A Parallel Reinforcement Learning Algorithm for Model Based Control of Transient Processes

Cole Dunbar and Aidan Rigby

Github: https://github.com/acrigby/ALPACA/tree/main/ME759

Abstract

The inherently sequential deep Q-learning algorithm is adapted to allow for parallelization on a HPC in an attempt to decrease total computational expense of training. The work performed is a preliminary analysis using OpenAI's CartPole model for future implementation using the Dymola solver in support of nuclear energy system research. A parallel implementation is developed using python and run on the Euler HPC by way of a slurm script. A scaling analysis of the number of processes affect on compute time showed promising speedup using at least five parallel processes. However, a tradeoff between number of processes and learning stability was shown in the scaling analysis. Extension of the CartPole model to better simulate the computational expense of Dymola showed that using six processes in parallel provides a 5x speedup compared to the sequential implementation when the computation is dominated by the C++ executable expense.

1. Introduction

1.1. The Project

This project looks to improve the execution speed of an existing sequential deep Q-learning algorithm based on the algorithm here by employing parallel computing. The result will be a parallel solution algorithm written using python that trains an agent to control a black box executable model based on the framework developed by OpenAI's CartPole model from the Gymnasium library. The solution will run the model on multiple threads (runs currently performed sequentially) on the high-performance computer Euler thus improving learning speed dramatically. The work undertaken will also attempt to perform the neural network update process in parallel with the parallel model evaluations and therefore gain speed performance benefits this way.

1.2. Motivation

Prior research between UW Madison and Idaho National Laboratory has shown that deep Q-learning algorithms can offer improvements in safety and economics of nuclear transient process controls by generating feed-forward signals for PID controllers. The transient based modelling and solver tool Dymola has a library of nuclear energy and integrated energy system models which can be exported as a (license locked) C++ executable. The aim of this project is to demonstrate that the existing reinforcement learning (RL) solution utilized with Dymola could be parallelized on a HPC to reduce the current run time (currently approximately 3 days). This run time is prohibitive to explore the range of feed-forward signals required to allow operation across the full operating regime of a nuclear power plant. To demonstrate that a speed up can be achieved the solver algorithm will be parallelized to control a "dummy" C++ executable model in place of the license locked Dymola model so that this implementation can be tested on Euler.

2. Methodology

2.1. Original Proposed Approach

To any RL solution there are two key processes: episodes (blue) and network updates (orange). Episodes consist of a series of environment evaluations based on the current policy network. In the algorithm investigated each episode generates around 200 combinations of action, states, and rewards with this process seen in Figure 1. The network update steps use these 200 combinations to update the policy network that is used to generate the agent's actions. The current algorithm runs an episode and at every step pushes the output result to the memory.

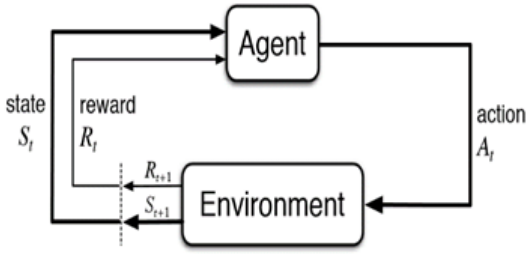

Figure 1: Reinforcement learning step methodology (https://towardsdatascience.com/reinforcement-learning-101-e24b50e1d292)

The update policy network runs after every 10 episodes and takes a random combination of these results to update the policy network. This is shown in time in Figure 2, the reason for the long run time with the sequential algorithm is evident.

The initially proposed solution from the project proposal is shown in Figure 3 it planned to run up to 16 models (shown for 8 different CPU threads) in parallel and then push all the action, states, and rewards to memory afterwards. The aim was to run the evaluations concurrently with the updating of the policy network based on the previous set of episode results.

2.1.1. Implemented Parallelization Strategy

While the sequential process described above holds for deep Q-learning an additional update step is completed within every step evaluation when the actions, states, and rewards are pushed to memory. This step is required for learning stability. This realisation makes the problem much more complex than initially proposed as the project becomes heavily sequential in order to maintain stability. Getting around this required some sacrifice in terms of stability (and consequently learning speed) in order to

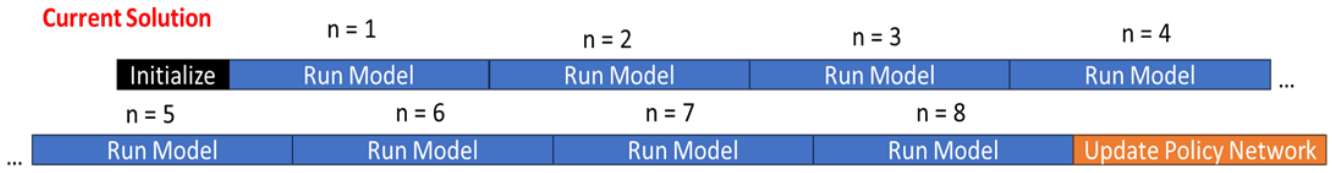

Figure 2: Sequential code algorithm schematic

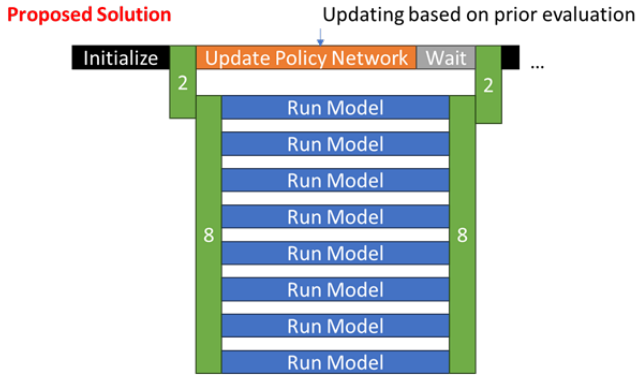

Figure 3: Proposed parallel solution algorithm schematic

implement a parallel solution. The eventual approach is seen in Figure 4.

This solution launches n threads (shown here for 3 threads). It runs an episode on each thread and as per the proposed solution runs a step and pushes the observations to a shared memory queue atomically. This shared queue is denoted the replay memory queue. In a change to the proposed solution every k steps of the episode a thread enters a critical construct. Thread safe access to a critical construct in python multiprocessing uses semaphores. A semaphore is obtained by a thread and the shared policy network is updated based on the current instance of the replay memory queue. The sempahore is then released for other threads to obtain.

This updated policy queue is used for any future action evaluations in all threads in future steps. Once all n episodes have been completed the target network is updated using the current replay memory and policy network. This is relatively short compared to the episodes and thus is completed sequentially. Performing this in parallel as was proposed has significant stability implications with limited time advantage and thus is not implemented.

This is in essence a manual version of pipelining with the n threads shifted by one update policy network step length. Provided the episode evaluation length dominates the length of the update step then the solution can be parallelized effectively. This however does have minor stability implications discussed in the results section.

2.2. Code Difficulties

The proposal suggested the use of PyOMP which aims to extend the functionality and taxonomy of OpenMC to python for thread based parallelism. Unfortunately this code is still in development and did not integrate well with the updated versions of packages required by the reinforcement learning. Given this it was decided to use pythons native multiprocessing support to parallelize the code based on the learning in ME759. Three key problems were overcome to achieve OpenMC like flexibility in the implemented solution. These were shared memory design and use, variable scoping and use of critical/atomic operations.

2.2.1. Shared Memory

Because of the lack of low level control of memory in python the use of shared memory between threads typically uses libraries that compile to c++ code. Python multiprocessing provides basic constructs for arrays of floats or integers but complex tensors or queues are not supported.

The policy and target neutral networks are formed of pyTorch tensors. PyTorch has a native method to place these in shared memory and this was used in this project. For the replay memory a novel workaround was found in which to create a new class based on the support for floating point arrays in multiprocessing library. This extended this class to allow for a maximum length queue and queue methods such as sample and length. This is held in the ParallelSolverClasses.py file. Lastly the rewards from each episode are also placed in shared memory for easy communication between threads. This is a simple array of floats.

2.2.2. Variable Scoping

Variables explicitly shared between threads were placed in shared memory as described above. Other hyperparameters were passed privately to each thread at a time. A global set of variables was defined in the main function but when a thread is launched each variable is instantiated to another variable within that thread of the same name. This caused problems until it was grasped that the global variables were not shared amongst threads and each thread maintains it's own private version which is destroyed when a thread is joined.

2.2.3. Critical and Atomic Operations

Because of the use of shared memory between threads race conditions easily arise especially with the use of a shared queue and neural network. The process of adding to the queue (appending) was carried out using an atomic append which was protected through the use of multiprocessing's native manager function. This allowed each thread to access and append to the custom shared replay memory without overwriting or interfering with other threads accessing the same queue.

Accessing and updating the shared policy network required more complex access requirements to prevent race conditions for an entire function. This was achieved using semaphores. A semaphore is an object created at the start of the program that can only be processed by one thread at a time. This prevents the need for explicit synchronisation allowing much faster code execution whilst preventing any

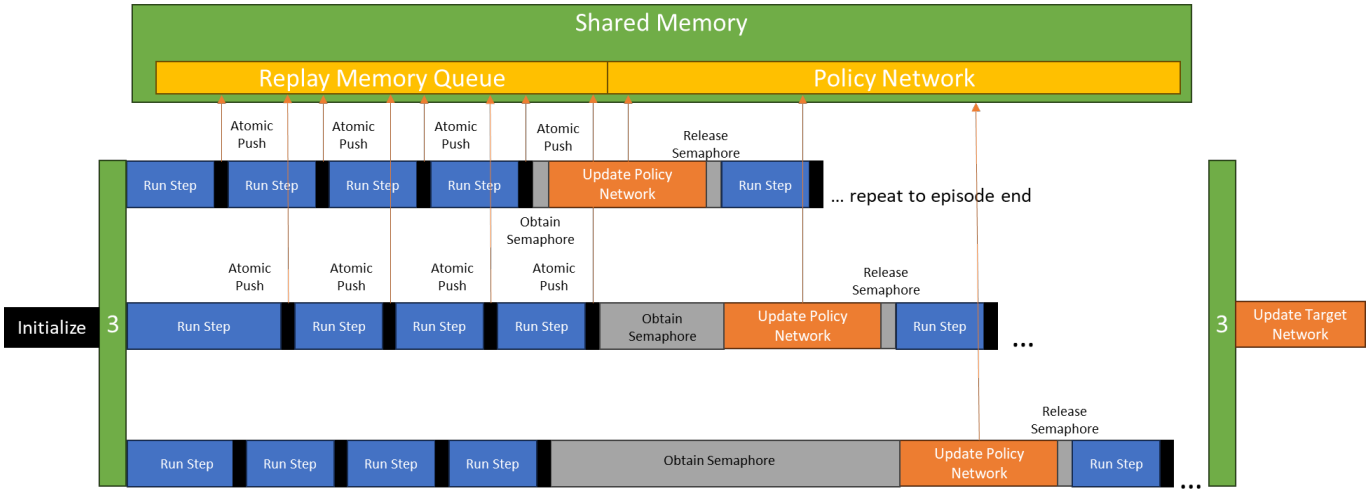

Figure 4: Implemented parallel solution algorithm schematic

two threads from calling a given function simultaneously. A thread acquires a semaphore as long as no other thread possesses it, performs the block of code after it, and then releases the semaphore for any other thread. This protects write access to the policy network. Read access is not explicitly protected as this would prevent paralellization and each thread can sample the policy network in read only access when required.

3. Running the code on Euler

To run the code on Euler, firstly the entire ALPACA repository main branch must be cloned to the relevant location on Euler. Once this is done navigate to the ME759 folder. The code can then be simply run by calling 'sbatch finalProjectcpp.sh'. This is a Slurm script which sets up the necessary environment, including all required packages, and executes the python script. The slurm script detail is described below.

In order to run python on the Linux-based system, a virtual environment must be constructed into which all the necessary packages are installed. Once the environment is created, it is activated such that all subsequent commands will be executed within the environment. Since changes were made to the Gymnasium library in a local branch to use a C++ executable in the python code, it is necessary to install this branch to the environment through the ALPACA repository. Some additional packages (PyTorch, matplotlib, and IPython) are installed to the environment using pip. Once these packages are installed to the environment they will remain there on subsequent runs, but they are left in the Slurm script for redundancy and the ability to cold start the code by a new user. Finally, the C++ executable which calculates the CartPole system dynamics using Runge-Kutta 4^{th} order model is compiled for redundancy. The following lines of the script are used to execute sequential or parallel implementations of the model. A single integer parameter is passed to the python script to define the number episodes between target network updates and, in the case of parallel implementation, the number of threads to launch.

4. Results and Discussion

4.1. A note on stability

Deep Q-learning is inherently sequential for maximal stability (referring to the code always converging); the next step has to be based on all the previous actions. In this parallel approach the next step in any parallel episode depends both on it's previous observations but also on none dependent observations form other episodes performed simultaneously with this episode. It can be seen for the learning graphs shown in Figure 5 and Figure 6 that this does impact convergence with the parallel solution converging slower than the sequential solution when measured against episode.

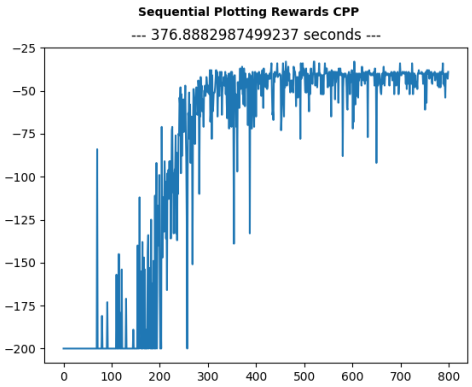

Figure 5: Convergence of sequential deep Q-learning

While for 800 episodes both solutions train a converged policy network the parallel approach is less stable and can result in more unreliable rewards once trained. One solution would be to increase the number of episodes used for training in the parallel solution while keeping it low enough to still yield an overall speedup of training time. Additionally, it is suggested that future work may look at some hybrid sequential and parallel training to obtain speed increases in the code whilst improving stability.

4.2. Scaling Analysis

First, a scaling analysis of the parallel implementation varying the number of processes launched and, conse-

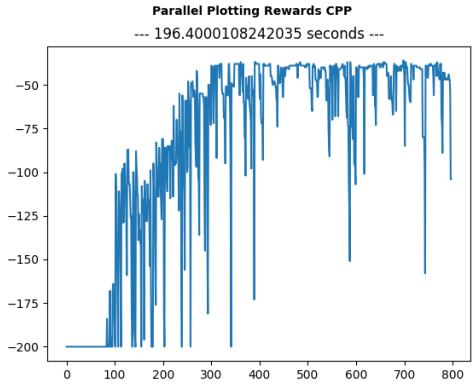

Figure 6: Convergence of parallel deep Q-learning

quently, the frequency of update steps in both implementations was performed. This analysis tested one through ten processes launched with the figures of merit being training time and training stability. Figure 7 shows the compute time results of the scaling analysis.

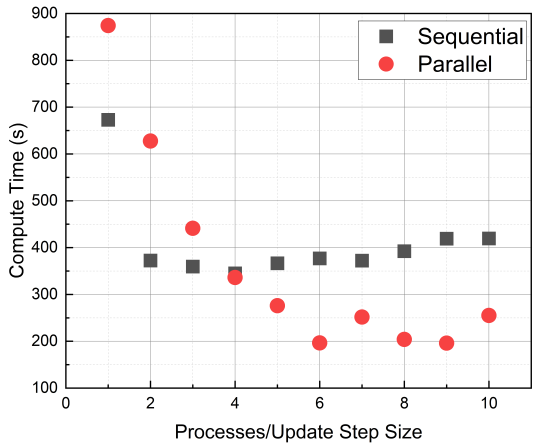

Figure 7: Scaling analysis of compute time as a function of number of processes launched

The scaling analysis shows that the compute time of sequential deep Q-learning is relatively stable regardless of the update step size. The parallel implementation yields a speedup when the number of parallel processes launched is greater than four. The cause for parallelization slowing down the computation at lower number of processes is the addition of significant computational overhead due to parallelization. This overhead must be compensated for by speedup due to the simultaneous processes.

Above six parallel processes, the compute time appears to converge to a computational limit. Stability analysis of the converged models showed that the overall stability of the learning model decreases with an increasing number of parallel processes. For this reason, six parallel processes is chosen for future analysis to minimize computational time while maintaining an acceptable level of stability.

4.3. Application to larger C++ executable

In the application of parallel deep Q-learning to Dymola, the largest computational step is running a C++ executable every step of every episode, which takes approximately 1 second to complete. While this is not computationally expensive in and of itself, when scaled to 200 steps per episode and 800 episodes for training, the total computational expense of this executable is nearly 45 hours! In the adapted CartPole model used in this work, the Runge-Kutta 4th order calculation is completed via C++ executable to act as a stand-in for the Dymola process. However, the computational expense is insignificant as implemented and does not properly represent a computationally expensive executable being called each step. Extending the present work to closer simulate Dymola is done by adding a sleep function to the RK4 executable. The results of compute time as a function of the added time delay is shown in Figure 8.

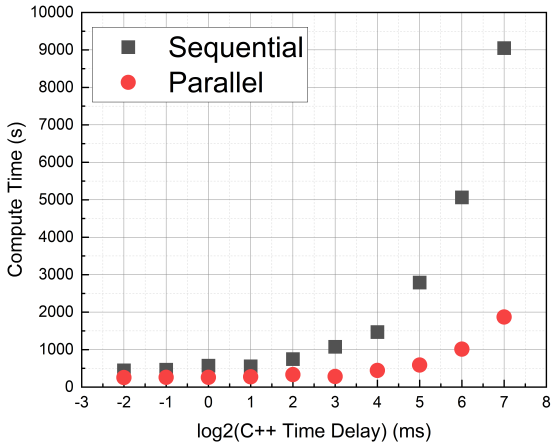

Figure 8: Compute time as a function of C++ executable size

At low time delays (¡4 ms), there is no appreciable difference in total compute time even as the time delay increases. This is likely due to the process by which python launches the C++ executable and subsequently reads its output being more computationally expensive for these fast executables. Above 4 ms time delay, the compute time beings to increase for both the sequential and parallel implementations, with the compute time doubling between 32 ms, 64 ms, and 128 ms delays. This proportionality of compute time increase to delay time increase shows that past 32 ms the execution of the model becomes dominated by the C++ executable, as is the case with Dymola. Figure 9 shows the same set of data as Figure 8, but in terms of overall speedup as a result of parallelization. It is clear from this plot that with large C++ executables, the there is an asymptote of 5x speedup due to parallelization. This makes sense intuitively as the parallel implementation launches six processes simultaneously. It is expected that varying the number of processes (n) would yield a speedup of (n - 1)x using parallel processes.

5. Conclusions

A parallel implementation of deep Q-learning using OpenAI's CartPole model was developed using python to investigate the applicability of parallelization on an HPC for deep Q-learning. The solution involves launching multiple processes which simultaneously run training episodes

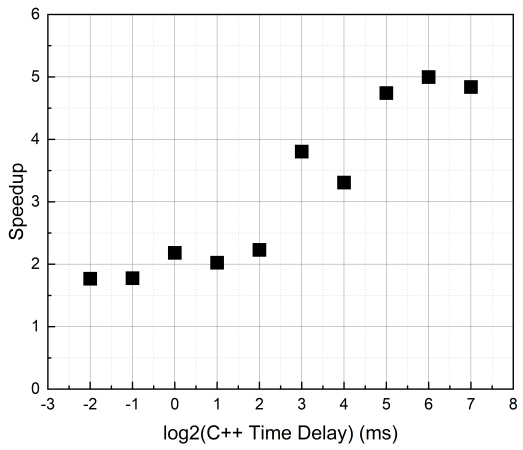

Figure 9: Speedup due to parallelization as a function of C++ executable size

while updating the policy in shared memory. A dummy C++ executable is used to calculate the state using a Runge-Kutta 4^{th} order method as a stand-in for future use of this implementation using Dymola. A slurm script is created to allow for any user to run both the sequential and parallel deep Q-learning models on the Euler HPC from the github repository without additional work required by the user.

A scaling analysis yielded increased speed of execution due to parallelization once the overhead of python parallel processes was overcome. A solution stability analysis showed that stability is highly dependent on the number of processes launched, so six parallel processes was chosen to adequately balance computational speed and solution stability. To better simulate Dymola, a sleep function was added to the dummy C++ executable to simulate a more computationally expensive executable. The results of a scaling analysis performed on the computationally expense of the C++ executable showed an overall speedup of 5x using the parallel solution with 6 parallel processes compared to the sequential model. In the case of Dymola, this would bring the computational time from 3 days to nearly half a day.

Future work will be focused on optimizing the compute time and increasing the solution stability. The proposed strategies for achieving this are slightly increasing the number of episodes in the parallel implementation to allow for slower convergence or using a hybrid sequential-parallel solution by balancing sequential sections for stability and parallel sections for speedup.

6. References

1. Pigott, K. Baker, S. A. Dorado-Rojas and L. Vanfretti, "Dymola-Enabled Reinforcement Learning for Real-time Generator Set-point Optimization," 2022 IEEE Power & Energy Society Innovative Smart Grid Technologies Conference (ISGT), New Orleans, LA, USA, 2022, pp. 1-5
2. Gymnasium, Zenodo, Towers, Mark and Terry, Jordan K. and Kwiatkowski, Ariel and Balis, John U. and Cola, Gianluca de and Deleu, Tristan and Goulão, Manuel and Kallinteris, Andreas and KG, Arjun and Krimmel, Markus and Perez-Vicente, Rodrigo and Pierré, Andrea and Schulhoff, Sander and Tai, Jun Jet and Shen, Andrew Tan Jin and Younis, Omar G., 2023, 10.5281/zenodo.8127026
3. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L. and Desmaison, A., 2019. Pytorch: An imperative style, high-performance deep learning library. Advances in neural information processing systems, 32.
4. https://towardsdatascience.com/deep-q-network-with-pytorch-and-gym-to-solve-acrobot-game-d677836bda9b