Training 2048 game AI with Parallel Architecture

Chia-Hung Yu
0411254 NCTU CS
yamiefun@gmail.com

Wei-Chieh Yu
0411228 NCTU CS
rftheitu0130@gmail.com

Li-Yang Wang
0416014 NCTU CS
osinoyan.cs04@g2.nctu.edu.tw

Participant

Chia-Hung Yu: architecture design, MPI coding, result analysis, reports discussion (34%)

Wei-Chieh Yu: architecture design, MPI coding, result analysis, reports discussion (33%)

Li-Yang Wang: architecture design, MPI coding, result analysis, reports discussion (33%)

Abstract

We propose a parallel architecture that can accelerate an reinforcement learning (RL) training process running on CPU.

In an RL training process, the agent usually has to spend a huge amount of time interacting with the environment, hence, we use a parallel architecture which allows many agents to interact with the environment asynchronously and in parallel. This helps speeding up the RL training process, and the result efficiency meets our expectation.

Introduction

In recent years, deep learning has already been used in multiple research area, for instance, object detection, machine translation and speech recognition. Most of these applications adopt supervised learning, therefore, numerous label data which demands plenty of time for researchers to arrange is needed. Moreover, the quality of these label data hardly affects the performance of agent.

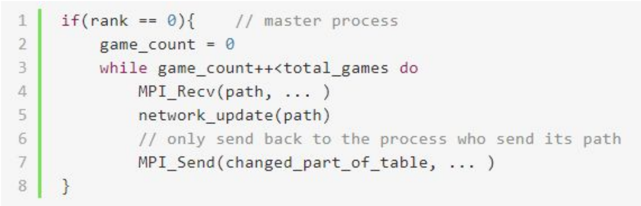
Reinforcement learning is a method that agents acquire experience by interacting with environment. However, reinforcement learning costs a bunch of time exploring in the environment, greatly increase the difficulty for training a neural network.

Reducing the time of training seems to be not only challenging but also meaningful in order to improve the efficiency at reinforcement learning field. In this project we accelerate training process by parallel exploring the environment through multiple workers.

Solution

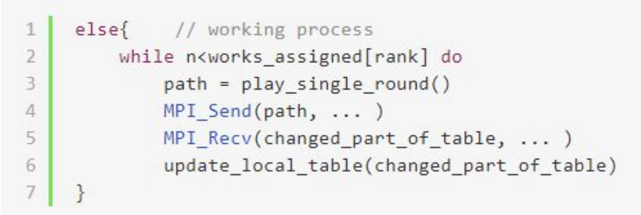
Our main idea is referred to the paper *Distributed deep q-learning* (Distributed deep q-learning; Chavez, K et al. 2015)[1], and modified according to our experiment. **Algorithm 1** illustrates what every slave processes should execute, the greatest difference from DQN algorithm is that worker threads send their experiences to the master for updating the neural network, get the updated part of the network back from master, then update their local network by themselves. That is, worker processes will simply work on four procedures :

1) Generating new data for its local shard
2) Send the experiences to the master
3) Receive the updated part of the neural network
4) Update its neural network



```
if(rank == 0){    // master process
    game_count = 0
    while game_count++<total_games do
        MPI_Recv(path, ... )
        network_update(path)
        // only send back to the process who send its path
        MPI_Send(changed_part_of_table, ... )
}
```

Algorithm 2 is the main part of the training process. Master thread receives workers' experiences, then pass the updated part of the network parameter to specific thread in order to continue on their works. Although the master process's updating part of code is not in parallel, we thought that the efficiency will still be considerably great because we notice that the time costs by updating the network is extremely less than costs by exploring the environment.



```
else{    // working process
    while n<works_assigned[rank] do
        path = play_single_round()
        MPI_Send(path, ... )
        MPI_Recv(changed_part_of_table, ... )
        update_local_table(changed_part_of_table)
}
```

In Figure 1, the parameter server will save a set of parameters. After every workers interacts with their own gaming environment asynchronously, they save their transitions in their experience replay memory and will send them to server end. After the server receives a set of transitions, it will calculate the gradient, update the parameter and send the updated part back to the worker.

Since every worker need to execute on a independent CPU core, there gaming environment and local memory are independent of each others, also there is no need of communications between workers which is similar to master-slave model, this enables the execution of workers to be parallelized.

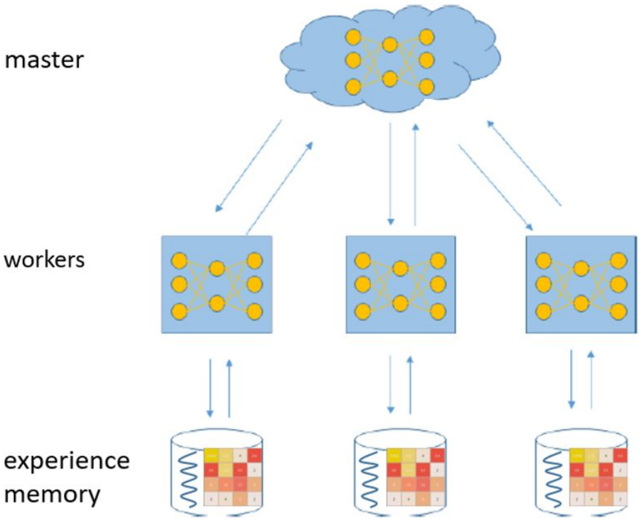


Fig. 1: The architecture of our algorithm model

Experiment Methodology

We take CGI-2048 game(C++)[4] as our experiment environment, it provides a complete 2048 gaming environment and a simple RL algorithm for training an agent.

We use a cluster with CPUs(Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz) for parallelizing our project. In our experiment, we train with different amount of processes (from 2 to 9) and each with a million games then compare their efficiency. Furthermore, to ensure the stability of the result, we train each cases 3 times and calculate the average.

Experiment Results

The following pictures are our experiment results, each of them are :

1. y: Win rate, x: Game count
2. y: Execution time (second), x: Game count
3. y: Speed-up ratio, x: Game count
4. y: Win rate, x: Execution time (second)

Fig.1 shows that although the number of process we use increase, the win rate still increases correctly (by comparing to the win rate progress np2, which is serial gaming)

Fig.2 & Fig.3 are the main purpose of our project, the time cost of each process number and the speed-up ratio of them. In figure.3, we found that when the game count increases the ratio slowly decrease and become steady. The reason of it is probably because after a few games, the agent begins to improve. The data size of gaming experience which includes every step's board increases then causes more time to send this with MPI.

In Fig.4, we found that within the same time, the greater the number of process we use, the higher the win rate will be. This proves that even with some MPI API calling, the result is still going well.

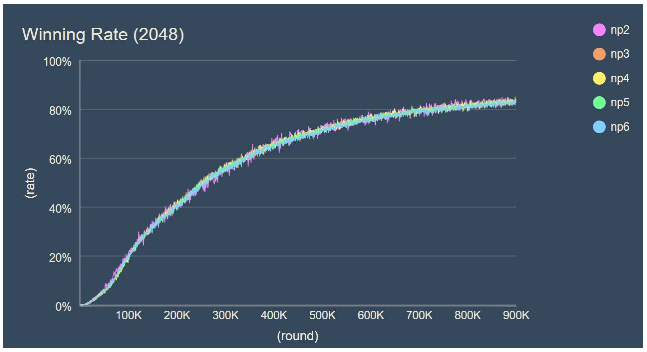


Figure.1

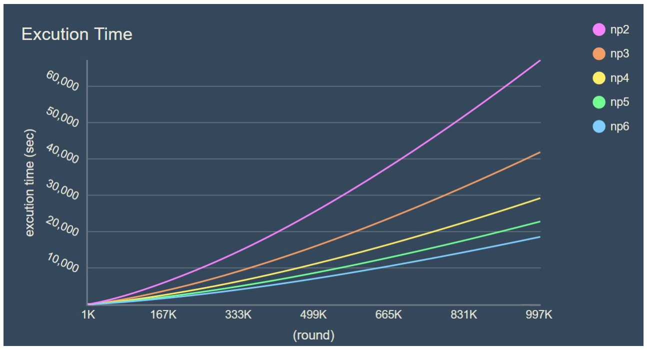


Figure.2

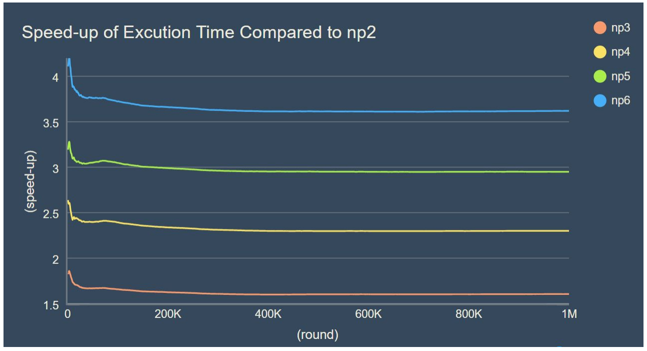


Figure.3

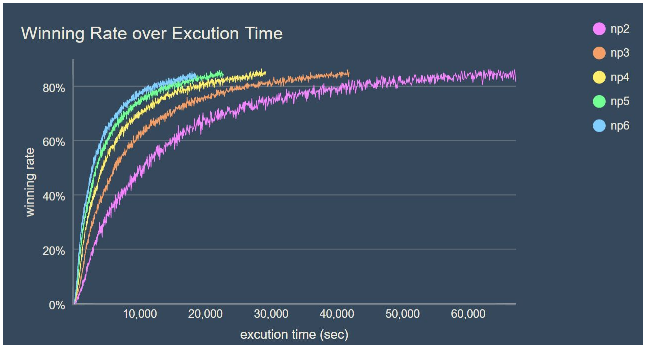


Figure.4

Related Work

A. Markov decision process

In an MDP, an agent chooses action A_t at time *t* after observing state S_t. The agent then receives reward R_t, and the state evolves probabilistically based on the current state-action pair.

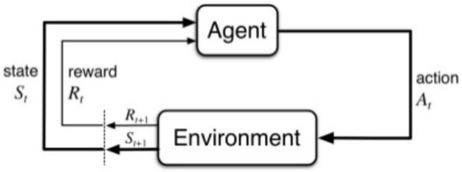


Fig. 2: Basic MDP diagram

The goal is to find a policy π^* that, if followed, the agent will maximize the expected sum of rewards given by the environment. π^* is related to the optimal state-action value function $Q^*(s,a)$, which is the expected value when starting in state *s*, taking action *a*, and then following actions dictated by π^*. Mathematically, it obeys the Bellman recursion

$$Q^\star(s,a) = R(s,a) + \sum_{s'\in\mathcal{S}} T(s,a,s') \max_{a'\in\mathcal{A}} Q^\star(s',a').$$

By using value iteration, we can compute the state-action value function. To obtain the optimal policy for state *s*, we compute

$$\pi^\star(s) = \operatorname*{argmax}_{a\in\mathcal{A}} Q^\star(s,a).$$

B. Reinforcement learning

The problem Reinforcement learning(RL) seeks to solve differs from the standard MDP in that the state space and transition and reward functions are unknown to the agent, hence, the agent needs to interacts with an environment through a sequence of observations, actions, and rewards and learns from the experience.

The basic idea is to estimate

$$Q^\star(s,a) = \max_\pi \mathbf{E}\left[R_t \mid s_t = s, a_t = a, \pi\right],$$

where π maps states to actions, with the additional knowledge that the optimal value function obeys Bellman equation

$$Q^\star(s,a) = \mathop{\mathbf{E}}_{s'\sim\mathcal{E}}\left[r + \gamma \max_{a'} Q^\star(s',a') \mid s,a\right],$$

where ε is the MDP environment.

C. Deep Q-learning

Google DeepMind proposed Deep Q Network(DQN) on Nature in 2015[2]. They combine nonlinear estimation in Deep Learning with the advantage of learning from experience in Reinforcement Learning and advance a new concept, Deep Reinforcement Learning(DRL). They apply it on Atari Games and successfully exceed human-level performance in most of the games, becomes a significant milestone in this field.

D. Asynchronous Advantage Actor-Critic

Asynchronous Advantage Actor-Critic(A3C; Google DeepMind, 2016)[3] is a training method advanced by DeepMind. It combines the asynchronous method, advantage value and deep deterministic policy gradient together to make the convergence result better.

In A3C, because the exploration work is done by multiple workers instead of one worker originally, result in better exploration property.

Conclusion

In this project, we propose a parallel architecture to accelerate the reinforcement learning process with C++ under CPU-only environment. The speed up ratio is reasonable and meets our expects.

Although we have tried some other parallel architectures that have better training performance, we notice that when execute the process on higher parallelize environment, too much overhead will make the performance worse than original. Hence, a better data structure and parallel architecture is needed to enhance overall performance.

References

[1] ONG, H. Y., CHAVEZ, K., AND HONG, A. Distributed deep q-learning. CoRR abs/1508.04186 (2015).

[2] Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M.; Fidjeland, A. K.; Ostrovski, G.; Petersen, S.; Beattie, C.; Sadik, A.; Antonoglou, I.; King, H.; Kumaran, D.; Wierstra, D.; Legg, S.; and Hassabis, D. 2015. Human-level control through deep reinforcement learning. Nature 518(7540):529–533.

[3] Mnih, V.; Badia, A. P.; Mirza, M.; Graves, A.; Lillicrap, T.; Harley, T.; Silver, D.; and Kavukcuoglu, K. 2016. Asynchronous methods for deep reinforcement learning. In International Conference on Machine Learning.

[4] CGI-2048 gaming environment. https://github.com/ccchang1023/CGI-2048