Fast Track – Massively Parallel Programming Techniques Applied to Reinforcement Learning Algorithms.

Dwight S. Bell

A Thesis in the Field of Information Technology

for the Degree of Master of Liberal Arts in Extension Studies

Harvard University

May 2011

**Abstract**

This project addresses the problem of applying massively parallel programming techniques to reinforcement learning problems with an empirical study of the advantages of a parallel approach. The benefits of using parallelism in the reinforcement learning domain are not obvious. There is a penalty associated with dividing a fixed number of agent actions across parallel agents. Each parallel agent has less information to guide its future actions compared to a single agent taking the same number of total actions serially. We explore strategies for offsetting the penalty with communication and knowledge sharing among parallel agents. Advantages of parallelism are found in the following ways:

* **Speed** – pure speed improvements by running multiple trials in parallel,
* **Information Sharing** –learning improves when agents share their individual experience with the group,
* **Differentiation** – parallel agents that differ can improve the overall learning rate, and
* **Evolution –** the best agents can be selected and varied over time while removing poor agents from the group improving learning speed and quality in complex domains.

We develop parallel implementations of Temporal Difference (TD) learning and other techniques for four problems using CUDA (NVIDIA Corporation, 2011), the framework for general purpose programming on graphical processing units (GPUs). We show empirical results demonstrating the ability of parallelism, with appropriately selected sharing, differentiation, or evolutionary strategies, to improve the quality and speed of reinforcement learning.

**Acknowledgments**

I would like to thank my thesis director, Dr. Ya’akov Gal who provided guidance throughout the process, including helping me focus the topic during the proposal phase and keeping me on track during the work phase. I also thank Dr. Jeff Parker, my research advisor who worked patiently with me during the early, middle, and end stages of proposal writing.

Further acknowledgment goes to the professors in my last three courses at Harvard Extension school: Mr. David Alpert (Artificial Intelligence), Dr. Sarah J. Finney (Intelligent Machines), and Dr. Hanspeter Pfister (Massively Parallel Computing). These three courses amplified my interest in the subject matter at the heart of this thesis.

# Table of Contents

Table of Contents vi

Chapter 1 Introduction 1

Motivation 1

Benefits of Parallelism in Reinforcement Learning 2

Empirical Study of Parallelism 3

Chapter 2 Background 6

Reinforcement Learning 6

Temporal Difference Learning 7

Learning the Value Function 8

Eligibility Traces 9

Other Considerations 11

Parallel programming on the GPU using the CUDA Framework 13

Chapter 3 Prior Work on Parallel Implementations of Reinforcement Learning 16

Chapter 4 Simple, Discrete Domains 20

Overview of the problem 20

Parallel Implementation 21

Sharing and Differentiation 23

Results 25

Chapter 5 Simple continuous domains 31

Overview of the problem 31

GPU Implementation 32

Sharing 33

Differentiation 34

Results 35

Chapter 6 Complex Continuous Domains 41

Overview of the problem 41

GPU Implementation 43

Sharing and Differentiation 45

Results 48

Chapter 7 Learning to Play a Game through Self-Play 52

Overview of the problem 52

Neural Net Design 53

GPU Implementation Issues 56

Replication with Variation and Summary of Results 59

Chapter 8 Conclusions and Future Work 63

References 67

Appendix 1 Glossary 69

Appendix 2 Random Number Generation on the GPU 71

# Introduction

## Motivation

The goal of this thesis is to apply GPU programming techniques to perform an empirical study of using parallel approaches for implementing reinforcement learning algorithms. The goal is an empirical analysis of the costs and benefits of parallel code when applied to reinforcement learning problems of increasing complexity. The problems chosen for this thesis are standard reinforcement learning problems of increasing complexity. In addition, the methods used to solve the problems also increase in complexity from problem to problem. Through this means, we will observe the potential of applying parallel techniques on the GPU over a wide range of applications.

It is not obvious that parallelism benefits reinforcement learning algorithms. Learning is an inherently serial process. The learning agent is continually receiving information and using the accumulated information to guide its future actions. There is a parallel penalty when a fixed number of agent/world interactions are spread across multiple, independent agents. Each parallel agent has less information available to it, since it has only seen a fraction of the total interactions, and it is natural to expect that less information will lead to slower learning. First the parallelism penalty is measured and then results are shown including the benefit from the raw speed improvement due to parallelism and the benefits of other strategies available to parallel agents, which are discussed next.

## Benefits of Parallelism in Reinforcement Learning

In reinforcement learning, which is discussed in greater detail in chapter two, an agent observes the state of the world, chooses an action and receives information on the new state as well as a reward value. The agent applies a reinforcement learning algorithm to update its internal state and uses the accumulated information to guide the choice of its next action. The process is repeated indefinitely until a terminal state or preset time limit is reached. The agent’s goal is to learn how to choose actions to maximize the total reward received over time.

Reinforcement learning is inherently sequential. The agent repeatedly executes a sequence of steps: observe state, take action, get reward and observe next state. There are, however, many ways a parallel approach can improve the speed or quality of learning:

* **Speed** -- Reinforcement learning experiments are usually repeated a number of times, starting from the same initial conditions, to get statistically credible results and to observe the variability of results. A parallel approach can provide a raw speed-up by running multiple trials in parallel. In addition, there can be opportunities to speed up complex calculations of a single agent with parallel threads.
* **Information Sharing** -- Parallel agents can potentially learn faster than single agents by sharing their experience. The learning process can be periodically paused to allow agents to share information based on their learning experience, thereby improving the overall quality. The success of this approach depends on the offsetting effects of the benefits to learning quality versus cost of the additional time to do the sharing.
* **Differentiation** -- Differentiating agents can potentially improve learning results. Agents can be differentiated in a number of ways. They may be assigned different portions of the state space, or choose different actions when presented with the same state of the world. Learning algorithm parameters can also be varied across parallel agents. With differentiated agents learning and sharing information, the best results can be shared across all agents.
* **Evolution** -- The results of learning may have some random elements, either in the state or in the agent’s selection of actions. In this situation, parallel agents can provide a benefit even in the absence of information sharing. Simply running multiple agents in parallel, testing them and selecting the highest quality agent at the end of training can improve learning results. In addition, other evolutionary techniques can be applied to a population of learning agents. By measuring the quality of agents during the learning process, the best agents can be selectively copied and poorer agents eliminated, improving the overall learning rate. The copies can be exact or they can include some variation of agent parameters that may produce an even better agent in the long run.

## Empirical Study of Parallelism

This thesis develops parallel implementations of reinforcement learning for four problems of increasing complexity. The problems are chosen from standard reinforcement learning problems with an incremental increase in problem complexity and in the complexity of the methods used to solve the problem. This allows an incremental approach to investigating the applicability of massive parallelism using GPUs to reinforcement learning problems. For each problem we develop empirical data to illustrate the potential improvements of parallel techniques using one or more of the strategies listed above. The learning results are measured and compared based on agent quality as a function of training time.

This thesis is the first comprehensive study of massive parallelism using the GPU applied to reinforcement learning problems. It makes the following contributions:

* Empirically demonstrates how massively parallel agents, running on a GPU with properly devised information sharing strategies can improve the speed and quality of learning, compared to a single agent implementation,
* Demonstrates how agent differentiation can be used to improve learning results in some problem domains,
* Combines an evolutionary approach with parallel agents yielding improved learning results, and
* Shows the advantages of massively parallel approach to learning to play a game through self-play.

The remainder of this thesis is organized as follows: Chapter two provides background information on reinforcement learning and parallel programming on GPUs with CUDA. Chapter three reviews relevant prior work in the area of parallel reinforcement learning. Chapters four through seven will present empirical results of applying parallel techniques to four reinforcement learning problems. The massively parallel approach will be applied to the N-armed Bandit, Pole-Balancing, Mountain Car problems and the problem of learning to play a game through self-play. The results of each implementation will be presented, with particular attention to the choices made for information sharing between parallel agents, agent differentiation, and ultimately agent selection and evolution. The final chapter will summarize the results, give conclusions, and discuss possible future work in the area.

# Background

## Reinforcement Learning

Reinforcement learning is an area within the machine learning field of artificial intelligence. The goal of reinforcement learning is to produce agents that can successfully choose optimal actions in a problem domain. The agent learns by observing its current state, taking an action, and receiving a reward. Figure 1 illustrates the simple, repetitive process that is used in reinforcement learning.

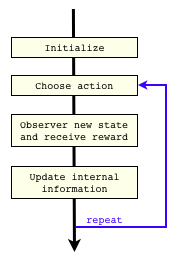


Figure 1:Basic reinforcement learning algorithm

The reward can be positive, negative or zero and the agent’s goal is to learn how to choose actions to maximize the total reward received over time.

### Temporal Difference Learning

The learning approach used here is based on Temporal Difference learning, a model-free approach where the agent does not attempt to build a model of the domain. Instead, the agent directly learns the present value of future rewards after taking action  from a state , which is represented by the function . With a good estimate of the -function, the agent can determine the best action from a given state by selecting  which maximizes . The -function defines a policy  that maps states onto the optimal action according to (1), and the value of a state  according to (2).

 (1)

 (2)

The -function can be stated in terms of , the expected reward received when taking action  from state , the discount rate , and the expected value of the next state, .

 (3)

Equation (3) shows that the -function can be expressed as the expected value of the reward plus the maximum value of the -function from the subsequent state. Whenever an agent takes an action from a state, receives reward and sees the next state, it gets a sample that can be used to improve its estimate of . The learning rate parameter, , is used to control the degree to which the estimates of  are updated according to equation (4), where  is the actual reward received.

 (4)

Equation (4) is the basic learning equation for one type of Temporal Difference learning sometimes referred to as -learning.

### Learning the Value Function

Sometimes the agent knows how to determine the next state after taking an action from the current state. When this is the case the agent can learn the value function  instead of learning . The value function defines a policy based on choosing the action that maximizes , where  is the known next state after taking action .

 (5)

 (6)

Now the agent can use its interactions with the environment to learn the value function using equation (7).

 (7)

Here  is the learning rate and  the discount rate as before.

### Eligibility Traces

When learning the -function, an agent takes an action  at time  from a state , receives the reward, and observes the next state. The new information causes the agent to change its estimation of  using equation (4). In principle, the change in the estimated value of  could then be retroactively used to update the estimated value for , the -value for the state and action that preceded the current one. Then the additional change in  could be used to update , etc., propagating the impact of the new information to states and actions further in the past that lead up to the current state.

To see an example of how eligibility traces work, consider the simple world illustrated in Figure 2. The agent occupies one of 5 states in the world, numbered 1 through 5 and has two available actions: LEFT and RIGHT. The reward is zero for all states and actions except for action RIGHT from state 5 that gives a reward of +100. Assume the agent starts with a value of 0 for  for all states and actions.

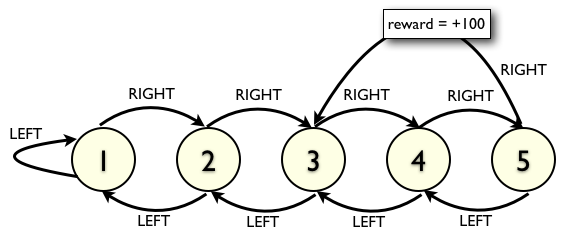


Figure 2: Simple world with 5 states and 2 actions and a single reward.

With normal temporal difference learning the agent will take actions in the world, perhaps randomly at first, and will at some point take the action RIGHT from state 5 and receive the reward of +100. This will cause the agent to increase the value of  using equation (4). During further exploration, if the agent happens to take action RIGHT from state 4 it will reach state 5. Now the application of equation (4) will cause  to increase. Eventually, in the same way, the value of  will increase when the agent takes action RIGHT from state 3 in the future, and so on until the action RIGHT from every state will have a positive value.

Eligibility traces increase the speed of learning by immediately propagating the change in to the state and action that preceded it, namely . The increase in  is also propagated back to the state and action that preceded it, and so forth. Eligibility traces provide an increase in learning speed by recording information about the history of states and actions and attributing new information back to those preceding states.

The impact of a change can be attenuated as it moves further into the past by multiplying by a factor . This method is called TD(λ). To implement TD(λ) we first calculate the amount of change,, and then use that value to update  for 

 (8)

 (9)

A computational shortcut uses an -value for each state/action, . When an agent takes an action from a state, the value of  is set to 1.0. Each time step after that, the -value is decayed by a factor of  according to equation (10).

 (10)

Then, after calculating  using equation (8), the update rule for all states becomes

 (11)

### Other Considerations

A problem for reinforcement learning in all but the simplest problem domain is the size of the state and action spaces. Frequently, the size of the state space makes it impossible for the agent to visit all states during the learning process. The agent must therefore generalize from the portion of the state space it has experienced to estimate the - or -functions over the entire domain. One approach, when the state space is continuous, is to partition the space into a manageable number of cells. The agent then estimates the function values for each cell. Another approach is to use neural nets to approximate the function of continuous state variables or of an encoding of a high dimensional state. Each of these approaches is demonstrated in later chapters.

Reinforcement learning can be done in an off-line mode or on-line mode. Off-line learning means the agent’s rewards received during the learning process do not matter. The agent’s quality is determined after learning is complete by testing the agent’s optimal policy function and the average reward it will produce. In on-line learning the agent’s goal is to optimize the reward received while it is learning, a more difficult task. In this thesis we deal with off-line learning and determine agent quality by pausing the learning process and separately testing the agent’s optimal policy. Note that sufficiently fast off-line learning methods can be used in an on-line manner by quickly running off-line learning and then selecting the optimal action in the on-line setting.

Reinforcement learning can be applied to a wide variety of problems. Problem domains for reinforcement learning are classified in a number of ways. First, problems are classified based on the state information that is available to the agent. If the agent always sees complete information about the current state of the world the problem the domain is classified as having observable states. If some information about the current state of the world is hidden from the agent then the problem is called partially observable. For example, chess is a game with completely observable states, while bridge or poker are domains with partially observable states. The four problems presented in this thesis have completely observable states.

Another important characteristic of a problem domain is whether the information about the current state that is available to the agent contains all the information that is relevant to predicting the future. This is referred to as the Markov property and can be stated in probabilistic terms as the future is independent of the past given the present. All problems in this thesis have the Markov property. An example of a problem domain that does not have the Markov property is blackjack with a finite deck of cards. The state may only contain information about the cards currently in play, but the complete history of cards played from the finite deck is relevant to predicting the probabilities of future cards. Craps is an example of a game that does have the Markov property, as the probabilities in each new game do not depend on prior results.

In general, the agent in a reinforcement learning problem starts with no information about the world it is in, other than the actions it can choose from. It simply takes an action from the current state and observes the next state that results from that action. The agent’s only information about how the world works is gathered through its interactions with the problem domain and observing the transitions between states. This is true for the first three problems in this thesis. In the last problem, the agent is learning to play a board game through self-play. The agent does know how the game works. It is programmed in advance to be able to determine the next state in the game for each of its possible moves from the current state. It is also programmed with the knowledge of which moves are legal from the current board position. This approach is common when reinforcement learning is applied to board games.

This brief introduction to reinforcement learning is intended to give a high level view to readers unfamiliar with the topic. A great source for learning more about reinforcement learning is a textbook by Sutton and Barto (Sutton & Barto, 1998).

## Parallel programming on the GPU using the CUDA Framework

Parallel programming using GPUs is on the rise, thanks to the combined influences of the leveling off of CPU frequency improvements, the increasing power of GPUs, and the wide availability of CUDA-capable GPUs , from the NVIDIA CUDA Programming Guide (NVIDIA Corporation, 2010), illustrates the rapid increase in GPU speed compared to CPUs. The NVIDIA Corporation has developed the CUDA programming framework to facilitate the use of NVIDIA GPUs for general purpose computing. The framework allows a programmer to write programs in C/C++ that run on the CPU in a normal way and call massively parallel portions of code that run on the GPU. The programmer is responsible for moving data to and from the GPU and calling into the GPU code, which is referred to as a kernel.

The multiprocessors on a graphics card are designed to execute the same code simultaneously on multiple sets of data. This is referred to as SIMD processing, for Single Instruction, Multiple Data. The goal when coding in the CUDA framework is to take the problem, isolate the portions that fit the SIMD hardware, and move them to kernels that run on the GPU. Simple graphics operations or other problems with numerous identical independent tasks can readily take advantage of the speed-up provided by GPUs. One of the challenges in this thesis was the implementation of the complex reinforcement learning algorithms on the GPU while maximizing the speed-up.

The code for this thesis was run on ‘resonance’, a General Purpose GPU cluster run by Harvard’s School of Engineering and Applied Sciences. The cluster contains 16 compute nodes with 2 GPUs on each node. Each compute node is controlled by a dual core Intel Xeon CPU running at 3.0 GHz with 8 GB of RAM. The applications for this thesis were programmed to use just a single GPU on one compute node. The GPU is the Nvidia Tesla T10 that contains and 4 GB of device memory and 240 processing cores. The GPU is designed to handle thousands of simultaneous threads, as it is very efficient at switching between threads to hide the latency associated with accessing global memory.

3 (NVIDIA Corporation, 2010)

Now we have introduced the basics of reinforcement learning and discussed the ability of GPUs to run massively parallel programs with CUDA. The next chapter discusses prior work in the area and we will then proceed to the four learning problems at the core of this thesis.

# Prior Work on Parallel Implementations of Reinforcement Learning

Parallelism techniques have been applied to reinforcement learning in the past. These applications typically ran on CPUs, but more recently, running on GPUs. There are also a number of different ways parallelism can be applied.

There are many works that combine multiple agents and reinforcement learning to address a multiagent learning problem. *A Comprehensive Survey of Multiagent Reinforcement Learning* (Buşoniu, Babuška, & De Schutter, 2008) describes some of the work in this area. In contract, this thesis addresses what are traditionally single agent learning problems, and parallelism is used to find the highest quality single agent as fast as possible. The output of the techniques in this thesis is a single agent even though multiple agents may be used in parallel to find it.

V. Palmer (Palmer, 2007) used GPU programming to speed up matrix calculations in the Least Squares Policy Iteration algorithm (Lagoudakis & Parr, 2003) applied to a multi-agent cooperative Pole Balancing problem. The GPU was used to speed up calculations on a fixed set of training data for from 1 to 4 agents where each agent had 20 basis functions. Here, the parallelism on the GPU was used to get a computational speed up of the policy evaluation and policy improvement steps of the Least Squares Policy Iteration algorithm for each agent. The implementation and interaction of multiple agents was done on the CPU. The batch processing using the GPU was compared to a Matlab implementation and found to improve calculation speed for training sets of at least 400 points. The GPU calculation time grew much slower than the CPU calculation time as the training set increased and also as the number of agents increased.

In this thesis we use the GPU to implement parallelism across multiple agents. In Palmer’s study, the GPU was used within each agent to speed up that agent’s calculations, but did not attempt to deal with multiple cooperating agents with varying experience and the issues of data communication on the GPU. Furthermore, the degree of parallelism applied in thesis is much greater, with agent group sizes in hundreds or thousands compared to a maximum of 4 agents in Palmer.

Grounds and Kudenko used a parallel approach with multiple CPUs for three reinforcement learning problems (Grounds & Kudenko, 2007). They parallelized across agents and studied the communication issues between multiple agents. In their approach, each agent communicated its information to all other agents in a staggered fashion over time. The agents took turns broadcasting their information to the other agents. The information communicated was limited to the most important parameters learned by that agent. The importance of a parameter was determined by how much it changed since the previous communication. Reducing the quantity of data communicated was necessary due to the relatively slow communication speed between CPUs running on separate machines. They tested from 1 to 16 agents on Pole Balancing, Mountain Car, and a Grid World problem, and demonstrated increasing quality of learning for a given training time as the number of agents increased.

This thesis differs by running the parallel agents on a single GPU instead of multiple CPUs. The GPU makes it possible to use thousands of agents while Grounds and Kudenko used a maximum of 16 agents. Kudenko had to severely limit the quantity of information shared between agents because of the high cost of communication between processes running on different CPUs, which is less of a problem for this thesis. On the GPU the cost of communication is simply the cost of synchronizing all agents at a sharing point and then calculating and storing values in global memory on the GPU. Communicating between threads on a single GPU can be less time consuming than communicating between processes on different CPUs.

Kretchmar (Kretchmar, 2002) applied a parallel approach to the N-armed Bandit problem, which we introduce in Chapter 4. Kretchmar used from 1 to 10 agents in parallel and the quality of learning was measured as a function of time steps per agent. After each time step, all agents shared information. Individual agents kept track of their own experience data and that of all other agents separately. This approach allowed the agents to share just the data from their own experience while still having access to the combined information. The results showed improved learning quality as the number of agents increased as a function of the number of steps in the learning process. Measurements were made as a function of action steps per agent, so total actions taken increased as the number of agents was increased. Results were not timed so the cost of communication between agents was not reflected in the quality measurements.

Kretchmar did not consider the time cost involved with parallel agents and information sharing, and by sharing after every time step the agents had complete information. In this thesis, time is an important component when measuring of the quality of learning. In this work agents spent most of the time operating independently, agents only share information periodically, and the agents do not have complete information.

Evolutionary techniques are a natural subject when multiple agents are used. Evolutionary methods and reinforcement learning are compared in (Whiteson, Taylor, & Stone, 2010). In that work the methods are each applied to two learning problems and the problem characteristics which favor reinforcement learning or evolutionary learning are discussed. Evolutionary methods and reinforcement learning are considered distinct approaches to solving a learning problem. In this thesis, we combine some evolutionary techniques with the reinforcement learning process implemented by multiple agents, with beneficial results.

The main differences between this thesis and prior work is the use of the GPU to apply massive parallelism to reinforcement learning problems allowing the use of hundreds or thousands of agents. Large numbers of parallel agents are shown to able to improve the reinforcement learning process through a number of techniques. Empirical measurements are made to demonstrate the improved learning due to information sharing, differentiation, and evolution of hundreds or thousands of agents.

# Simple, Discrete Domains

## Overview of the problem

The first problem is the N-Armed Bandit, a standard problem in reinforcement learning, described by Sutton and Barto (Sutton & Barto, 1998). The N-Armed Bandit problem gets its name from the "one-armed bandit" slot machines in casinos. The -armed bandit has more arms, as the name suggests. The agent must select one of the arms, 'pull' it, and then it receives a variable reward. The reward is based on a normal distribution that is different for each arm. The learning problem is to determine which arm has the highest average reward by pulling arms and observing rewards. The quality of an agent is measured as the percent of times the agent has correctly identified the highest value arm at the end of the learning period.

This problem has only one state and  actions since the agent can choose one of the  arms to pull. The agent must decide which arm to pull based on the information it has accumulated from previous rewards. The agent needs to try all arms to get some information about each of them, but should ultimately try to concentrate on the higher-reward arms to find the best one. The traditional single-agent approach is to keep track of the average reward produced by each arm. The agent then decides whether to select the arm with the highest reward, or to explore the arms and choose one at random. The learning parameter, , is used to set the probability that a random arm is chosen. The best arm is chosen with probability . After the training period, the arm with the highest average reward is the agent’s prediction of the best arm.

TD learning in this domain simplifies to learning  since there is only one state. The learning rate, , is typically set to  where  is the number of times an action has been taken in the past. This gives the prior estimated value of  a weight of  and the updated value equals the mean of all observed rewards for action .

Experiments were run using 10-, 100-, and 1000-armed bandits, with qualitatively similar results. Results shown below are for the 100-armed bandit.

## Parallel Implementation

In the parallel implementation for this problem the agents share their information periodically. The processing is split between periods of learning where each agent operates independently and a sharing period where the information is consolidated from all agents and then shared. The actual learning and sharing take place on the GPU while the CPU controls the overall process. The high level division of activities between CPU and GPU is shown in Figure 4. This diagram shows the basic paradigm I used for the parallel implementation of learning algorithms using the GPU. It will be updated as the need arises as domains become more complex.



Figure 4: high-level sequence diagram showing activities on CPU and GPU

Key learning parameters for the multi-agent parallel approach for the -Armed Bandit are:

* The exploration probability, , same as in the single agent algorithm,
* the number of parallel agents, and
* the frequency of information sharing.

## Sharing and Differentiation

Each agent keeps track of the number of pulls for each arm and the arm’s average payout. When sharing occurs an overall count and average is calculated for the block of agents, and then shared back to all agents. Let  be the number of pulls stored by agent  for arm , and  be the value estimated by agent  for arm . The total values are calculated and each agent is updated based on the following equations for each arm :

 (12)

 (13)

For each agent :

 (14)

 (15)

Note that in equation (15) the number of pulls stored by each agent after sharing is their share of the total number of pulls, not the total number of pulls for the entire block. This gives less weight to the averages stored by the agent and allows those values to change more dramatically after sharing has occurred. It also means that the sum of the number of pulls across all agents is equal to the actual total number of pulls. This is useful when re-calculating the total values at future sharing points. The result is that some agent's will have their average value for the best arm go down due to the random nature of rewards, perhaps causing another arm to become the best for that agent. This has the ultimately beneficial effect of causing agents to explore other arms with high average values.

The next time sharing occurs, the block average is recalculated and quality is improved for all agents. This gives the agents a periodic burst in learning quality at the point of sharing. The quality may drop between sharing points as the agent’s own values are allowed to vary. At each sharing point the quality of each agent is restored and actually improves based on the combined information.

Experiments were run with a number of agent differentiation strategies:

* partitioning the arms among the agents so when an agent was exploring it would only explore its assigned arm,
* random bias given to each agent used when choosing the best arm to pull,
* giving some agents in a group a high epsilon value so they would explore with a much greater probability than other agents, and
* assigning some agents to randomly choose between the top 2 or top 4 arms when selecting the best arm instead of choosing the actual best arm.

There was no dramatic improvement in learning quality from these strategies. For some, a small improvement in learning quality was more than offset by the increased processing time to implement the more complex logic, resulting in poorer learning as a function of learning time.

## Results

Experiments were run for 10-, 100-, and 1000-armed bandits using a single agent and groups of from 4 to 8,192 parallel agents.

The results are broken down into two components to gain a better understanding of the cost and benefits of parallelization. The first component is learning quality as a function of the number agent-time steps. An agent time-step is one interaction between an agent and its domain, which equates to one cycle of the learning process executed by one agent. This will measure the “parallelization penalty” which is the expected reduction in learning quality when a fixed number of agent-time steps are spread over multiple parallel agents, as compared to a single agent operating for all of the time steps. The penalty occurs because the parallel agents operate independently and only periodically share partial information about their experience. The single agent is expected to have better learning results, when measured on this basis, since it has complete information available at every time step.

Offsetting the parallel penalty is the speed-up in real time of the parallel implementation. A fixed number of agent-time steps can run faster if they are executed in parallel, so for a fixed learning period, the parallel approach can execute more agent-time steps than the single agent approach.

The first experiment is for a fixed number of agent-time steps. Learning is run for the fixed number of agent-time steps and then repeated for 1,024 trials. Learning quality is calculated as the fraction of the 1,024 trials where the agent correctly identified the bandit with the highest expected payout, which can be determined by inspection. Measuring learning as a function of agent-time steps gives an understanding of the algorithmic ‘parallel penalty’ from splitting up a fixed number of actions over parallel agents. The results for the 100-armed bandit are shown in Figure 5 and Figure 6. The two graphs are the same except for the difference in agent group sizes. Learning quality on the y-axis is measured by the probability of correctly identifying the best arm over 1,024 trials. The blue line with circles is the result for a single agent running on the CPU and the other lines are for different numbers of parallel agents.

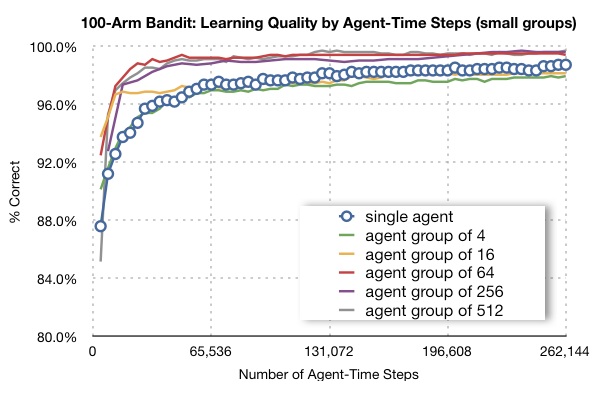


Figure 5: Learning quality by agent-time steps, small agent groups

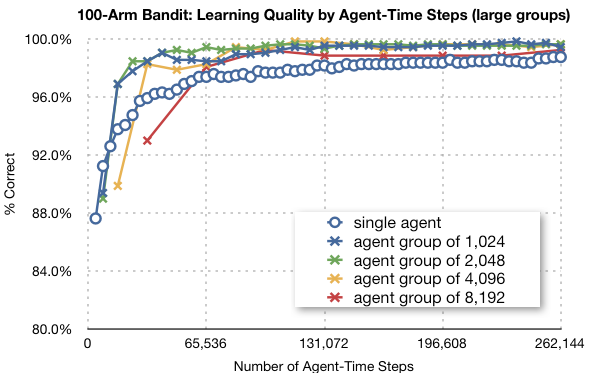


Figure 6: Learning quality by agent-time steps, large agent groups

The results are interesting as there is no clear penalty for the parallel approach, except for agent groups of 4 or 16. When parallel agents numbered 64 or more the parallel implementation actually produced higher quality using a fixed number of agent-time steps. The difference appears to diminish for extremely large groups as the learning quality drops for 8,192 agents. The multi-agent advantage is likely coming from more effective exploration due to the lack of complete information. Each agent in the group makes an independent choice of best arm and the next action to take based on limited information, or perhaps some new unique information it has uncovered but not yet shared, leading to beneficial exploration of the state space.

A single agent approach can, of course, be modified to match the multi-agent results by using a more complicated exploration strategy, even mimicking the calculations as if there were multiple parallel agents. Nonetheless, it is interesting that the normal, simple exploration strategy using the  parameter can produce better results for a fixed number of agent-time steps when executed in parallel then a single agent using the same algorithm with the same number of agent-time steps.

The next step is to measure learning quality as a function of learning time. Learning time is the real elapsed time the agent spends interacting with the environment, updating its internal state, and sharing information in the case of parallel agents. The timing values are for a single learning trial run on the CPU for a single agent, and on the GPU for parallel agents. For parallel runs, the number of time-steps is determined so that the parallel agents took at least as much time as the single agent run for 262,144 time steps. To get credible quality measurements the results were averaged over 1024 trials for both single agent and parallel agent runs. Learning quality as a function of learning time is shown in Figure 7. For the two smallest numbers of parallel agents, groups of 4 or 16, the parallel implementation on the GPU did poorly compared to the single agent CPU run. Since the time values were for a single trial, there were only 4 or 16 threads running concurrently on the GPU. Given the complexity of the calculations and the need to frequently synchronize threads to share information, the poor results are not surprising. For the larger numbers of parallel agents, with hundreds or thousands of threads, the parallel GPU results clearly surpass the single agent CPU results when learning quality was measured as a function of learning time.

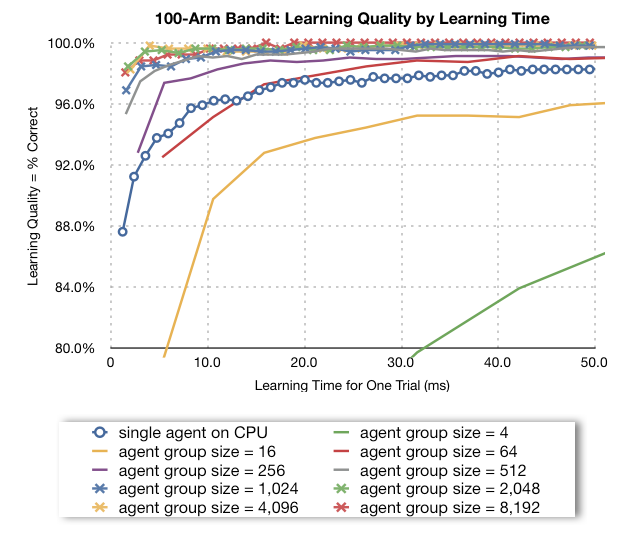


Figure 7: Learning quality as a function of learning time for 100-arm bandit

The last chart, Figure 8, summarizes the results of the parallel implementation of N-Armed Bandit on the GPU. The blue bars are the learning quality for 131,072 agent-time steps. Comparing the multi-agent blue bars to the CPU blue bar shows the improvement due to improved algorithm performance for the fixed number of agent-time steps. The green bars are the learning quality after 37.7ms, which is the time it took to run 131,072 time steps for a single agent on the CPU. It can be seen that with 256 or more agents, the advantage of the parallel implementation on the GPU over the single agent CPU run increased when measured as a function of time. The drop in the fixed agent-time steps bar (blue bar) for 8,192 agents compared smaller agent groups shows that the algorithmic advantage for parallel groups does not continue to increase as group size increases. This is also seen in Figure 6.

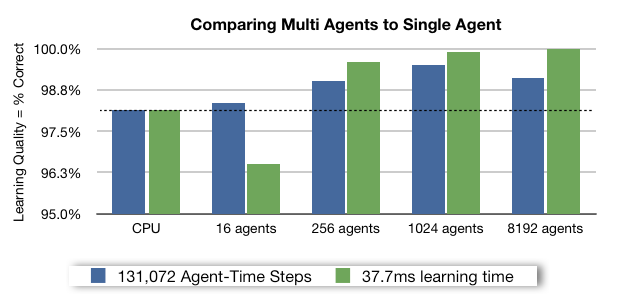


Figure 8: Summary of multi-agent GPU learning compared to single agent, CPU

In summary, the massively parallel approach to the N-armed Bandit problem shows the surprising result that there is no parallel penalty for agent groups of size 64 or larger (Figure 5 and Figure 6). Splitting a fixed number of agent-state interactions across parallel agents can actually improve the learning quality through more effective exploration. The parallel results improved further when learning quality is measured as a function of learning time (Figure 7) as the raw speed improvement provided by the GPU allow more agent-time steps to be executed with greater speed in parallel.

# Simple continuous domains

## Overview of the problem

The Pole Balancing problem, sometimes referred to as the inverted pendulum, is another canonical problem of reinforcement learning. The paper “The Pole Balancing Problem, A Benchmark Control Theory Problem” (Brownlee, 2005) contains a detailed description of the standard definition of the problem that was used in this project. In the Pole Balancing problem, the state consists of a cart on a track with a vertical pole attached to a hinge on the cart. The goal is to keep the pole balanced within a certain tolerance of the vertical position and to prevent the car from going off either end of the track by applying a fixed force to either end of the car (see Figure 9). The state of the world consists of four continuous values: the position and velocity of the cart and the angle and angular velocity of the pole. The agent has no knowledge of how the world works and chooses between two actions: apply the fixed force to the left or to the right.

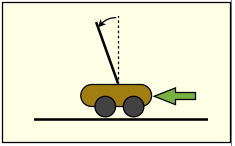


Figure 9: Pole balancing problem. The agent can exert a force on the car in either direction. The agent’s goal is to keep the pole balanced within a certain tolerance of vertical and to prevent the car from running off the either end of the track.

A reward is introduced to make this a reinforcement learning problem. The reward is 0 for every time step that the car is on the track and the pole is in the allowable vertical range. The agent receives a large negative reward if there is a failure due to the car running off the track, or the pole falling beyond the vertical tolerance. Each trial begins with the state variables being initialized with random values.

This problem poses a new challenge due to the continuous state variables. There are a number of ways that continuous variables can be handled. One approach would be to learn a continuous function, , over the continuous domain of the states. This approach will be used in Chapter 6 . For this problem we choose to partition the continuous states into a small number of cells. The position and angle values are partitioned into 3 value ranges. The cart velocity and angular velocity of the pole are partitioned into two value ranges, positive and negative. This creates 36 possible discrete states of for the system. Given the two possible actions, there are a total of 72 state-action values. This manageable number of state-action values makes it reasonable to try to learn the  function using TD(λ) learning. In this problem the agent attempts to learn the value of  for each state-action pair using eligibility traces. Eligibility traces should speed learning on this problem as reaching the failure state is intuitively attributable to many prior actions and not just one bad choice.

## GPU Implementation

Temporal difference learning with eligibility traces is a complex algorithm to implement on a GPU. Each agent must maintain a number of values for each state-action pair: the estimated -value, the eligibility trace, the weight for the -value that will be used during sharing, and the random bias amount (discussed in next section). In addition there are random number seeds and stored values for previous state and action. The processing within the learning GPU code is more complex than the n-Armed Bandit problem, as illustrated in Figure 10.

It is more difficult to monitor the learning process as well. The agent’s quality cannot be determined by inspection. It can only be determined by testing, so the learning process must be paused and the agent tested for a statistically significant number of time-steps. For this problem learning quality is defined as the average time to failure over a trial of 8,192 time-steps. Testing was done following each sharing phase.

## Sharing

Figure 10: Pseudo code for TD learning implemented by learning kernel on GPU.

// learning kernel for Pole Balancing

initialize state, s

choose action a, storing Q(s,a)

for each time step

take action a, returning reward r and new state s’

choose next action a’ from state s’

delta = difference between Q(s,a) and r + Q(s’, a’)

update all Q values based on delta and eligibility trace

update all eligibility trace values

a := a’ and s:= s’

next time step

Sharing among agents for this problem is done in a straightforward manner. After a period of learning, the -values are averaged over all agents. The average is done on a weighted basis. Each agent maintains a weight for each state-action pair. The weight for agent  is  and it is initialized to a constant value. During the learning process, the agent weights are updated as follows:

 (16)

During sharing, an average value for all agents for  is calculated based on the following equation:

 (17)

The value  is shared with all agents and their weights are reset after sharing to the same constant initial value.

 (18)

 (19)

## Differentiation

The sharing described above produces identical agents at the start of each learning period. Differentiation was added to the agents for this problem with beneficial results. With differentiation, a small random bias amount is added to each agent’s  values. The amount of bias is added after sharing and it decreases over time. The sharing update rules with differentiation are as follows, where  is a uniform random number over the specified interval,  is the maximum bias amount and controls the rate of reduction in the maximum bias:

 (20)

 (21)

Randomly biasing the values of  has the most impact on states where  is close to , i.e. when the values learned so far do not clearly favor one action over the other. In this situation, the random bias will cause some agents to choose  as the best action and others to choose , generating new information about both actions from that state. This, hopefully, leads to identifying one action as clearly the best when the agents pool their information at the next sharing point. The next section shows the improvement in learning due to differentiation.

## Results

Agent quality was measured for this problem as the average time to failure over 8,192 time steps. Testing is required to determine agent quality as it cannot be determined by inspection. Each agent is given a random starting state and runs for 8,192 time steps, noting the average time to failure during this period. The best possible score would be 8,192, which means no failure during the testing period. Each trial of 8,192 was repeated 1,024 times to get a statistically significant quality measurement. Doing the quality testing on the GPU sped up the testing since the 1,024 trials can be done in parallel.

The first set of results measure the quality of learning as a function of agent-time steps. This measurement shows the algorithmic penalty, if any, of the parallel approach. The algorithmic penalty is the reduction in learning quality due to the fact that agents operating in parallel have less information available to them, after a given number of agent-time steps, than a single agent. The single agent will have complete information about all actions, rewards, and state transitions, while the parallel agent must wait until a sharing point to gain any information from the experiences of the other agents.

The first graph, Figure 11, shows results by agent-time step for varying numbers of parallel agents and compares them to a single agent. In these runs, there was no differentiation of parallel agents after sharing. The jump in quality as agents shared information is clearly visible in the graph, but only the smaller agent groups, up to 256 parallel agents, could exceed the single agent learning quality after 262,144 agent-time steps. The 4,096 agent group, did have an anomalous drop in learning quality at its second sharing point at 262,144 total agent-time steps. The cause for this was not investigated and the learning quality quickly recovers when the number of time steps per agent increases, as can be seen in Figure 12.

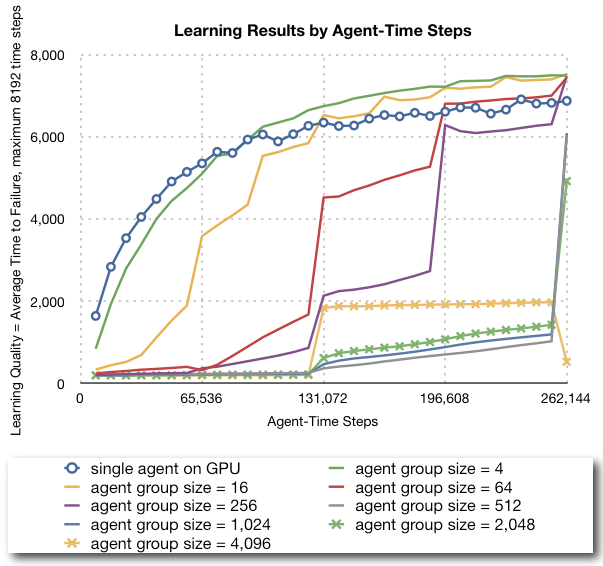


Figure 11: Learning quality for Pole Balancing as a function of agent-time steps without agent differentiation.

The frequency of sharing is a learning parameter that is determined separately by agent group size through experimentation. As agent group size increases, the optimal number of time steps per agent between sharing decreases.

The next measurements tested learning quality as a function of learning time. As with the previous problem, learning time is measured based on a single run with the single agent on the CPU or group of parallel agents on the GPU. The time spent testing the agent is, of course, excluded from the measurement. The learning quality is then calculated by averaging the learning measurements over 1,024 trials. The quality of learning with parallel agents is dramatically better than single agent learning for groups of 256 or larger when measured over a learning period of 400ms, as can be seen in Figure 12.

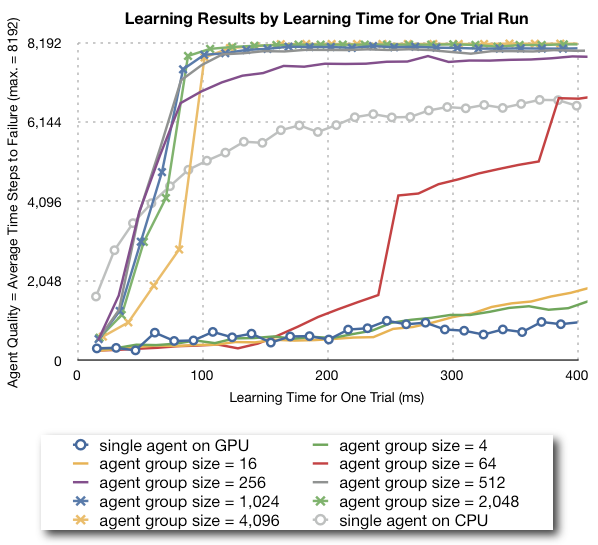


Figure 12: Learning quality as a function of learning time for Pole Balancing, without agent differentiation.

Agent differentiation is beneficial to parallel learning for the Pole Balancing problem, as mentioned before. The differentiation for each agent is a small random bias added to the -values shared with each agent. The amount of random bias was determined by experimentation and decreased exponentially over time. Differentiation dramatically improves the speed of learning for agent groups of 256 or larger. Figure 13 shows the improvement that occurred during the first 100 ms of learning. Ultimate quality does not change much because average quality is close to the best possible value of 8,192 and there is no room for improvement. The final learning quality including the impact of differentiation is shown in Figure 14, clearly showing the success of the parallel approach.

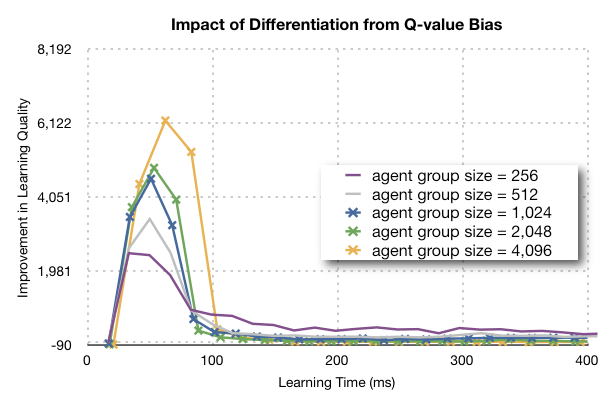


Figure 13: Increase in learning quality as a result of agent differentiation by random bias of Q-values.

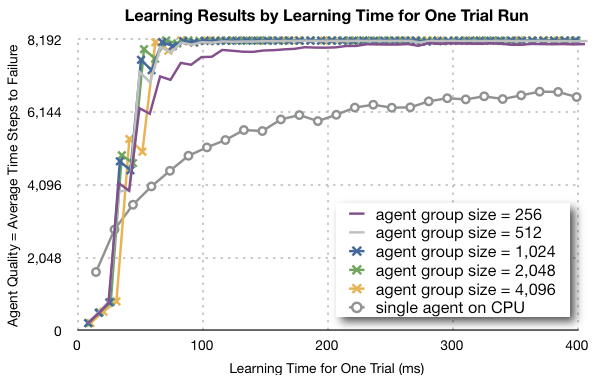


Figure 14: Learning quality as a function of learning time for Pole Balancing with differentiation.

In summary, the massively parallel approach produces a dramatic improvement in learning quality as a function of the learning time for the Pole Balancing problem (Figure 14). This improvement is obtained by using information sharing and agent differentiation with groups of at least 256 parallel agents.

# Complex Continuous Domains

## Overview of the problem

The mountain car problem is a simulation of a car in a valley between two mountains. The goal is to reach the top of the mountain to the right, but the car's engine is not powerful enough to drive straight up the mountain. The only way to reach the top is, at some point, to back up, toward the left, up the opposite mountain, and then to drive forward and use the momentum gained to reach the goal. A negative reward is received each time step until the car reaches the mountain top goal. The agent can choose between driving forward, backward, or coasting and starts with no knowledge of how the world works.

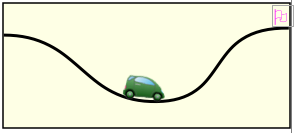


Figure 15: Mountain Car - the goal is to reach the flag on top of the mountain on the right.

In this problem, there are two continuous state variables, the car’s position and velocity, and three possible actions: drive forward, drive backward, or coast. The continuous state variables pose the same difficulty as in the previous problem, but for this problem we chose to use a neural net to approximate the -values as a function of the continuous variables. Using a neural net on this problem is not necessary, but it was chosen to increase the complexity of the calculations done by the GPU to in parallel. The three neural nets each have a single hidden node and one output node. The hidden node uses a sigmoid activation function while the output node is linear. There are 15 weights that must be trained during the learning process.

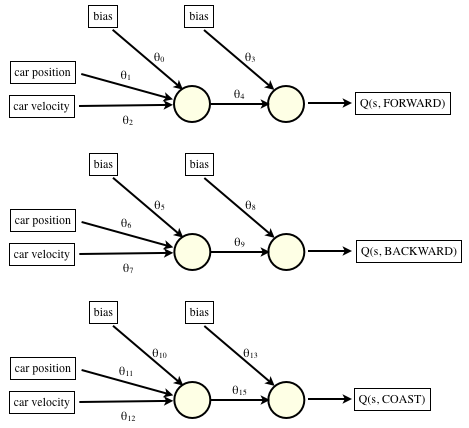


Figure 16: Three neural nets used in Mountain Car to estimate *Q(s,a).*

The learning algorithm for this problem is TD(λ), same as the previous problem, but now changes in the -values are used to update the neural net weights using back propagation. The value for  is calculated as in equation (8) and used to update neural net weights by back propagation. Eligibility traces can also be used with neural nets and back propagation, but the normal calculations can be a problem. Each state/action pair leading up to the current state had its own activation of the neural net and varying gradients of the output with respect to neural net weights. The gradients are used in back propagation so a straightforward implementation requires storing all the gradients with respect to each weight for each time step. Luckily, there is a convenient calculation approach that avoids this problem.

 (22)

 (23)

In equation (22)  is the gradient of the neural net output with respect to , the weights at time . Accumulating the gradient values in  and decaying them by a factor of  yields a factor that can be used in a single calculation to update neural net weights using back propagation and eligibility traces, shown in equation (23). This calculation approach is described in (Juan, Sutton, & Ram, 1997).

## GPU Implementation

The GPU agents for this problem continue to use one thread each to implement the learning algorithm. With hindsight, it may have been better to use multiple threads per agent to speed up the calculation of . The agent calculates all three values, , , and , whenever it has to determine the best action from a current state and multiple threads per agent could speed-up these calculations. Multiple threads per agent are used in the final problem, Chapter 7 Learning to Play a Game through Self-Play.

The learning kernel for Mountain Car is similar to the Pole Balancing kernel. The differences come from accumulating gradient values that incorporate the eligibility trace and then using those accumulated gradient values to update neural net weights during the back propagation (Figure 17).

Figure 17: Mountain Car learning kernel.

// learning kernel for Mountain Car

initialize state, s

choose action a, storing Q(s,a)

for each time step

**accumulate gradient and for current state and action**

take action a, returning reward r and new state s’

evaluate Q-values for possible actions using neural net

choose next action a’ from state s’

delta = difference between Q(s,a) and r + Q(s’, a’)

**update neural net weights using delta and accumulated gradient**

a := a’ and s:= s’

next time step

The agents must be tested to determine their learning quality, as in the Pole Balancing problem. Here, the testing method divided the two-dimensional state space into a grid and tested the agent at each point in the grid. The number of time steps to reach the goal is measured and averaged over all points in the grid. The GPU sped up the agent testing process by running tests in parallel.

## Sharing and Differentiation

Sharing among agents cannot be done as it was in the first two problems. Averaging the neural net weights across the agents may produce a result with lower quality than any of the individual agents. If we averaged the agents in that way we would be forcing all agents into one form for the solution that may not be optimal.

Multiple agents can, however, improve results in this problem even without sharing information. There is a wide variation in the quality of the individual agents after a period of learning. The graph in Figure 18 shows the average, median, and best learning quality for a group of 128 agents with no sharing or differentiation. Testing the quality of all agents and selecting the best one can augment the parallel learning approach. The best agent, as revealed by the testing process becomes the result of the parallel learning approach and the time used to test agents is included in learning time.

Learning quality for Mountain Car is the average number of time steps to reach the goal starting from points on the testing grid. In Figure 18, the lower values for time steps to reach the goal correspond to higher quality learning.

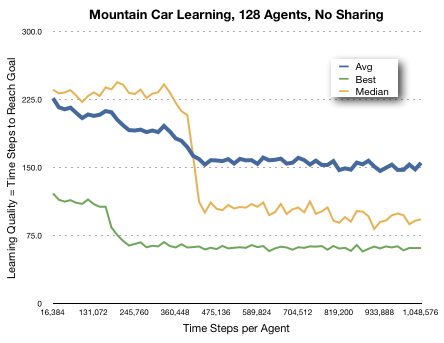


Figure 18: Mountain Car, average, median, and best results for 128 agents with no sharing.

The graph above (Figure 18) shows how identifying the best agent through testing can dramatically improve the speed and quality of learning compared to the single agent result. The multiple agent approach with testing produces quality consistent with the green line, which is the best agent, while single agent results vary widely, centered on the median line and are significantly worse than the best agent.

There is no direct, agent-to-agent, sharing of information in this parallel implementation. We can, however, replace poorer performing agents with copies of the best agent. This form of sharing, “sharing the best”, provided a small additional improvement in learning.

Some problems may not have an easy way of testing the quality of agents on an absolute basis. An alternative to calculating the absolute quality of agents is to test their relative quality through competition. This can be done with Mountain Car by giving two agents identical starting conditions and determining which agent reaches the goal more quickly, then repeating the process a number of times. Determining the best agent through competition and then sharing that agent produces results essentially the same as absolute quality measurement.

This parallel learning approach requires the identification of the best agents so the testing time, or competition time, is included in the learning time for the parallel algorithm and the testing phase is added to the high-level sequence diagram in Figure 19.

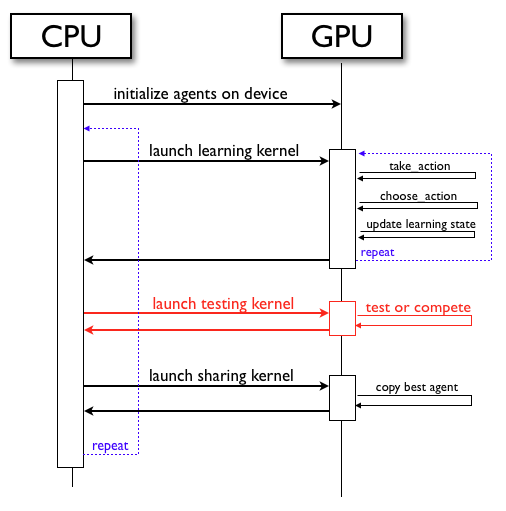


Figure 19: Mountain Car, CPU and GPU processing with additional testing kernel

## Results

Mountain Car is the first problem where we are not sharing calculated values based agent’s learning experience. The benefit from parallel agents on this problem comes from the variability of individual agent results and the ability to identify the best agent from a group of parallel agents either through testing or through competition.

The untimed results of learning quality as a function of agent-time steps is shown in graph Figure 20. The agent-time steps count just the number of interactions between agents and the domain during the learning process. They do not count any time-steps during the testing process. Still, the very large agent groups, 2048 and 8192 performed poorly compared to the single agent as a function of agent-time steps. The 8192 did so poorly no data points are visible on the graph. Groups of 128 and 512 agents did better than the single-agent.

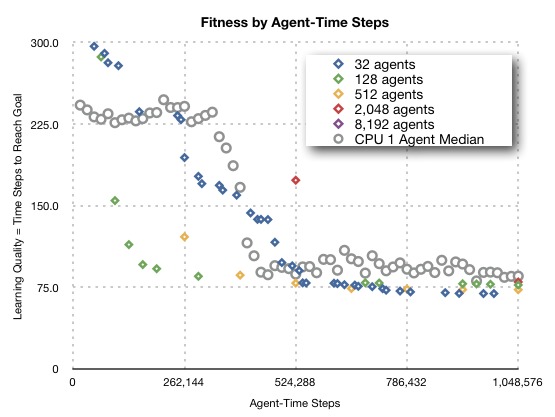


Figure 20: Mountain Car learning quality as a function of agent-time steps for varying numbers of parallel agents.

The results for parallel learning as a function of time, Figure 21, now show two additional factors not reflected in Figure 20. The speed-up of learning due to parallelization is included, as in previous problems, plus the time cost of testing agents. The time value along the x-axis is learning time plus testing time. The graph shows that all agent group sizes, from 32 to 8,192, outperform the single CPU agent in quality as a function of time. The best agent group sizes are from 128 to 2048. The timed results for very large agent groups are hurt by the testing time that grows with the number of agents once the GPU is saturated. The smallest agent group with 32 agents outperforms a single agent, but not as fast as the best group sizes.

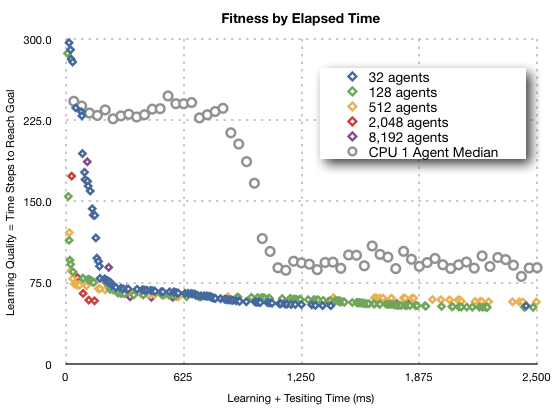


Figure 21: Mountain Car, learning quality by elapsed time, which includes learning time and testing time.

Ultimate fitness level does not vary much by size of agent group, which can be seen in although all the multiple agent results are better than the single agent median result.

The last graph, Figure 21, clearly shows the advantage of parallel agents on the GPU for the Mountain Car problem. The single agent CPU median line, indicated with gray circles, shows the typical learning pattern for a single agent. The agent learns for a while without much gain in fitness, and then has a short period of fast improvement. After the steep drop the learning curve levels out. By running multiple agents in parallel and testing for the best agent, the parallel approach experiences the period of fast learning much earlier than the one agent approach. For agent group sizes of 128, 512, and 2048, the steep learning takes place roughly within the first 100 ms of training, while the single agent median learning burst does not finish until close to one second after the start of training.

In summary, the parallel approach is able to improve on a single agent on the Mountain Car problem without directly sharing information among agents. Instead, the variability of individual agent learning combined the ability to test agents and determine the best one allow the parallel approach to learning more quickly than the single agent.

# Learning to Play a Game through Self-Play

## Overview of the problem

The final problem is to learn to play a game through self-play using parallel agents. The focus is on the ultimate quality of the agent produced, with less emphasis on the speed of learning. In this problem the agent is pre-programmed with knowledge about how the game works. That is the agent can determine the next state of the game for any of the actions that are possible from the current state. It also can determine which actions are legal actions from the current state. The agent does not know the goal of the game and can not determine in advance what board positions are wins or losses, which must be learned through the reward signal. The game uses the knight piece from chess on a 5 x 7 chessboard. The game’s piece movement pattern, number of pieces per side, and board size are variable problem parameters but the results shown here are for knights on a 5 x 7 board and the game starts with the random placement of 5 pieces for each player. Each player makes a move, as in chess, and a player wins if all opponent’s pieces are captured. The game has a maximum number of turns and the result is a drawn game when that limit it is reached.

Learning through self-play was advanced by the famous work by Tesauro (Tesauro, 1994) who used TD(λ) and self play to train a program to play backgammon at a master level. This was dramatic accomplishment and has not yet been duplicated in other domains. Tesauro points out in his work that it may be the random element that is present in backgammon that drives the learning agent into wide exploration of the state space, enabling high quality learning to take place during self-play. The random element helps to avoid the problem of converging prematurely to a sub-optimal strategy that can happen with deterministic games. In those games, the agent may only experience a small fraction of the state space and optimizes for this region only, since by playing itself, it never has to deal with other regions of the state space.

This game is discrete but has a large state/action space. There are  possible starting positions with random placement of 5 Xs and 5 Os on a 5 by 7 board. The total number of game states is even larger since pieces can be removed during the game. The knight can move in up to 8 directions so there may be up to  possible actions. Clearly, there are too many values to learn either  or  for every point and generalization is required. For this problem I chose to use a neural net to approximate the value function . The value function can be used by the agent to pick optimal moves with one move look ahead. The agent loops through possible moves, calculates the board position after each one and selects the move that results in the highest value board position.

### Neural Net Design

The neural net must take a description of the board position and output the agent’s estimated value for that board position, . I coded the board position as two series of binary values for each square on the board. The first series had the value 1 when player X had a piece on the corresponding square and the second series indicated squares with an O piece. A single layer of 4 hidden nodes with sigmoid activation is used and the single output node uses sigmoid activation as well. The number of hidden nodes is a parameter that can vary. The neural net design is shown Figure 22.

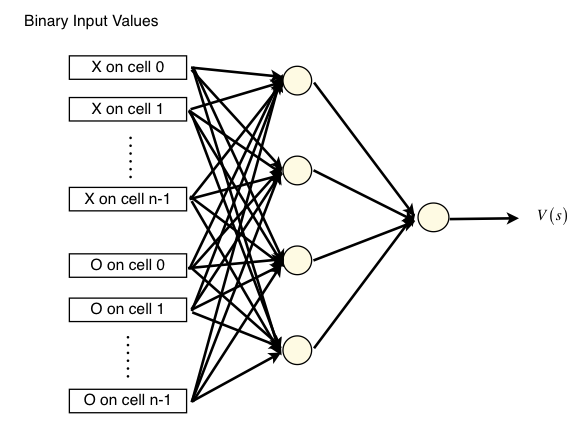


Figure 22: Neural net design for learning *V(s)* on game board of size *n.*

The learning algorithm for this problem is TD(λ), building on the work done for the previous problems and adapting it to learn the  values. The weight update rules combine the temporal difference equation for the value function with eligibility traces and back propagation updates for neural net weights.

The basic framework for learning is to have a group of agents learn by playing a set of games against each other. The relative quality can be gauged by recording the agent’s winning percentage during the learning episodes. This is shown for an 8-agent group in Figure 23. The absolute measurement of agent quality requires some outside benchmark or test to be performed. A benchmark opponent was used to gauge agent quality in Figure 24. The quality measure is winning percentage calculated as .

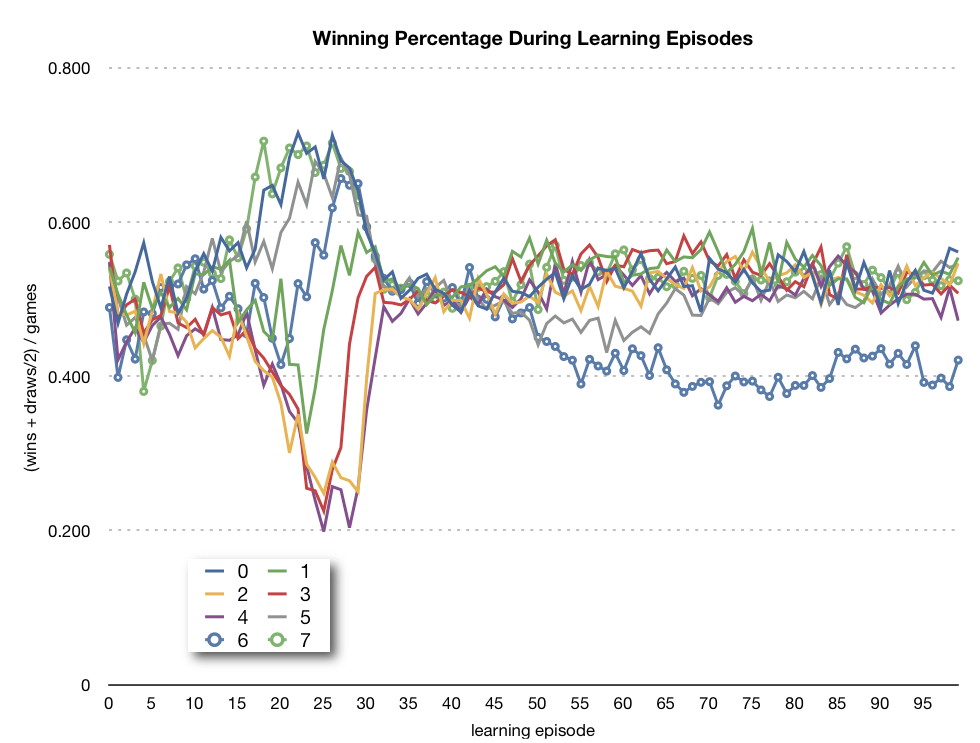


Figure 23: Agent winning percentage during learning, 8-agent group.

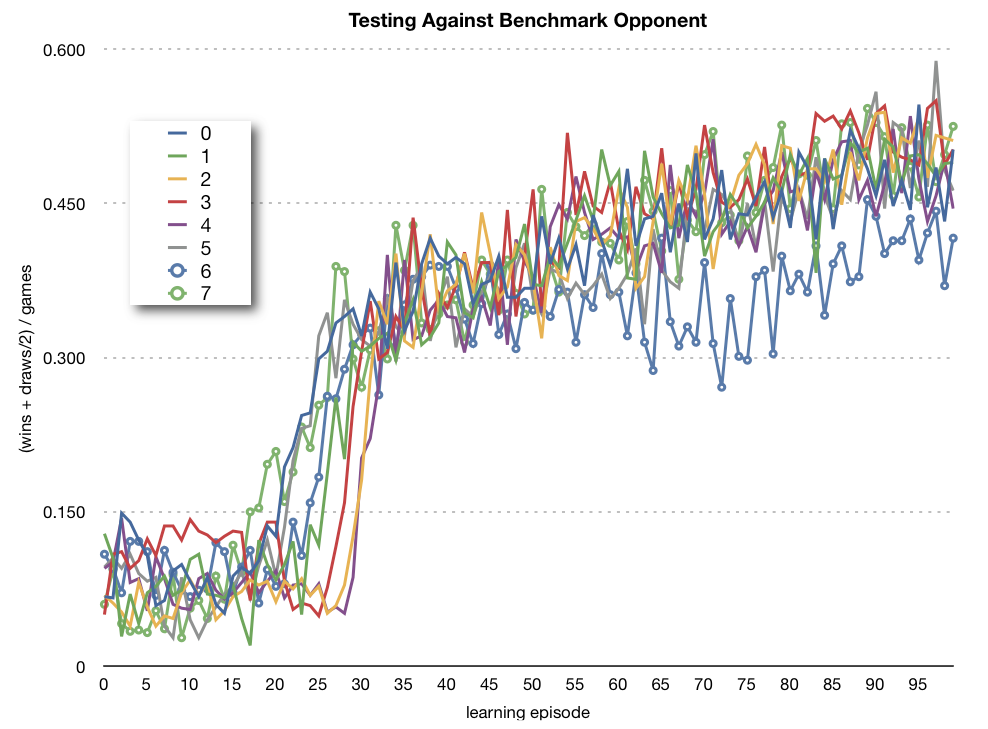


Figure 24: Agent quality measured against a benchmark opponent.

## GPU Implementation Issues

The algorithm used on this problem is very compute intensive. For each turn, the agent must look at all possible moves and estimate the value of the next board position by using the neural net estimate . The GPU implementation uses multiple threads per agent to speed up the neural net calculations and to speed up the weight update calculations that are needed each turn. Each agent has one thread per board square. The entire group of threads within an agent will be active during some portions of the processing. Other times, only a single thread or a smaller group of threads is needed for the calculations. Thread activity for the choose move function is illustrated in Figure 25. In that diagram, the green arrows represent active threads and dashed arrows are inactive threads.

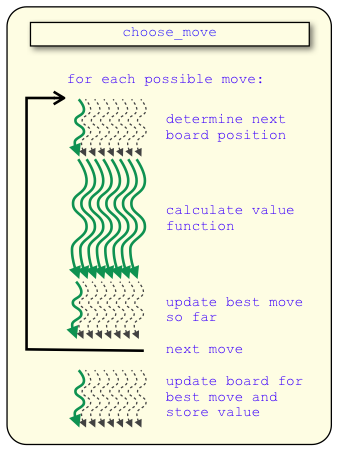


Figure 25: Thread activity within an agent during choose\_move function.

Agents can compete against an opponent in parallel too. The next diagram, Figure 26, illustrates having  agents compete against one opponent at the same time.

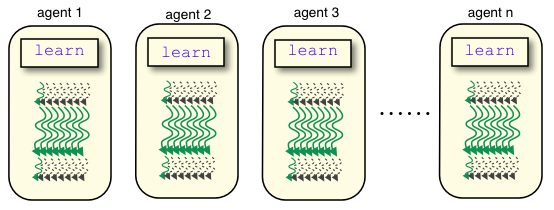


Figure 26: n agents learning by competing in parallel.

Having each agent compete against multiple opponents simultaneously increases parallelism further. This approach requires creating multiple copies of the agent’s weights. Each copy then competes against a different opponent and updates its own weights. At the end of an episode of learning, the change in weights across all the copies of an agent is accumulated and the master copy is updated in one batch, illustrated in Figure 27.

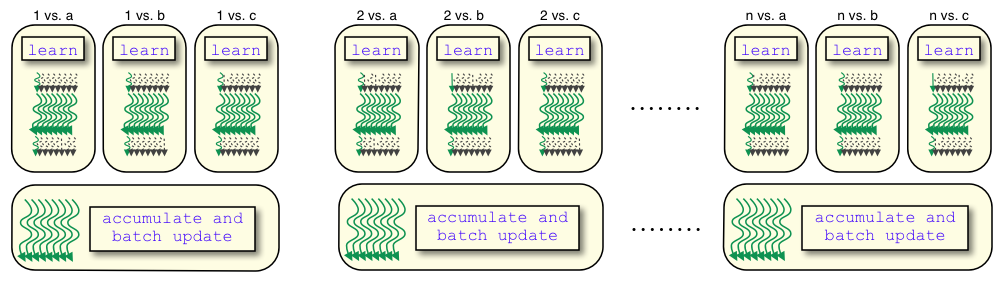


Figure 27: agents playing multiple opponents simultaneously with batch update at the end.

The processing on both the CPU and GPU is more complicated on this problem than on previous problems. The GPU is much slower than the CPU for a single agent, but equals CPU speed with 4 agents and is faster for agent groups of 16 or more, as shown in Figure 28, which displays the learning time for one million turns.

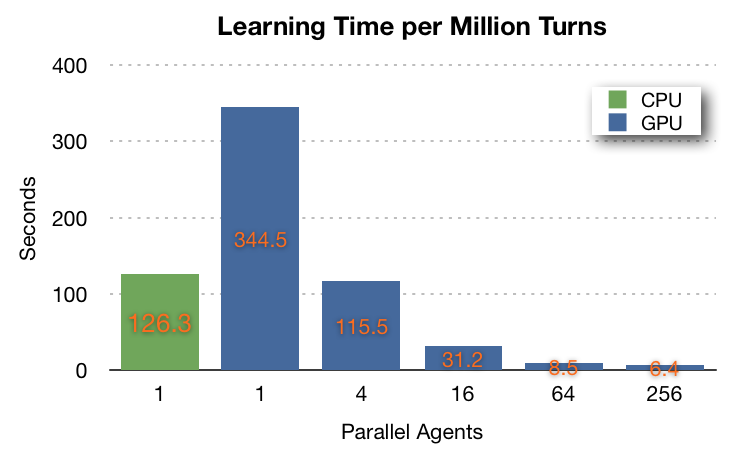


Figure 28: comparison of CPU and GPU speed by number of parallel agents.

## Replication with Variation and Summary of Results

There is no direct sharing between parallel agents in this problem. Indirect sharing happens by agents competing against each other and learning. If one agent’s quality improves, it becomes a better opponent for the other agents and the other agents should improve as a result as well.

This problem produces a wide variation of agent quality, similar to Mountain Car problem. Selecting the best agent from a group of parallel agents, based on internal competition between agents or testing against a benchmark opponent, will improve the quality compared to the average result. Variation in quality can be seen in Figure 29, which shows the best, worst, and average agent quality for a group of 64 parallel agents with quality measured against a benchmark opponent.

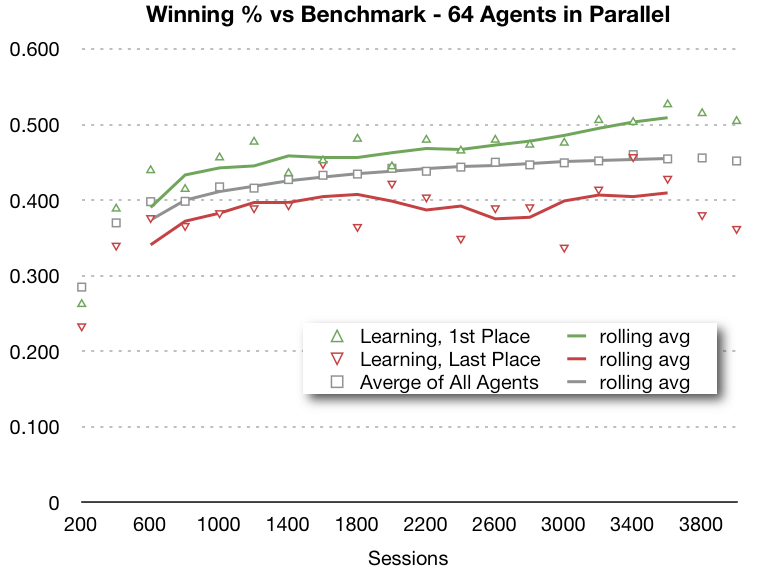


Figure 29: Variation in learning quality for a group of 64 parallel agents.

Another technique that works on this problem is the use of selective replication to improve the overall quality of the agent population. After each learning episode a round-robin competition determines the relative quality of the agents. The best agents are copied and the worst agents dropped. In addition, this has a secondary effect of improving the opponents for all agents in the future. Figure 30 shows the high-level sequence diagram for CPU and GPU coordination.

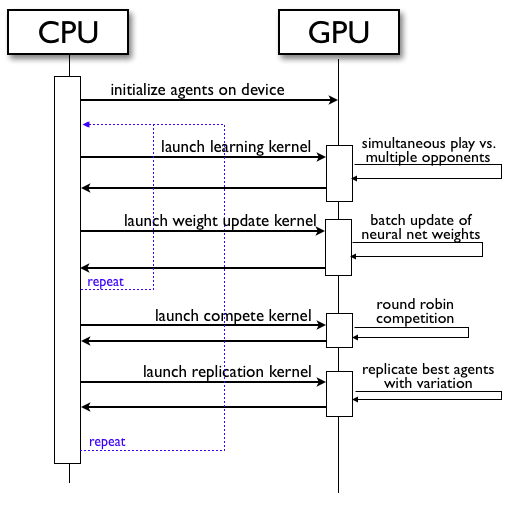


Figure 30: High-level sequence diagram for learning through self-play.

The replicated copies of the best agents can include some differentiation. They can have variation in the learning parameters or slight random bias applied to the neural net weights. Variation helped the population of agents to continue to improve over a long training session. The last graph, Figure 31, shows the typical results when using selective replication with variation with a group of 64 agents.

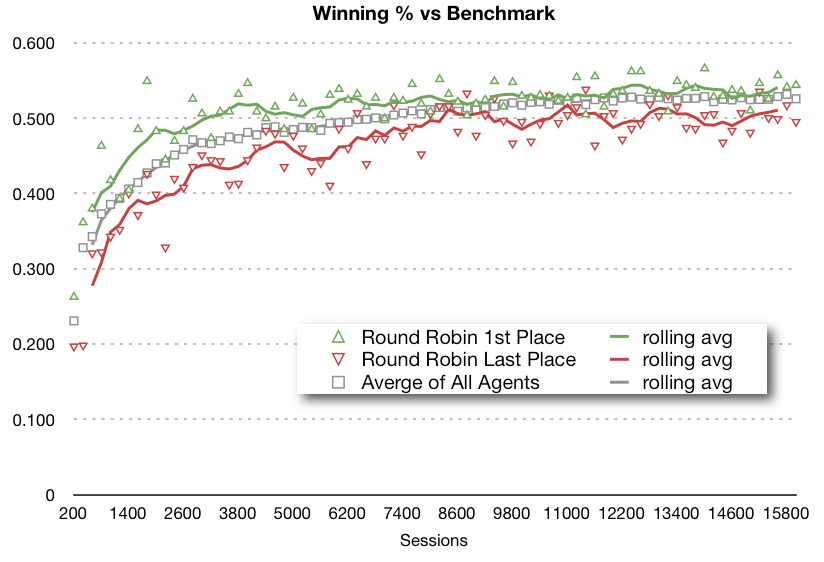


Figure 31: Learning against benchmark opponent for group of 64 parallel agents using selective replication with variation.

The parallel approach to this problem provides a variety of opponents for agents to learn from. This can help overcome the problems with self-play using a single agent. In addition, the evolutionary techniques of selective replication with variation allow the population of agents to continue to improve over time.

# Conclusions and Future Work

This project applies parallel programming techniques on the GPU to four reinforcement learning problems of increasing complexity, demonstrating the variety of benefits that parallelism provides. The algorithms applied to these problems and implemented on the GPU also span a range of complexity. Benefits are observed arising from a number of factors:

* raw speed improvement due to parallelism,
* improved learning through appropriate information sharing strategies,
* differentiation of agents to improve learning results, and
* the benefit of evolutionary techniques with a group of parallel agents.

For the first three problems we looked at the parallel penalty. This penalty is the reduction in learning quality due to dividing a fixed number of agent-time steps over multiple agents, as compared to a single agent learning for the same number of agent-time steps. We saw that, in some problems, the parallel penalty is non-existent, or there is actually an algorithmic benefit to splitting up agent-time steps over multiple agents, most likely due to improved exploration of the state space.

This project demonstrates that the complex algorithms for Temporal Difference Learning with eligibility traces and using neural nets for state space generalization can be successfully programmed on the GPU using the CUDA platform. We also show this platform can handle groups of parallel agents in the thousands.

Offsetting the benefits are the inherent complexity of parallel programming and the additional requirements to design effective sharing, differentiation, and evolutionary strategies to maximize the benefits of the parallel approach. There is also a proliferation of additional learning parameters that must be optimized, such as agent group size, frequency of sharing, testing methodologies, etc.

The application of massively parallel techniques to reinforcement learning algorithms is an exciting area of study with great possibilities to improve on the traditional single agent approaches to reinforcement learning. The application of parallelism to self-play represents a possible avenue to avoiding some of the problems frequently encountered by single agent self-play. This, combined with the rapid improvement of parallel hardware, and in particular GPUs, point to many great applications of massively parallel techniques to reinforcement learning in the future.

Some problems are more suited to implementation on a GPU or may yield greater benefits from the GPU implementation. In general, the GPU is best suited to algorithms that perform the same sequence of steps on each iteration. Algorithms that have many branching points are not as suited to GPU implementation as the processing for agents that take one branch must generally pause and sit idle while the GPU handles agents that take the other branch.

Another issue that impacts the speed of a GPU implementation is memory access patterns. The best performance occurs, in general, when consecutive agents are accessing consecutive memory locations on the graphics card. An algorithm that requires random access to data structures in memory that depends on the agent’s current state will see smaller speed improvements when compared to an algorithm that allows threads to access memory in a coherent fashion.

The GPUs used in this thesis generally have their best performance when they are handling tens of thousands of threads. Parallel algorithms that benefit from this many threads are more suited to GPU implementation than algorithms that derive benefits from a much smaller number. GPU technology and other parallel hardware is evolving quickly and it is likely that the limitations of today’s hardware will diminish over time, enabling the application massively parallel techniques to an increasing range of problems.

There are a number of ways the last problem, parallel learning through self-play, can be extended or improved. The game-playing agent produced by parallel learning can easily be incorporated into a typical game playing algorithm that uses multiple move look ahead and the mini-max algorithm for game tree search. The agent’s value function would then be used as the estimated board value when the maximum depth is reached in the game tree search.

Another possible technique is to improve the value function itself through look ahead. That is take a sampled board position and recalculate it’s value using multiple move look ahead. The calculated value is then used as a training example to improve the weights of the neural net. This self-improvement could even be done in parallel on a second GPU while the agent is learning through self-play on the first GPU.

A major challenge in implementing the programs for this thesis was the difficulty of parallel programming and GPU programming in particular compared to traditional single-threaded programs on the CPU. Correctness of the code is the first problem and was overcome here by coding the identical process on CPU and GPU, then verifying their consistency. The second problem is code optimization on the GPU. There are many ways to organize the threading structure and the GPU memory usage patterns and choosing the best approach is difficult. But the processing power of GPUs is increasing at a fast pace and programming tools are improving as well. This will lead to the increased application of the parallel techniques demonstrated in this thesis to more complex problems in reinforcement learning and other areas of artificial intelligence.

# References

Brownlee, J. (2005 July). The pole balancing problem: A benchmark control theory problem. *Technical Report 7-01* . Swinburne University of Technology.

Buşoniu, L., Babuška, R., & De Schutter, B. (2008, March). A Comprehensive Survey of Multiagent Reinforcement Learning. *IEEE Transactions on Systems, Man, and Cybernetics Part C: Applications and Reviews* *, 38* (2), 156-172.

Grounds, M., & Kudenko, D. (2007). Parallel reinforcement learning with linear function approximation. *The Sixth Intl. Joint Conf. on Autonomous Agents and Multi-Agent Systems (AAMAS 07)*, 213-215.

Howes, L., & Thomas, D. (2008). Efficient Random Number Generation and Application Using CUDA, Chapter 37. In H. Nguyen, NVIDIA Corporation, & H. Nguyen (Ed.), *GPU Gems 3* (Vol. 3). Addison-Wesley.

Juan, C., Sutton, R., & Ram, A. (1997). Experiments with Reinforcement Learning in Problems with Continuous State and Action Spaces. *Adaptive Behavior* *, 6* (September), 163-217.

Kretchmar, R. (2002). Parallel reinforcement learning. *Proceedings of the 6th World Conference on Systemics, Cybernetics, and Informatics* .

Lagoudakis, M., & Parr, R. (2003). Least-squares policy iteration. *The Journal of Machine Learning Research* *, 4*, 1107-1149.

NVIDIA Corporation. (2011). *What is CUDA?* Retrieved 2011 27-February from http://www.nvidia.com/object/what\_is\_cuda\_new.html

NVIDIA Corporation. (2010 йил 10-November). *NVIDIA Cuda C Programming Guide.* Retrieved 2011 27-February from CUDA Toolkit 3.2: http://developer.download.nvidia.com/compute/cuda/3\_2\_prod/toolkit/docs/CUDA\_C\_Programming\_Guide.pdf

Palmer, V. (2007). *Scaling Reinforcement Learning to the Unconstrained Multi-Agent Domain.* Dissertation, Texas A&M University, Computer Science.

Sutton, R., & Barto, A. (1998). *Reinforcement learning: an introduction‎.* Cambridge, MA: The MIT Press.

Tesauro, G. (1994). TD-Gammon, a self-teaching backgammon program, achieves master-level play. *Neural computation* *, 6*, 215-219.

Whiteson, S., Taylor, M. E., & Stone, P. (2010). Critical factors in the empirical performance of temporal difference and evolutionary methods for reinforcement learning. *Autonomous Agents and Multi-Agent Systems* *, 21* (1), 1-35.

1. Glossary

|  |  |
| --- | --- |
| agent-time steps | The sum of all time steps taken by all agents. |
| alpha | A learning parameter,  controls the speed of learning. |
| eligibility traces | A method to attribute the information gained from the latest action to previous states and actions, thereby speeding up the learning process. |
| epsilon | A learning parameter,  specifies the frequency of exploration. |
| lambda | A learning parameter, λ specifies the amount of credit given to past actions for the information just learned. |
| Markov property | Future states only depend on the current state and future actions, not on the past states or actions that led up to the current state. |
| observable states | Complete information about the current state is available to the agent |
| off-line learning | Learning without trying to optimize performance. The agent can explore and take sub-optimal actions to gain the most information for improving it’s learning. |
| on-line learning | Learning while simultaneously trying to optimize performance |
| partially observable states | Complete information about the current state is not available to the agent. |
| policy function | A function, , that maps states to actions. An agent’s goal in learning is to find the optimum policy function. |
| Q-value: | The expected present value of future rewards based on taking action  from state . Assumes the agent will follow some specified policy for future actions. |
| V– value function: | The present value of future rewards based on being in state . Assumes the agent will follow some specified policy for future actions. |

1. Random Number Generation on the GPU

Random numbers play an important role in running the experiments in this thesis. They are used to simulate the random aspects of the problem domain, or to select the agent’s action when its strategy is non-deterministic. The initial implementation of each algorithm is implemented on both the CPU and GPU to produce identical results for testing purposes. Random numbers must also be generated identically on both platforms. The uniform and Gaussian random number generators used in this project are based on methods described in Chapter 37 of GPU Gems 3, made available on the NVIDIA developer website (Howes & Thomas, 2008). The authors describe a Hybrid Generator that combines a Tausworthe Generator with a Linear Congruential Generator to produce random numbers with good statistical quality using 4 32-bit values for seeds.

The agents on the GPU do not execute in a predefined sequence so it is not possible to use one random number generator, with one set of four seeds, and produce consistent results that can be verified by the CPU. Instead, each agent must maintain its own set of random number seeds that are used whenever that agent needs to generate a random number.

Gaussian random numbers, when needed, were generated using a Box-Muller Transform (Howes & Thomas, 2008) applied to two uniform random numbers.