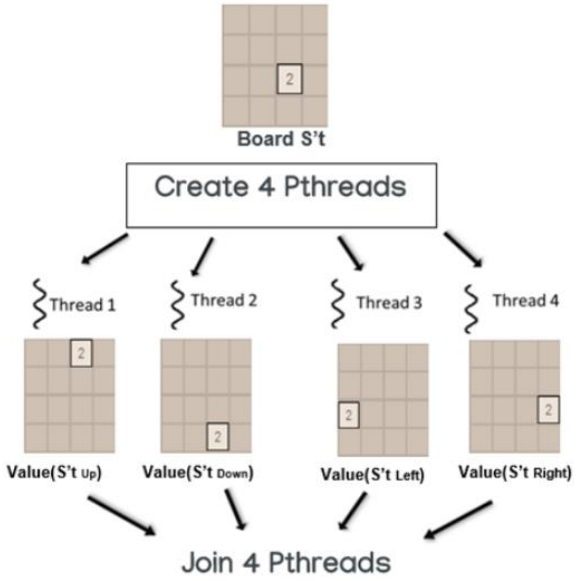
MPI

In this parallelization approach, we initially implemented a master and three workers. The key focus is on allowing workers to make decisions on the value estimation of the board in different directions and returning the results to the master for decision-making. Therefore, our first step involves gathering the evaluated values from each worker to the master. Subsequently, as our workers estimate values based on the board state, when the master decides on a direction, we need to update the board state and broadcast it to each worker. This ensures that the chosen direction by each worker remains meaningful. In the MPI context, we utilize MPI_Gather and MPI_Bcast to facilitate direction determination and board updates for each worker.



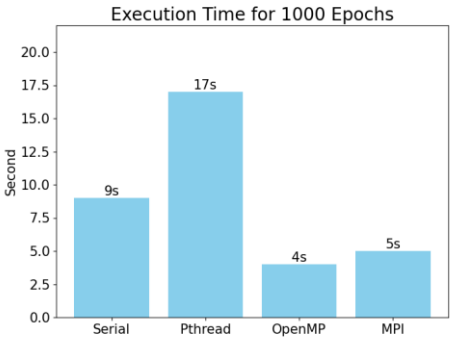
Pthread

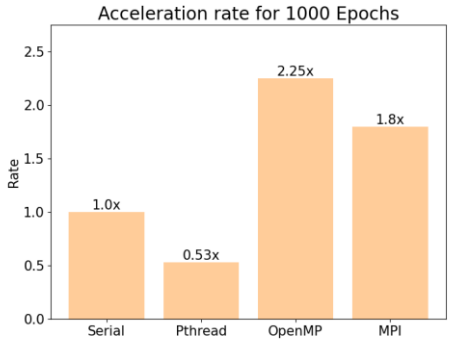
At each step, there is an unpopped board S't. To determine the direction in which the board should be moved, we have listed the layouts of the board S't in the up, down, left, and right directions, named S't up, S't down, S't left, and S't right, respectively. We have allocated 4 threads created by the Pthread Library to calculate their values. Finally, the results of the calculations from the 4 threads are joined together, and the layout with the highest value is selected to continue the game.

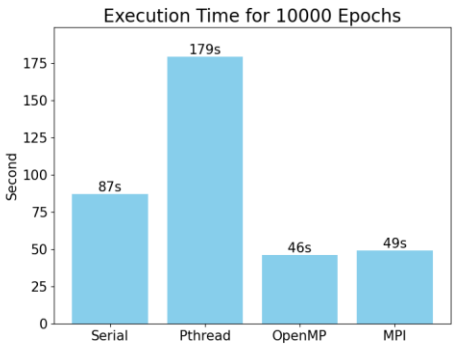


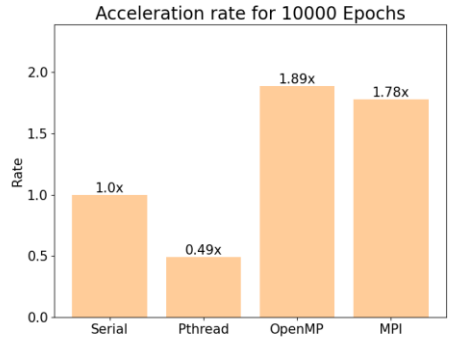
Experimental Result

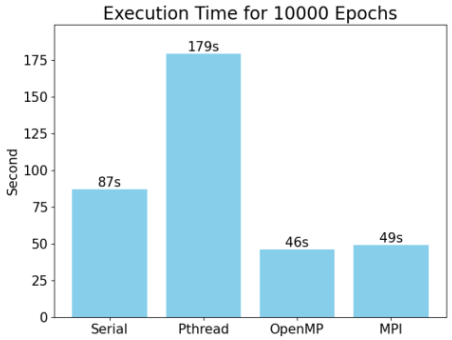
In this comparison, we examined the execution times and speedup ratios of using Pthread, OpenMP, and MPI for 1000, 10000, and 100000 iterations. We observed that Pthread exhibited the longest execution times, even surpassing the execution times of the serial version. We speculate that this is due to Pthread lacking a thread pool mechanism, unlike OpenMP. Therefore, Pthread creates threads when selecting the movement direction in each round, leading to continuous thread create and join. This results in significant overhead. Additionally, we noticed that the performance of OpenMP is slightly better than MPI. This is attributed to MPI's need for a master to collect information from other workers, update the board, and then broadcast the updated board to all workers. The communication processes (scatter and broadcast) in MPI introduce some delays, causing its performance to be inferior to OpenMP.

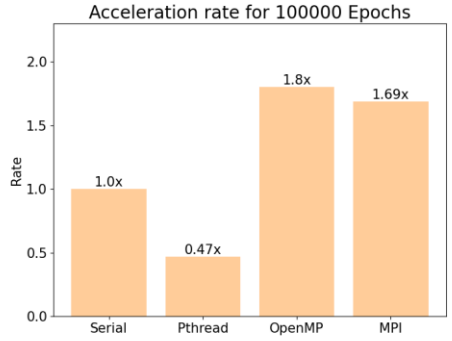












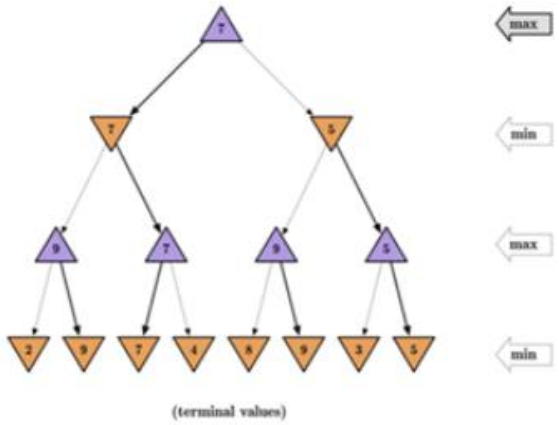
Related work

In the paper titled “Parallel Minimax Agent: Implementation of Game 2048” [1], parallel programming is also employed to accelerate their decision-making algorithm. The paper primarily focuses on implementing a parallel version of the minimax algorithm for the game 2048 using Haskell. The minimax algorithm, commonly utilized in decision-making and game theory, is adapted for a two-player turn-based scenario.

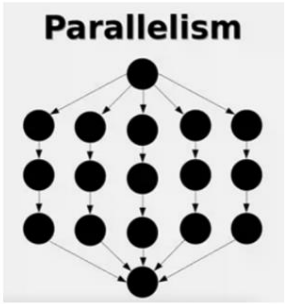
The algorithm assumes a two-player setting, where each player aims to maximize their own utility to secure a victory. It operates on the principle that both players, MAX and MIN, choose optimal options in each state. The maximizer function strives to achieve the highest possible value, while the minimizer function aims for the lowest score possible. These functions are recursively employed to determine decisions in a given state.

As illustrated in Figure [1], the minimax algorithm is utilized in search trees, progressing from the top node to the terminal node, and then backtracking. Each node in the tree represents an option with an associated utility value, and decisions are made by comparing values in node pairs. Terminal values are located in the leaf nodes of the last level.

The tree assumes the presence of two players, MAX and MIN. The MAX player endeavors to maximize the value and win the game by selecting the highest terminal values (e.g., 9, 7, 9, 5 in the example). On the other hand, the MIN player aims to minimize the MAX player’s value and win by choosing the lowest values from the MAX player’s options (e.g., 7, 5 in the example). This process of MAX maximizing and MIN minimizing continues until a final value (e.g., 7 in the example) is determined.

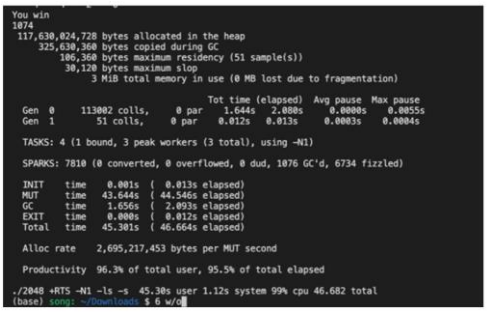


As shown in Figure [1], this paper primarily focuses on the parallel execution of algorithms. Once these strategies are implemented, the program carries out multiple parallel executions for recursive search after initiation and consolidates the results in a unified form. The Strategies method embodies deterministic parallel programming, and this project exclusively concentrates on employing this method.



Paper Execute Times

In below Figure[1], the method of paper parallelization was applied for 1074 epochs, taking a total of 45.301 seconds.

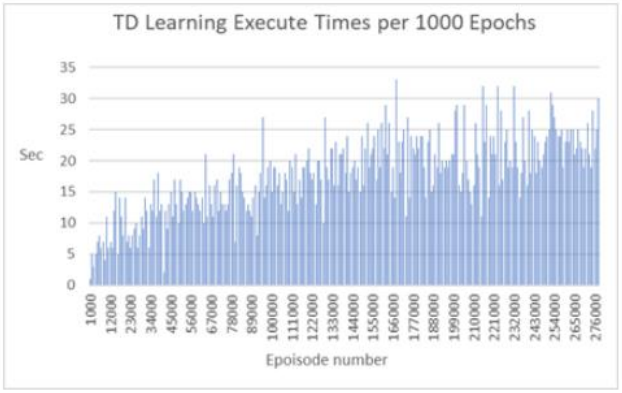


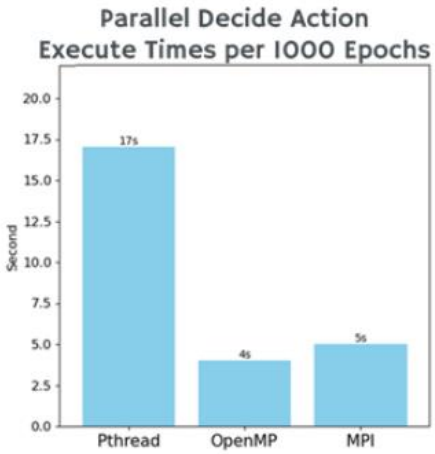
Our Parallel Design and TD Learning Execute Times in an Episodes
below first Figure illustrates the execution times of TD Learning per 1000 Epochs. However, since the TD Learning Execute Times vary in each 1000 epochs, we take the average value of 17.73 seconds to represent the TD Learning Execute Times in every 1000 epochs.

In below second Figure, it depicts the execution times of our three parallelism strategies deciding actions to play the 2048 game per 1000 Epochs. Among them, Pthread took 17 seconds, OpenMP took 4 seconds, and MPI took 5 seconds.

Therefore, in 1000 epochs, including TD Learning and the parallelized execution times for playing the 2048 game, the Pthread version took a total of 17.73 + 17 seconds, the OpenMP version took 17.73 + 4 seconds, and the MPI version took 17.73 + 5 seconds.

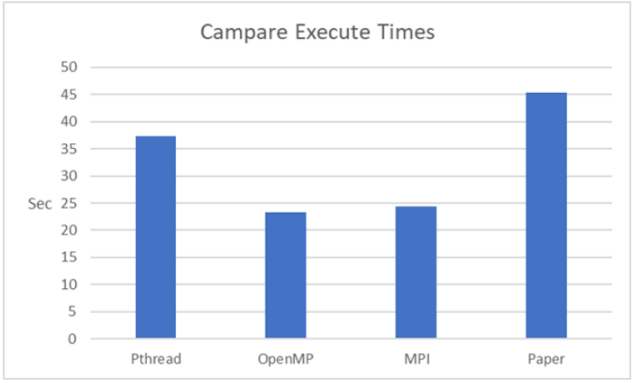
Then, in a single epoch, the Pthread version took a total of 0.003473 seconds, the OpenMP version took a total of 0.002173 seconds, and the MPI version took a total of 0.002273 seconds.





Compare

Therefore, after 1074 epochs of parallelization, the Pthread version took a total of 37.3 seconds; the OpenMP version took 23.338 seconds; and the MPI version took 24.412 seconds. Compared to the paper's time of 45.301 seconds, our acceleration results outperformed the paper's version.



Conclusion

The primary focus of our investigation is the time-consuming nature of training a Reinforcement Learning (RL) model. We aimed to explore potential parallelization strategies to accelerate the process. In our implementation, we used RL to play the game 2048 and parallelized independent components. The experimental results demonstrated some success with OpenMP and MPI. However, it's important to note that RL training involves backpropagation, and accounting for the time spent on backpropagation may reduce the overall speedup achieved. Despite the potential limitation in achieving high speed up for the entire RL process, we believe that parallelizing RL computations is crucial. Without parallelization, RL training becomes excessively time-consuming. Therefore, even

a slight acceleration is deemed necessary. Regarding future work, we propose further exploration of parallelization within each thread or worker. Maximizing parallelization opportunities, such as parallel decision-making when selecting directions, or allowing multiple threads / workers to make direction choices before returning the analysis, may optimize our parallelization efforts. For example, while four threads can concurrently make four direction choices, sixteen threads could make two rounds of direction choices before returning their analysis. This approach could potentially enhance the efficiency of our parallelization. Furthermore, future work may delve into exploring how to parallelize dependent components to achieve a more effective implementation of RL training.

REFERENCES

[1] COMS 4995 Parallel Functional Programming, Fall 2021 Final Project Report, Jeeho Song (js4892), " Parallel Minimax Agent : Implementation of Game 2048 "