A Scalable Parallel Q-Learning Algorithm for Resource Constrained Decentralized Computing Environments

Miguel Camelo Jeroen Famaey Steven Latré

University of Antwerp - iMinds
Department of Mathematics and Computer Science
Middelheimlaan 1, 2020 Antwerp, Belgium
Email: {miguel.camelo, jeroen.famaey, steven.latre}@uantwerpen.be

ABSTRACT

The Internet of Things (IoT) is more and more becoming a platform for mission critical applications with stringent requirements in terms of response time and mobility. Therefore, a centralized High Performance Computing (HPC) environment is often not suitable or simply non-existing. Instead, there is a need for a scalable HPC model that supports the deployment of applications on the decentralized but resource constrained devices of the IoT. Recently, Reinforcement Learning (RL) algorithms have been used for decision making within applications by directly interacting with the environment. However, most RL algorithms are designed for centralized environments and are time and resource consuming. Therefore, they are not applicable to such constrained decentralized computing environments. In this paper, we propose a scalable Parallel Q-Learning (PQL) algorithm for resource constrained environments. By combining a table partition strategy together with a co-allocation of both processing and storage, we can significantly reduce the individual resource cost and, at the same time, guarantee convergence and minimize the communication cost. Experimental results show that our algorithm reduces the required training in proportion of the number of Q-Learning agents and, in terms of execution time, it is up to 24 times faster than several well-known PQL algorithms.

Keywords

High Performance Computing, Machine Learning, Reinforcement Learning, Q Learning, Parallel Computing

1. INTRODUCTION

Centralized data centers offer an easy and on demand access for High Performance Computing (HPC) applications such as batch processing, big data, web applications, etc. However, these infrastructures are not suitable to support the deployment of the new Internet of Things (IoT)-based applications that have very strict requirements in terms of response time and mobility or simply lack a link with a centralized infrastructure. Examples of such applications are intelligent control systems for autonomous vehicles, electric grids, robot path planning, network management, and traffic lights [1, 2]. Instead, we are seeing an increasing trend towards a highly decentralized infrastructure in which the computing nodes are either deployed at the network edge (e.g. fog computing) or on the embedded constrained resources devices themselves (e.g. mist computing).

In general, intelligent control systems work by solving the related optimization problem that lie at the heart of most Machine Learning (ML) approaches. One of these approaches is Reinforcement Learning (RL) [3]. RL algorithms learn how to optimize a long-term objective by directly interacting with the environment. Q-Learning (QL) is one of the most efficient and simpler RL algorithms that does not need any previous knowledge about the problem or the environmental dynamic. An important aspect of QL is that it converges to an optimal policy if a tabular representation of the action-value function is used for learning [4]. However, most of the RL algorithms are sequential, designed for centralized environments and time and resource consuming. Therefore, they are not applicable to such constrained decentralized computing environments. Additionally, QL itself entails two main problems that limits its applicability in those environments. First, it requires a long training time in order to learn the optimal policy and second, since convergence is only guaranteed if a tabular representation is used, the table size can quickly grow resulting in scalability issues.

In order to reduce the required training of QL to converge, two approaches have been proposed: Collaborative Multi-Agent Q-Learning (C-MAQL) [5] and Parallel Q-Learning (PQL) [6]. However, the impact of distributing the computation (communication cost) and synchronization among agents has not been considered in most of the studies. On the other hand, the limitations of applying QL on large-scale problems has been mainly addressed by using function approximators [7], such as Neural Networks (NN) [8]. They provide a compact parametrized representation of the action-value function that allows solving problems with very large state spaces and finding average action-selection policies faster than table-based approaches. However, function approximators do not have theoretical convergence guarantees [9] and it is slower to find more accurate (or optimal) action-selection policies than table-based approaches [10].

In this paper we propose a multi-agent learning approach that supports large-scale problems without losing the convergence guarantee. The contributions of the paper are two-fold. First, we provide a multi-agent framework for Q-Learning. This framework can be used both for C-MAQL and PQL algorithms. Second, we present a table-based PQL algorithm for decentralized environments that minimizes the communication cost by combining a table partition strategy with an efficient co-allocation of processing and storage. Together with our multi-agent architecture, this algorithm has guarantee of converge and is highly scalable in terms of both memory and processing.

The remainder of this paper is structured as follows. Related work on PQL is discussed in Section II, while section III introduces the key concepts behind PQL. The Multi-Agent framework for QL is presented in Section IV, and the scalable PQL algorithm is described in Section IV. Evaluation results are provided in Section V. Finally, conclusions and future work are presented in Section VI.

2. RELATED WORK

While most of the research has been done on single agent QL [11] and C-MAQL [5], PQL has been less studied. Kretchmar presents the complexities of sharing information among different agents and proposes an algorithm that accelerates the learning process by using multiple agents in parallel [6]. This implementation requires that each agent keeps two tables, one for storing its own experiences, and one for combining knowledge among agents. As a result, the number of training episodes is reduced by adding more agents. Although this proposal is scalable in terms of processing (i.e. adding more agents is translated into less training episodes) its memory scalability is poor since it requires maintaining two different tables per agent.

Further on, Kretchmar presents two new sharing algorithms that extend its previous work [12]. The first algorithm, the Constant Share RL (CS-RL), assumes that there is a common Q-Table (QT) which all the agents can access and update. The advantage of this algorithm is that it allows instantaneous sharing of experience among agents but its high communication cost, due to a naive distribution of processing, and the bottleneck of having a central shared QT penalizes its deployment in distributed environments. The second algorithm, called the Max-Shared RL (MS-RL), assumes that agents communicate much less frequently. Each agent has two tables: a local QT and a second table to keep track of the experience of the agent. At the time of sharing, the Q-value of the agent with most experience is shared among the whole population. However, this algorithm allows that the experience obtained by other agents is completely discarded and is hence wasted computation.

As an improvement to CS-RL, Printista et. al. propose a parallel implementation of Q-learning via a communication scheme with local cache [13]. We call this implementation PQL with Cache (PQL-C). Based on a master-slave model, this implementation deploys agents that work on different parts of the QT. In this way, each agent will be focused on learning over only one subgroup of states and their associated actions locally.. If the agent requires non-local states, then it will contact the agent that owns the non-local state. In order to improve the performance of the implementation, each process keeps the most recently used values in a local cache. A value in this cache is updated when the number of requested is larger than a given threshold. However, this implementation has several disadvantages. First, it is not memory scalable since it keeps a global QT in the master that is synchronized and updated with the partitions on the slaves periodically. Second, the policy used to update the cache allows that the processes work with either outdated values of non-local states to reduce communication cost or updated information at the cost of a communication overhead according its communication model.

In PQL-C once a partition is assigned to an agent, it does not change until the learning ends. As a way of increasing the performance and focusing on those partitions with more activity, Kushida et. al. presents the Prioritized Field Learning Method (PFLM) [14]. In PFLM, each partition of the QT gets a priority, which changes dynamically, and those partitions which has big changes in Q-values obtains the largest priority, and neighbor partitions obtain the next priorities. The partitions are assigned to agents according to the priority. Another similar master-slave approach is presented by Mannion et. al. [15]. This implementation, called State Action Space Partition (SASP), uses a centralized global QT that is shared by the master and slave agents. The main difference with PQL-C is that the slave agents only update the subset of Q-values assigned to them and never go further their own partition. Although both PFLM and SASP have shown a slightly improvement in the performance compared to PQL-C, they still suffer the same drawbacks.

In summary, the design and deployment of PQL algorithms has focused on reducing the required training to converge by increasing the number of agents used to solve the problem and defining strategies to share and merge their experiences. However, the communication cost that entails decentralized environments and the spatial and time complexity of those algorithms have not been object of research. Most of the work assumes either specialized parallel hardware, which is very expensive, or distributed environments with centralized storage with unlimited memory capacity and very fast interconnection networks. As a result, most of the algorithms are neither implementable nor scalable with a poor performance in distributed environments.

3. THE PQL PROBLEM

In traditional sequential QL, an agent perceives its current state $s \in S$ by sensing the environment, and then selects an action $a \in A$. As a result of the selected action, the agent moves to a new state $s' \in S$ and receives a reward $r \in R$. The goal of the agent is to maximize its total reward over the time. The obtained rewards are used to estimate the action-value function as the expected utility of taking a given action in a given state and following the optimal policy thereafter. The optimal policy is constructed by selecting the action with the highest value in each state. The algorithm execution is based on episodes that start in a given state and finishes when either the agent reaches a goal state or a given condition is met. The move from s to s' is called an algorithm iteration. QL stores the estimate of the expected sum of future rewards, which the agent probably will obtain by starting in state s and selects the action a, in a table of Q-values $Q(s, a)$. This table is called the QT. Algorithm 1 shows the pseudo-code of the QL algorithm.

Algorithm 1 Q-Learning

```
Require: Initialize Q-Table with $|S||A|$ values
 1: repeat(for each episode)
 2:     Initialize $s$
 3:     repeat(for each iteration)
 4:         Use behavior policy to choose $a$
 5:         Take action $a$, observe $r$, $s'$
 6:         $TDerror = \max_{a'} Q(s', a') - Q(s, a)$
 7:         $Q(s, a) \leftarrow Q(s, a) + \alpha \cdot (r + \gamma \cdot TDerror)$
 8:         $s \leftarrow s'$
 9:     until s is goal state
10: until max episodes or convergence is True
```

Where $0 \leq \gamma \leq 1$ is the discount factor, a parameter to trade-off the importance of sooner versus later rewards, $0 < \alpha \leq 1$ is the learning rate, a parameter that determines how the new information will override the old information, and $TDerror$ is the temporal difference error between the actual Q-value and the estimate of future optimal value. QL is one of the most efficient and simpler RL algorithms that does not need any previous knowledge about the problem or the environmental dynamic. However, it requires a long training time in order to learn the optimal policy and its convergence is only guaranteed if a tabular representation is used, resulting in possible memory scalability issues. The ability of a QL agent to acquire more experience and improve the actual estimate of the action-value function has a direct impact on its performance.

Collaborative Multi-Agent Q-Learning (C-MAQL) [5] and PQL [6] are two frameworks that aim to reduce the required experience of each QL agent by increasing the number of agents solving the problem and designing strategies for sharing and merging knowledge among them. In C-MAQL, multiple instances of QL (i.e. intelligent agents) are solving different parts of the problem or are working in an environment that is altered by the actions of other agents. In PQL, the agents learn the same task in isolation and independently interact with the environment. Focusing on PQL, it has been already proved that the number of episodes is reduced proportional to $1/n$ [16], where n is the number of agents updating the unique table. Figure 1 shows a naive implementation of the Constant Share Reinforcement Learning algorithm (CS-RL) [12].

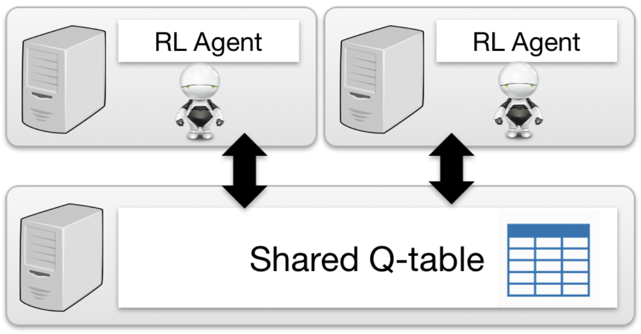

Figure 1: A naive approach for the Constant Share Reinforcement Learning algorithm (CS-RL) [12].

While the performance, in terms of number of episodes, depends on the strategies and frequency of sharing experience between agents, centralized implementations are only partially scalable since the centralized QT becomes a bottleneck and the execution time is not reduced at the same speed. As example, Figure 2 shows the speed-factor in the number of episodes and execution time for a full centralized implementation of the the CS-RL algorithm solving the Cliff problem (see example 6.6 in [3]) in a grid of 100x100 and using up to 64 agents on a computer with 20 cores. Notice that increasing the number of QL agents is not translated into an improvement of the QT response time and deploying more agents will imply a performance decrease (after a certain breaking point). This behaviour is expected because multiple agents are trying to update a unique centralized table, i.e. concurrency is increased, and the processing resources are used for running both the QL algorithm and the table management algorithm (get, put, update, etc.). The centralized table is thus becoming a bottleneck.

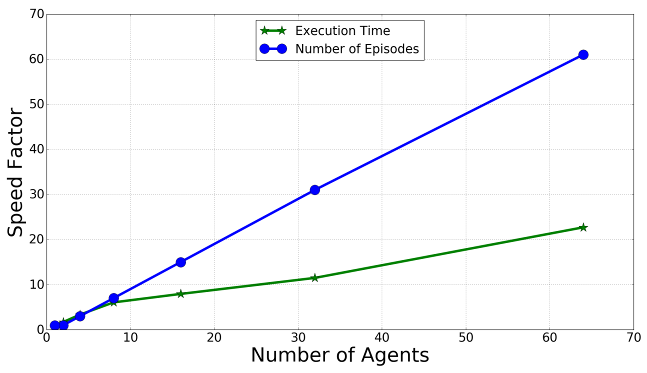

Figure 2: Centralized CS-RL reduces the number of episodes proportional to number of agents. However, the execution time is reduced in a lower rate due to the table bottleneck.

4. A MULTI-AGENT MODEL FOR PQL

The memory and processing scalability problem of PQL running on centralized environments could be solved by distributing both the QT and the agents among multiple computer nodes. However, distributing both agents and table entails several challenges in terms of synchronization among agents, table partitioning, table concurrency control and convergence guarantee based on a distributed table. In addition, distributing the computation on different machines connected through a traditional interconnection network adds a latency on the table operations of at least hundreds of microseconds, which is translated into iterations of 2-3 orders of magnitude slower than the centralized version running in-memory storage (assuming no latency when the system is sensed). Therefore, it is also needed to minimize the network communication as much as possible, while the concurrency control provides the underlying mechanisms to guarantees the correctness of the PQL algorithm.

In order to define a PQL algorithm that addresses those challenges, we propose a general model for supporting Multi-Agent RL algorithms. Note that running a PQL algorithm will imply the execution of several task simultaneously: while it is running parallel instances of the QL algorithm, it also manages both the concurrent access to the QT and the related events of the agent deployment in background such as the creation, allocation of resources (storage and processing), deployment, synchronizing, and data collection. Based on these tasks, we define the following three kinds of agents that interact to create our multi-agent model for RL.

- **Processing agent (PrA)**: This agent is the (traditional) QL agent that works on an external QT.
- **Storage agent (StA)**: This agent stores (part of) the QT. Each agent will be in charge of serving requests from processing agents. These requests could be getting a Q-value $Q(s, a)$ or put/update it, where $s \in S$ and $a \in A$. Note that each agent is only responsible of Q values stored locally.
- **Management agent (MngtA)**: This agent is in charge of 1) Build and distribute the QT among the StAs, 2) Deploy PrAs through the network on the available processing resources, and 3) Get statistics about the agents, network and execution performance, etc.

By defining these three types of agents, we are able to distribute the typically centralized QT among our different agents. If this distribution is done intelligently, it can significantly speed up the QL convergence. It is important to highlight that, up to the best of our knowledge, this is the first multi-agent model that defines the RL agents as function of management, processing and storage tasks. This model allows the designing and deployment of different kinds of RL algorithms, including C-MAQL and PQL, based on the combination of these different agent types. As examples, the CS-RL algorithm can be deployed by using one central StA and n_p PrAs and the PQL-C can be deployed by using one central StA and n_s PrAs with a small local cache. In all the cases, a single MngtA is required. Our model allows the assigning of exclusive resources for each task in order to provide scalability in terms of memory and processing. Figure 3 shows a deployment of the CS-RL algorithm using the proposed model.

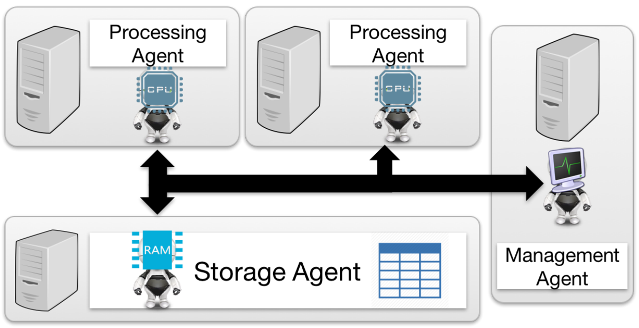

Figure 3: Our multi-agent framework allows that C-MAQL and PQL algorithms can be modeled as function of management, processing and storage tasks. A naive approach of the CS-RL algorithm can be deployed as shown in this figure.

5. A SCALABLE PQL ALGORITHM

One of the advantages of the CS-RL algorithm is that by using a single StA to keep the QT, all the PrAs are updating and using the same policy. As a result, the number of episodes is reduced in a factor of $1/n_p$, where n_p is the number of PrAs solving the problem, and there is guarantee of convergence since there is a unique QT for all the PrA. However, in addition to the poor memory scalability of a centralized table, this approach has a high communication cost since a single QL iteration requires at least 4 round trips through the network (from PrAs to StAs) in order to execute the following operations: get $Q(s, a)$, get $\arg\max_a Q(s, a)$, get $\max_{a'} Q(s', a')$ and update $Q(s, a)$. For this reason, we propose to augment the PQL with a Co-allocation of Storage and Processing (PCSP) algorithm: a scalable (in memory and processing) algorithm that provides convergence guarantees together with a reduced communication scheme based on our multi-agent framework presented in Section 4. The goal of this algorithm is to intelligently distribute processing and storage among the different nodes. By aiming to co-allocate both processing and storage of the same part of the QT we can ensure that most updates of the QT only need to be performed locally, hence reducing communication overhead. The co-allocation algorithm thus decides on the best way split the QT and assigns the processing accordingly. Algorithm 2 shows the pseudo-code of the proposed algorithm.

Algorithm 2 PQL with Co-allocation of Storage and Processing (PCSP)

```
Require: Number of states $|S|$ and actions $|A|$,
Require: Set of physical nodes $M$
 1: if $|M| > 0$ then
 2:     Deploy a MngtA
 3:     $q \leftarrow$ total available memory across physical nodes.
 4:     if $O(|S||A|) \leq q$ then
 5:         Compute table partitions $P$
 6:         Deploy $n_s = |P|$ StA
 7:         Initialize the QT
 8:         Create and deploy $n_p$ PrA
 9:         while $p$ has not converged, $\forall p \in PrA$ do
10:             Execute QL on $p$, $\forall p \in PrA$
11:         end while
12:         Obtain statics from $p$, $\forall p \in PrA$
13:         Retrieve the optimal policy $\pi$ from QT
14:     end if
15: end if
16: return $\pi$
```

Our algorithm is composed of two main procedures: creation and deployment of the multiple agents (lines 4-8), and the QL algorithm execution (lines 9-11). A detailed description of these steps is given below.

5.1 Storage agent deployment

The algorithm inputs are the number of states $|S|$ and actions $|A|$ together with the set of available physical nodes M. If there is any available machine, then we can deploy our MngtA. This agent will gather all the information related to the memory and number of cores available on each $m \in M$. Based on that information, the MngtA can determine if the QT fits in the memory available across all the nodes. We use a unique distributed QT since this approach achieves a reduction in the number of episodes proportional to $1/n_p$ and guarantees memory scalability. In addition, it also provides the flexibility of having StA of different memory capacities working together. When the physical nodes are homogeneous, the table partition is load balanced among StA and every StA will store exactly $\Omega(|S| * |A|)/n_s)$. In heterogeneous environments, if s_i is the available memory to store (part of) the QT on node i, then $\sum {s_i}_{i=1}^{i=n_s} = O(|S| * |A|/n_s)$, where $0 \leq s_i \leq |S| * |A|$.

The table partition is based on the idea of allocating consecutive states into the same StA. We assume that the partition algorithm has access to some domain knowledge of the problem and the (possible) structure of the solution. Take as an example the Cliff walking problem [3]. If the partition algorithm knows that a (possibly optimal) solution would be composed by consecutive states connected by the action "move to right", then we can split the grid world horizontally and create the set of partitions P as shown in Figure 4. If c_i is the available processing resources (e.g. processors or cores) on node i, then we can create $|P| = \lceil c/2 \rceil$ partitions, where each partition contains $\lceil |S||A|/|P| \rceil$ Q-values. The number of StA to deploy will be equal to the number of partitions. In this way, we assign exclusive processing resources to run the StA and management each partition efficiently. In terms of the table data structure, the distributed QT stores states instead of single Q-values $Q(s,a)$. Each state keeps internally the associated Q-values of each available action of the state. This data structure allows that operations such as $\max_a Q(s,a)$ are executed locally.

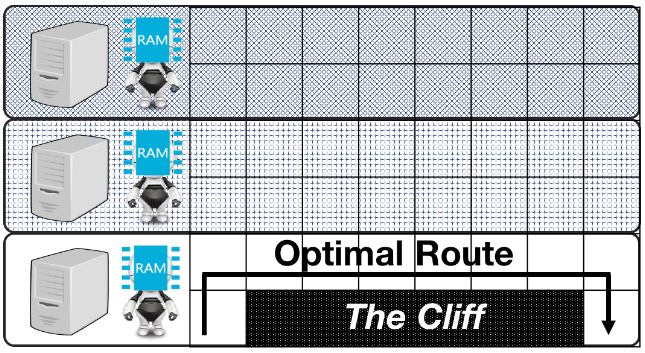

Figure 4: In the Cliff walking problem, a (possible optimal) route is composed by "move to right" actions. A horizontal partitioning of the table allocates states that will be consecutively visited with high probability.

5.2 Processing agent deployment

As the PrAs and StAs are independent entities, the way they are linked with each other will define the communication cost in the system. Deciding if the resources are used exclusively for one specific task will depend highly on two factors: the available resources of the node, i.e. memory and number of processing unites (cores), and the interconnection network. Previously, we have discussed that the MngtA uses the partition algorithm to determine the allocation of each partition on each node $m_i \in M$ with available storage resources and deploys $\lceil c_i/2 \rceil$ StA per partition. Once the StA have been deployed, the MngtA must determine where the PrAs must be located. Note that in traditional QL, the algorithm execution has minimum impact on the QT I/O operation since each iteration is completely sequential. Therefore, deploying $\lceil c_i/2 \rceil, \forall m_i \in M$ PrA on nodes where StA have been already deployed will minimize the concurrency and maximize the use of resources.

Note that when a PrA is exploring states that are managed by the local StA, the communication cost is minimum since everything is done in memory. However, once the agent is exploring states that are located on an external node, both the communication cost and the total execution time are increased. In order to minimize this cost, the PrA will deploy a temporal PrA on the node where the explored state is stored as shown in Figure 5. Once it is deployed, the temporal PrA will continue the execution until it reaches either the goal state, or it gives up (reaches a maximum number of iterations), or it requires a state that is not local. After one of those conditions is met, the temporal agent is removed and the original PrA gets the last state visited. Note that this strategy will have a positive impact on the performance only if the partition strategy has allocated near states on the same partition and the (possible) optimal policy tends to visit states on the same node. Algorithm 3 shows the modified QL running on the PrA.

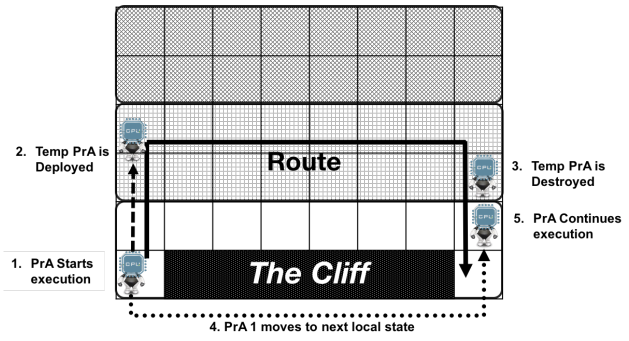

Figure 5: When a PrA visits a state that is located in another partition, a temporal PrA is deployed over there. It reduces the number of remote calls to that QT partition.

Algorithm 3 Modified Q-Learning

```
Require: Access to Distributed Q-Table with $|S||A|$ values
 1: repeat(for each episode)
 2:     Initialize $s$
 3:     repeat(for each iteration)
 4:         Use behavior policy to choose $a$
 5:         Take action $a$, observe $r$, $s'$
 6:         if $s'$ is not stored in any local StA then
 7:             Deploy temp PrA on node where $s'$ is stored.
 8:             repeat
 9:                 Wait
10:             until temp PrA has finished
11:             $s \leftarrow s'$
12:         else
13:             $TDerror = \max_{a'} Q(s',a') - Q(s,a)$
14:             $Q(s,a) \leftarrow Q(s,a) + \alpha \cdot (r + \gamma \cdot TDerror)$
15:             $s \leftarrow s'$
16:         end if
17:     until s is goal state
18: until max episodes = true or convergence = true
```

5.3 Concurrency and Convergence

Concurrent access to update values in the QT implies several challenges in order to guarantee data consistency in distributed environments. If two (or more) agents are visiting a state that is locally stored and its next state is also locally stored, then there is no need of concurrency control since both states are stored in memory as volatile values, i.e. they will never be cached thread-locally. It allows that the Q-values can be accessed on real-time, and therefore the update rule of QL (Algorithm 3 line 10) will always be applied using the newest value on the table.

Note that using atomic updates in memory allows a better performance than using transactional updates, like in distributed data-bases, because there is no need of locking states in the QT. However, it also allows that the selection of an action may be based on old Q-values. A simple analysis of the e-greedy policy on this specific case will show that the use of this out-of-date value will not affect the algorithm convergence. Note that if a given action a was selected by considering old Q-values but such selection would be different if we use newest one, then the selection is still e-greedy. The main reason is that if a was a greedy selection based on oldest values, e.g. $\max Q(s, a), \forall a \in A$, then a may be also selected using the e-greedy policy based on the newest ones. In other words, an iteration for exploitation becomes an iteration for exploration.

The convergence of our PQL can be ensured if it is serializable, i.e. the result of running multiple agents in parallel is the same as the one obtained by running a single agent QL. Note that multiple agents updating simultaneously different non-related states can be seen as independent operations, and therefore these operations are serialized. On the other hand, reading a non-updated Q-value to select an action is also a serializable operation since an exploitation iteration can be seen as a exploration one as we explained before.

6. PERFORMANCE EVALUATION

This section presents the experimental results on the performance of our PCSP algorithm. We compare PCSP against the following state of the art algorithms:

- The single-agent Q-learning algorithm (SQL) [3].
- A naive implementation of the Constant Share RL (CS-RL) algorithm [12].
- An extended version of the CS-RL with local cache on the processing agents (CS-RL-EXT).
- The PQL algorithm via a communication scheme with local cache (PQL-C)[13].

The performance of the algorithm was measured using two dependent variables:

- **Number of episodes**: This variable counts the number of times an agent reached a goal state, or a given condition is met, before the optimal policy is learned. This metric only depends on the experience obtained during the training. This metric is hardware-independent and lower values imply better performance.
- **Total execution time**: This variable measures the clock time elapsed until the algorithm converges. It includes the time of requesting and updating values on the table (maybe through a network), the time to sense and modify the environment and the time for checking the quality of the actual policy. This variable is hardware depended and lower values imply better performance.

The first metric is used to compare different PQL algorithms against a single agent QL. On the other hand, the second metric allows a fair comparison between PQL algorithms running on similar hardware platforms. Based on these metrics, the results of this paper are presented in terms of the speed factor of the algorithm:

Speed Factor: Let P be the set of Processing agents (PrAs). Let $x_{A,n}$ and $x_{B,n'}$ be two measurements of the same performance variable (i.e. episodes or execution time) of the algorithms A and B with $n, n' \in P$, respectively. The speed factor of algorithm A with respect to B is defined as the ratio $x_{A,n}/x_{B,n'}, \forall n \in P$ and fixed n'. Higher values imply better performance.

All the evaluations were executed 10 times on a computing infrastructure with 8 Intel Quad-Core computers (Intel Q9650) interconnected by a gigabit ethernet network. The multi-agent architecture and all the algorithms were implemented on top of the Apache Ignite framework [17].

6.1 Algorithm Configuration

In all the algorithms, every episode starts locating the agent in the start state and finishes when the agent reaches either the goal state or a given number of iterations. PCSP and PQL-C algorithms use a QT distributed on 8 physical nodes, and PQL-C synchronizes each partition with a central QT each time a local (cached) value is modified. The cache size for CS-RL-EXT and PQL-C is 1% of the QT size. All the agents are configured with α= 0.9, δ= 0.9, and ϵ= 0.1 in the $\epsilon-$greedy behavior policy. An agent stops when it has reached consecutively the goal state 10 times and the accumulative changes on the QT values on those episode is 0. When all the agents have stopped, the number of episodes and the execution time are computed as the average among all agents on the 10 trials.

6.2 Problem Description

In order to compare the algorithms, we selected the cliff walking problem [3], which is commonly used in RL. In this problem, an agent travels a virtual grid-world. The world has a start state, a goal state, and a cliff on which the agent will die if it falls. The objective of the agent is to reach the goal state without falling off the cliff. The size of the grid is MxN and the agent starts at the leftmost cell in the bottom, that is, (M-1, 1). The goal is the rightmost cell in the bottom, that is, (M-1, N-1). All the cells between (M-1, 1) and (M-1, N-1) is the cliff. The agent can take 4 actions (move UP, DOWN, LEFT, RIGHT) in each state. If the agent falls into the cliff, then the agent will die. In our experiments, the size of the grid is 100x100 and the reward is -100 if an agent falls into the cliff, -1 for each movement of the agent, and +100 if the agent reaches the goal. Figure 6 shows a 8x6 grid and two solutions that solve the problem.

6.3 Number of Episodes

One of the important features about PQL is that by adding more Processing agents (PrAs), the reduction in terms of number of episodes should be proportional to the number of agents that are solving the problem. Figure 7 shows the speed factor, in terms of the number of episodes, obtained by our algorithm PCSP-8 (8 physical nodes, 2 Storage agents (StAs) per node), CS-RL, CS-RL-EXT and PQL-C, with respect to SAQ. In all the algorithms, the reduction in the number of episodes is closed to $1/n_p$, where n_p is the number of PrAs, which is similar to the performance observed in a full centralized PQL (see Figure 2).

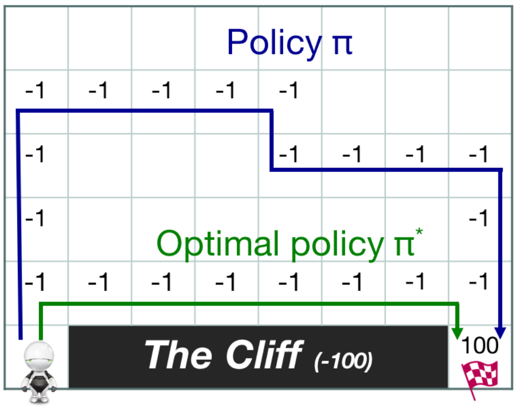

Figure 6: Two possible solutions for the cliff problem. The optimal solution is the shortest route that avoids the cliff.

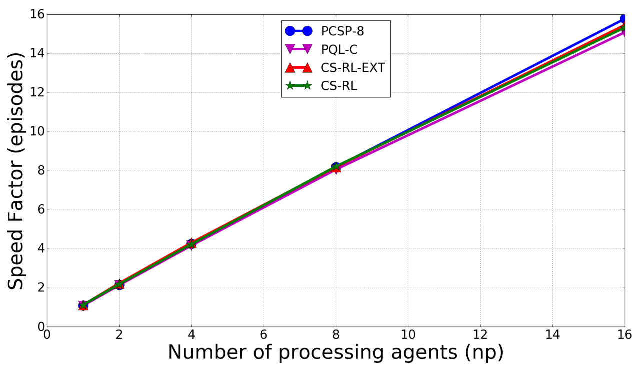

Figure 7: Since all the algorithms are updating an unique table, they can reduce the number of episodes proportionally to the number of PrAs solving the problem.

6.4 Execution Time

While a speed factor in terms of number of episodes is a first indication of the speed up of parallelization, the most important metric is the speed up in terms of execution time. A comparison between our algorithm and both centralized and distributed PQL algorithms is presented below.

6.4.1 *Comparison with a centralized PQL algorithm*

One of the main drawbacks in centralized PQL is often the fast decreasing of the speed factor in terms of execution time (see Figure 2), which is due to the bottleneck on the centralized QT. Figure 8 shows that the partition strategy used by our algorithm to distribute the QT reduces the concurrency and minimizes its impact on the execution time. As shown, when increasing the number of processing agents, both a centralized PQL algorithm and our PCSP algorithm is able to achieve higher speed factors. However, PSCP benefits from a much higher speed factor. While moving from 8 to 16 PrA gives an improvement of 176% in PCSP-8, it is only 131% in a centralized version of the CS-RL algorithm. This is a speed factor of 40% more. Figure 8 also shows that we are close to the theoretical optimum. Also PCSP suffers from a drop in speed factor due to an increasing in the communication cost of the algorithm (i.e., 15% below the optimum for 16 processing agents). This drop is however considerably less then the centralized PQL algorithm due to the intelligent table partitioning strategy and co-allocation process, which thus minimizes the communication cost considerably.

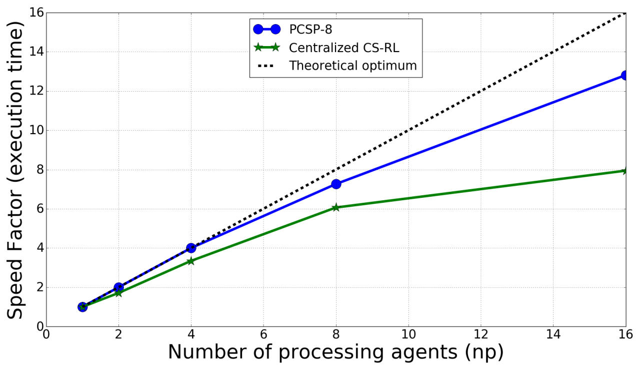

Figure 8: In term of execution time, the performance of our algorithm is 15% below the theoretical optimum but it is still much better than the centralized PQL approach with an speed factor 40% higher with up to 16 PrAs.

6.4.2 *Comparison with distributed PQL algorithms*

In order to compare the performance of our PCSP algorithm against other algorithms that attempt at solving the scalability problem of centralized QL (i.e., CS-RL, CS-RL-EXT and PQL-C), the speed factor of each algorithm was computed with respect to the execution time of the distributed CS-RL algorithm with one StA and one PrA. We can see in Figure 9 that the speed factor of our algorithm outperforms by far the performance of CS-RL, CS-RL-EXT, and PQL-C. By assuming the same number of PrAs, our algorithm is (on average) 24.8 times faster than CS-RL, 17.4 than CS-RL-EXT, and 8.09 than PQL-C. It shows that the co-allocation of PrA close to the data that they are updating reduces the communication among them and minimizes the total execution time considerably.

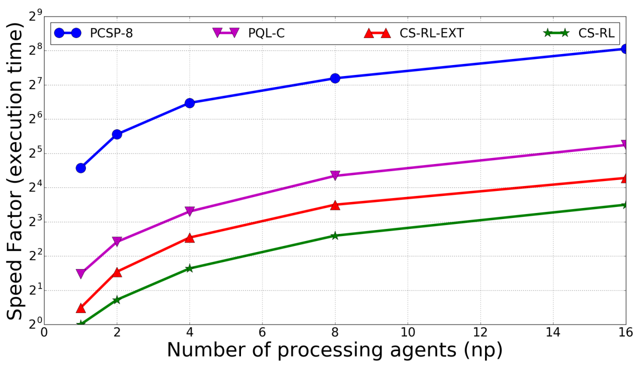

Figure 9: In terms of execution time, our algorithm overcomes the performance of all evaluated algorithms. It is (on average) 24.8x faster than CS-RL, 17.4x than CS-RL-EXT, and 8.09x than PQL-C.

6.5 Impact of number of storage agents

The previous section illustrated the speed factor for a PQL environment with 16 StAs. In this section, we evaluate the impact of the number of StAs on the speed factor. Figure 10 shows the speed factor of our algorithm in terms of execution time with 2 (PCSP-2), 4 (PCSP-4), and 8 (PCSP-8) physical nodes that store the QT. As can be observed, PCSP-2 outperforms the other algorithms in terms of speed factor for 2, 4 and 8 PrAs. However, when using 16 PrAs, PCSP-8 is significantly better. This result can be explained as follows. If we increase the number of PrAs (np) and the number of StA (ns) simultaneously, then the algorithm is able to compensate the communication cost of inter QT synchronization by an increase in the overall number of physical nodes available. On the other hand, having $n_p << n_s$ implies an insufficient use of existing resources and increases the communication cost as a lot of PrAs will need to be deployed on remote StAs. Finally, a $n_s << n_p$ decreases the execution time by execution more operations in local memory, but requires more memory per node to store part of the QT.

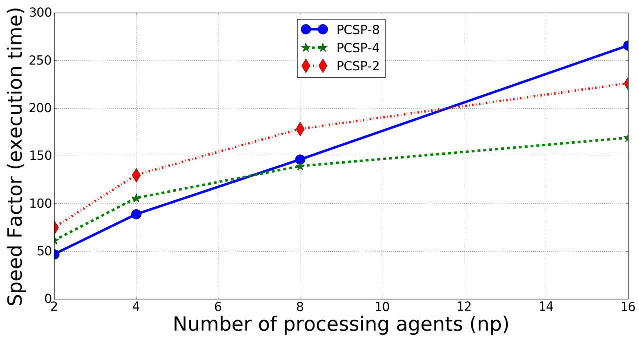

Figure 10: By increasing both the number of PrAs and the number of StA simultaneously, the algorithm is able to compensate the communication cost of inter QT synchronization

7. CONCLUSION AND FUTURE WORK

In this paper, we proposed a multi-agent framework to support large-scale problems using Q-Learning (QL) and a scalable Parallel Q-Learning (PQL) algorithm for resource constrained environments. The multi-agent framework abstracts different functions to support the deployment of multi-agent Reinforcement Learning (RL) algorithms. These functions are exposed as three main agents: manager, processing and storage. Built on top of this framework, we proposed the PCSP algorithm, which combines an intelligent table partition strategy with a co-allocation of both processing and storage nodes. The experimental results showed that our algorithm reduces the number of episodes proportional to the number of processing agents that are solving the problem. In terms of execution time, the speed factor of our algorithm is up to 24 times higher than several well-known PQL algorithms. Additionally, the resulting communication cost of adding more physical nodes to store the distributed Q-Table (QT) is compensated by adding more processing agents. In general, our algorithm significantly reduces the individual resource cost, since there is no need of specialized parallel hardware, guarantees convergence, and minimizes the overall system communication cost.

8. ACKNOWLEDGMENTS

Part of this research has been funded through the SMILE-IT project, funded by IWT.

9. REFERENCES

[1] Daniele Miorandi, Sabrina Sicari, Francesco De Pellegrini, Imrich Chlamtac, Internet of things: Vision, applications and research challenges, Ad Hoc Networks, Vol. 10, issue 7, pp. 1497-1516, Elsevier, 2012.

[2] Ovidiu Vermesan and Peter Friess, ”Internet of things-from research and innovation to market deployment”, River Publishers Aalborg, 2014.

[3] S. Sutton and A. G. Barto. Introduction to reinforcement learn- ing. MIT Press, 1998.

[4] Christopher Watkins and Peter Dayan, Technical Note: Q-Learning, Journal Machine Learning, May 1992, 8(3) ISSN 0885-6125, pp. 279-292.

[5] L. BuÅ§oniu, R. BabuÅąka and B. De Schutter, Multi-agent reinforcement learning: An overview. In Innovations in Multi-Agent Systems and Applications, pp. 183-221, 2010, Springer Berlin Heidelberg.

[6] R. Matthew Kretchmar, Parallel reinforcement learning, 6th World Conference on Systemics, Cybernetics, and Informatics, 2002.

[7] J. Tsitsiklis and B. Van Roy. An analysis of temporal-difference learning with function approximation. IEEE Transactions on Automatic Control, 42(5), pp. 674-690, 1997.

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

[9] Sebastian Thrun and Anton Schwartz, Issues in using function approximation for reinforcement learning. In Proceedings of the 1993 Connectionist Models Summer School Hillsdale, NJ. Lawrence Erlbaum. Citeseer, 1993.

[10] Peter Finnman and Max Winberg, Deep reinforcement learning compared with Q-table learning applied to backgammon, Bachelor’s Thesis in Computer Science, KTH Royal Institute of Technology, Sweden, 2016.

[11] L. P. Kaelbling, M. L. Littman and A. W. Moore, Reinforcement learning: A survey. Journal of artificial intelligence research, 4, 237-285, 1996.

[12] R. Matthew Kretchmar, Reinforcement learning algorithms for homogenous multi-agent systems, Workshop on Agent and Swarm Programming, 2003.

[13] Alicia Marcela Printista, Marcelo Luis Errecalde, and Cecilia Ines Montoya. A parallel implementation of Q-Learning based on communication with cache, Journal of Computer Science & Technology (2002).

[14] M. Kushida, K. Takahashi, H. Ueda and T. Miyahara, A Comparative Study of Parallel Reinforcement Learning Methods with a PC Cluster System, 2006 IEEE/WIC/ACM International Conference on Intelligent Agent Technology, 2006, p.p. 416-419.

[15] Patrick Mannion, Jim Duggan, and Enda Howley, Parallel Reinforcement Learning with State Action Space Partitioning, Proceedings of the 12th European Workshop on Reinforcement Learning. 2015.

[16] Steven D. Whitehead, A Complexity Analysis of Cooperative Mechanism in Reinforcement Learning, Proceedings of the AAAI’91. pp. 607-613, 1991.

[17] Apache Ignite: In-Memory Data Fabric. 2016. https://ignite.apache.org. Accessed: 2016-08-10.