Exploring Parallel strategies in Deep Reinforcement Learning

Tejonidhi Raghunath Deshpande
Graduate Student in Robotics
Georgia Institute of Technology
Atlanta, USA
tdeshpande33@gatech.edu

Moises Shalimay de Souza Andrade Sr.
Graduate Student in Computer Science
Georgia Institute of Technology
Atlanta, USA
mandrade@gatech.edu

Surya Prakash Senthil Kumar
Graduate Student in Robotics
Georgia Institute of Technology
Atlanta, USA
skumar671@gatech.edu

***Abstract*—Reinforcement Learning combined with Deep Learning (DRL) has emerged as a powerful paradigm. The exceptional performance of such models on tasks including gaming, robotic simulations, autonomous driving and multitude of other applications showcase their flexibility and promise to tackle complex real-world problems. However, training DRL models presents at least two significant challenges: (i) training times and (ii) effective exploration of the parameter search space. Efficient exploration of the parameter space is critical in achieving optimal performance, and is especially challenging in environments with sparse rewards, as in RL settings. These challenges call for solutions that allow reasonable learning with limited resources. Hence, in this work, we try investigating various parallel strategies to speed up training and effectively explore the parameter space to improve the baseline performance of DRL networks on a set of Atari games.**

***Index Terms*—Distributed training, Parallel computing, Deep Reinforcement Learning, Robotics**

I. Introduction and Problem Statement

Reinforcement Learning (RL) is an interdisciplinary area of machine learning and optimal control where an agent interacts with the environment and changes its current state by taking some actions. While doing so the agent receives a reward and the goal of the agent is to maximize the cumulative reward. There are two main types of RL techniques, model-based RL and model-free RL. Model based RL methods require the model of the environment and primarily rely on planning. Model-free RL methods are explicit trial-and-error learners and primarily rely on learning.

In DRL we use neural networks (usually fully connected) as function approximators to represent the optimal policy as well as the value function. Furthermore, due to cost and limited learned behavior from physical trials, the training experiments for an RL agent are usually carried out in a physics based simulator (e.g. Gazebo, Mujoco & Isaac Sim).

One of the main advantages of deep learning is that computation can be easily parallelized. In order to exploit this scalability, deep learning algorithms have made extensive use of hardware advances. A myriad of such techniques has been proposed for cheaper and faster training in the past. In the scope of this project, we consider Actor-Critic, a model-free, off-policy RL algorithm suitable for continuous state and action space with different parallelization strategies. We plan to investigate different levels of parallelism in a DRL model for some specific tasks. For instance, we can parallelize the model at an environment-level, multiple agents and environments are spawned in parallel, each one independently exploring different parts of the state-space. Another strategy could be to spawn multiple agents in a single environment, reducing the memory and computational requirements. The parallelism can also be achieved at network level by distributing multiple agent-environment instances across computing nodes, each receiving different hyper-parameter initialization.

Although RL training is implemented at much higher scale in industry, our plan was to get some insights into the effect of parallelism in training of RL models and hopefully serve as a starting point for future research in this direction. By conducting thorough experiments and analyses, our objective is to uncover effective parallel methods for accelerating RL training thereby reducing computational costs and improving their scalability. In this project, we successfully implemented data- level parallelism, where multiple compute nodes are used to train the policy represented by the neural network, network- level parallelism where multiple agent-environment instances were distriubuted across computing nodes (CPUs and GPUs) and a slightly advanced population based training PBT variant of network level parallelism, where the worst performing agents are replaced by the best performing agents. Our implementation for respective parallization strategies can be found at following github links:-
PAAC:-$https://github.gatech.edu/tdeshpande33/PAAC$
DNNP:- $https://github.gatech.edu/tdeshpande33/DDNP$
PBT:- $https://github.gatech.edu/tdeshpande33/PBT\_Atari$

Section II describes a brief literature review of the previous work done in the domain of Reinforcement learning and various parallelization strategies used. Section III gives a brief background about the project with section IV discussing the impacts and contributions of our work. Section V describes the results obtained and comparative evaluations of different parallelization strategies. Section VI gives conclusion with section VII stating the contributions from each team member to the project.

II. Relevant Work/ Backgound

A. Parallelizing Artificial Neural Networks

Several techniques have been proposed for facilitating a cost-effective training regime for deep neural networks. This include model and data-level parallelism which have been proved to accelerate training times and enhance computational efficiency, as reviewed by Tal et al. [2].

DistBelief [4] is a framework that, utilizes numerous machines to enable parallelized training of large models. This parallelism is achieved through the implementation of Downpour SGD, an asynchronous version of vanilla Stochastic Gradient Descent, which computes gradients and updates the parameters across model replicas distributed among multiple compute clusters. The authors have achieved state-of-the-art performance on networks with over a billion parameters while minimizing overall training times.

Alfredo et al. [3] designed a framework for effectively parallelizing networks on GPU with synchronous SGD and eliminated the need for multiple replicas of the parameters across the cluster. The proposed approach is generalized to both on-policy and off-policy algorithms and demonstrated state-of-the-art performance on different tasks in the Atari environment. Another work [6] on neural network training, introduces a *population-based training* scheme wherein the best-performing model is discovered by parallelizing the hyperparameter search across distributed clusters.

B. Deep Reinforcement Learning

In addition to the previous literature on model and data-level parallelism in deep neural networks, several methods have been proposed for distributing the training of RL models. Gorila [7] employs parallel instantiation of environments for multiple agent processes to gather individual experiences, which are then aggregated in a shared experience buffer. Each learner process maintains a replica of the policy to monitor changes in its parameter subset and periodically communicates these changes to the shared parameter server. This approach has demonstrated superior performance over traditional sequential DQN (Deep Q-Network) in various ATARI games, achieving a 2x training speedup. Shixiang Gu et al. [5] have also shown reductions in training durations by asynchronously parallelizing the RL algorithm across multiple robots, which eventually pool their policy updates.

C. Population Based Approach

Jaderberg et al. introduced another approach called population based training (PBT). Apart from being just asynchronous and parallel, PBT takes into consideration the performance of model under training. Importantly, it discovers a schedule of hyper-parameters rather than just following a sub-optimal strategy of trying to find a single fixed set to use for the whole course of training. The worst performing models are replaced by best performing models during training. The authors also show that, PBT achieves faster wall-clock convergence.

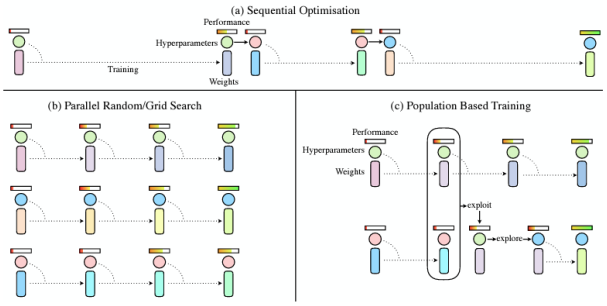

Fig. 1. Population based training proposed by [6] and comparison to Sequential and Parallel Approach

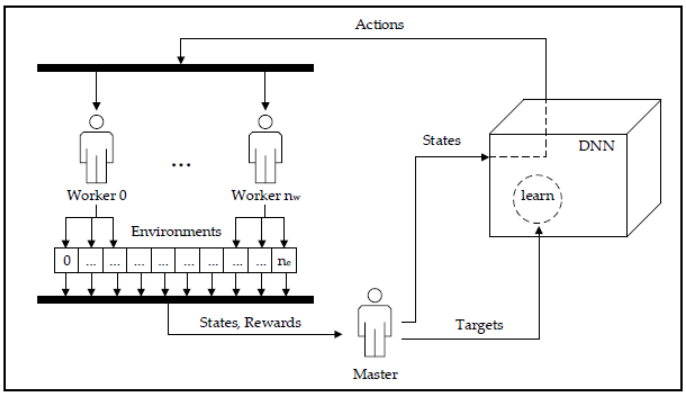

Fig. 2. Parallel architecture proposed by [3]

III. Proposed Approach

The baseline setting of our experiments is founded on [3]. Fig. 2 summarizes their proposed architecture, which in this work we call a *cell*. In summary, this setting supports multiple workers interacting with the environment in parallel and they communicate their changes to the master node that governs the global updation of the critic and the value network. The updated policy is then broadcasted among the workers thereby closing the loop. We will experiment with two augmentations to this architecture, both separately and potentially in combination.

A. DNN-level Parallelism

In this approach, we parallelize the computations of DNN responsible for learning the policy and value functions, as in Fig. 3. There are at least three strategies to realize this: (i) data-level parallelism, and model-level parallelism with (ii) intra-layer parallelism, and/or (iii) inter-layer parallelism. Data-level parallelism splits the network across multiple workers, each holding a copy of the model weights. Inter-layer parallelism assigns different network layers among workers, while intra-layer parallelism divides the tasks of a single layer among several workers. While model-level strategies presents a smaller memory footprint, they have a higher communication overhead

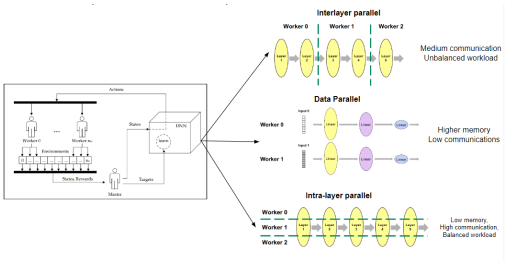

Fig. 3. DNN Parallelism vs Single-GPU Baseline

	16 Env	32 Env	64 Env
	Steps/Second		
Baseline	530	565	521
Data Parallel	1076	1417	1596
Speedup	***2.03x***	***2.51x***	***2.96x***
	Average Reward		
Baseline	94.4	97.2	95.3
Data Parallel	93.4	98.2	98.2

* Steps/Second: overall average at step ~63M

*Average Reward: exponential moving average at step ~63M

Fig. 4. DNN Parallelism vs Single-GPU Baseline

in comparison to data-level strategies, making the latter more suited for smaller Neural Networks.

For this project, we utilized Data-Level parallelism due to the relative small size of the of the DNN, which consists of a CNN with four layers. Nonetheless, our implementation utilizes Pytorch's DDP mechanism, and is easily extendable for parallelization at the model level and allows for training in multi-GPU multi-nodes environments.

More specifically, worker-environment block pairs are divided evenly among the GPU devices and at the end of each episode their gradient is synchronized before the model is updated.

Due to resource constraints associated to PACE, we only ran the experiments using 2 GPUs in the same node; nonetheless, it was sufficient to verify the correctness and improvements of our method relative to the baseline implementation.

Figure 4 shows the results of our experiments considering the first 63 Million steps (approximately 10 hours of training for the single-GPU case)[1]. For a small number of environments we achieve a speedup of approximately 2x, increasing to 3x as the number of environments grows. This increase in speedup can be attributed to the fact that, with fewer environments, there are fewer parallel operations to exploit. As a result, there is a smaller offset of the higher communication overheads, leading to smaller gains relative to the single-GPU case when more environments are considered. Finally, both methods yield similar rewards, demonstrating that our approach successfully accelerated training without compromising model performance.

B. Choice of programming language

We utilized PACE machines with CUDA GPUs for our experiments. For the programming language, we mostly utilized Python and Pytorch with occasional references from [6] and [3].

IV. Project Contribution

In this project, we try to contribute via carrying out a comparative analysis of state-of-the-art parallelization strategies. For the benefits parallelization offers to the field of reinforcement learning, we study different parallelization strategies and their strengths and weakness via thorough experimentation. We come to a conclusion that the most basic advantage actor critic (A2C) algorithm, a synchronous version of asynchronous advantage actor critic(A3C) is suitable for environments where limited computational resources are available or running asynchronous processes is not possible/feasible.

The parallel advantage actor critic algorithm (PAAC) is another asynchronous variant of A3C, well suited for training deep RL agents in parallel across multiple CPU cores.It leverages parallelism to accelerate training by running multiple copies of the agent in parallel, each collecting experiences and updating the model independently. Use of PAAC is recommended when the RL trask under training environments with high-dimensional state and action spaces, where collecting experiences from agent-environment interaction is particularly expensive. It finds its uitility the most in scenareos when distributed computaitonal resources such as clusters are available.

Population Based training (PBT) is a meta-algorithm (i.e. it decides how to combine two or more algorithms) that combines reinforcement learning and evolution algorithms. It dynamically adjusts the hyper-parameters and shares knowledge across multiple agents to accelerate learning. It is specifically useful when hyper-parameter tuning is crucial for achieving good performance.

V. Evaluation Plan and Datasets

We evaluated our implementation based on the simulation results on the proposed parallel strategies. As far as our dataset is concerned, we collated agent-environment experience tuples from the open-sourced implementation of multiple Atari environments by Gymnasium. To keep the implementations simple and results consistent we finalized our environment to be the Breakout Atari environment provided by Gym. But to test our hypothesis we mainly used runtime (model convergence) and scalability as an evaluation metric. We measured how our implementation scales as we increase the number of environments (in distributed implementation case this corresponded to increasing the number of machines used for computation) and compare the scores and runtime of our method using the framework mentioned in [3] as a baseline. In summary, our goal was to achieve a better runtime and scalability than the baseline approach.

[1]We selected this stage because, due to constraints with PACE, we could not run all the experiments for the same number of hours/steps, and this is a common stage for all the experiments considered in the table. Nonetheless, as 5 shows, the model is well-performing and stable and at this stage.

A. Multi-threaded parallelism

1) Description: Firstly, we started our analysis with the implementation of PAAC [3] abbreviation for Parallel Advantage Actor Critic, on the Breakout-v4 Atari environment [1]. The implementation of PAAC involves training of a single model loaded on the GPU that is *shared* across multiple concurrent CPU threads. Each CPU thread can be considered a *learner* n_w which in turn contains n_a number of actors (RL environment instances) and hence the total number of environments instances, n_e would be $n_w \times n_a$. This is a variant of data-level parallelism in which each *worker* or *actor* thread is responsible for the collection of experiences from their respective environments and passing it on to the master for training the model as shown in Fig.2. We use a *fork-server* framework for instantiating learner threads along with a data structure to queue the experience tuples from the actors within. Synchronisation is done at the end of one step in each environment by placing locks and eventually concatenating the experiences gathered to train the model loaded on the master thread. To preserve flexibility in the design, we developed the entire framework in PyTorch and used its multiprocessing module to set up the fork-server for parallel training. Our implementation closely follows the structure outlined in the provided code, yet it has been entirely replicated using PyTorch.

Algorithm 1: Parallel Advantage Actor Critic (PAAC)

Initialise timestep counter $N = 0$ and network with weights θ, θ_v in the master thread.
Initialise shared containers R_s, M_s, S_s and setup a set e of n_e environments in a fork-server framework.
while $N \leq N_{max}$ **do**
  for $t = 1$ **to** t_{max} **do**
    Sample a_t from $\pi(a_t|s_t;\theta)$;
    Compute v_t from $V(s_t;\theta_v)$;
    parallel for $i = 1$ **to** n_e **do**
      Execute action $a_{t,i}$ in environment e_i
      Collect new state $s_{t+1,i}$, reward $r_{t+1,i}$ and mask $m_{t+1,i}$ in shared containers.
    end
    mask is an indicator function 1, denoting the end of an episode
  end
  $R_{tmax+1} = \begin{cases} 0 & \text{if terminal } s_t \\ V(s_{tmax+1};0) & \text{otherwise} \end{cases}$
  for $t = t_{max}$ **to** 1 **do**
    $R_t = r_t + \gamma R_{t+1}$
  end
  Compute $d\theta, d\theta_v$. Update parameters θ & θ_v.
  $N \leftarrow N + n_e \cdot t_{max}$;
end

2) Experimentation and Results: We used the PACE Phoenix Cluster for our experimentation and we chose our network hyperparameter configuration as in [3]. We tested our parallel implementation with $n_w = 8$ learners while varying the number of actors in the range of $n_a \in \{2, 4, 8\}$. NVIDIA A100 GPU was used for training the model and Intel Xeon Gold 6226 CPU @2.70 GHz with 64 threads was requested for data collection from parallel environments out of which only 12 learner threads were used. N_{max} was fixed as 80M with a wall time of 15 hours for training and episode duration of 10000 steps. The number of steps t_{max} before pushing an update to the model was fixed as 5.

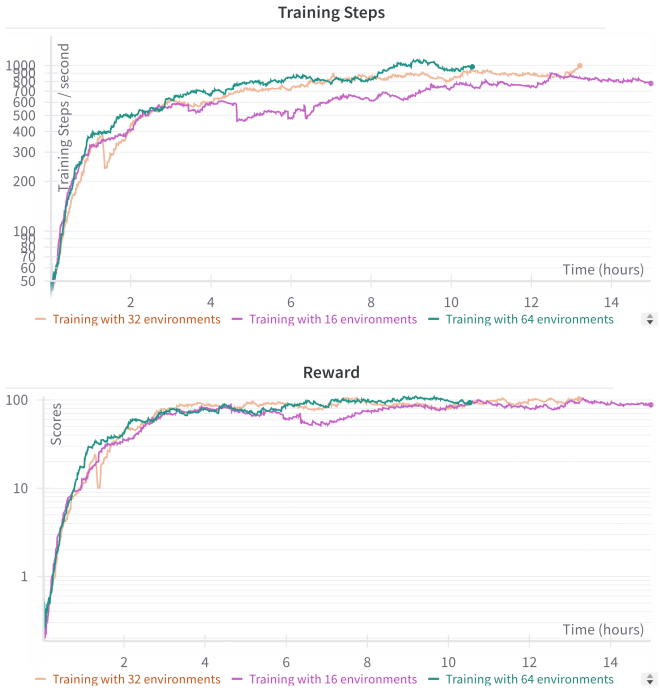

Fig. 5. Results describing the training steps/s and rewards accumulated across environments

From the 5, we can infer that the model with 64 environments (env) reaches the maximum score much quicker than the other two versions. The time taken by 64 env version to reach a score of 100 was estimated to be around **7** hours, 32 env was around 9.5 hours & 16 env was around 12 hours and we obtain an average speedup of **1.3X**. With increase in the total number of environments, the number of steps performed per second increases as rightly indicated by the training steps plot.

B. Distributed parallelism

1) Description: To evaluate the effectiveness of PAAC across clusters, we employed distributed training, utilizing the Remote Procedure Call (RPC) framework. This approach facilitates model training across multiple processes in implementationa topology by providing mechanisms for remote communication. We extended the idea of PAAC to a distributed setting where we define two entities, *Agent* and *Observers*. Each observer process or the *callee* will contain an instance of the environment to run and will pass the state vector $(s_{t+1}, r_{t+1}, m_{t+1})$ to the agent when requested. The

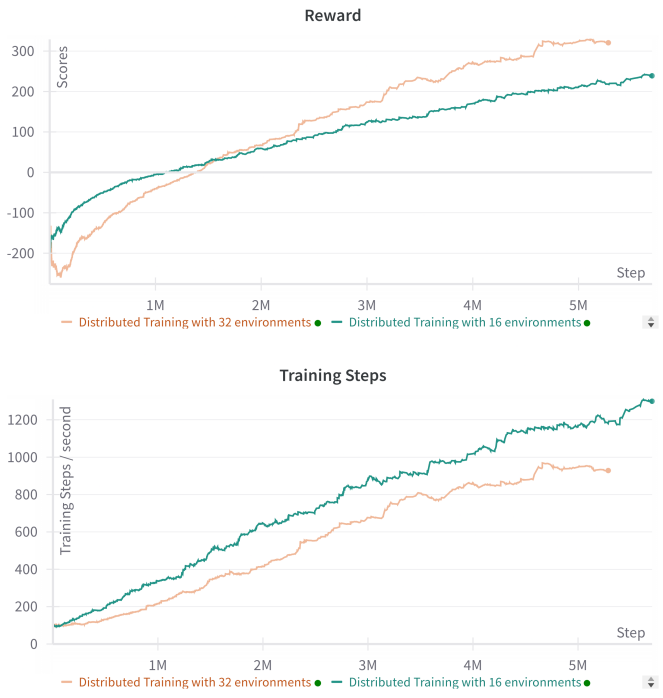

Fig. 6. Results describing the training steps/s and rewards accumulated across environments

Agent process or the *caller* on the other hand will hold the model that needs to be trained and will use the remote reference object from RPC to make calls to the remote machines. The framework for the distributed setting is shown in Fig. 7.

2) Experimentation and Results: We used the PACE Phoenix Cluster for our experimentation and we chose our network hyperparameter configuration as in [3]. Since we had some restrictions on the number of GPUs, we chose to train the model on CPU. The observations for the Breakout Atari environment are images and training a convolutional neural network on CPU is a tedious task and so we resorted to train a much simpler environment called the *LunarLander*. This is justified because our objective is to verify the effectiveness of PAAC in a distributed setting and thus our hypothesis is environment agnostic. We tested our distributed implementation with $n_e \in \{16, 32\}$ observers and we requested for 8 compute nodes with 8 tasks per node thereby handling 64 processes in total. The model is currently being trained for 80M with a wall time of 15 hours for training and episode duration of 10000 steps. However, we will attach the model performance till 7M steps.

From the 6, we can infer that the model trained by collecting experiences from 32 observers reaches the score of 200 within **4h 40m** whereas the time taken by 16 env counterpart to reach the same score was estimated to be around **7 hours**, giving a **1.5x** speedup. This proves that, distributing the training can help in faster training of an RL environment and also

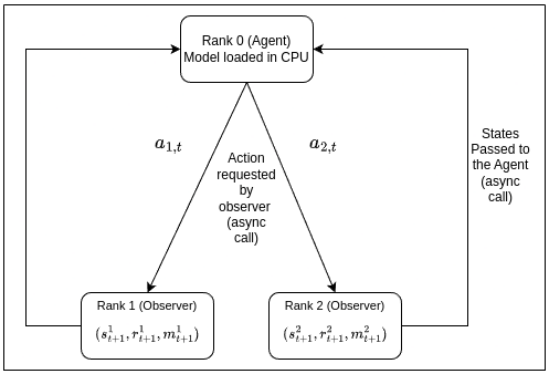

Fig. 7. Distributed PAAC

inherently encourages exploration.

Algorithm 2: Distributed PAAC

Initialise timestep counter $N = 0$ and network with weights θ, θ_v in the agent process (rank:0) by starting the RPC framework.
Initialise n_e observer process with rank:(1,n_e - 1).
Observer processes:
  Reset Environment and get the state s_t
  Request action by calling the agent along with s_t
  Receive action and step to observe s_{t+1}, r_{t+1}, m_{t+1}
  Record observations in the agent.
Agent process:
  Wait until all the observers have finished recording.
  Train the model with gathered data.

C. Population Based Training

1) Description: : As mentioned earlier in the introduction section, this is an meta variant of PAAC algorithm described in Multi-threaded parallelism section. In this section, although not totally bug free, but we were able to implement population-based-training algorithm in a GPU based environment, given that no such prior implementation exists. In this implementation, the worst x % of the workers (agents) are replaced by best x% of the agents. After the replacement, the hyper parameters for newly replaced workers are mutated by some factor. In our case, the worst 20% of the workers were replaced with best 20% ones. The bestness factor of a model was decided based on the cumulative reward it has generated over the episode.

Following the terminology, from [6], we say the model is exploiting its current knowledge when a poor performing model gets replaced by a better performing model and we say the model is exploring when the hyper-parameters are perturbed. Unlike other methods, which usually only exploit current knowledge, this techniques also considers exploration making it possible to explore previously un-explored parts of the state-action-space. In terms of evaluation and comparison to base-line, the model was able to achieve scalability where it was able to run multi-threaded agent-environment simulation (simulating multiple environments in parallel). The exploration-exploitation aspect of the project achieved a robust

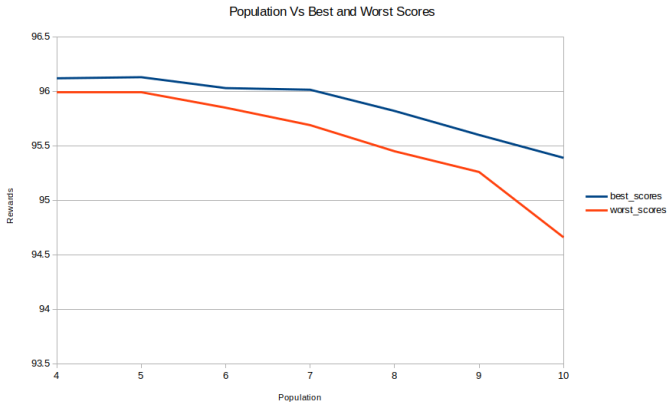

Fig. 8. Population Vs Best and Worst Scores

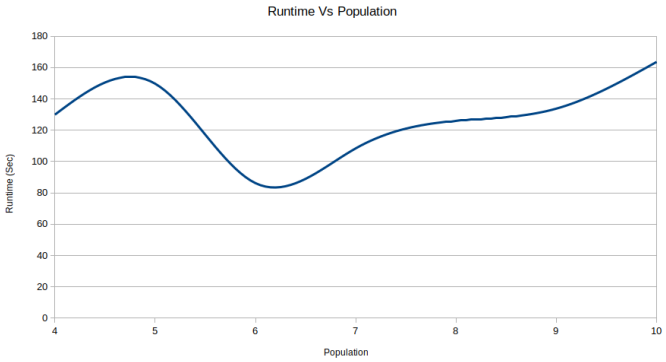

Fig. 9. Population Vs Runtime

policy learning with increase in population but increasing the population also resulted in slightly increased runtime. The policy model architecture used in this implementation is same as used in the PAAC.

2) Experimentation and Results: For experimentation, we used the PACE Phoenix Cluster for our experimentation and we chose our network hyperparameter configuration as in [6]. We tested our parallel implementation from a population of $n_p = 4$ to $n_p = 10$. NVIDIA A100 GPU was used for training the model and Intel Xeon Gold 6226 CPU @2.70 GHz with 24 threads was used. N_{train_max} was fixed as 500k with a wall time of 1 hour for training and episode duration of 10000 steps. We used PyTorch and its multiprocessing module to implement the PBT for Breakout Atari game.

From Fig. 8, we can infer that the overall best score of the model slightly decreases with population may be due the fact that he model/ policy is exploring more and more previously unexplored parts of the action-reward space (exploration). This is also evident from the explore-exploit trade off commonly discussed in RL literature. [8]. From Fig. 9, it can be also inferred that the runtime slightly increases with increase in the population. This can be attributed to the fact that with increase in population the GPU has now more models to simulate in parallel. In conclusion, even though the cumulative reward slightly decreases and runtime increases, the model is able to learn a robust policy over the period of time.

VI. Conclusion

In conclusion, we carried out a comparative analysis for various parallelization strategies namely multi-threaded PAAC, distributed PAAC, DNN level parallel and population based training methods for training an RL agent in Breakout Atari environment along with learning their strengths and weaknesses. Though we did not achieve a significant improvement in terms of speedup between the distributed and the multi-threaded version of PAAC, we can be certain that parallel training can bring down training run-times and encourage exploration of the state space. Further research can be done in the directions combining these parallel strategies together as they offer orthogonality and observe their effect on overall performance and speedup.

VII. Breakdown of Contributions

Tejonidhi Deshpande:- Carrying out initial experiments for PAAC, implementing PBT for Breakout Atari in PyTorch, carrying out experiments with PBT as mentioned in V-C, formatting and documenting the final report.

Moises Andrade:- Implementation of the DNN parallel strategy for PAAC training with DDP; carrying out experiments and comparisons to baseline as in III-A. Formatting and documenting the final report.

Surya Prakash:- Worked on the implementation of PAAC in PyTorch and conducted experiments for benchmarking purpose. Developed a *distributed* version of PAAC V-B and setup clusters for experiments alongside formatting and documenting the final report.

References

[1] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, June 2013.

[2] Tal Ben-Nun and Torsten Hoefler. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis, 2018.

[3] Alfredo V. Clemente, Humberto N. Castejón, and Arjun Chandra. Efficient parallel methods for deep reinforcement learning. 2017.

[4] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc' aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc Le, and Andrew Ng. Large scale distributed deep networks. In F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[5] Shixiang Gu, Ethan Holly, Timothy Lillicrap, and Sergey Levine. Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. 2016.

[6] Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M. Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, Chrisantha Fernando, and Koray Kavukcuoglu. Population based training of neural networks. 2017.

[7] Arun Nair, Praveen Srinivasan, Sam Blackwell, Cagdas Alcicek, Rory Fearon, Alessandro De Maria, Vedavyas Panneershelvam, Mustafa Suleyman, Charles Beattie, Stig Petersen, Shane Legg, Volodymyr Mnih, Koray Kavukcuoglu, and David Silver. Massively parallel methods for deep reinforcement learning. 2015.

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