Q-Learning Parallelization

Jacopo Di Simone

Politecnico di Milano, Milano, Italia

5 May, 2017

Introduction

- The goal of the project was to parallelize the Reinforcement Learning algorithm Q-Learning, used for solving tasks modeled after Markov Decision Processes (learn the optimal policy). The technique is described in the paper "A parallel implementation of Q-Learning based on communication with cache" of Printista et al.
- The idea of the algorithm is to let an agent act in an environment, collect its reward and update the q-function.
- The main characteristics of Q-Learning method is the ability to learn the optimal q-function (and the optimal policy) following a non optimal policy.

Sequential Q-Learning

Initialize $Q(s, a)$ $\forall s \in S$ and $\forall a \in A(s)$
for all episodes **do**
  Initialize s
  repeat
    Choose a from s using a ϵ-greedy policy derived from Q
    Take action a, observe resultant state s' and the reward r
    $Q(s, a) \leftarrow Q(s, a) + \alpha + [r + \gamma \max Q(s', a') - Q(s, a)]$
  until s is terminal
end for

Parallel Q-Learning

- In the paper the author propose a parallelization of the algorithm of the type Domain Data Decomposition.
  - The data are decomposed in different chunk. In our case the data is the q-function.
  - Each task will work on a chunk.

Proposed Implementation: Process

In the algorithm there are 2 type of tasks: *Slave* and *Master*.

- The tasks *Slave* will handle the computational part, applying the Q-Learning method to the assigned domain.
- The task *Master* will handle the communication in the program and will conserve a global version of the q-function.

Proposed Implementation: Communication

The communication are always between Slave and Master and never between Slave and Slave.

- *reqmsg* is sent from task *Slave* to ask for value of the q-function out of its domain. To avoid high communication overhead it was implemented a cache. It keeps the last value provided by the master until a certain requirement is reach. After that a new *reqmsg* is sent.
- *infmsg* is periodically sent from the *Slave* to the *Master* with their q-function partition. Also in this case to avoid the overhead the message is sent every *tot* epoch.

Implementation: Idea

- The algorithm is implemented in C++ using the pthread library.
- Unlike the implementation proposed in the paper, the *Master* was eliminated. In its place the concept of shared memory was adopted: a memory shared between every tasks.

Implementation: Q-Learning

The Q-Learning function is implemented in the Agent class through the function *learn*

```
/*
static function used to run the q learning function by an agent
@param agent: agent which learn the function. Passed as a pointer
of void just to satisfy some constraints of pthread
*/
static void * learn(void * agent);
```

Implementation: Problem

To test the algorithm I selected the same problem used in the paper; the grid. Each transition in the grid give a reward of 0, except for the 2 goal state which give a reward of 100. The game terminate when the agent reach a goal. The disocunt factor is γ is 0.95.

```
|# #  ##    # G|
|#  #   ## #   |
|             #|
|# # #     ####|
|## #   #  #  #|
|#  #   #      |
|   ####    #  |
|# ### #      #|
|       #     #|
|    ## ##    #|
|    ###  #   #|
|            ##|
| #    #   # # |
|G#    #  ##   |
```

Implementation: Environment

To handle the interaction of the Agent in the grid I created the class Environment. Through many functions allow the agent to act in the grid. One of this functions is *step*.

```
/*
make a step in the grid (modify also the curr_state variable)
@param action to be performed
@return the observation: next_state (integer (-1 if the
                          state is goal)), reward, done
*/
observation step(Actions a);
```

Implementation: Optimal Value Function

To test the algorithm and obtain the optimal Value function I used the Value Iteration method. This is implemented in the Environment class through the function *valueiteration*.

```
/*
@param env: environment to learn
discount factor
theta: precision of the algorithm (default 0)
@return value function as a (pointer to) vector of double
*/
std::shared_ptr<Eigen::VectorXd> valueIteration(
                                  double discount_factor = 0.95,
                                  double theta = 0);
```

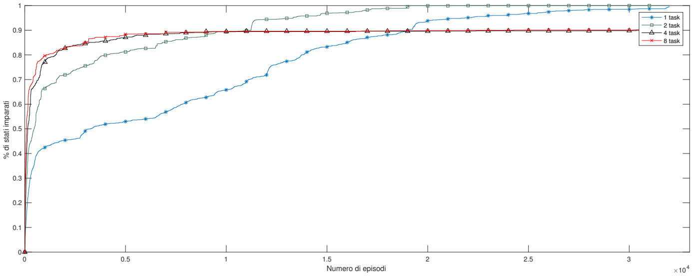
Results analysis: Execution Times

The results are obtained doing a mean over 30 execution for each number of tasks (1, 2, 4). Then I collect these measurements with the ones of 10 other different grids, doing a mean of the obtained results.

np	1	2	4
Tempo	3.8167	2.8311	1.9521
Speedup	———	1.346	1.783

Results analysis: Learning rate

In the graph are represented the learning rate curves for 4 different number of processes (1,2,4,8). These are computed doing a percentage of the learned states in each episode and computing the mean over all 11 grids.



Conclusions

The obtained results regarding the execution time are what I expect, according also to the results showed in the paper.

Instead, the learning rate did not give the expected results. This is due to the ϵ value used in the experiment (0.1). Being this value so low, it give to much credit to the initial explored state, assigning a probability too low to action without the maximum expected value. Using an ϵ that change its value episode per episode remove this problem.