Parallel Reinforcement Learning with Linear Function Approximation

Matthew Grounds and Daniel Kudenko
Department of Computer Science
The University of York
{mattg,kudenko}@cs.york.ac.uk

ABSTRACT

In this paper, we investigate the use of parallelization in reinforcement learning (RL), with the goal of learning optimal policies for *single-agent* RL problems more quickly by using parallel hardware. Our approach is based on agents using the SARSA(λ) algorithm, with value functions represented using linear function approximators. In our proposed method, each agent learns independently in a *separate* simulation of the single-agent problem. The agents periodically exchange information extracted from the weights of their approximators, accelerating convergence towards the optimal policy. We present empirical results for an implementation on a Beowulf cluster.

Categories and Subject Descriptors

I.2.6 [**Artificial Intelligence**]: Learning

General Terms

Algorithms, Performance, Experimentation

Keywords

Reinforcement learning, value function approximation, parallel algorithms

1. INTRODUCTION

In this paper we investigate a parallel approach to reinforcement learning. The primary goal of this approach is to find good solutions to *single-agent* learning problems more quickly by exploiting parallel hardware. This focus differs from most previous work on multi-agent learning which is primarily concerned with agents that share an environment and learn to coordinate or compete. While there have been preliminary analyses [3, 7] demonstrating the promise of a parallel approach to RL, the cost of inter-agent communication has usually been ignored. There are currently no parallel algorithms for RL that are practical for solving large-scale problems using real parallel hardware.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
AAMAS'07 May 14–18 2007, Honolulu, Hawai'i, USA.
Copyright 2007 IFAAMAS .

We present an approach where the parallel agents learn using identical simulations of a given single-agent RL domain. Each parallel agent uses the SARSA algorithm and a *linear value function approximation*. Information about the value functions is then exchanged by asynchronous message passing. By aggregating the information each agent has learned, the agents converge more quickly towards the optimal policy. Empirical results show the performance of this algorithm on a Beowulf cluster of Linux computers.

2. RL BACKGROUND

A reinforcement learning problem can be described formally as a *Markov Decision Process* (MDP). An MDP is a tuple $< S, A, T, R >$, where S is a set of problem states, A is a set of actions, $T(s, a, s') \rightarrow [0, 1]$ is a function which defines the probability that taking action a in state s will result in a transition to state s', and $R(s, a, s') \rightarrow \mathcal{R}$ defines the reward received when such a transition is made. Functions T and R are initially unknown, so the agent must learn about the environment by direct interaction.

SARSA is an RL algorithm which has been shown empirically to work well with linear approximation [5]. The SARSA update rule is defined as:

$$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha(r + \gamma Q(s', a'))$$

If the agent's control policy tends towards greedy choices over time, the $Q(s, a)$ estimates will converge to the *optimal* value function $Q^*(s, a)$ [4].

We use an ϵ*-greedy* exploration strategy, where $\epsilon \in [0, 1]$ gradually decays towards zero. A *linear* function approximator is used to represent $Q(s, a)$, based on a set of n learning features $\{\phi_i(s, a)\}$ (generated by *tile-coding* [6]) and a set of n weights $\{\theta_i\}$. The approximation $\widetilde{Q}(s, a)$ is defined:

$$\widetilde{Q}(s, a) = \sum_{i=1}^{n} \theta_i \phi_i(s, a)$$

Our implementation of the SARSA algorithm with linear function approximation and a replacing eligibility trace is based on the pseudocode given in [6].

3. PARALLELIZATION

In related MDP planning research, parallel methods which *partition* the state space have predominated [1]. In our work we have pursued an alternative approach, where each parallel agent learns an approximate value function for *the whole*

978-81-904262-7-5 (RPS) ©2007 IFAAMAS

state space. This means that agents do not have exclusive specializations, so there may be some duplication of effort. The advantage of this approach is that all the agents can focus on the states with a high probability of visitation.

We assume that an *identical* simulation of a single-agent learning problem is available to each agent. While this restricts our approach to learning in simulation only, almost all practical applications of RL involve some degree of environment simulation, so this assumption is reasonable.

To accelerate learning, the agents periodically *exchange information* over a communication channel, allowing one agent to exploit information learned by another. The environmental knowledge acquired by each agent is expressed in the weights $\{\theta_i\}$ of the linear value function approximation. Therefore we need to define a communication and update mechanism which allows one agent's weights to be affected by another agent. We assume that all the agents use the same set of learning features $\{\phi_i\}$. This means that a weight θ_i has the same meaning for every agent.

4. ALGORITHM

Each of n_a parallel agents start with the same initial weights, perform learning episodes in parallel and communicate every p time steps. They all finish after a fixed time period of real time t_{end}. Originally we considered an approach where each agent broadcast *all* of its weight values every p time steps. This resulted in a lot of redundant information being exchanged between the agents.

For example, consider the change $\Delta\theta_i$ in some agent's value for θ_i since the last communication. If the change is close to zero, it could mean one of two things: (a) feature ϕ_i has not been active since the last merge, or (b) feature ϕ_i has been active, but weight θ_i is already a good prediction of the value of the feature. In either case there is little benefit in communicating the change.

Conversely, the largest values of $\Delta\theta_i$ occur when an area of state space is encountered that hasn't been explored by any of the agents. In these cases, some weights change rapidly to approximate the value function structure in the new area. This is valuable information to communicate.

A variable θ_i^{ref} is used to store a *reference value* for each weight θ_i at various stages. The weights $\{\theta_i\}$ are modified during learning, but the variables $\{\theta_i^{ref}\}$ are not affected, allowing $\Delta\theta_i = \theta_i - \theta_i^{ref}$ to be tracked for each feature. The weights can now be ranked in significance according to $|\Delta\theta_i|$.

Each agent broadcasts a message after every p time steps. We choose a number (n_{com}) of weights to communicate out of the total number of weights (n_{tot}). Parameters p and n_{com} must be selected carefully to trade off communication costs against an improved convergence rate. Even if n_{com} is much smaller than n_{tot} we can achieve significant convergence speed-ups.

When an agent receives a message, each change $\Delta\theta_i'$ received from the remote agent is compared with the local change $\Delta\theta_i$. If $\Delta\theta_i'$ and $\Delta\theta_i$ have *equal signs* it is important that part of the local change is *cancelled out* in response to the remote change (see lines 15–18 of Listing 1). Otherwise it is likely the agents will *overshoot* the true weight value.

Since our algorithm is based on *asynchronous* message passing, the agents are not required to broadcast messages at the same fixed time. In fact, we have found that the best results are achieved by spreading out the messages over time. We employ a *staggered* approach, where the first agent broadcasts p/n_a steps after the start of the parallel run, the second agent after $2p/n_a$ steps, and so on. Each agent then broadcasts after every p steps counting from its first broadcast. In the absence of synchronization, small variations in the dynamic load of processors will disrupt this uniform pattern. As long as random variations keep the broadcasts fairly well spread out this is not a problem.

Listing 1 An asynchronous parallel RL algorithm.

1: Initialize all θ_i and θ_i^{ref} to 0
2: **while** time elapsed $< t_{end}$ **do**
3: Execute one simulation step and update weights $\{\theta_i\}$
4: **if** p steps have been taken since last broadcast **then**
5: Calculate $\Delta\theta_i = \theta_i - \theta_i^{ref}$ for all i
6: *best* ← indices of n_{com} largest values of $|\Delta\theta_i|$
7: $m \leftarrow \{(i, \Delta\theta_i) \mid i \in best\}$
8: Send message m to the other agents
9: **for all** $i \in best$ **do**
10: $\theta_i^{ref} \leftarrow \theta_i$
11: **for all** messages m received since last check **do**
12: **for all** $(i, \Delta\theta_i') \in m$ **do**
13: $\Delta\theta_i \leftarrow \theta_i - \theta_i^{ref}$
14: $\theta_i^{ref} \leftarrow \theta_i^{ref} + \Delta\theta_i'$
15: **if** $sign(\Delta\theta_i') \neq sign(\Delta\theta_i)$ **then**
16: $\theta_i \leftarrow \theta_i + \Delta\theta_i'$
17: **else if** $|\Delta\theta_i'| > |\Delta\theta_i|$ **then**
18: $\theta_i \leftarrow \theta_i^{ref}$

5. EVALUATION

Our evaluation used a Beowulf cluster of Linux computers. Each node of the cluster had a 1GHz Pentium III processor and 768MB of memory. The nodes were connected with a switched 100Mbs Ethernet network. The method was implemented in C++ using the MPICH implementation of the MPI parallel programming interface.

Two of our evaluation domains are established RL benchmark problems: the *Mountain-Car* and *Pole-Balancing* tasks as described in [6]. In addition, we required a domain where problems of increasing difficulty could be defined in order to investigate our method's performance in large-scale problems. For this purpose, we used a stochastic grid world domain which is similar to the *Puddle-world* [5].

In this domain, an agent learns to move along a near-optimal path from a starting position to a goal position in a 2D environment. There are impassable walls in the grid, as well as *sticky areas*, which cause the agent's move actions to fail with a specified probability. An instance of the grid world is defined by a bitmap image file. Note that in contrast to other grid world domains, the state space is not a discrete grid, but a *continuous* 2D state space $\{(x, y) | x \in [0, x_{max}], y \in [0, y_{max}]\}$. This allows a domain instance defined by a single image to be made progressively more difficult by reducing the distance each action moves the agent.

6. RESULTS

The graphs in Figures 1–3 plot the performance achieved (i.e. the mean episode length) against the real time elapsed since the start of the parallel run. In each case we chose the number of runs to firmly establish statistical significance

(clearly separating the confidence intervals), but did not include the error ranges in the graphs to keep them readable.

Figure 1 shows results for a stochastic grid world instance of size 256x256. The linear function approximator used 16,384 weights. The graph shows for different numbers of agents how quickly the optimal policy (with average episode length of about 800 time steps) is approached. A linear speedup cannot be obtained due to increasing communication costs, but quite large speedups can be obtained with 8 or 16 agents. Figure 2 shows similar results for the Mountain Car task, using an approximator with 2430 weights.

Figure 3 shows the results of a slightly different experiment with the Pole-Balancing task. Instead of trying to learn a policy of a given quality more quickly, the goal in this experiment is to use a fixed time period to achieve the best quality possible. The results show that with more agents a policy of greater quality can be learned in the limited time.

For full details of the parameter settings for these experiments, and an in-depth empirical evaluation of our algorithm, the reader is referred to [2].

7. CONCLUSIONS

In a range of domains our parallel RL algorithm as has been shown to learn good policies more quickly with parallel hardware. Empirical analysis has shown that the *quantity* of information exchanged between agents is the main factor limiting performance on a Beowulf cluster. To our knowledge, this is the first parallel RL method appropriate for speeding up learning using real parallel hardware.

In future work, we plan to conduct similar experiments on parallel systems with different network characteristics. We will also examine whether performance can be improved further using alternative criteria for ranking the weights.

8. ACKNOWLEDGMENTS

This research was partially supported by QinetiQ, under the terms of Contract CU004/26749 for the Future Systems & Technologies Division, QinetiQ Ltd, Farnborough.

9. REFERENCES

[1] T. Archibald. Parallel dynamic programming. In L. Kronsjö and D. Shumsheruddin, editors, *Advances in Parallel Algorithms*. Blackwell Scientific, 1992.

[2] M. J. Grounds. *Scaling Up Reinforcement Learning using Parallelization and Symbolic Planning*. PhD thesis, The University of York, UK, 2007.

[3] R. M. Kretchmar. Parallel reinforcement learning. In *Proceedings of the 6th World Conference on Systemics, Cybernetics, and Informatics (SCI2002)*, 2002.

[4] S. Singh, T. Jaakkola, M. L. Littman, and C. Szepesvari. Convergence results for single-step on-policy reinforcement learning algorithms. *Machine Learning*, 38(3):287–308, 2000.

[5] R. S. Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In *Neural Information Processing Systems*, volume 8, 1996.

[6] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998.

[7] S. D. Whitehead. A complexity analysis of cooperative mechanisms in reinforcement learning. In *Proceedings of the 9th National Conference on Artificial Intelligence (AAAI-91)*, pages 607–613, 1991.

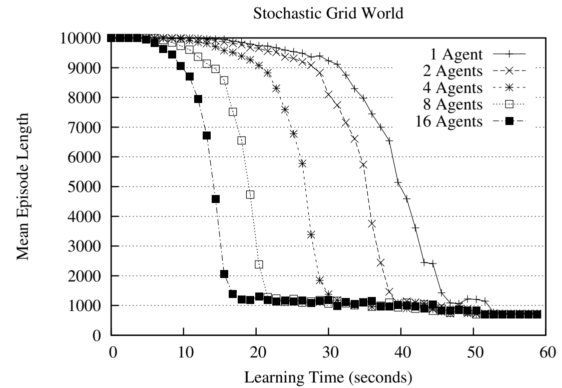

Figure 1: Algorithm ASM in the Grid World task (results averaged over 10 runs).

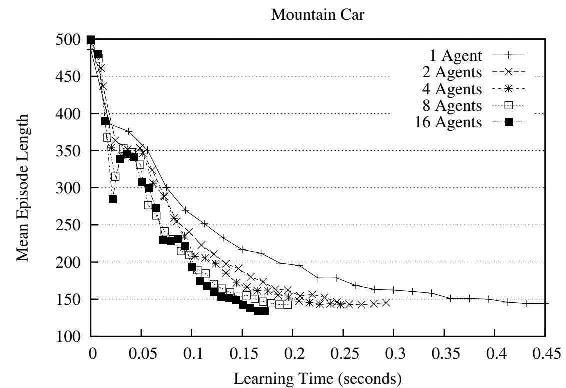

Figure 2: Algorithm ASM in the Mountain Car task (results averaged over 100 runs).

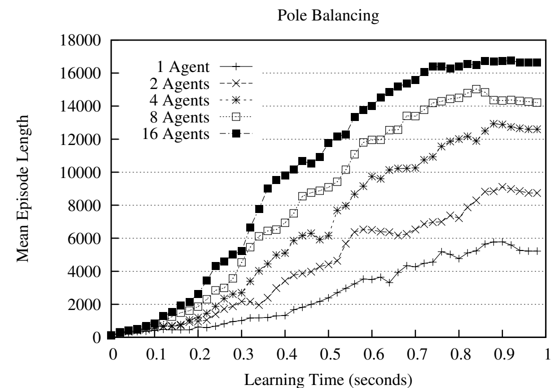

Figure 3: Algorithm ASM in the Pole Balancing task (results averaged over 100 runs).