Reinforcement Learning: A Distributed Implementation using MPI

Aleksandra Obeso Duque

1 Introduction

Machine learning is a group of bio-inspired algorithms that have been designed in order to tackle problems that are still very hard to solve or that do not have a deterministic solution method yet. One example of such techniques is know as Reinforcement Learning [Engelbrecht 2007], which is an algorithm bio-inspired on the way human beings learn to solve a particular task by experience.

In reinforcement learning, an agent learns by interaction within a particular environment in order to maximize some long-term measure of success or minimize a cost function.

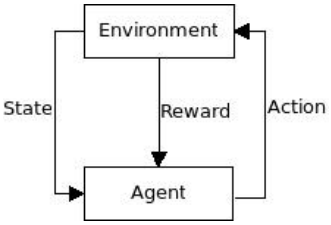

Figure 1: Reinforcement learning: Agent's interaction within an specific Environment.

As shown in Fig. 1, the environment can be described by a number of states and a reinforcement signal know as reward that is a quality measure of the environment's state. The agent can then execute a set of actions within the environment

in order to explore it. Its actions modify the environment and indirectly the future states and rewards.

The interaction between the agent and the environment under a trial and error scenario, leads to the learning of a particular task through time. Fig. 2 shows the agent's diagram which is characterized by a Learning System and an Action Selector.

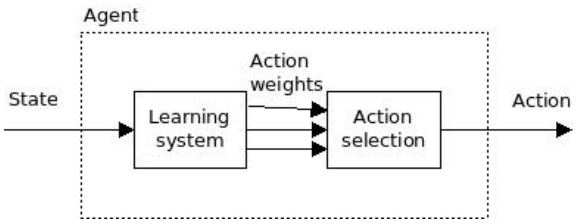

Figure 2: Reinforcement learning: Agent's diagram.

Due to the fact that for more realistic scenarios, the search space could grow affecting the performance of the algorithm, it might be interesting to analyze how to come up with a parallel implementation for the exploration algorithm.

The present work gives a more detailed description of the regarding problem, a solution proposal for a distributed implementation of reinforcement learning using MPI and some performance analysis. Finally, a future path is drawn in order to improve the results here obtained.

2 Problem description

In this particular work, the *environment* the agent interact with, is represented by a 2D grid-world (Fig. 3). The *agent* receives an immediate *reward* when it reaches the trophy state and it can execute 4 possible *actions*: Move north, south, west or east. As a side effect, the agent also discovers the optimal path to reach the trophy from every state (red arrows).

The *learning system* uses a temporal difference learning methodology known as *Q-learning*, which associates Q-values to state-action pairs. $Q(s, a)$ is the estimated value of doing action a in state s, assuming that all future actions are greedy. The update rule is then given by:

$$Q(s, a) \leftarrow Q(s, a) + \eta[r + \gamma \max_{a' \in A} Q(s', a') - Q(s, a)] \tag{1}$$

Where η represents the learning rate, r the immediate reward, γ the exploration rate. As it is stated before, γ represents the relation between the exploration vs. the exploitation strategy the agent uses at taking actions.

Eq. 2 must hold if all Q-values are correct. It can be proved that the Q-values will converge to their correct values, given that all states are visited enough amount of times.

$$Q(s,a) = r + \gamma \max_{a' \in A} Q(s', a') \tag{2}$$

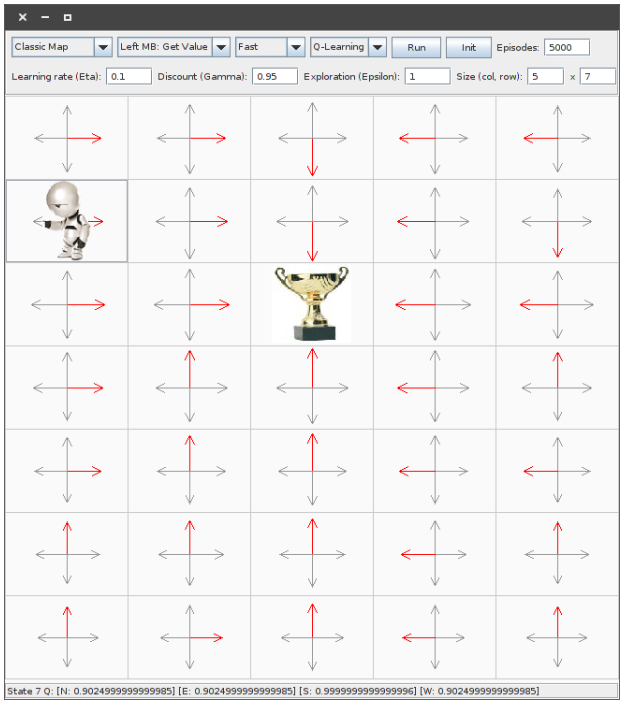

Figure 3: Reinforcement learning: Small grid-world scenario.

Since the state-action search space could be very large as the grid-world and/or the available actions grow, the performance of the learning process could be improved using some parallel implementation for the exploration algorithm.

3 Solution method

3.1 Reinforcement Learning Implementation

For the implementation of the reinforcement learning algorithm, there are mainly four matrices contiguously allocated as arrays in memory, one per each Q-value action in each state of the 2D grid-world. Based on the Eq. 2, an estimation matrix containing the converged Q-value for each state is generated in order to use it as convergence measure based on a maximum error given as an input parameter.

For each simulation, the trophy location and the initial point for the agent are chosen randomly. When the agent reaches the trophy a reward is given, it is then absorbed by this state and reallocated into a random starting point to run the exploration algorithm one more time, until all the states converge to the expected Q-value. The strategy used in this work for *action selection* is known as ϵ-greedy, with probability ϵ of exploration by selecting at random, otherwise be greedy.

The basic input parameters are:

- m, n: Grid-world size [2, ∞)
- eta: Learning rate (0, 1]
- $gamma$: Discount factor (0, 1]
- $epsilon$: Exploration rate: Greedy -> Random [0, 1]
- max_error: Convergence error (0, ∞)
- $reward$: Reward value [1, ∞)

3.2 Distributed Implementation using MPI

To parallelize the reinforcement learning algorithm, the 2D grid-world is partitioned in order to assign subgrids to each processor, also considering non-uniform cases using static partitioning. The processors are arranged in a 2D grid using *MPI_Cart_create()*.

Nodes generate their own convergence submatrix based on the global location of the trophy that is communicated using *MPI_Bcast()*. Each processor then represents an agent exploring a subgrid in a cooperative work where their experiences are shared among them via the Q-values. Since the reward should be propagated through the whole grid, it is necessary to share halo points among adjacent nodes. The Cartesian communicator is split into rows and columns using *MPI_Comm_split()* and specific data types are created using *MPI_Type_vector()* to ease the communication process between them.

Two distributed implementations are developed in order to communicate the Q-values for all four actions on the halo points; one using blocking and one using non-blocking communication. However, the non-blocking communication needs some future refinement in order to split the inner points (overlapping allowed) and halo points (non-overlapping allowed due to data races).

In order to check convergence, a global convergence flag is communicated among nodes using *MPI_Allreduce()* with an *MPI_LAND* operation on the local convergence flags.

Since the workload of adjacent nodes depends on the update of the halo points, at the first time steps the workload is going to be very low due to the initial zero values compared to the overhead of the communication. To overcome this problem, $comm\_steps$, a basic parameter is introduced to specify how frequently

nodes communicate among them. However, a better approach is proposed as future work.

Some important points considered at the time of parallelizing reinforcement learning in this work are:

- If an agent reaches a halo point it is absorbed but no reward is given. It is only reallocated into a random point to continue with the exploration process inside its subgrid.
- If a reward is randomly located in the halo points, some kind of indication should be communicated to the neighbor node so the algorithm can converge. In this work, due to time constraints, a restriction was given in order for the algorithm never generates the reward on the halo points (that also restrings the minimum size of an inner subgrids to 4x4 considering at least one cell for the trophy and one for the agent).
- Since this implementation only consider one reward state in the whole grid, when this one grows too big, the Q-values for states very far from the reward become negligible when using the typical reward value of 1, which also means an almost negligible workload. To overcome this problem, a very high reward value is assigned when working with huge grid-worlds.
- In pro of energy efficiency, a first implementation considering local convergence was done where nodes that converged were stopped modifying the neighbor flags when needed, so the neighbor nodes do not get stuck waiting for the communication with the converged ones. However, the error propagation does not allow the convergence of the whole algorithm under the specified max_error, that is why the global convergence strategy already mentioned was used instead.

The input parameters added to the parallel algorithm are:

- $comm\_steps$: Number of time steps between communication $[1, \infty)$
- px, py: Processor grid size $[1, \infty)$

4 Experiments

For testing performance, the following Q-learning parameters were used:

- $eta = 1.0$
- $gamma = 0.99$
- $e\_greedy = 1.0$
- $max\_error = 0.0000001$
- $reward = 100000000$

The given grid-world sizes were in [100x100, 250x250, 500x500, 750x750, 1000x1000] and the processor grids in [1x1, 4x4, 5x5, 6x6, 7x7]. Since it is important to check how the *comm_step* parameter affects the performance, its values were also varied in [100, 1000, 10000].

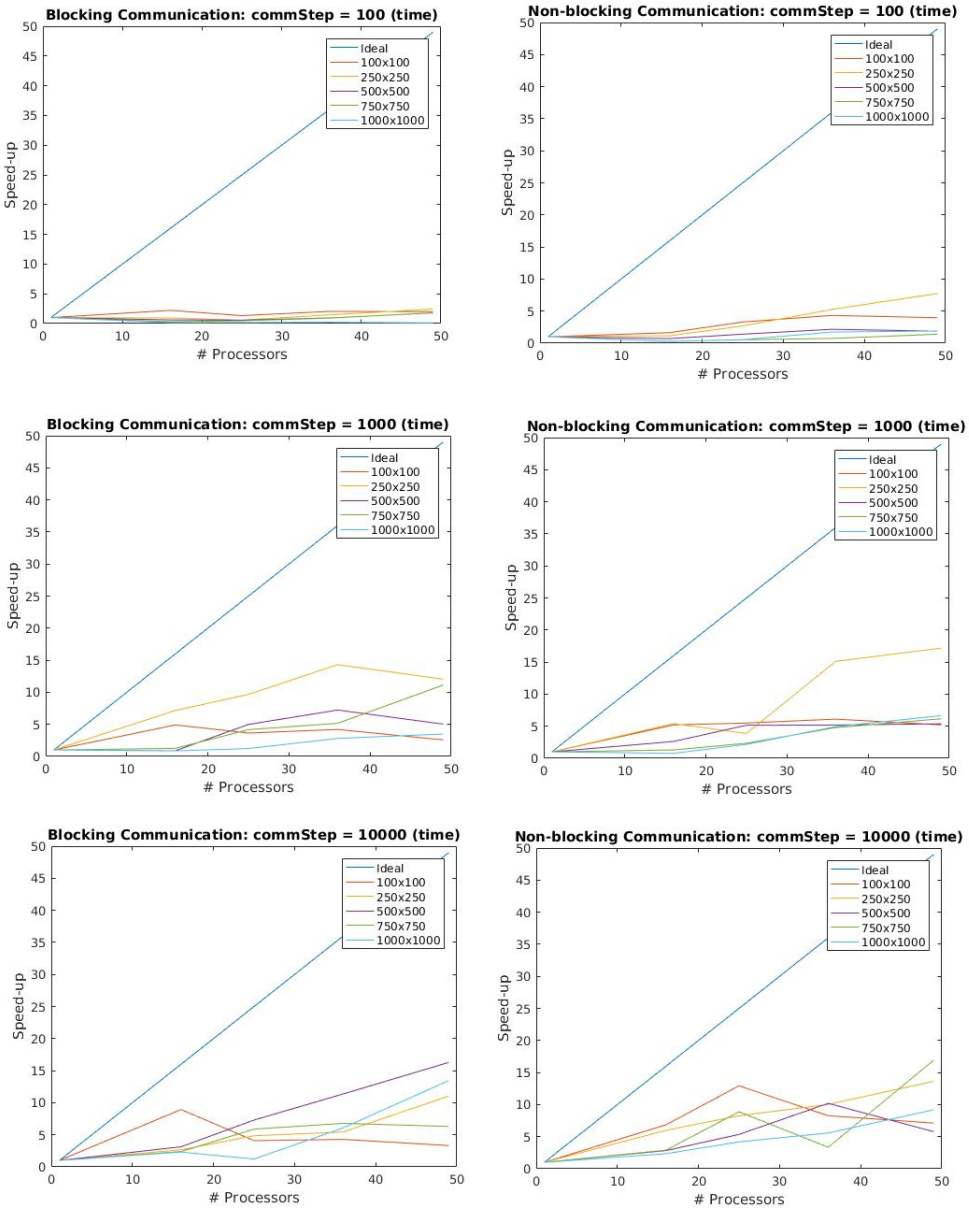

Figure 4: Distributed reinforcement learning: Strong scalability based on time.

Figs. 4 and 5 show the obtained strong scalability for both blocking and non-blocking communication, for different values of $comm\_steps$ based on time and in number of epochs or time steps. As it can be observed, the randomness of the exploration algorithm makes hard to establish a comparison point. However, some conclusions and improvements are presented in the next section.

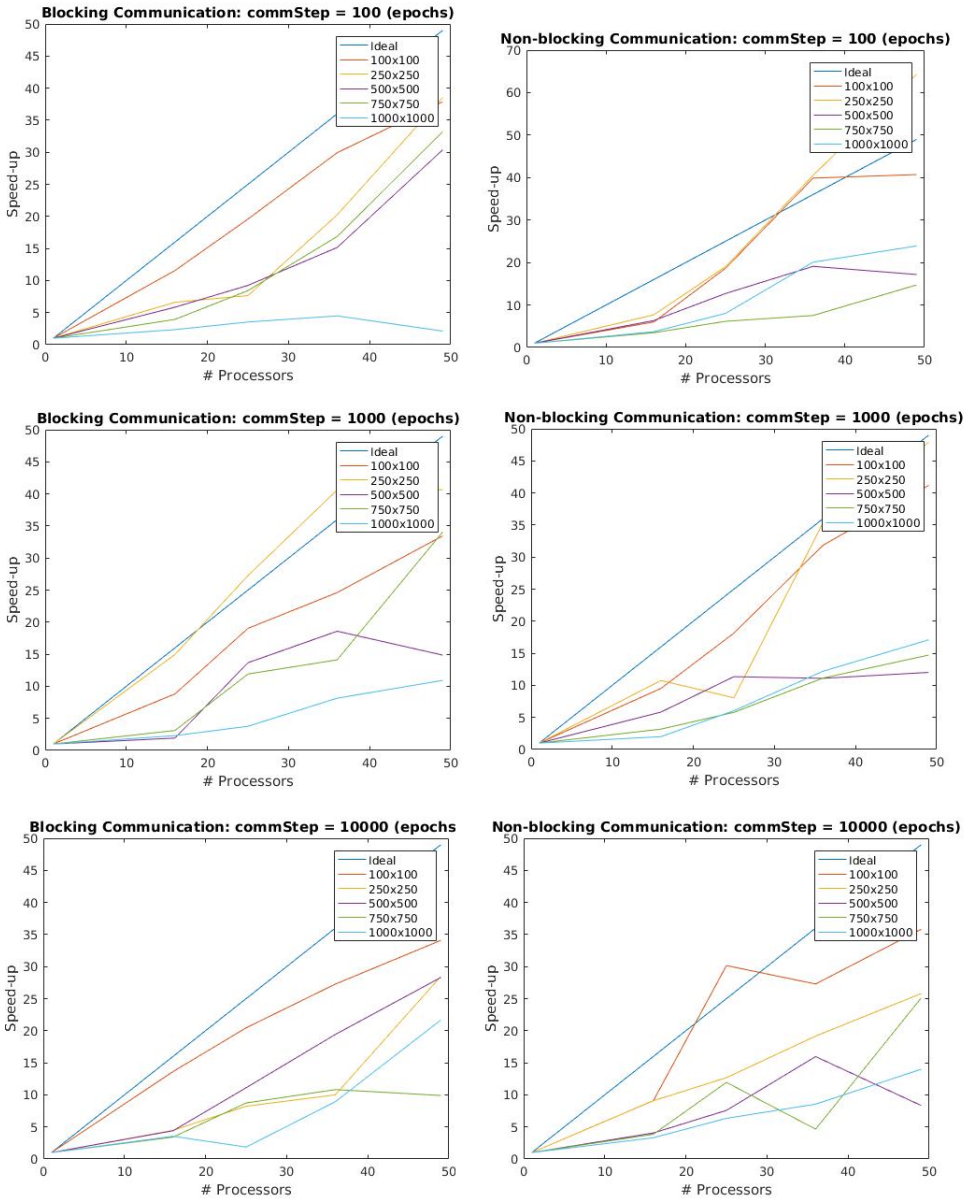

Figure 5: Distributed reinforcement learning: Strong scalability based on epochs.

5 Conclusions and Future Work

As it was said before, the randomness of the exploration algorithm makes hard to compare the performance results. It would be good to run the algorithm many times and take the average but since the algorithm takes long time to converge, makes hard to fulfill all the tests in the time given for this project.

From Figs. 4 and 5, it can be pointed out that there is no big difference in the performance obtained using blocking vs. non-blocking communication as it was expected, since the implementation of the non-blocking communication does not split the inner points from the halo points. However, even implementing a better solution would not give a significant improvement due to the very fast interconnection of Rackham nodes.

Regarding the effect of the $comm\_steps$ parameter, Fig. 4 shows how the performance is affected when a low value is given for this parameter. Also it is important to notice how the performance is degraded when the grid-world size increases. Both shows how important is to keep a good ratio workload vs. overhead. A way to improve it would be implementing several rewards spread through the grid-world that would help to balance the workload.

Another important improvement, would be to evaluate the inclusion of a change measurement $\delta Q(s, a)$ in order to trigger communication only when the Q-values in the halo points have changed at least by a particular value (input parameter) [Barron 2009]. This strategy would help to balance the ratio workload vs. communication overhead in a more proper way thus improving performance.

References

Engelbrecht, Andries P. (2007). *Computational Intelligence: An Introduction. Chapter 6.* 2nd. Wiley Publishing.

Barron, Jonathan, Golland, Dave, Hay, Nicholas (2009). *Parallelizing Reinforcement Learning*. https://jonbarron.info/Barron-ParallelizingRL.pdf.