A parallel implementation of Q-Learning based on communication with cache

Alicia Marcela Printista, Marcelo Luis Errecalde, Cecilia Inés Montoya

Proyecto UNSL Nº 338403[1]
Departament of Informathic
Universidad Nacional de San Luis
Ejército de los Andes 950 - Box 106
5700 - San Luis - Argentina
E-mail: {mprinti, merreca, cmontoya}@unsl.edu.ar
Fax: +54 652 30224

Abstract

Q-Learning is a Reinforcement Learning method for solving sequential decision problems, where the utility of actions depends on a sequence of decisions and there exists uncertainty about the dynamics of the environment the agent is situated on. This general framework has allowed that Q-Learning and other Reinforcement Learning methods to be applied to a broad spectrum of complex real world problems such as robotics, industrial manufacturing, games and others.
Despite its interesting properties, Q-learning is a very slow method that requires a long period of training for learning an acceptable policy.
In order to solve or at least reduce this problem, we propose a parallel implementation model of Q-learning using a tabular representation and via a communication scheme based on cache.
This model is applied to a particular problem and the results obtained with different processor configurations are reported. A brief discussion about the properties and current limitations of our approach is finally presented.

Keywords: Parallel Programming, Communication based on cache, Reinforcement Learning, Asynchronous dynamic programming.

[1] The research group is supported by the UNSL (Universidad Nacional de San Luis) and ANPCyT (National Agency for promoting technology and Sciences).

1. Introduction

This work proposes a parallel implementation of the Q-Learning algorithm on a massively parallel machine using the Parallel Virtual Machine (PVM) message passage library with an efficient communication scheme based on cache.
Q-learning [21, 22] is a Reinforcement Learning (RL) method[4, 7, 8, 11, 15, 16] that deals with the problem of learning to control autonomous agents. The learning process works based in interactions by trial and error with a dynamic environment which provides reward signals for each action the agent executes.
Q-Learning is well adapted for solving sequential decision problems, where the utility of actions depends on a sequence of decisions and there exists uncertainty about the dynamics of the environment in which the agent is situated on. This general framework has allowed that Q-Learning and other Reinforcement Learning methods to be applied to a broad spectrum of complex real world problems such as robot navigation, , industrial manufacturing, games and others.
Q-learning and other RL methods are attractive because they are based on a formal mathematical model (Markov Decision Processes) and therefore a precise definition of the task to solve and its solution is possible. It is probably the easiest method to understand and implement among all the RL methods. It does not assume any previous knowledge about the environmental dynamics and by this reason it turns very attractive to be used in partial or fully unknown domains to the system designer. Besides this, convergence to an optimal policy is guaranted if a tabular representation of the function to learn is used.
In spite of all previous properties presented above, Q-learning suffers the problem of requiring a long period of training for learning an acceptable policy and it is considered very slow fotr practical ends.
In order to solve or at least reduce this problem, this work proposes a parallel implementation model of Q-Learning keeping the standard tabular representation in order to maintain convergence conditions. This model is applied to a particular problem and results obtained with different configurations of processors are reported. A brief discussion about the properties and current limitations of our approach are finally presented.
This paper is organized as follows: section 2 (Mathematical Model) presents a formal model (MDP's) normally used as the underlying theoretical framework for the kind of problems Q-Learning is applied on. Section 3, overviews the main methods for solving MDP's paying special attention on Q-Learning. Both sections have a tutorial and can be skipped by experienced people in these topics. Section 4 shows general issues and implementation details about our parallel model of Q-Learning. Section 5 describes a particular problem and empirical results, concluding this work with some conclusions in section 6.

2. Mathematical Model

Q-Learning is a machine learning algorithm for solving tasks modeled after Markov Decision Processes (MDP).
In this framework, the decision maker (also called the *agent*) is connected via perception and action to something usually named the *environment,* comprising everything outside the agent. At each time step t the agent receives as input from its environment some representation of the environment's state s and selects one of its available actions a in state s. When the agent realizes this action the environment's state changes and one time step later, the agent receives the representation of the new environment's state and a signal of immediate reward r, as a consequence of the previous action.

Figure 1 diagrams this agent-environment interaction:

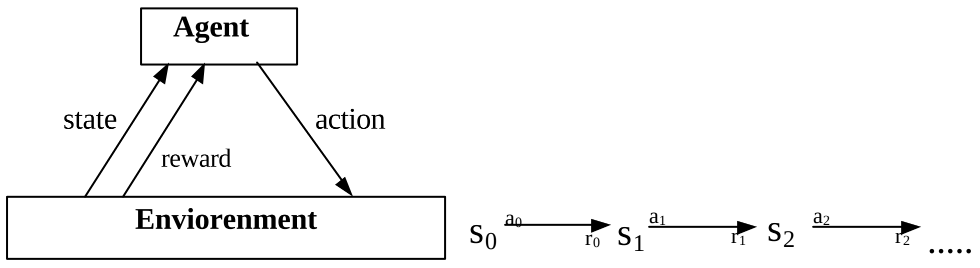
Agent
state
reward
action
Enviorenment
$s_0 \xrightarrow[r_0]{a_0} s_1 \xrightarrow[r_1]{a_1} s_2 \xrightarrow[r_2]{a_2}$ .....

Figure 1

If the agent’s goals are defined by this immediate reward function, the agent’s task is reduced to find out a behavior in which the agent can decide (in each state) which action should be selected in order to maximize the total amount of reward it receives over the long run.
When the environment’s responses are based only on the current state where the agent is situated on and the particular action executed, without influences of previous states and actions, the environment and the task as a whole, are said to satisfy the *Markov property* and can be formally defined as a MDP consisting of:

- A set of possible states S.
- A set of possible actions A
- The one-step dynamics of the environment given by:
  - ∗ The *transition probabilities* $P(s,a,s')$
    $P(s,a,s') = \Pr\{ s_{t+1} = s' \mid s_t = s, a_t = a\}$
  - ∗ The *expected immediate rewards* $R(s,a,s')$
    $R(s,a,s') = E\{r_{t+1} \mid s_t = s, a_t = a, s_{t+1} = s'\}$[2]

For all s, $s' \in S$ and $a \in A(s)$, where $A(s)$ denotes the set of valid actions in s .

And the *solution* for this MDP, consists in obtaining a policy

$$\pi^* : S \rightarrow A$$

that in each time step t, selects the action a_t so that the sum of the discounted rewards that the agent receives over the future is maximized. In particular, it chooses to maximize the *expected discounted return,* R_t:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots\ldots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

where $0 <= \gamma <= 1$, is called the *discount rate.*

[2] The terminology used is the suggested in [16]. Others authors consider to *R* such a 2- variable function: $R(s,a) = \Sigma_{s'} P(s,a,s') R(s,a,s')$

3. Resolution techniques for MDP's. Q-Learning

Previous to the presentation Q-Learning algorithm and its parallel version, it is necessary to understand some general concepts that apply to all techniques usually used for MDP's resolution.
Solving MDP's is normally based on the estimation of *optimal value functions*, defined over either the set of states or the set of actions. Basically, a value function calculates *how good* it is for the agent to be in a given particular state or perform an action in a particular state. Obviously, the *how good* notion will depend on the policy π used by the agent.
Because the goal is to find out a optimal policy π^*, methods trying to solve a MDP deal with the way of learning the *optimal value-state function* (denoted V^*) or the *optimal value-action function* (denoted Q^*). They are formally defined as:

$$V^*(s) = \max_\pi E_\pi \{ R_t \mid s_t = s\} = \max_\pi E_\pi \{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s\}$$

$$Q^*(s,a) = \max_\pi E_\pi \{ R_t \mid s_t = s, a_t = a \} = \max_\pi E_\pi \{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s, a_t = a \}$$

In essence, V^* determines for each state which is the *maximum expected return* that can be obtained from this state, on every possible policy. Policies reaching this maximum are named *optimal policies*.
Q^* on the other hand, measures the expected return for taking action *a* in state *s* and thereafter following an optimal policy.
Although more than one optimal policy can exist (we denote all of them by π^*) they share the same value functions.
Usually, once one has V^* or Q^* it is easy to determine an optimal policy:

$$\pi^*(s) = \max_{a \in A(s)} \sum_{s'} P(s,a,s') [R(s,a,s') + \gamma V^*(s')]$$

Or

$$\pi^*(s) = \max_{a \in A(s)} Q^*(s,a)$$

However, it is often more convenient to learn Q^* than V^* because the *optimal action-value function* allows optimal actions to be selected without having to know anything about possible successor states and their values.
There are several methods for solving MDP's through value function estimations differing essentially on some of the following issues below:

- Learn V^* or Q^*
- Information about one-step dynamics of the environment is available or not
- The optimal policy is obtained in advance to be used by the agent (off-line methods) or it is learned *during* the execution phase, using information obtained as the agent interacts with the environment (on-line methods).
- Updating of value function estimations proceed in systematic exhaustive sweeps of the whole problem's state set (synchronous version) or it is focused onto parts of the state set that are most relevant to the agent in possibly different instants of time (asynchronous version).

The main contribution for solving MDP´s came from the optimal control theory area, through

what is known as *dynamic programming methods*.
The term dynamic programming (DP) refers to a collection of algorithms that can be used to compute optimal policies. These methods successively approximate optimal evaluation functions making use of recurrence relations in these functions.
DP methods have historically worked off-line, focusing on approximations of V^*. They assume knowledge of a complete and accurate model of the environment (one-step dynamics) and normally work in a synchronous way on the *whole* set of states and this reason turn them inadequate for solving big problems.

Asynchronous dynamic programming (ADP) attempted to solve this limitation focusing value function updates on subsets of the set of states. ADP principles are important for our work because its asynchronous features turns these methods as serious candidates for parallel processing[2,3].
Although ADP works in an off-line way, recently an on-line version has been proposed in [1] called *Real Time Dynamic Programming* (RTDP).

When the one-step dynamics of the environment is unknown, the problem to solve is referred as *incomplete information MDP* and methods for its resolution are named in optimal control terminology as *adaptive control methods*. When these methods learn V^*, they have furthermore *estimate* a model of the environment in order to build optimal policies and are called *indirect adaptive methods*. An example of such methods is the on-line method named *Adaptive Real time dynamic programming* [1].

When the adaptive method builds the optimal policy without using an explicit system model, it is called a *direct* or *model free* method.
These methods usually approximate Q^*, and by this reason they can build an optimal policy in a direct way because the one-step dynamics of the environment are not required.

3.1. Q-Learning

Q-Learning is a method proposed by Watkins [21, 22] for solving incomplete information MDP's. From a control theory point of view it is an adaptive direct method and as its name indicates, it is based on learning Q^*, using the following algorithm [16]:

```
1   Initialize (with 0's or random values) $Q(s,a)$ for all $s \in S$ and for all $a \in A(s)$
2   Repeat (for each episode)
3           Initialize $s$
4           Repeat (for each step episode):
5                   Choose $a$ from $s$ using a policy derived from Q (e.g., $\in$-greedy)
6                   Take action $a$, observe resultant state $s'$ and the reward $r$.
7                   $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
8                   $s \leftarrow s'$;
9           until s is terminal
```

Figure 2

When the agent moves forward from a old state to a new one, Q-learning propagates the Q estimations backward from the new state to the old.

In spite of in theory Q-learning cycles forever, in practice the learning task is split in *episodes* (or *trials*) where each episode starts in a initial state given and ends when the agent reaches some condition defined by the learning system designer (e.g., to arrive to a goal or absorbing

state, to exceed a maximum number of iterations).
The α parameter, is named *step-size parameter*, and usually take values from 0 to 1. This parameter is usually decreased step by step.
Under the assumption that all pairs continue to be updated and α is properly decreased, this algorithm guarantees that Q estimates converge with probability 1 to Q^*.
A special remark of Q-learning algorithm deserves the step where the agent chooses an action *a* in state *s* (see Figure 2 – line 5). If Q^* estimations (Q table) were the true values of Q^*, the best option for the agent would be obviously to select the action with maximum Q value (*greedy policy*). In this case we say that the agent is *exploiting* its current knowledge of the values of the actions. Unfortunately, this is not a realistic scenario because the Q-table is just an estimation of Q^* and this estimation will be accurate if the agent *explores* new states and actions and not just the suggested by the current values of Q.
Because it is impossible for the agent to simultaneously explore and explode it with a single selection of action this conflict is often named as the *exploration/exploitation dilemma*.

The stochastic or changing nature of the environment with possible locally optimal actions, requires a minimum of exploration selecting actions not suggested by the greedy strategy.
A lot of approaches have been suggested to face this problem[19, 20]. We selected in our experiments a simple but effective strategy named *Î-greedy*. This strategy works in a greedy way by default but with probability *Î* chooses a random action.

4. Parallel Q-Learning

The main problem of Q-learning algorithm is that the agent requires a great number of trainig episodes to learn an acceptable value function [4]. Today, there exist two principal approaches for speeding up of the learning process:

1) To allow the input of information provided by external observer [9, 10, 13] and
2) To integrate learning with planning processes [9, 12, 13, 17, 18].

The first one, consists in allowing an external observer to be able to incorporate pieces of advices in order to help the agents to learn complex aspects of the environment in an efficient way.
In the second case, the agent learns a model of the environment simultaneously with the learning of the policies to make a more intensive use of the limited amount of the experience.

In this work, an alternativer approach is proposed, which is based upon a parallel implementation of the Q-learning algorithm. The tasks will be distributed among processors that have the capacity to work in a parallel way. The principal motivation on the development of this implementation has been to increase the processing speed. But, in some cases, the parallel processing introduces an extra motivation. This motivation suggests that the parallel processing betters the quality of the learning process output because it allows to discover internal relationships of the problem that a conventional sequential approach is not able to explore.
The parallel programming is based on some partitioning strategy which divides the problems in different parts. Once this parts are completed, their partial results must be combined to obtain the desired result. The partitioning techniques can be applied to the function of the program (called *functional decomposition*). In this approach, to divide the original sequential program in parallel, independent and synchronic units, implies to identify inter-module relation to determine where the parallel implementation is safe [6].
Another techniques can be applied to data (called *domain decomposition*). This partition strategy was the one used to obtain a parallel version of the Q-learning algorithm. This implementation keeps two or more identical process operating over different fractions of the Q-table. This means that each process will be focused on the learning process over only one

subgroup of states and their associated actions. Nevertheless, some interactions will be needed; at some point of this learning, the process may require non local states information that belong to a subgroup of neighboring states.

Depending on the specific data partitioning strategy and the number of partitions is that this interaction could considerably speed up the communication between neighboring processes. This process includes strong synchronization requirements in order to prevent that the processors remain waiting for each others. So, this suggests that an unique process of specific purpose executes this operation and therefore only one synchronization operation will be necessary for exchanging information about Q-table with the global master processor. Due to the later, the resultant structure of the algorithm is Master/Slaves[5, 23].
The master will be responsible of:

- Deciding over which portion of the Q-table will each slave work,
- Distributing the information among the slaves processes and later,
- Maintaining an update version of the Q-table.

On the other hand, this table will be accessed and updated by the slaves through the following requirements to the master:

- *req_msg:* each slave executes the process shown in fig.2. A process slave will establish communication with the master when it requires the maximum value of an foreign state (see line 7, in fig.2). After a *req_msg* was sent, the slave process receives from the master the required information. In the Q-learning parallel version this situation is considered as an other condition of the end of episodes, besides the two used in the sequential algorithms (to reach an goal state or overcome a maximum number of steps).
  In the original implementation of Parallel Q-learning, a *req_msg* was sent for any information requirement which resulted in a high communication overhead. To lessen this problem, it was implemented a *cache* that keeps the last value provided by the master, which is only updated if the maximum value of requirement reaches a specified value by the *cache_size* parameter.
  If too high values were chosen for *cache_size* then the processors would execute with very little communication among themselves, but the performance of the learning algorithm would be relatively low, because most of the time the process would be working with outdated values of their neighboring states.
  If, on the other hand, too low values were chosen, the performance of the algorithm, regarding the quality of results, would be close to a sequential Q-learning. In this case, the communication overhead would be considerably high because the amount of messages exchange among slaves for working with updated information .

- *inf_msg:* each slave processes sequentially over its local groups of states. To keep the global Q-table updated in the master, the slave processors would periodically send their Q-table partition. To reduce the number of *inf_msg*, this approach was better conditioning the sending if *inf_msg* to the existence of some change of the local Q-table of each slave.

The program was executed on a *Power Mouse Parallel Machine*. It is a “host-based” system. Its Operating System is Solaris with the *PARIX* environment[14]. PARIX (PARallel extension to unIX) is the operating system of Parsytec parallel computer, which gives an environment for message passing programming and SPMD (Simple Program - Multiples Data) programming by means of its *PowerPVM* systems. In this system the communication is based in threads.

5. Experimental results

This section presents a comparative analysis of results obtained using the traditional sequential approach of Q-Learning and our parallel implementation.
The test for optimality was performed by comparison with the control law obtained from full dynamic programming (Value Iteration method) using the true simulation.

5.1. Problem description

The selected problem for performance comparisons is a two-dimensional maze similar to the kind of problems commonly used in this area.
In this case, we used a maze consisting of 135 cells (states) (see Figure 3). Each state has four possible actions: RIGHT, LEFT, UP, DOWN, which results in the agent deterministically moving to the corresponding neighbor cell. Blocked actions (by a barrier or a maze border) do not move the agent which stays in the same state. Each transition produces a reward of 0 from the environment, except when the agent reaches one of both terminal states (named **G** in Figure 3). In this case the agent receives 100 units of reward. The discount factor γ is 0.95. The episodes always start in a random non terminal state. This process is repeated at the end of each episode .

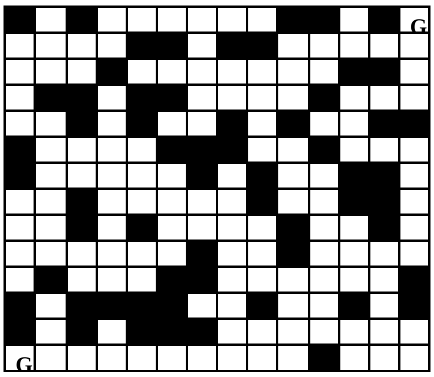

Figure 3

In our experiments the end of the episode are determined when the agent reaches a goal state or overcomes a maximum number of steps per episode (we will refer this value as MSE)
The agent´s task is maximize the sum of the discounted rewards received over the future. For this problem that is equivalent to reach the nearest terminal state **G** (with reward 100) starting from a arbitrary state of the maze and using the minimal possible number of steps.
For this problem, the values corresponding to the optimal policy (obtained with the Value Iteration method) are:

- Average of V^* values over all non-terminal states = 58.3419.
- Average of minimal number of steps required by the agent to reach the nearest G state, starting from every non terminal state = 12.2556.

5.2. Results Analysis

The parameters used in the sequential and parallel Q-learning algorithms were α=1 (step size) and *MSE = 10000* (maximum steps per episode), *Î = 0,1* ($\in$-greedy method as exploration policy) and for testing the convergence to an optimal policy was chosen *ME = 32000* (maximum episodes number). All of these parameters were selected from the best results obtained in the sequential version of Q-learning algorithm.
The parallel versions include another condition for the end of episode needed when a slave

process requires information about one foreign state (this state belongs to another processor).

Sometime, the works related with Q-learning algorithms use as a reference the results obtained in each episode. But, this kind of measures was not possible in the parallel version because it has multiples conditions of end of episode. In order to test the results, we used a time period of 8 microseconds called *epoch*. The algorithm begins in the initial state (the non terminal states), then the agent processes in a *greedy* way and for each *epoch* the current state of learning is registered for the optimal policy.

The parallel Q-Learning version, was executed with *np*= 2,4 and 8 processors. The whole set of states was equality partitioned among current processors following the numeration of states. The result was a horizontal partition of the maze.
In the experiment the influence of *cahe_size* parameters whose values were 1,2,4 and 8 was studied.
Tigure 4 shows the average times taken for the parallel and sequential algorithms where they converged to the optimal policy. Those times correspond with the best results on different *cache_size* values: *cache_size* =8 for *np*=2*; cache_size* =4 for *np* =4 and *cache_size* =8 for *np* =8.

np	1	2	4	8
time	4.326607	3.40059	1.85736	2.659
Speedup	-----------	1.27	2.32	1.62

Figure 4

The parallel algorithm exhibits a speedup up to *np=4*. However, increasing *np* (8, 16, ...) does not produce an increment in the speedup because the overhead of communication is greater than the load assigned to each processor.

The performance of the parallel version is very satisfactory. Nevertheless, the figure 5 give us a more detailed idea about of the learning process. The figure shows the percentage of optimal states learned in different time stamps (every 50 epochs).

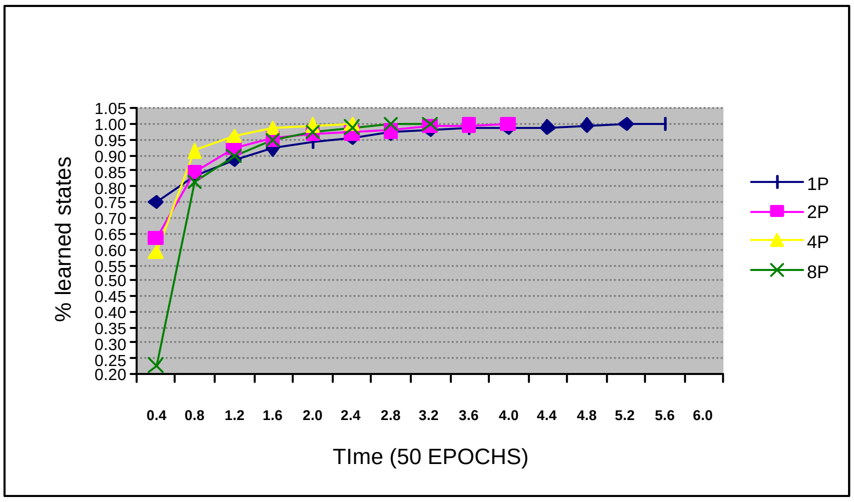

Figure 5

Figure 5 shows the global percentage of learning that consists in to take the percentage of states in which the action suggested for Q-table agrees with the optimal policy.
This percentage is inversely proportional to *np* in the firsts epochs, because the slaves processes that work on the intermediate states of the maze have no relevant information from the neighboring states. However, when the information is available on the whole set of processors, the convergence to the optimal policies is faster than the sequential one and consequently the parallel time will be lesser than the sequential time.

The last thing to be considered is about the confidence level of convergence to the optimal value.
The result on figure 4 does not tell anything in relation to the percentage of running in which they converged exactly to the optimal policy.

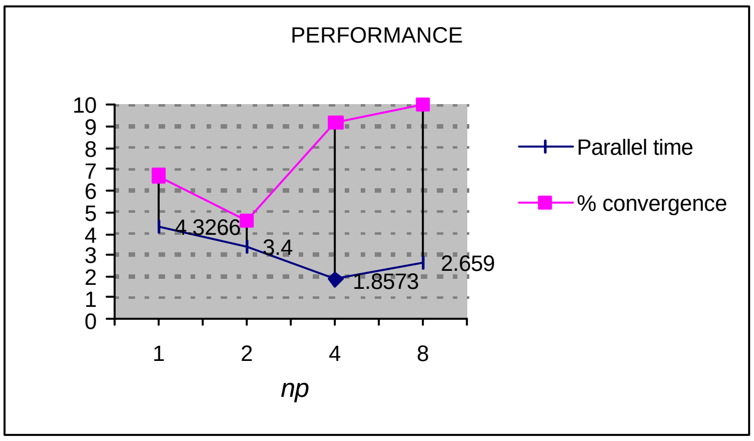

Figure 6

We believe that an analysis more realist should take into account this aspect. Then, the figure 6 shows the convergence time to the optimal policy and the percentage on running (multiplied for 10) in which they converged exactly to the optimal policy.

It can observe that the parallel version with *np=4* reaches good times and its convergence percentage is near to 90% (for 32000 episodes). However for *np=8* the time is greater than the last one, but the percentage is 100%. This indicates that an increment in the number of processors cannot improve its time but can reach a better convergence percentage when the number of episodes is restricted.

6. Conclusions

In this paper, we have presented a parallel implementation of Q-learning algorithm based on an communication system with cache. The results obtained show the feasibility of the approach related to the convergence percentage and also related to the speedup. Even the problem developed was small (only 135 states) reaches goods speedups. It is important to note that current results will have a fully impact when the approach be applied to a real sequential decision problem with thousand or million states.

Finally, we have also shown how one domain partitioning strategy did easy the task and in spite of that it will be possible to scale to larger problems.

7. References

[1] A. G. Barto, S. J. Bradtke y S. P. Singh. "Learning to act using real-time dynamic programming". *Artificial Intelligence*, 72: 81-138, 1995.
[2] D. P. Bertsekas. "Distributed dynamic programming". *IEEE Transactions on Automatic Control*, 27: 610-616, 1982.
[3] D. P. Bertsekas y J. N. Tsitsiklis. "Parallel and Distributed Computation: Numerical Methods". Prentice Hall, Englewood Cliffs, NJ, 1989.
[4] M.Errecalde, M. Crespo y C. Montoya. "Aprendizaje por Refuerzo: Un estudio comparativo de de sus principales métodos". Proc. del II Encuentro Nacional de Computación (ENC´99). Sociedad Mexicana de Ciencia de la Computación. México, 1999.
[5] I. Foster "Designing and Building Parallel Programs". Addison Wesley – 1995.
[6] R. Guerrero, F. Piccoli y M. Printista. " Parallelism and Granularity in an Echo Elimination System". Proceedings of 12th. International Conference on Control Systems and Computer Science. Vol. II – pags. 232-237, Bucharest, Romania, 1999.
[7] L. P.Kaelbling, M. Littman y A. Moore. "Reinforcement Learning: A Survey". Journal of Artificial Intelligence Research 4 (1996) - 237-285 - Mayo 1996.
[8] L.P.Kaelbling. Learning in Embedded Systems. MIT Press. Cambridge, MA, 1993.
[9] L. - J.Lin. "Self-Improving Reactive Agents Based on Reinforcement Learning, Planning and Teaching". Machine Learning - Volumen 8 - Número 3/4 - Mayo 1992.
[10] R.Maclin y J. W. Shavlik. "Creating Advice-Taking Reinforcement Learners". Machine Learning - Volumen 22 - Págs. 251 - 282. 1996.
[11] T.Mitchell. "Machine Learning". Capítulo 13. (Versión preliminar).
[12] A.Moore y C. Atkeson. "Prioritized Sweeping: Reinforcement Learning with Less Data and Less Time ". Machine Learning - Volumen 13 - Número 1 - Octubre 1993.
[13] J.Peng y R. J. Williams. "Efficient learning and planning within the Dyna framework". Adaptative Behavior, 1(4), Págs. 437-454, 1993.
[14] Power Mouse.in details "http://www.parsytec.de/top/products/pm-detail.htm
[15] S.Russell y P. Norvig. "Artificial Intelligence. A modern Approach".Prentice –Hall –1995.
[16] R.Sutton y A. Barto. " Reinforcement Learning: an introduction". The MIT Press, 1998.
[17] R.Sutton. "Dyna. an Integrated Architecture for Learning, Planning, and Reacting" Working Notes of the AAAI Spring Symposium, pp.151-155, 1991.
[18] R.Sutton. "Integrated Architectures for Learning, Planning, and Reacting Based on Approximating Dynamic Programming". – Proceedings of the Seventh Int. Conf. On Machine Learning, pp. 216-224, Morgan Kaufmann, 1990.
[19] S.Thrun y K. Moller. "Active exploration in dynamic environments". Advances in Neural Information Proccesing Systems, 4, pags. 531 - 538. San Mateo, CA, Morgan Kaufmann, 1992.
[20] S.Thrun. "The role of Exploration in Learning Control". Handbook in Intelligent Control: Neural, Fuzzy, and Adaptative Approaches, White, D. A., & Sofge, D. A. (Eds.).
[21] C. J. C. H. Watkins. "Learning from Delayed Rewards". PhD thesis, Cambridge University, 1989.
[22] C. J. C. H. Watkins y P. Dayan. "Q-Learning". *Machine Learning*, 8: 279-292, 1992.
[23] B. Wilkinson y M. Allen "Parallel Programming: Techniques and Application using Networked Workstations and Parallel Computers". Prentice Hall. – 1999.