Parallelizing Tabular Q-learning using OpenMP. Using a grid world as our Environment

Author: DENNIS SHPITS

1. Introduction

Reinforcement learning (RL) is an area of machine learning that deals with intelligent agents taking actions in an enviornment to maximize a cumulative reward. In this paper we focus on a specific Reinforcement learning algorithim called Tabular Q-learning. We will show how Q-learning can be implemented in C++ and then parallelized using OpenMP. We use a grid world (a matrix) to replicate an enviornment. Grid cells are labeled 1 to n. An Agent can perform a series of actions within the grid world with the goal of reaching a specified winning grid cell (usually the n^{th} grid cell).

2. Markov Decision Process

Our RL problem consists of states, actions, and an enviornment. A State s_t ϵ S where S is the set of all possible states within our enviornment. In our grid world, a state is the current cell position of the agent in the matrix. An Action a_t ϵ A where A is the set of all possible Actions. A transition function $s_{t+1} = f(a_t, s_t)$ returns the following state after an action is taken from the current state. A reward function $r_{t+1} = r(a_t, s_t)$ returns the reward of taking an action from our current state.

The goal in a Markov decision process is to find a good "policy" for the agent: a function π that specifies the action a_t that the agent will choose when in state s_t. Our policy function is $a_t = \pi(s_t)$.

Experience during learning is based on s_t, a_t pairs together with the outcome s_{t+1}. We can put this in english terms using the following sentence "I was in state s_t and I tried doing a_t and s_{t+1} happened".

3. Algorithim

The algorithim performs episodic learning. An episode ends when the agent reaches the winning state, or it is "stuck" and can't move. We define the total number of episodes that will run as ITERATIONS. Our ITERATIONS value should be high enough that the agent is eventually ”exploiting” a policy. A future improvment to the algorithm can be made by having ITERATIONS be dynamically calculated.

We create a q table which is responsible for holding our Q learning values. After the agent takes an action it receives a reward. Rewards can be negative if the agent lands outside of the enviornment. The agent receives a large award if it lands on the winning position. Eventually the agent stops exploring the enviornment and exploits the q table using a greedy approach.

An example of how the agent starts to exploit the q table using a greedy approach is if we have the following enviornment:

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}$$

If we are in position 7 and our q table looks like the following:

$$\begin{bmatrix} s_t & a_1 & a_2 & a_3 & a_4 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 7 & 0.4 & 1.2 & 4.1 & 0.8 \\ \cdots & \cdots & \cdots & \cdots & \cdots \end{bmatrix}$$

The agent will always choose action a_3 because it is the optimal action-value. If all the action-values are equal, then the agent ”explores” and performs a random action.

4. Classes

I have broken the problem down with the use of C++ classes. I created two classes: A **State** class and an **Action** class. The **State** class holds the current cell position the agent is in as a private variable. Using information hiding the main code does not need to know about matrix positions. It also makes the main code easier to read. We have also added other functions that perform searching in a matrix.

The **Action** class is more interesting. It uses inheritance to create new actions. We can create a new a_t by inheriting from the **Action** base class making our code strongly modular. Once we have created a new a_t we can just add it to our vector of actions: `vector<Action*> av`.

Another benifit of the **Action** class is it defines the reward function. For each derived **Action** class we have defined transition functions. We have functions for each derived **Action** class that tell us if the returned state from a transition is valid. Creating **Action** classes allows the main code to focus on the task at hand and abstracts away the operations needed to be made on the grid world.

5. q function

$$q(s,a) = (1-\alpha) * q(s,a) + \alpha * (r(s,a) + (\gamma max_a, q(s',a')))$$

The q function has the future built into it. $\gamma max_a, q(s',a'))$ is the maximum amount of return based on the q table for an action.

You can find the q function call in the code. It looks like this:

```
qtable->find(mystate.CurrentPosition())->
second.find(takeaction->GetName())->second
= ( (1- alpha) * qtable->
find(mystate.CurrentPosition())->
second.find(takeaction->GetName())->second)
+ (alpha * (reward + vprime));
```

You will notice our q table is of type:

```
map<int, map<string, double>>
```

The rows of the q table are all the cell identifiers from 1 to n. The columns of the table are the actions.

The q function is also called the optimal action-value function. α in the function describes the learning rate where 0 means the q table never changes, and 1 means adopt a new observation and throw away what you knew before. In our code we have set this value to 0.5.

6. Parallelizing Tabular Q-learning

I was able to parallelize various parts of the code. In the **State** class two functions: (`getcurrentrow()`, `getcurrentcol()`) were parallelized. These functions were responsible for returning the the row or column for a given grid label. These functions use nested for loops to search the matrix. After parallelization each thread would search a column.

We utilized `#pragma omp cancel for` when any of the threads found the cell in question. We periodically called `#omp cancellation point for` to check if `#pragma omp cancel for` was called by any of the other threads.

In the main we have to initilize our q table. Since our q table was of type `map<int,map<string, double>>`, we are able to allow each thread to initilize each row in the table in random order. I do not expect there to be a data race for inserting elements because each time we are inserting a new key. The documentation for the `operator[]` states that "Concurrently accessing other elements is safe."

Another area that I parallelized was a section of code that checks if the agent is "stuck". There may arise situations where any next action the agent performs leads to a previous cell in the grid that has already been explored. For example if our set of actions A={left,down,right,up} and our grid enviornment is the following:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

If the agent decided to take the path $1 \rightarrow 2 \rightarrow 5 \rightarrow 4$ it is "stuck" because it can not move to a valid position after 4. In this case the episode ends. We parallelized this by having each action run on a thread and increment a counter variable if it results in a state that is not valid. If the counter is equal to the number of actions we are "stuck".

The line `#pragma omp parallel for reduction(+ : stuckcount)` is parallelizing the loop over all actions and doing a reduction on our counter variable. We can have a large number of actions, so parallelizing this "stuck" check is beneficial. We also add a parallel break statement if any thread finds that we are not "stuck" using `#pragma omp cancel for`.

The final and the most important parallelization was having multiple agents by updating the q table with multiple threads. Running a single agent did not always lead to an optimal solution. With a single agent, it takes the best action based on what it knows so far (agents are greedy). Once the agent knew any solution it would exploit that solution. By adding multiple agents we converge to the real q learning function, thus leading to more optimal solutions. The reason having multiple agents leads to more optimal solution is because multiple agents add random exploration and thus we have a more complete q table.

Since multiple agents share the q table we added a mutex lock using `omp_lock_t q_table_lock`. Each agent calls `omp_set_lock(&q_table_lock)` to lock our data structure and `omp_unset_lock(&q_table_lock)` to unlock.

6. Scalability

We can analyze scalability by running scalability tests. A scalability test that may be of interest is how well the RL program performs in terms of finding the optimal solution for larger environments. The larger the enviornment, the harder it is for the agent to find one of the optimal paths to the winning state. By increasing the enviornment we are growing the amount of work.

We create a series of grid enviornments in increasing size. We then test how the program performs by adding more threads. For our performance metric we are looking for the hit accuracy of how often the agent correctly finds an optimal path which will be a ratio $\frac{hits}{hits+misses}$. We are trying to answer the question of whether the program can handle growing amounts of work. The Grid sizes are 10, 20, 30, 40. We have Thread counts of 1, 10, 20, 30, 40, 50. We also created a bash script to automate the repetitive trials. We perform 100 trials and record the optimal path hit ratio afterwards. We keep the ITERATIONS value high enough to ensure the agent is eventually "exploiting" a policy.

Threads	Grid Size	Optimal Path Hit Ratio	Agent Start Cell	Winning Position
1	10	0.43	1	10
10	10	0.83	1	10
20	10	0.85	1	10
30	10	0.84	1	10
40	10	0.80	1	10
50	10	0.90	1	10
1	20	0.21	1	20
10	20	0.93	1	20
20	20	0.95	1	20
30	20	0.90	1	20
40	20	0.93	1	20
50	20	0.92	1	20
1	30	0.12	1	30
10	30	0.80	1	30
20	30	0.82	1	30
30	30	0.91	1	30
40	30	0.87	1	30
50	30	0.80	1	30
1	40	0.12	1	40
10	40	0.89	1	40
20	40	0.82	1	40
30	40	0.87	1	40
40	40	0.90	1	40
50	40	0.89	1	40

Figure 1: Scalability Table

We can see from the table that our RL program can handle increasing workload size. Adding threads improves outcome because we are essentially adding random exploration. I would argue that our RL program is highly scalable because we do not need to add proportional Threads to a growing workload to see high optimal path hits.

In our scalability tests we needed a way to find the optimal path length to determine if a path choosen by the agent is optimal. We decided to create a tree structure that modeled all the possible combinations of paths. Taking the branch with the shortest length that reaches the winning position is considered an optimal path. Each branch in the tree is a unique combination of actions that either leads to a winning position, or a ”stuck” position.

7. Version Control

All of the code used can be found on github:

https://github.com/dennisshpits/ToyQLearning

We used github to encourage other developers to make contributions to our RL project. If someone is very inquisitive, the specific commit where we added OpenMP can be found here:

https://github.com/dennisshpits/ToyQLearning/pull/1

8. Concluding Remarks

We have presented a way to converge to the optimal Q learning function using OpenMP by allowing multiple agents to explore a grid world simultaneously. The algorithm is quite simple and only makes use of standard directives in OpenMP and is thus easily portable. The algorithm was implemented in C++ using OpenMP and compiled with gcc using the -Wall -g flags. OpenMP can replace the need for adding a random probability ϵ for choosing an action (not the greedy choice) for exploration in Q learning which leads to a faster algorithm.

REFERENCES

[1] Matei Ciocarlie, *An Introduction to Deep RL from a Robotics Perspective*, Columbia Engineering short course, Summer 2020

[2] *APMA 4302 - Methods in Computational Science*, Columbia Engineering course, Fall 2021