Laboratory: Value Iteration on a Grid World using Python.

Author: Carlo Lucchesi
E-mail address
carlo.lucchesi@edu.unif.it

Prof: Marco Bertini
Course: Parallel Computing

Abstract

Synchronous Value Iteration is the first algorithm for model-based Reinforcement Learning, suited for worlds with finite state and action space. Synchronous update requires the use of two arrays, lacking of data flow dependency and allowing for an easy parallelized version. The asynchronous variant aims for a faster convergence, but a parallel version requires a synchronization to be adherent to the original algorithm. In this report, we will compare different version written in Python, both exploiting multiprocessing and multithreaing, now suited for CPU bound tasks using the GIL free build.

1. Introduction

1.1. Problem

Value Iteration (VI) is a basic algorithm to learn a Value function in a model base context, where the state is finite. It consists in resolving the Bellman's Optimality Equations with an iterative algorithm. The $k+1$-th approximation of the value function, that can be represented as a vector, can be obtained by applying the Bellman's Optimality Equations on the previous approximation. It can be demonstrated that, whatever is the starting vector, this algorithm can converge to the optimal value function. Once obtained, it's possible to define an optimal policy.

In asynchronous VI, in every iteration only one state is updated, while the other remains the same. The idea it that the following iterations can exploit the updates made in previous one, instead of waiting for every state to change. It can be demonstrated that even this converges to the optimal value function, given that every state is sampled infinitely often.

1.2. Machine Specifications

The computer used for executing the experiments is a laptop, which was always connected to AC during the runs. The full specifications are the following:

- **CPU**: AMD Ryzen 5 7520U 4 core (8 threads) 2.8 GHz.
- **Cache**: 64 B of cache line, 256KB L1, 2MB L2, 4MB L3.
- **RAM**: 8GB.
- **OS**: GNU/Linux (Pop_OS! specifically, an Ubuntu derivative).
- **Interpreter**: 3.14.2 free-threading build
- **Environment Manager**: Conda.

For these tests, CPU boosting was disabled to avoid thermal throttling. In fact, modern CPUs can boost to higher clocks speed for a brief time to meet spikes in compute loads, but with a substantial increase in thermal dissipation. To preserve CPU longevity, when it gets too hot maximum frequency is limited. For long computational times (like the hours needed to perform all the runs), this mechanism is not predictable and advantages the first runs, skewing the results.

2. Experiments Description

Every experiment was run 6 times, the first excluded and then mediated.

2.1. Grid World

Grid World is a basic artificial model used for studying RL algorithm. The environment is composed by regular cells, and the agent state corresponds to one of the possible free cells. Generally, inside the environment there are four types of cells:

- **Free Cells**: one of the possible state of the agent. Most of the time, they are associated with a 0 reward, or slightly negative to favor short paths.
- **Obstacles**: the agent can't go in these cells, so them aren't inside the state space.
- **Goals**: one or multiple cells that give a positive reward to the agent. The agent should aim to these cells.
- **Traps**: cells that have negative reward and that make the episode stop, preventing the agent to reach the goal.

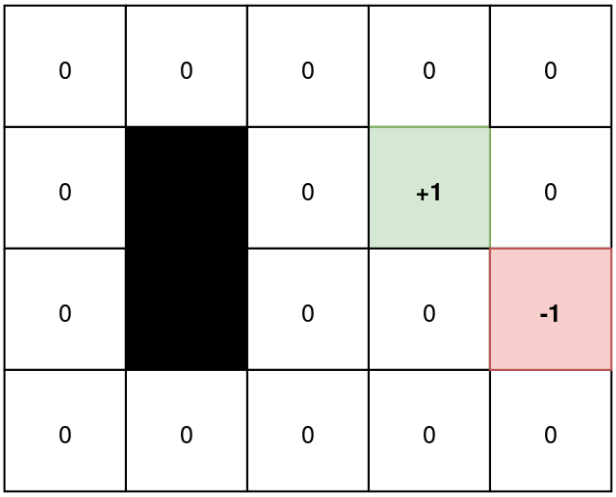


Figure 1: Example of rewards in a grid world.

RL task are used in uncertain environments called Markov Decision Process (MDP), so that the result of an action made by the agent is not deterministic. In this implementation of grid world, given an intended action, there is a 10% probability to move sideways (if the action is *UP*, the sideways moves are *LEFT* and *RIGHT*). If the action tries to go against an obstacle or outside the map, the agent simply bounces on it, remaining in the starting cell.

Bellman Equations requires the use of a discount factor to guarantee algorithm convergence and make push agent towards shorter path. In this case, a discount factor of $\gamma = 0.95$ was used.

2.2. Synchronous Value Iteration

In this project, the algorithm implemented has the aim to *resolve* Bellman's Optimality Equations for every state, instead of focussing on maxing returns from experiments of an agent in a particular spot. This choice derives from the goal of this project: using different levels of parallelism to speed-up tasks. In the context of an agent on a particular starter point, other types of algorithm are used, but much harder to parallelize.

If the state space is finite and not too large, the value function $V(x)$ can be seen as a vector, with dimension equal to the state space cardinality. Synchronous VI applies the Bellman Operator in equation 1 starting from an approximation V_k and get V_{k+1}.

$$\forall x \in \mathbb{X}$$
$$V_{k+1}(x) = \max_u \{r(x,u) + \gamma \sum_{x' \in \mathbb{X}} \phi(x'|x,u) V_k(x')\} \tag{1}$$

To be correct, this update cannot append in-place, requiring at least two arrays.

As a stopping condition, the relative difference in norm between two iteration must be smaller that a certain threshold, as shown in figure 2.

$$\frac{||V_{k+1} - V_k||}{||V_k||} < \epsilon \tag{2}$$

2.2.1 Joblib

Calculation of different coefficients of the new value function (V_{k+1} in equation 1) doesn't have any flow dependence for different state $x \in \mathbb{X}$, so they can be easily parallelized.

The standard approach for parallelizing CPU bound tasks in python is multiprocessing. To easily handle process spawning and join, Joblib library was used. The python code implementing the Bellman Operator was already done to be applied only on a slice of states. So, for every iteration it will be spawn a certain amount of worker, and will be created an equal amount of slices to elaborate, so to minimize worker scheduling.

The sequential version used an external `numpy` array for writing the result, but in-place changes cannot be done using processes by default. To make the code easier, every worker will return its slice of output, and then it will be reassembled.

Joblib[1] offers the possibility to swap backend to use different constructs to parallelize. The default one is `Loky` and is based on multiprocessing, but it can be instructed to use threads. Both the backend was used.

In addition, Joblib offers patterns for an easy *memoization* of function result. After some testing with the Markov Transition Density generator of the grid world, it was able to generate the results much quicker than a disk cache. For this reason, memoization was not used.

2.2.2 Python Threads,

After the Joblib implementation using the `Parallel` construct, an equivalent version is done using the `ThreadPoolExecutor` directly from python. To obtain a real parallelism with python's threads, the release of GIL is necessary. Even if the code tries to use as much as possible `numpy` data structures, it doesn't release the GIL significantly (or maybe at all). So, to obtain any speed-up the free-threading python build was used during the experiments.

2.3. Asynchronous Value Iteration

VI can be implemented with an asynchronous update, were only one state is changed on every iteration. The idea is to have quicker but more iterations, and still converge faster than synchronous. This algorithm it's meant for much higher space state, considering that the updates can be done inplace.

A comparison between asynchronous and synchronous VI on the convergence of every state in the value function vector isn't fair. In general, the two condition of termination are different, making them not directly comparable. So, asynchronous version will be treated separately.

Lastly, due to a lack of a simple condition of termination found in literature, the following approach was used: a synchronous instance is executed to obtain a reference, then the asynchronous version tries executing until the results are close enough to the target. The condition to meet is described in equation 3.

$$\frac{||V^* - V_k||}{||V^*||} < \epsilon \tag{3}$$

2.3.1 Thread Safe Implementation

In this case, there is a data flow dependency if two distinct threads samples the states x_i and x_j, and if them are both possible following state of each other. In terms of Markov Transition Density this results in equation 4.

$$\begin{aligned} &\exists u \in \mathbb{U} \text{ so that } \phi(x_j|x_i, u) \neq 0 \\ &\exists u \in \mathbb{U} \text{ so that } \phi(x_i|x_j, u) \neq 0 \end{aligned} \tag{4}$$

Notice that in the case of a grid world, two adjacent cells meet this condition.

To avoid that, every thread before reading from the vector it should acquire all locks from the adjacent cells (with Manhattan distance of 1).

2.3.2 Neglecting Locks

After implementing a proper thread safe implementation of the asynchronous VI, also a not

thread safe one was tested. The idea is that, after all, value iteration calculates sequences of approximations. So having an implementation that is not exactly reducible to and equivalent sequential one can be acceptable.

In fact, both version are able to meet (with different vectors) the stopping condition, with a negligible difference between them.

3. Results

In the following sections are reported the result obtained. The sizes of the problem used in the experiments are quite small. This is due to the limited compute power of the machine used, and the general slowness of interpreted languages like python.

The size reported in the following figures refers to the width and height of the grid, so that is a square of $size^2$ cells. Every world is randomly generated, but using the same seed, so world of same sizes are identical. Every world has exactly one goal and one trap, plus 5% of the cells are walls.

3.1. Synchronous Value Iteration

The result of the first test can be seen in figure 2. As expected, speed-up increases together with the number of threads/processes used, making a top of 4 a then slightly decrease. When the number of concurrent streams exceed the number of physical cores, the streams starts to share the ALUs inside the core, adding overhead from task scheduling. Still, speed-up increases, but with a lower rate, up to 8 concurrent streams.

From this result, it seems that using python threads (with a No GIL interpreter) is more efficient than the multiprocessing made by Joblib. This result confirms empirical experience when confronting threads with processes. The worse performance of Joblib with threads was not expected, but is a phenomenon observable in all the experiments.

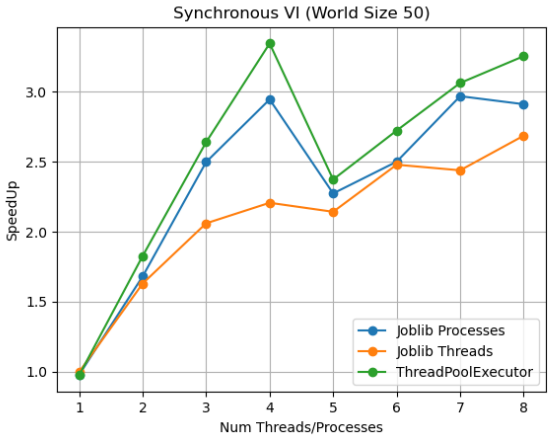


Figure 2: Speed-up changing parallelism grade with a small world.

When incrementing the size of the grid (4 times the number of cells) the relation between multithreading and multiprocessing reverts. In fact, in figure 3 we can see that up to 5 concurrent streams threads and processes are identical in performance. With higher level of parallelism, Joblib with processes are more performant than the thread version using `ThreadPoolExecutor`. Still, the thread backend of Joblib is inferior in terms of performance.

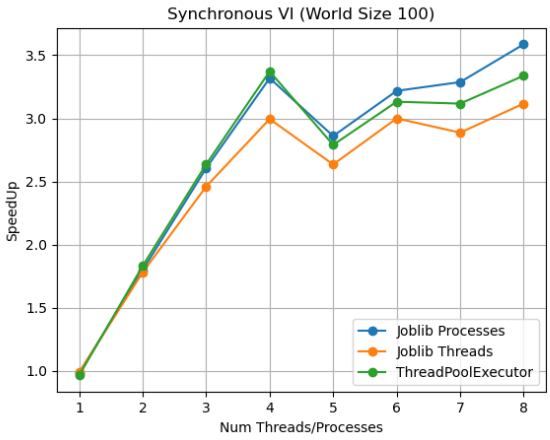


Figure 3: Speed-up changing parallelism grade, but with a greater world.

This effect of Joblib handling better higher amount of processes occurs across different sizes. In fact, if we limit the number of threads/pro-

cesses to 4, figure 4 shows how threads are more performant for smaller sizes, but almost identical with bigger sizes.

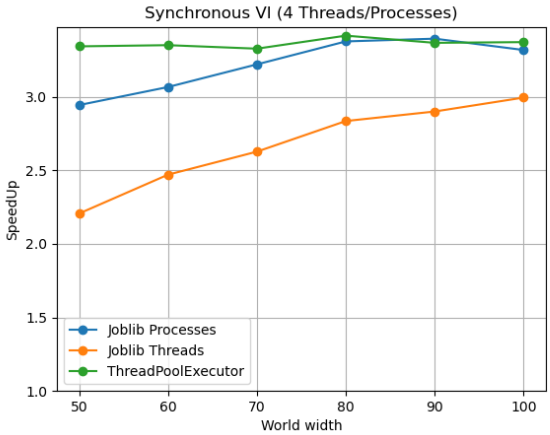


Figure 4: Speed-up achived wth 4 thread, at different world sizes.

With higher parallelism, figure 5 shows that threads obtains the same speed-up, while the Joblib with processes is able to a higher to.

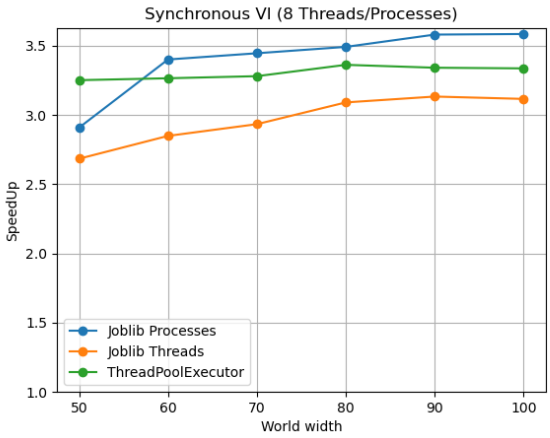
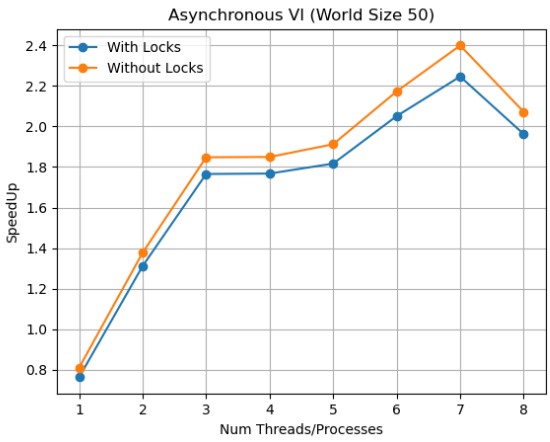


Figure 5: Speed-up achieved with 8 thread, at different world sizes.

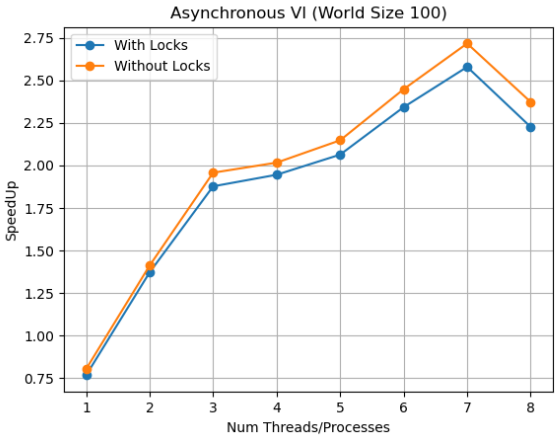
From one perspective, these results confirm the empirical experience that parallelism obtains better results with bigger problems. But, on the other hand, we obtain a higher speed-up using Joblib with processes than threads. This *inversion* can be imputed to a not optimal implementation of the thread version of the code. Also, Joblib allows setting the maximum number of workers to use, and this doesn't imply that uses all of them constantly. We can fairly assume that Joblib is well optimized and capable of finer optimizations than a naive implementation using `ThreadPoolExecutor`. Still, it's impressive to see how well optimized `Loky` is, for even outperforming the Joblib's thread backend.

3.2. Asynchronous Value Iteration

In figures 6a and 6b we can see the speed-ups of the parallel of the asynchronous VI algorithm. In this case, both the variants are made using threads.



(a) Speed-up with a world size 50.



(b) Speed-up with a world size of 100.

Figure 6: Speed-up obtained in asynchronous case.

The first thing that we can notice is how speed-up is below one in the case of one single threads, showing how thread handling overhead has a greater impact. This derives on how the parallelization is done. In fact, for every iteration of the algorithm the threads must be created, started and then joined. As said before, asynchronous VI works by executing much more iteration. So, this overhead magnifies much more than the synchronous one.

Also, can be noted that the implementation that neglects race condition has a little speed-up, as expected. However, the difference is very small.

In figure 7 can be seen that the version without locks has a higher speed-up consistently. This effect was expected, a for this reason not discussed any further.

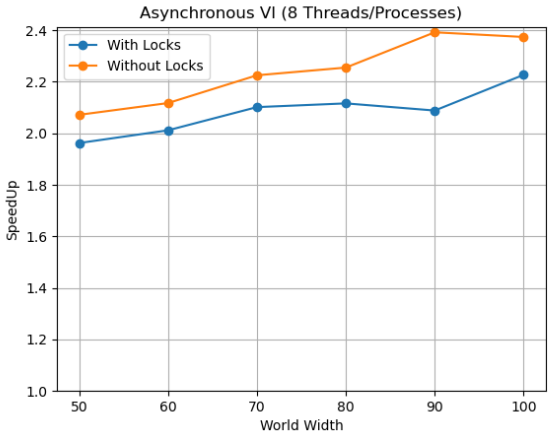


Figure 7: Speed-up for the asynchronous version, with different world sizes.

4. Conclusion

In this report, we saw a few implementations of synchronous and asynchronous VI with different parallel approaches.

Synchronous VI, based on how it updates the value function, it's much easier to parallelize and reaches higher speed-ups than the asynchronous one. Even if GIL free python interpreter shows a meaningful performance boost, it's not a trivial task to outperform using threads already established parallel libraries based on processes. As 4 shows, with smaller sizes where overhead has a greater impact, multithreading beats multiprocessing, stating once again that threads are *less expensive* than processes. Moreover, the `Loky` backend is capable of even better performance than multithreading, even if both are used with Joblib. This shows how mature and well optimized libraries can beat naive implementations, making them an easy and compelling tool to use.

For asynchronous VI, parallelization is harder, and has diminished benefits. From a practical point of view locks acquiring can be ignored, considering that the result are approximation and differ a little between them. Nevertheless, the benefit from this relaxation is disappointing. In fairness, someone that ponders to use asynchronous VI is probably in a different setting than this (optimize the reward of an agent in a particular spot), and so he should focus more on faster variants of the algorithm instead of parallelization.

Eventually, python true multithreading allows new possibilities for CPU bound tasks, but current state-of-the-art technologies didn't become irrelevant, or even remaining the way to go for python parallelism.

References

[1] Joblib docs. https://joblib.readthedocs.io/en/stable/. 3