PYTPG: A DISTRIBUTED AND MULTITHREADED IMPLEMENTATION OF TANGLED PROGRAM GRAPHS FOR REINFORCEMENT LEARNING

by

Bryce MacInnis

Submitted in partial fulfillment of the requirements
for the degree of Bachelor of Computer Science, Honours

at

Dalhousie University
Halifax, Nova Scotia
August 2024

© Copyright by Bryce MacInnis, 2024

Table of Contents

List of Tables

List of Figures

Abstract

Tangled Program Graphs (TPG) have been shown to be competitive with Deep Q-Learning Networks (DQN) at reinforcement learning benchmarks such as the OpenAI's Gym tasks and games in the Atari Learning Environment (ALE). This paper presents `pytpg`, a multi-threaded and distributed open-source implementation of the Tangled Program Graph genetic model. Previous single-threaded implementations of TPG are limited by their slow training performance. This paper provides details about the implementation of `pytpg` such that it can be reproduced by others, as well as providing a benchmark of its performance compared to its single-threaded counterpart. The benchmarks used for this paper are the *CartPole-v1*, *LunarLander-v2*, and *CarRacing-v2* tasks from OpenAI's Gym. The metrics used for benchmarking are throughput, time vs generation, and CPU utilization.

Chapter 1

Introduction

Tangled Program Graphs (TPG) are a genetic programming model capable of solving reinforcement learning problems. In other papers, TPG has been shown to be competitive with Deep Q-Learning (DQN) reinforcement learning models in benchmarks such as the Atari Learning Environment (ALE) [1] [2] and VizDoom [4][5]. Unlike Deep Q-learning models which are trained by each step (an interaction with the environment), as a genetic model, TPG is trained only after the completion of each episode. To achieve performance parity with state-of-the-art DQN methods, it is required to train TPG models exhaustively.

While some research interest has been found in improving the computational efficiency of the Tangled Program Model itself, such as by implementing the TPG algorithms through CUDA. The direction of research in this paper addresses another bottleneck which limits the training even more so than the algorithms' implementations. This bottleneck is the interaction with the task environments.

To address the vocabulary used in this paper, a *step* is a single interaction between the model and the task environment. An *episode* is a consecutive series of steps taken until the task environment terminates signalling that the task is complete. A *policy* is a chosen set of actions for a given set of observations provided by the task domain. A *team* is the fundamental unit of TPG. Teams are candidate policies for interacting with the environment. Finally, a *generation* is a set of episodes where each team in the Tangled Program Graph attempts to solve the problem.

To train the Tangled Program Model until it is competitive with other state-of-the-art reinforcement learning models, the model needs to interact with the task domain through hundreds or even thousands of generations. Single-threaded implementations of Tangled Program Graphs start each episode serially. In other words, the next episode does not begin until the preceding episode finishes. In order to overcome this lost time efficiency, this paper proposes a multi-threaded implementation of Tangled

Program Graphs where multiple episodes train simultaneously. This would allow for a vertically scaleable improvement in training efficiency proportional to the hardware's ability to parallelize the execution of the training environments.

As a novel contribution, this paper extends the performance of the multi-threaded implementation by leveraging distributed hardware. Such that the burden of running environments in parallel can be distributed among multiple CPUs on a network which would allow for TPG to train in even further reduced time. In Chapter 2, a high-level overview of what reinforcement learning is provided, the difference between Gradient-based methods and Evolutionary Algorithm methods are explained, and Tangled Program Graphs are explained in detail. Chapter 3 details how experiments are conducted. These details include the hardware used to run the experiments, a description of the database, details on the distributed computing framework are given, as well as details on diversity maintenance, environment seeding, and some concerns around multi-threading. In Chapter 4, the results of the experiments are provided. These experiments benchmark the performance of three metrics: throughput, time per generation, and CPU utilization. In Chapter 5, a summary of the results and findings of this paper is provided and potential areas of future research are detailed. In Chapter 6, there is a small appendix providing the source code to `pytpg`.

Chapter 2

Background

2.1 Reinforcement Learning

Reinforcement learning is an emerging subfield of machine learning. Where an agent must learn how to interact with a dynamic environment. Problems are formally modelled as Markov Decision Processes. A Markov Decision Process is a set of states S, a set of actions A and a transition matrix of probabilities that each action will be taken from each state SXA. When an action is taken, a reward (positive or negative) is provided to the agent. Reinforcement learning is finding the optimal actions to take from a given state in order to maximize the reward. Markov Decision Processes may be finite or infinite. In the case of a finite Markov Decision Process, there is a terminal state such that when an agent reaches this state, the interaction with the environment is finished.

2.2 Gradient-based Methods

While reinforcement learning is formulated around Markov Decision Processes. Typically, the set of states and the transition matrix are unknown a priori. An agent must interact with the environment to learn the complete set of states and the rewards for taking each action while in a given state.

In one of the seminal reinforcement learning works, Andrew Barto and Richard Sutton proposes using the Bellman Optimality equation [6] to calculate V(s), the expected reward of being in that particular state. From this, it is possible to derive a Q-value, Q(s,a), which is the expected reward for taking an action a from the given state s.

There are many reinforcement learning algorithms that learn the Q-values for each state. One solution to reinforcement learning is to choose the action with the maximum Q-value for a given state.

Many solutions use neural networks to learn an estimator function for V(s) and Q(s,a), these are called Q-learners. The neural network is initialized with random weights. Then the output of the neural network is used for the Q-value of each action for the state that the agent is in. The /Temporal Difference/ error, which is the difference between the estimated Q-value and the actual reward received from the environment is computed after each step of interacting with the environment.

The neural network is trained by gradient descent to minimize the temporal difference error until the neural network becomes a good approximator of the Q(s,a) function.

2.3 Evolutionary Models

Unlike gradient-based methods, evolutionary models are Monte-Carlo based solutions. An evolutionary model is initialized with a population of agents, where each agent is initially a random solution to the task domain. After the entire population of agents interacts with the environment, the total cumulative reward is assigned to the agent as a score, usually called fitness. Then each agent is ranked by their fitness. The best-performing agents are kept while a percentage of agents are removed from the population. This is inspired by nature's natural selection. The removed percentage of agents is then replaced by clones of the best-performing agents and mutation is applied to them. After each generation, the average fitness improves. The final action selection policy is the policy of the best-performing agent in the final generation.

2.3.1 Tangled Program Graphs

In this paper, we are implementing Tangled Program Graphs (TPG). TPG is an example of the previously mentioned Monte-Carlo based evolutionary models. In TPG, we have two co-evolving populations of teams and programs. The agents of the TPG model are known as *root teams*. Where root teams are defined as teams that are not referred to by any other teams. A root team represents a candidate solution to the task domain.

Teams and Programs

In a Tangled Program Graph, a team has two components, the first is a set of registers capable of storing double-precision floats. The second is a set of programs that when executed modify the team's registers. A program is a set of register-based instructions. Each instruction is made up of a source register, a destination register, and an operation to be applied to the destination register. The destination register defines which register within the team is being modified, while the source register may either be one of the team's internal registers or it can be an observation from the set of observations provided by the environment after each step.

The implementation of Tangled Program Graphs in this paper has a limited instruction set that allows programs to add, subtract, multiply, divide, take the cosine, and negate based on a condition. This instruction space allows for programs to alter their team's registers in both linear and non-linear ways.

Operation	**Description**
1	$R[X] = R[X] + \text{Source}[Y]$
2	$R[X] = R[X] - \text{Source}[Y]$
3	$R[X] = R[X] * \text{Source}[Y]$
4	$R[X] = R[X] \div \text{Source}[Y]$
5	$R[X] = \text{COS}(\text{Source}[Y])$
6	IF $R[X] < \text{Source}[Y]$ THEN $R[X] = -R[X]$

Table 2.1: The PyTPG Instruction Set

Within the team, each program is assigned an action from the task domain's action space. A team chooses which action to use for each step by a bidding process. Each program's first register, R[0], is assigned a special role. The first register represents the program's confidence that given the observations provided to the program, the action assigned to the program is the most optimal to choose. This confidence value is dubbed the program's *bid*.

Training

Given the set of observations provided by each step, the observations are fed by the team to its programs. The programs execute their instructions using the observations and internal registers as source and destination registers. The first register of each

program is then populated with a bid. After all programs have been executed, the program with the largest bid provides the action that the team will use to interact with the environment. The programs are trained to assign high bids with actions most correlated with greater cumulative reward and likewise, are trained to assign low bids with suboptimal actions. The correct bidding behaviour is an emergent property gained during the training of the model.

When the generation has been completed, teams are ranked by their cumulative reward. A fixed subset of the population is deemed survivors and remains unchanged. While teams outside of that subset are considered uncompetitive and removed from the population. The surviving population is uniformly sampled. The sampled teams are cloned copies of the originals. Finally, variation is introduced to the model by applying mutation to the clones. This mutation changes the instructions, the programs and the team itself for each cloned team. After each episode, this natural selection and mutation process changes the bidding behaviour of the programs. This is repeated after every generation such that the teams converge on an optimal policy for solving the task domain.

Mutation

Teams in TPG can be mutated at the instruction level, the program level, and the team level itself. The following mutations may occur.

1. Instructions may be mutated, such that the operation may be randomly changed to any other operation supported in the instruction set.
2. Instructions may have their source registers changed to any other register which could be a state value coming from the observation space, or another internal register within the program.
3. The destination register, similarly, can be changed to any other register including state values from the observation space or another internal register.
4. Programs may be mutated by adding instructions
5. Programs may be mutated by removing instructions
6. Programs may be mutated by swapping instructions

7. Teams can be assigned more programs from programs that already exist in the population.
8. Teams can be given new programs that are created during the mutation phase.
9. Teams may have programs randomly deleted.
10. Teams may point to other teams, such that the team's action is the pointed team's action

The most significant mutation that can occur is when a team changes its action to that of another team. This allows teams to significantly grow in complexity. It has also been found that specialist teams develop at solving specific parts of the task domain, which other teams can later reuse by pointing to those specialists. [2] This effectively allows for problem decomposition.

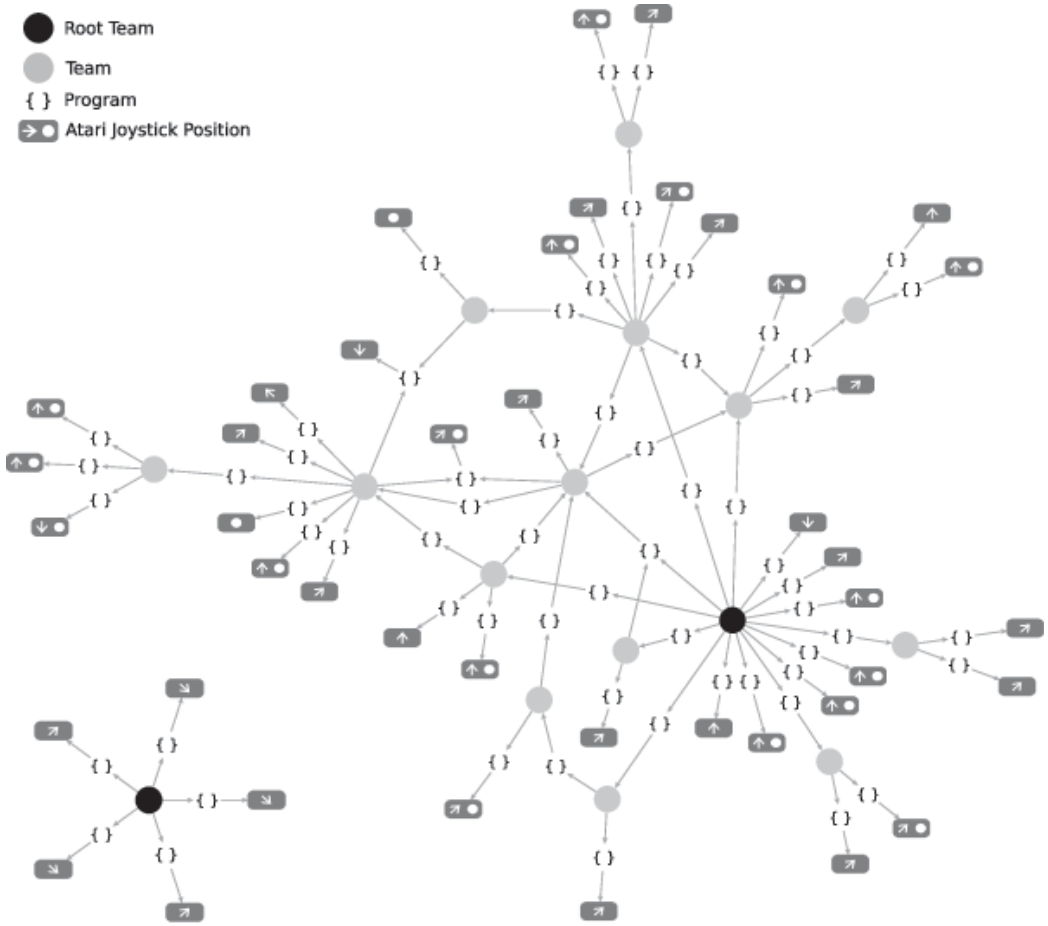

Figure 2.1: A visualization of an evolved Tangled Program Graph [3]

Training Algorithm

Algorithm 1 TPG's Action Selection Algorithm

```
1: Input: All teams, current team, programs, state
2: Add the current team to the visited list
3: Execute programs with the given state
4: Sort programs by bid in descending order
5: Set highestBidProgram as the program with the highest bid confidence
6:
7: Set secondHighestBidProgram as the program with the second highest bid confidence
8: for each program in sorted programs do
9:   if program.action is atomic then
10:    Return program.action
11:  else
12:    Find the team referenced by program.action
13:    if referenced_team is not in visited list then
14:      Add referenced_team to visited list
15:      Return getAction(referenced_team)
16:    else
17:      Return the action of secondHighestBidProgram
18:    end if
19:  end if
20: end for
```

Algorithm 2 TPG's Training Algorithm

1: **Initialize** model, teams, and programs
2: **for** each generation from 1 to *num_generations* **do**
3: **for** each team in the population **do**
4: **Reset** the environment with initial observation *obs*
5: **while** environment is not terminated **do**
6: **Send** the observation to the team
7: **Select the action** as determined by the team
8: **Step** the environment to get new state, reward, and the termination boolean
9: **end while**
10: **end for**
11: **Rank** teams by fitness (cumulative reward)
12: **Select** the top `POPULATION_GAP` percentage of teams
13: **Remove** teams that were not selected
14: **while** number of root teams $<$ `POPULATION_SIZE` **do**
15: **Clone** a selected team
16: **Mutate** the clone
17: **Add** the mutated clone to the population
18: **end while**
19: **end for**

Chapter 3

Methodology

3.1 Hardware

All experiments were conducted on machines hosted in Azure. The worker machines are running on Standard D8s v3 virtual machines using the Intel(R) Xeon(R) CPU E5-2673 v3. The clock speed for that processor is 2.4GHZ and it has 8 vCPUs. Each machine is given 32 GiB of RAM. The supervisor, database and message broker are running on a separate Standard D2s v3 virtual machine with 2 vCPUs and 8GiB of memory. The Standard D2s v3 virtual machine uses a Intel(R) Xeon(R) CPU E5-2673 v3 CPU with a clock speed of 2.40GHz

3.2 Database

To make the training data collected from the simulators more accessible to visualization and further analysis. A centralized database is used to store the information collected from the environments. In order to minimize the performance impact of I/O, all data is collected in a buffer until the entire batch of parallel teams is completed then all data is inserted in a batch operation before starting the next generation.

3.3 Celery

In order to facilitate training on multiple CPUs, a Python framework called Celery is used. Celery handles the back-end communication between computers on a network. In order to train the model with the simulators, it is required that each simulator receives a copy of the model. Celery transmits this data through a message broker running in the background. In this implementation, Redis is chosen as a message broker but the choice is arbitrary.

3.4 Diversity Maintenance

As with any genetic programming model, it is essential that the population is sufficiently diverse in its phenotypic presentation. This means that in TPG, each team should have an independent bidding behaviour and accordingly, its own solution to solving the problem. Through the process of natural selection, it is possible to lose the diversity of the population in a period of premature convergence if the model remains stuck in a point of local optima.

In this situation, the majority of TPG's team population may exhibit the same bidding behaviour. Any novel solutions that are not fully developed yet would be out-competed by the locally optimal teams with higher fitness. If no novel solutions are allowed to develop, and after each round natural selection is applied. Then it is possible for each team to converge to similar bidding behaviour.

Special care is taken throughout this experiment to ensure that diversity is maintained in the team population. In order to do so, a *diversity cache* is implemented. In this diversity cache, the last 50 observations received from the environment is stored within the database. During the mutation that follows after to the natural selection, the bidding profile of each team will be determined by feeding the observations from the diversity cache to each team. A *profile* of a team in the team population will be the actions selected over the last 50 observations received from the environment.

If a team has the same profile as another team in the population, then the second team will experience an additional round of mutation. This effectively is the same as having similar teams mutate twice as fast. Thus encouraging the TPG model towards having more diversity than without the diversity cache.

Additionally, to further prevent the premature convergence of the model. A technique called *rampant mutation* is applied to the model's team population. Rampant mutation will increase the mutation rate, such that the team mutates `N` times as much every `M` generations. This shakes the teams up sufficiently such that if the model was stuck in a local optimum before the rampant mutation then after the series of mutations diversity is re-introduced.

3.5 Environment Seeds

In order to fairly evaluate the fitness of teams in a generation, the OpenAI simulators are initialized with the same random seed given to each team in the generation. A new seed is provided every generation. Setting the seed for the environment ensures that each starting state is the same for each team and the stochasticity inherent to each environment is consistent among each episode. The end result is such that each team is given the same starting state. All teams will receive the same subsequent observations and rewards when their action policy is the same. When the action policies diverge, it is only at that point that the observations and rewards will change.

This is a necessary adjustment to each simulator for natural selection. Otherwise, if teams were evaluated using different starting states, then it would be possible for some teams to be provided with an objectively easier starting position than others. Because of this, the fitness of the teams would skew not by the efficacy of the action policy but instead by luck.

3.6 Multi-threading and Team Batching

Each environment is run in parallel but choosing the number of environments to run simultaneously remains a hyperparameter. To maximize CPU utilization, the number of environments to run simultaneously is determined by the number of CPU cores the hardware provides. It is ideal to run as many simulators as possible in parallel but due to context switching, it is recommended to dedicate one core per environment.

Since hundreds of teams need to be run per generation, it is required to run hundreds of simulators per generation. This means the implementation of Tangled Program Graphs must batch the teams into groups with the same size as the number of cores of the CPU to prevent said context switching.

Chapter 4

Experiments

4.1 Parameterization

Parameter	Value
MAX_NUM_STEPS	1500
POPULATION_SIZE	360
POPGAP	0.5
DELETE_INSTRUCTION_PROBABILITY	0.7
ADD_INSTRUCTION_PROBABILITY	0.7
SWAP_INSTRUCTION_PROBABILITY	1.0
MUTATE_INSTRUCTION_PROBABILITY	1.0
ADD_PROGRAM_PROBABILITY	0.7
DELETE_PROGRAM_PROBABILITY	0.7
NEW_PROGRAM_PROBABILITY	0.2
MUTATE_PROGRAM_PROBABILITY	0.2
TEAM_POINTER_PROBABILITY	0.5
MAX_INSTRUCTION_COUNT	96
MAX_INITIAL_TEAM_SIZE	5
MAX_PROGRAM_COUNT	200
DIVERSITY_CACHE_SIZE	50
RAMPANT_MUTATION_INTERVAL	10
RAMPANT_MUTATION_COUNT	10

Table 4.1: The hyperparameters used in benchmarking the following experiments

The `MAX_NUM_STEPS` parameter defines the maximum number of steps the agent can have with the environment before timing out. The `POPULATION_SIZE` parameter defines the number of root teams (candidate solutions) to keep at every generation. The `POPGAP` parameter defines what percentage of the root teams to keep during a natural selection phase. The following parameters `DELETE_INSTRUCTION_PROBABILITY`, `ADD_INSTRUCTION_PROBABILITY`, and `SWAP_INSTRUCTION_PROBABILITY` define the probabilities that an instruction is deleted, added, or swapped respectively when an instruction is mutated. The `MUTATE_INSTRUCTION_PROBABILITY` defines the probability that an instruction is mutated. Likewise, parameters `ADD_PROGRAM_PROBABILITY`,

DELETE_PROGRAM_PROBABILITY, NEW_PROGRAM_PROBABILITY, and MUTATE_PROGRAM_PROBABILITY do the same at the program level. The parameter TEAM_POINTER_PROBABILITY defines the probability that a team will reference another team during a round of mutation. The parameters MAX_INSTRUCTION_COUNT, MAX_INITIAL_TEAM_SIZE, and MAX_PROGRAM_COUNT constrain the size of Tangled Program Graphs by setting a cap on the number of instructions, capping the number of programs assigned to a team upon creation, and the maximum number of programs assigned to a team respectively. The DIVERSITY_CACHE_SIZE defines the number of observations to retain to generate diversity profiles for. The RAMPANT_MUTATION_INTERVAL parameter specifies how frequently rampant mutation occurs and the RAMPANT_MUTATION_COUNT parameter defines how many mutations are applied during a round of rampant mutation.

4.2 CartPole-v1

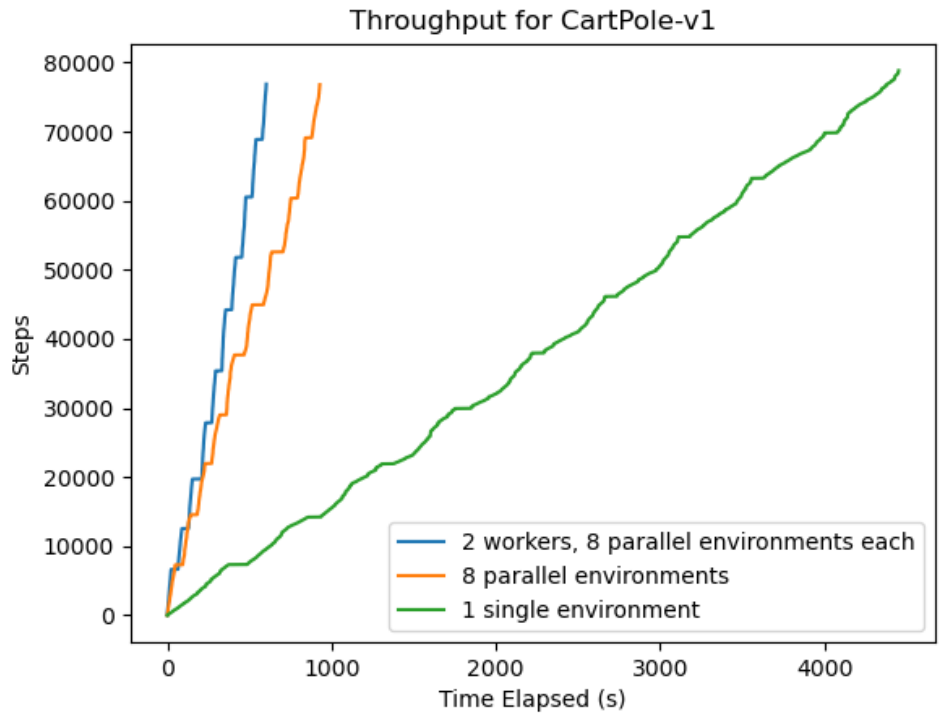

Figure 4.1: Throughput for CartPole-v1

As shown in Figure 4.1, a total of approximately 80,000 steps are performed by the agents, although this number varies since each run is independent of each other. In

the single-threaded implementation of TPG, 78,832 steps were performed in 4452 seconds. This is a throughput rate of 17.7 steps per second. In the multi-threaded implementation of TPG, 76,794 steps were performed in 928 seconds for a throughput rate of 82.8 steps per second. This is an overall throughput speed up of 4.67x over the single-threaded implementation. The multi-threaded and distributed implementation processed 76,866 steps in 602 seconds for a rate of 127.7 steps per second. Which is 1.54x higher than just the multi-threaded implementation alone and 7.2x higher than the single-threaded implementation. Given that the multi-threaded and distributed model is theoretically capable of performing 16x faster as 8 parallel environments are running simultaneously on 2 machines, the actual speed up of 7.2x underperforms. This can be explained both by network latency, where the overhead of coordinating work between two machines would slow down the training performance, and Amdahl's law where despite increasing the training performance the time to send the information to the database remains the same as the single-threaded implementation.

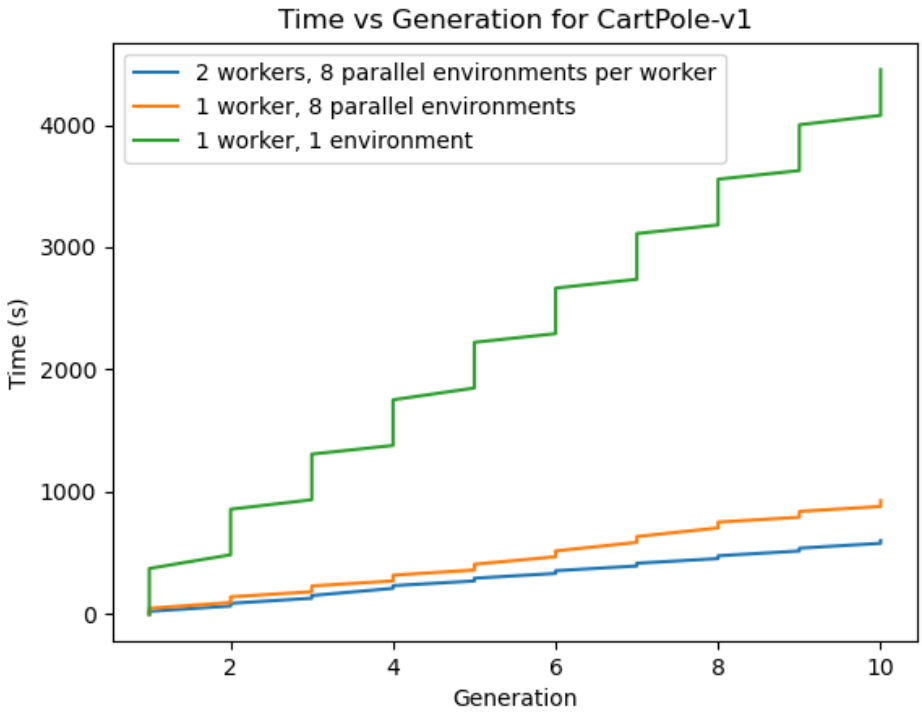

Figure 4.2: Time vs Generation for CartPole-v1

In Figure 4.2, each implementation is trained for 10 generations. The single-threaded implementation is capable of training 10 generations within 4452 seconds or 74 minutes. The multi-threaded implementation takes 928 seconds or 15 minutes to train 10 generations. This makes the multi-threaded implementation 4.8x faster to train than the single-threaded implementation. The distributed and multi-threaded implementation takes 602 seconds or 10 minutes to train 10 generations which is 1.5x faster than the multi-threaded implementation and 7.4x faster than the single-threaded implementation.

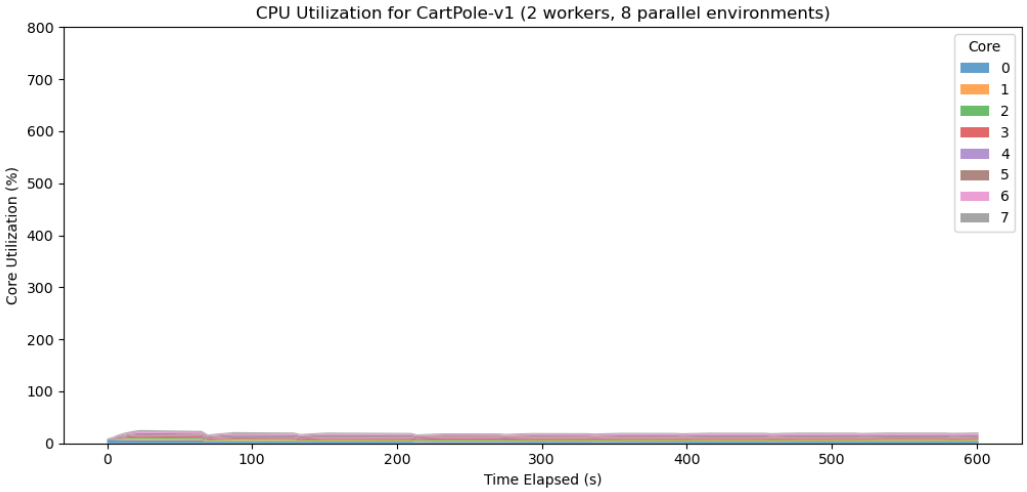

Figure 4.3: CPU Utilization of the first CPU of the 2 Distributed CPUs for CartPole-v1

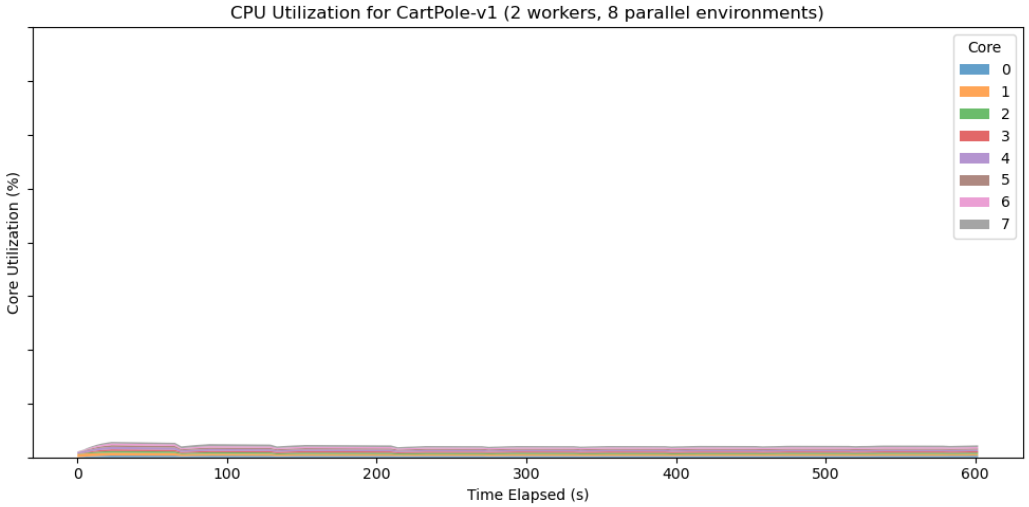

Figure 4.4: CPU Utilization of the second CPU of the 2 Distributed CPUs for CartPole-v1

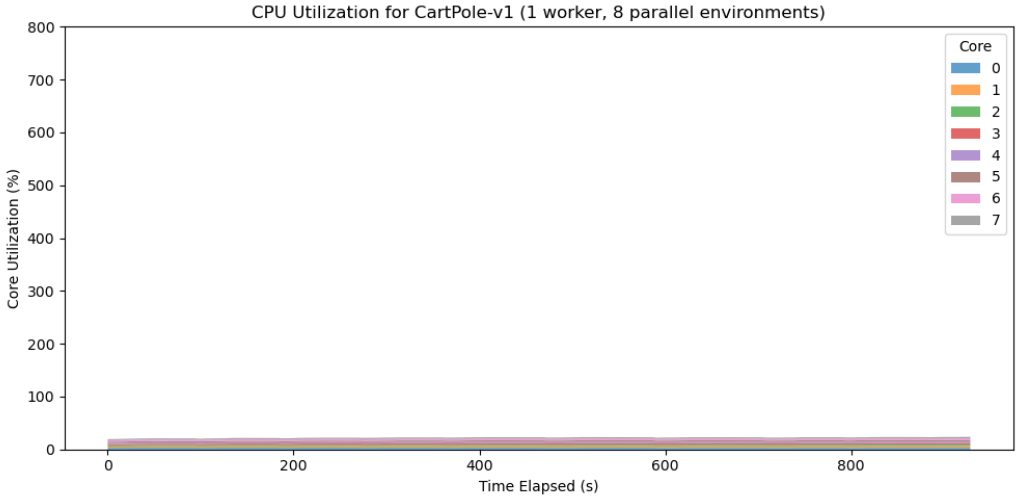

Figure 4.5: CPU Utilization of 1 CPU running 8 parallel environments for CartPole-v1

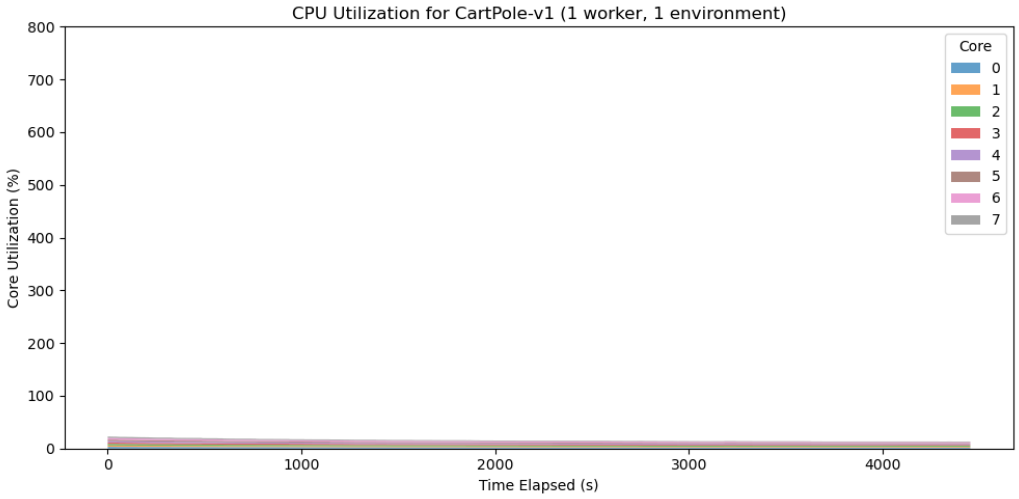

Figure 4.6: CPU Utilization of 1 CPU running 1 environment for CartPole-v1

In Figures 4.3, 4.4, 4.5, 4.6 we see that across all implementations the CPU is under-utilized. Because *CartPole-v1* has only four observations, it does not require a tremendous amount of CPU effort. This means that we can overcome the time inefficiency of running a single environment without placing any additional load on the computer, which demonstrates how inefficient the single-threaded implementation can be.

4.3 LunarLander-v2

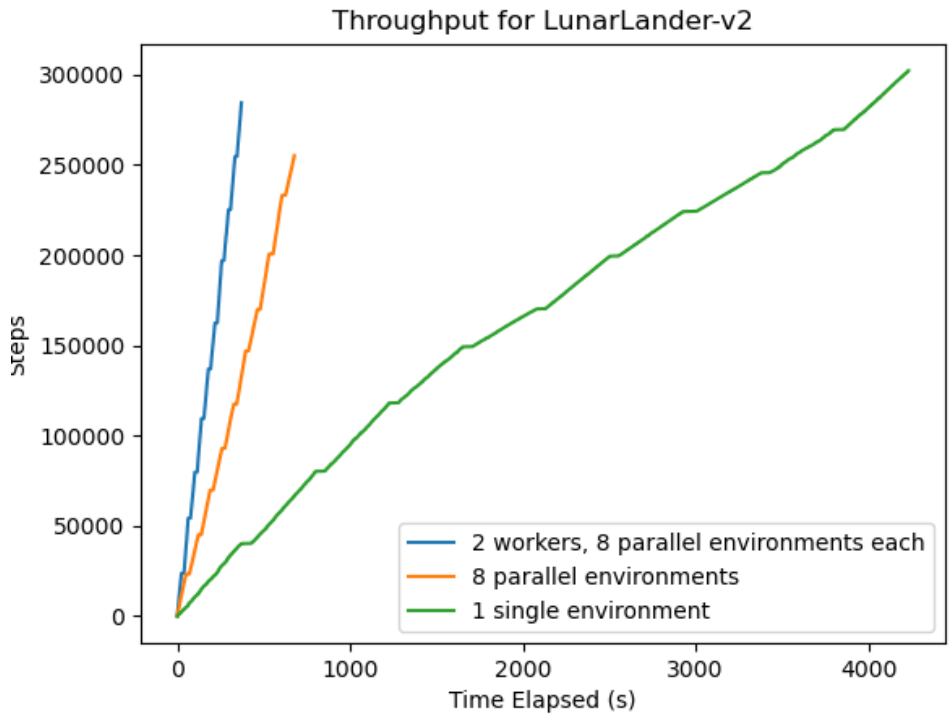

Figure 4.7: Throughput for LunarLander-v2

In Figure 4.7, the single-threaded implementation processes 302,192 steps in 4233 seconds. This is a throughput rate of 71.4 steps per second. The multi-threaded implementation processes 255,178 steps in 677 seconds for a throughput rate of 376.9 steps per second. Which is an overall rate increase of 5.3x that of the single-threaded implementation. Finally, the multi-threaded and distributed implementation processes 284,409 steps in 371 seconds for a rate of 766.6 steps per second. This means that the multi-threaded and distributed implementation has a speed up of 2.0x that of the multi-threaded implementation and 10.7x that of the single-threaded implementation.

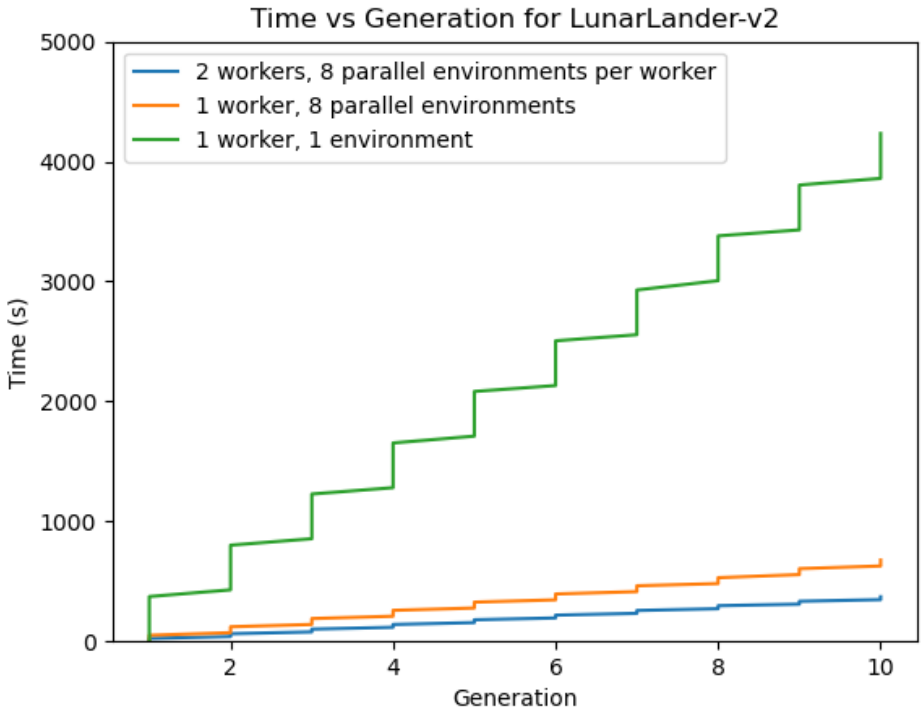

Figure 4.8: Time vs Generation for LunarLander-v2

In Figure 4.8, the single-threaded implementation completed 10 generations in 4233 seconds or 71 minutes. The multi-threaded implementation completed 10 generations in 677 seconds or 11 minutes. This is a speed up of 6.4x compared to the single-threaded implementation. The distributed and multi-threaded implementation completed 10 generations in 371 seconds or 6 minutes. This is a speed up 1.8x compared to the multi-threaded implementation and 11.4x compared to the single-threaded implementation.

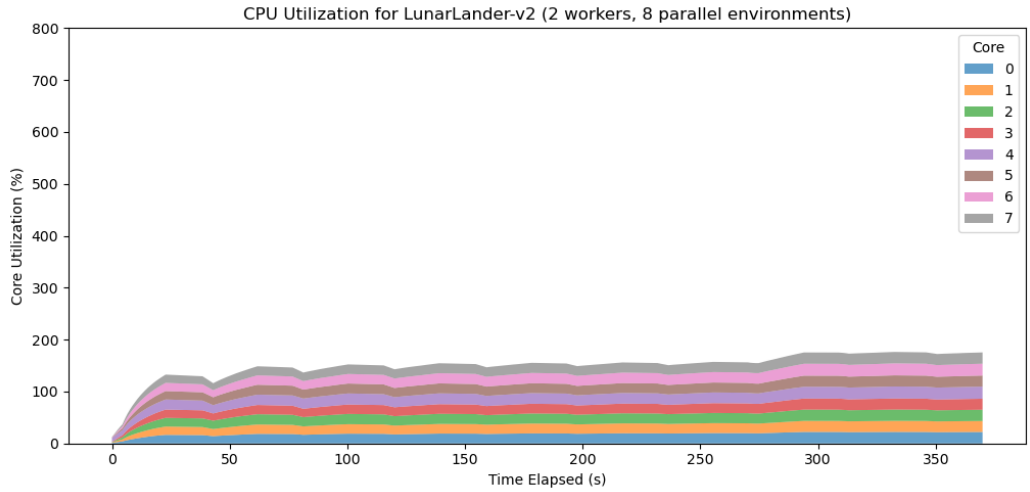

Figure 4.9: CPU Utilization of the first CPU of 2 Distributed CPUs for LunarLander-v2

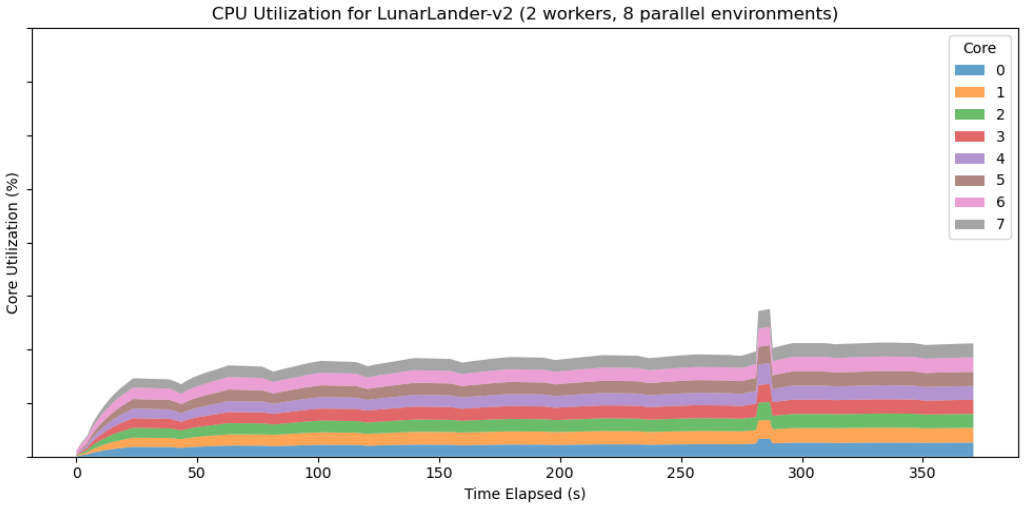

Figure 4.10: CPU Utilization of the second CPU of 2 Distributed CPUs for LunarLander-v2

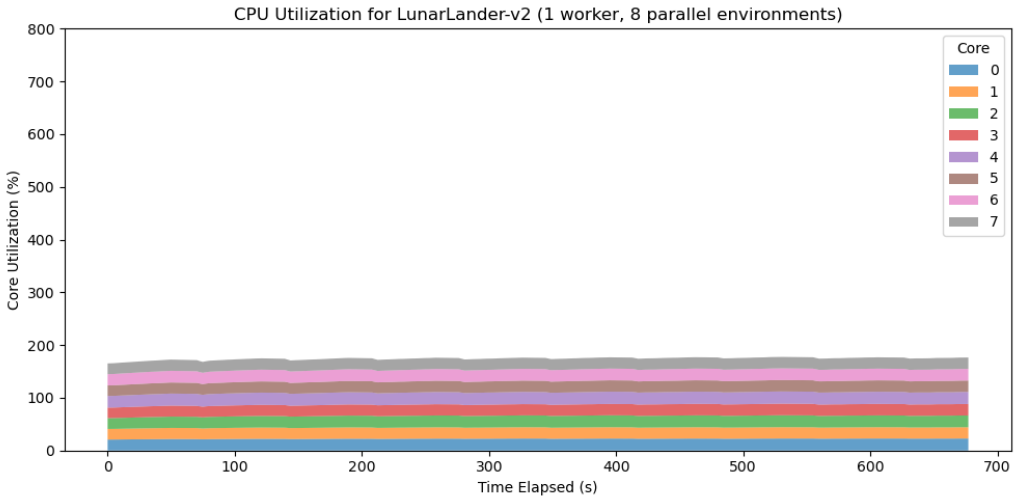

Figure 4.11: CPU Utilization of 1 CPU running 8 parallel environments for LunarLander-v2

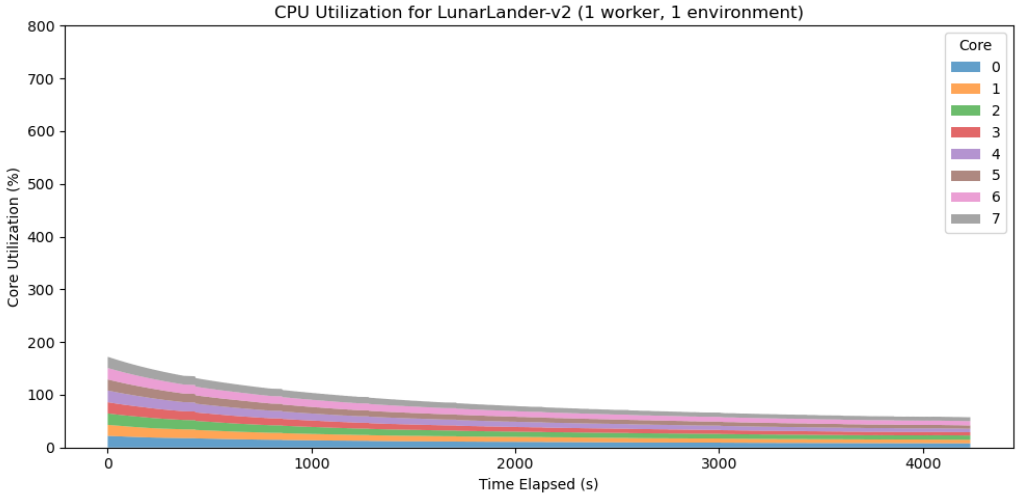

Figure 4.12: CPU Utilization of 1 CPU running 1 parallel environment for LunarLander-v2

In Figures 4.9, 4.10, 4.11, 4.12 we see that while all CPUs remain underutilized, the multi-threaded implementations utilize approximately twice the CPU as the single-threaded implementations demonstrating that parallelizing the environments is using the hardware more efficiently.

4.4 CarRacing-v2

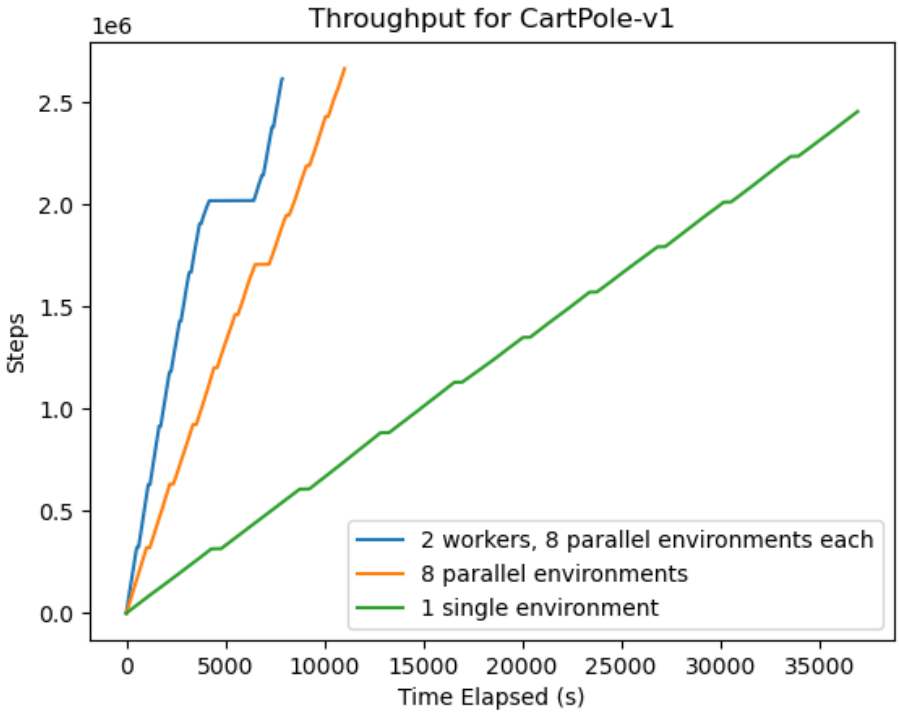

Figure 4.13: Throughput for CarRacing-v2

In Figure 4.13, the single-threaded implementation processes 2,453,637 steps in 36,950 seconds. That is a throughput rate of 66.4 steps per second. The multi-threaded implementation processes 2,663,340 steps in 11,027 seconds for a throughput rate of 241.5 steps per second. This is an increase of 3.6x from the single-threaded implementation. Finally, the distributed and multi-threaded implementation processes 2,613,253 steps in 7875 seconds. This is a throughput rate of 331.8 steps per second which is a speed up of 1.37x compared to the multi-threaded implementation and 5.0x that of the single-threaded implementation. This does; however, seem skewed by an event where the throughput dropped for some time and no steps were being processed possibly due to a network disruption during training.

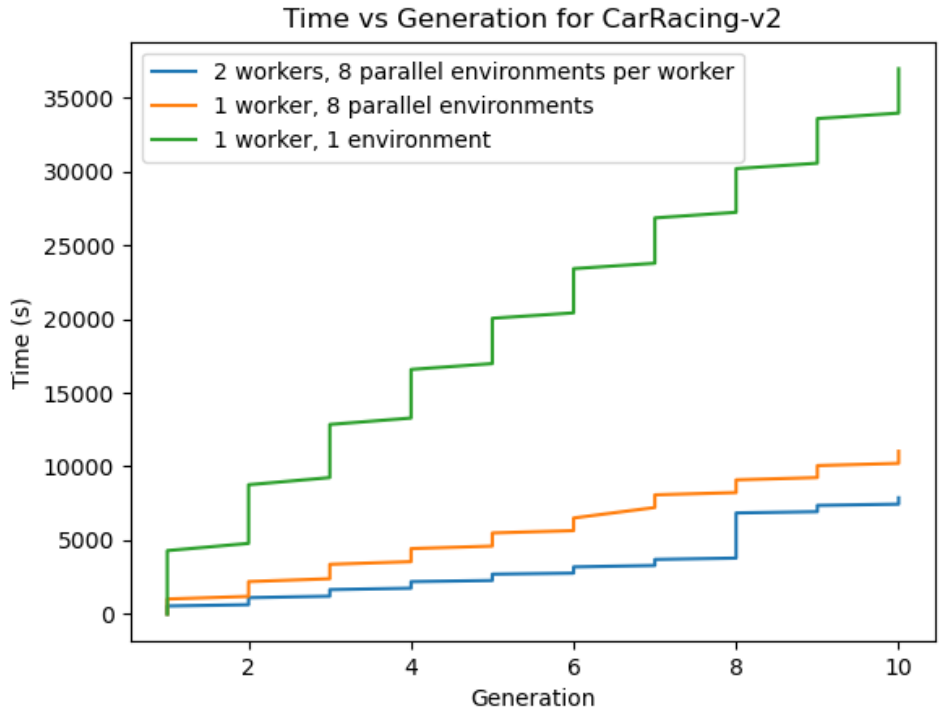

Figure 4.14: Time vs Generation for CarRacing-v2

In Figure 4.14, the single-threaded implementation completes 10 generations in 36,950 seconds. The multi-threaded implementation completes 10 generations in 11,027 seconds or 3 hours and 4 minutes. This is a speed-up of 3.4x. The distributed implementation completes 10 generations in 7875 seconds or 2 hours and 11 minutes. This is a speed-up of 1.4x to that of the multi-threaded implementation and 4.7x that of the single-threaded implementation.

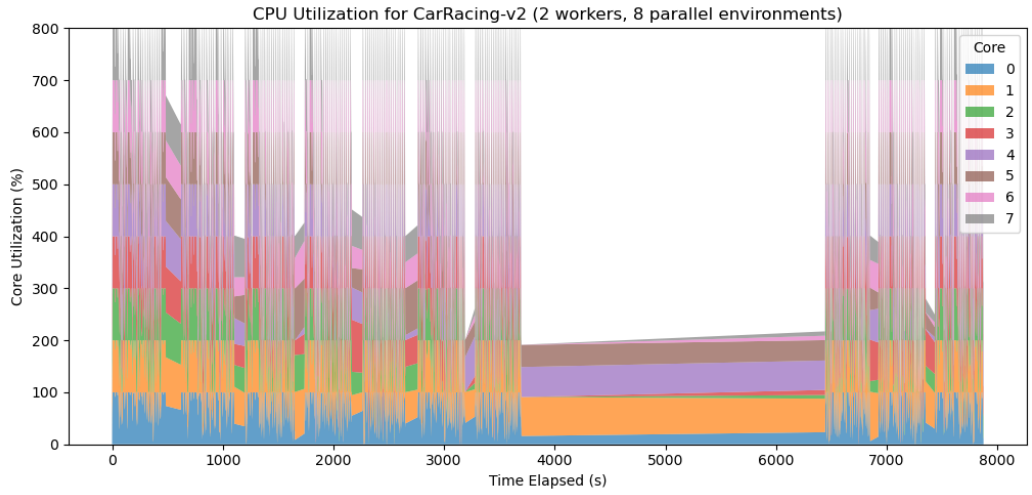

Figure 4.15: CPU Utilization of the first CPU of 2 Distributed CPUs for CarRacing-v2

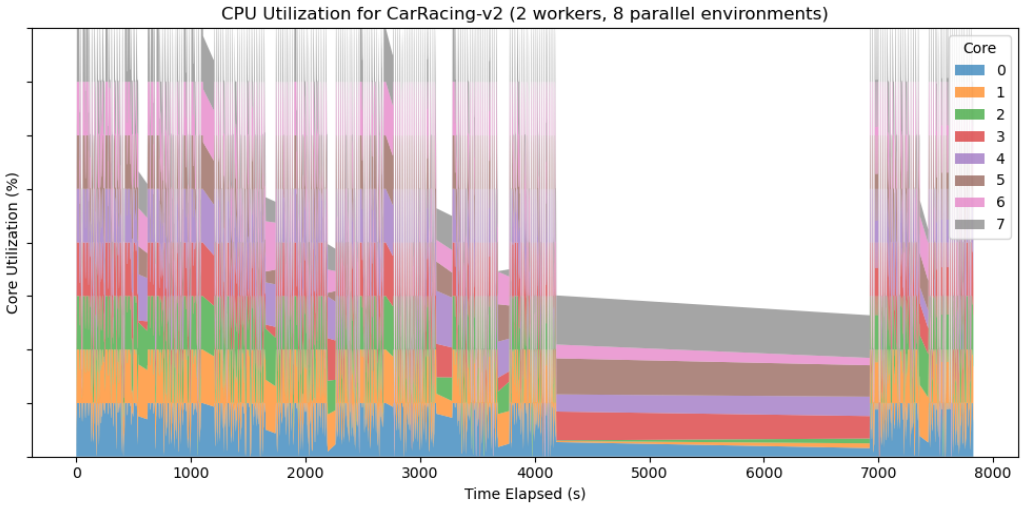

Figure 4.16: CPU Utilization of the second CPU of 2 Distributed CPUs for CarRacing-v2

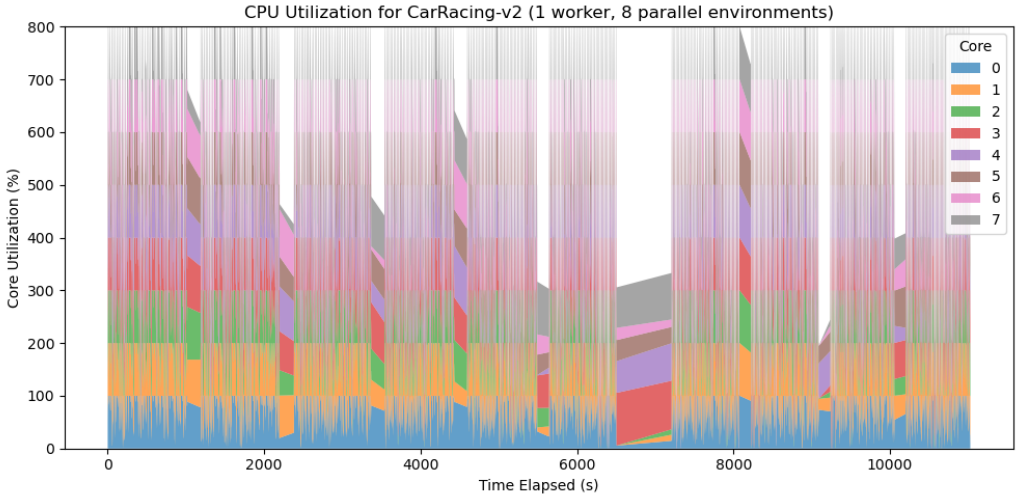

Figure 4.17: CPU Utilization of 1 CPU running 8 parallel environments for CarRacing-v2

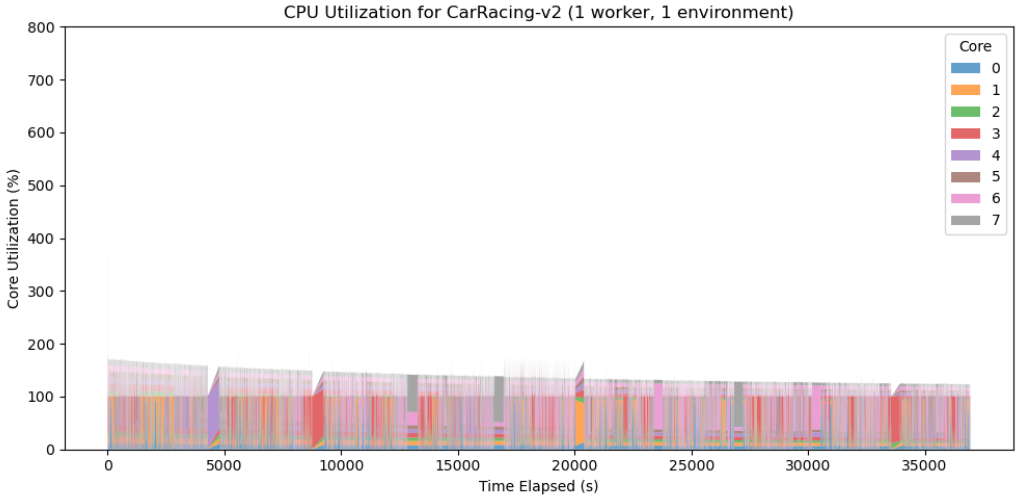

Figure 4.18: CPU Utilization of 1 CPU running 1 parallel environment for CarRacing-v2

In Figures 4.15, 4.16, we see that running 8 environments in parallel utilizes the entire CPU. This could explain why the parallelization was not as effective for *CarRacing-v2* as it was *LunarLander-v2*. Potentially, decreasing the batch size from 8 to 6 may show improvement if the batch size was too large for the CPU to handle.

In Figures 4.17, 4.18, we see that running only one environment does not effectively utilize the CPU while running 8 environments is possibly overloading the CPU.

Chapter 5

Conclusion

In conclusion, we've demonstrated that by running multiple environments simultaneously. We can significantly reduce the time required to train TPG models. We have shown that it is possible to increase the training throughput and benchmarked this by comparing the single-threaded and multi-threaded implementations using the Steps vs Time metric. We have also shown through the Time vs Generation metric that we significantly reduce the training time per generation. We also demonstrate through the comparison of the CPU utilization graphs, that we make more efficient use of our hardware leading to the previously mentioned improvements.

Beyond multi-threading, we show similar improvements by distributing the training among multiple CPUs. This allows us to horizontally scale the performance out. We show that doing so is capable of improving the training time by orders of magnitude.

5.1 Future Work

Future research could explore more environments than that of the three Gym environments tested in the experiments. These new environments could include the *VizDoom* [4] environment, the *Atari Learing Environment* [1] and also the *Mujoco* [7] environments.

In order to demonstrate, the true capability of the multi-threading that this implementation offers, future work in this area should be done to benchmark the performance on the model on better, more specialized hardware. By introducing a CPU with hundreds of cores, you could get much better performance than the 8-core CPUs used in this paper.

Other areas of research would include a focus on increasing the performance of the Tangled Program Graph model. Currently, all of the program execution is being done through the CPU. However, this program execution would be much faster on

specialized hardware such as a GPU. An implementation of the Tangled Program Graph model through CUDA would be a significant enhancement.

Finally, a significant limitation of the Tangled Program Graphs is its sample inefficiency. Tangled Program Graphs are an online machine-learning model which requires interaction with the environment for training. After each episode, all observation data is no longer used. By utilizing historical data to train the model without interacting with the environment directly, the bottleneck of interacting with the environment would be largely removed.

Chapter 6

Appendix

6.1 Source Code

The source code for `pytpg` is available at: https://github.com/bmcnns/distributed-tpg

Bibliography

[1] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[2] Stephen Kelly and Malcolm I. Heywood. Emergent tangled graph representations for atari game playing agents. In *Proceedings of the 20th European Conference on Genetic Programming*, pages 101–117. Springer, 2017.

[3] Stephen Kelly and Malcolm I. Heywood. Emergent solutions to high-dimensional multitask reinforcement learning. *Evol. Comput.*, 26(3), 2018.

[4] Michał Kempka, Marek Wydmuch, Grzegorz Runc, Jakub Toczek, and Wojciech Jaśkowski. Vizdoom: A doom-based ai research platform for visual reinforcement learning, 2016.

[5] Robert Smith and Malcolm Heywood. *Scaling Tangled Program Graphs to Visual Reinforcement Learning in ViZDoom*, pages 135–150. 03 2018.

[6] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[7] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033. IEEE, 2012.