Scalable Reinforcement Learning on Cray XC

Ananda V. Kommaraju, Kristyn J. Maschhoff, Michael F. Ringenburg, Benjamin Robbins
Cray Inc., 901 Fifth Avenue, Suite 1000
{akommaraju,kristyn,mikeri,brobbins}@cray.com

***Abstract*—Recent advancements in deep learning have made reinforcement learning (RL) applicable to a much broader range of decision making problems. However, the emergence of reinforcement learn workloads brings multiple challenges to system resource management. RL applications continuously train a deep learning or a machine learning model while interacting with uncertain simulation models. This new generation of AI applications imposes significant demands on system resources such as memory, storage, network, and compute.**

In this paper, we describe a typical RL application workflow, and introduce the Ray distributed execution framework developed at the UC Berkeley RISELab. Ray includes the RLlib library for executing distributed reinforcement learning applications. We describe a recipe for deploying the Ray execution framework on Cray XC systems, and demonstrate scaling of RLLib algorithms across multiple nodes of the system. We also explore performance characteristics across multiple CPU and GPU node types.

***Index Terms*—Reinforcement Learning; Cray XC Systems; High Performance Computing; Performance;**

I. Introduction

Recent advances in deep learning and reinforcement learning (RL) have revolutionized domains such as game playing, autonomous driving, and scientific discovery. For example, DeepMind's AlphaGO [1] and AlphaFold [2] demonstrate the applicability of RL to complex game playing and protein folding problems.

However, this new generation of AI applications imposes significant demands on system resources such as memory, storage, network, and compute. The emergence of this paradigm brings multiple challenges to system resource management as RL applications continuously train a deep learning or a machine learning model while interacting with uncertain simulation models. These simulation models are required to execute at high speed and low latency to provide inputs to the training models. Many of the RL algorithms are stateful models where the state information is transferred between training models and simulations. The stateful nature of the computation requires substantial data storage and high speed data transfers. In a multi-agent, multi-policy based models, the entire computation graph can be replicated and run in parallel while consuming multiple variants of state information. The need for low latency enforces dynamic execution of the task graph by scheduling tasks as soon as the dependent tasks are completed. Given the wide range of system requirements, we showcase that these workloads are suitable candidates to leverage Cray technologies. These requirements motivate us to enable RL applications on Cray systems and conduct a study into the system resource utilization of these workloads.

The UC Berkeley RISELabs Ray framework [3] is an active open source Python library that provides a distributed framework to develop concurrent applications that are dynamic, fine grained, and heterogeneous. Ray also provides fault-tolerance and an in-memory object store for dynamic execution of task graphs. The RISELabs RLlib [4] is a library built on top of Ray, which provides an abstraction to RL application entities such as simulation environments, actors, state, and agents. The combination of these two frameworks provides a rich environment to develop and run RL applications. In addition to abstraction support for actors, state, and agents, RLlib and Ray provide multiple techniques for policy optimization to enable distributed training across a large set of experiences.

In the first part of the paper, we discuss distributed reinforcement learning. We showcase how to deploy the Ray execution framework on Cray XC systems, and demonstrate scaling of the framework across multiple nodes of the system. Next, we demonstrate state of the art RLlIb algorithms on Cray XC systems using the Ray execution framework. We demonstrate the performance characteristics of these applications on single node and multi-node CPU and GPU architectures. We then provide a comparative analysis of RL algorithms running on Cray XC single GPU nodes and Cray CS Storm dense GPU nodes. For both platforms, we discuss the various configurations used to evaluate these algorithms to exploit parallelism at different levels.

II. Distributed Reinforcement Learning

A typical RL application consists of three entities: a simulation environment, a training module (*policy optimizer*), and a serving module (*policy evaluator*) which are part of an agent. Figure 1 shows a typical RL application. The simulation environment operates on actions and generates new state information in the form of new data points. These environments are based on a Markov Decision Problem (MDP). The training module performs policy improvement using a gradient descent algorithm with the state and reward data generated by the simulation environment at each step. The optimization evaluates a value function for state and actions generated from that state. This process trains a policy to predict the actions for a specific state which can achieve a maximum final reward. The serving module acts as an evaluator of the trained model and provides feedback to the simulation environment in the form of actions. At every step of this entire loop, these modules perform a reward based optimization to achieve a final optimal outcome.

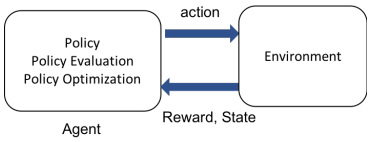


Fig. 1. A Reinforcement learning loop

The state space of these environments is extremely large, and computationally expensive for an agent to explore. In a single node multi-CPU or a multi-GPU setting such environments can be explored in parallel by multiple actors. RLlib provides a framework to explore these environments using Ray actors, launching multiple instances of these environments in workers running in parallel. Each of the workers collects experiences with a specific policy graph, consisting of a neural network model and a loss function. The workers use their current policy graph to decide which action to take in the current environment. The workers' experiences are gathered and evaluated with a loss function. In some training algorithms, the loss values are computed locally by the workers and communicated to a centralized agent. In others, the experiences (actions and rewards) are instead transmitted to the central agent, which evaluates the loss function itself. The centralized agent then performs a gradient descent optimization based on the actions and loss values to train a global policy graph. The new policy graph is then distributed to the workers, and the process repeats until convergence.

RL algorithms leverage parallelism at multiple levels. These algorithms involve deep nesting of distinct components where these components exhibit opportunities for distributed computation. For example, multiple simulation models or environments with perturbed state variations may run in parallel to cover a wide range of state spaces. Multiple workers running in parallel can explore a subset of these environments using a trained policy. Further, the trained policy or a target policy can employ distributed deep learning.

State of the art techniques like Deep Q Networks (DQN) [5], Importance Weighted Actor-Learner Architecture (IMPALA) [6], Ape-X [7], Proximal Policy Optimizer (PPO) [8], Advanced Actor Critic (A2C), and Asynchronous Advanced Actor Critic (A3C) [9] have different architectures and data flows. We discuss some of these techniques in this paper in relation to their scalability on Cray XC systems. RLlib captures the right abstractions and patterns of these multiple data flow architectures. These techniques employ a variety of communication patterns to update target policies by launching multiple actors. In some cases, like IMPALA, multiple learners train multiple target policies. In techniques like DQN or any of its variants, the algorithms utilize store and replay of experiences in order to train a centralized global policy. The replay buffers are sampled by a centralized learner. These complex and multiple different ways of training agents can be challenging in terms of availability of system resources. In this paper, we map these abstractions provided by RLlib and Ray

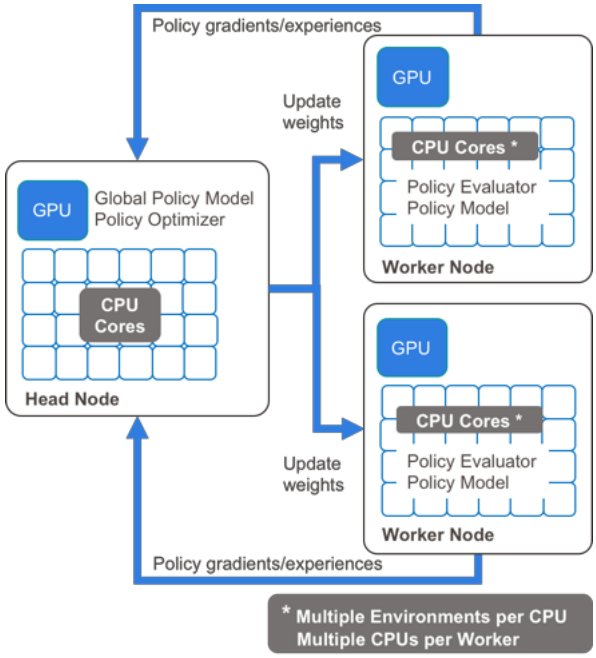


Fig. 2. A Reinforcement learning loop mapped onto an XC system

onto a Cray XC system and explore different RL algorithms to understand their resource usage and performance.

For example, consider a scenario where a PPO agent is being trained to learn to play the game Pong on a single node of an XC system. The target policy of PPO agents is trained though online learning without replaying experiences. Hence during training a continuous stream of data is transferred between workers and the driver. Multiple workers with a single instance or multiple instances of the target environment can be launched in parallel across the available CPUs on a single node. These workers perform policy evaluation and communicate the resultant states and rewards to a centralized agent. The centralized agent's policy optimizer can utilize the rest of the CPUs or potentially a GPU to perform policy optimization. Such training can be extended to multiple nodes by collating all the resources that are available across the nodes.

A. *Reinforcement Learning - A HPC workload*

RL algorithms like PPO or A3C perform policy evaluation through multiple SGD passes over sampled data and perform gradient optimization on a central policy. By distributing the RL application across multiple nodes or on multiple GPUs of a single node, significant data transfer can cause bottlenecks leading to increase in training time due to all-reduce operation on the policy gradients or in some algorithms all-reduce operation on experiences. We have also observed that sample collection, sample distribution, and optimization incur communication overhead. Further, these agents model the decision making policies as deep neural networks. These neural network policies are compute and memory intensive.

In Figure 2 we show how an RL application can be mapped onto a three node XC allocation: one head node and two worker nodes. In this case, each XC node has a single GPU and multiple CPU cores. The head node will perform global policy optimization through variety of optimizers. The worker(s) nodes perform experience gathering and replay sampling through memory buffers. The number of workers can be easily scaled up enabling distributed experience gathering and hence distributed learning. In some algorithms, the experiences are relayed back directly to a centralized policy optimizer running on the head node. It can be noted that such operations are all-reduce operations and therefore the performance of these operations is dependent on the interconnect architecture. After training a central policy, the head node updates weights on each of the copies of the policies running on the workers. These operations are broadcast functions over the interconnect. RLlib provides vectorization of each worker by running multiple instances of environments in each of the worker. Thus with the availability of large number of CPU cores on a single node, we can assign multiple CPU cores per worker. The workers can leverage this additional level of parallel compute and launch multiple environments.

Cray XC systems support both CPU-only nodes and hybrid nodes with a single CPU-hosted NVIDIA GPU. Not all of the RL algorithms require or target multiple GPUs, for example Ape-X. These algorithms can launch the workers on nodes with all CPU cores. The variety of node architectures on an XC system can enable efficient resource utilization for RL algorithms.

This hierarchical distribution of tasks with the need of high-speed connectivity motivate us to consider RL applications are an important HPC workload.

III. Deploying a Ray cluster on the Cray XC platform

The RLLib library is built on top of Ray distributed execution framework. Ray provides a framework for scheduling workers, managing resources, parallel simulation of environments, and training policies. Ray consists of three major components: a global control store, an in-memory distributed object store, and a distributed scheduler. Most RL applications are developed as distributed applications. Moreover, these applications can be modelled as hierarchical parallel tasks. The hierarchical model of execution of RL applications comprising of agents, policy graphs, environments, and policy optimizer leverage Ray's distributed execution framework. In order to train RL agents in a distributed environment, the first step is to setup a Ray cluster. In this section we describe how to deploy a Ray cluster on the XC.

A. Preliminary Steps

Ray and RLlib are python libraries which are under active development. In our experiments, we have first created a conda environment named *ray*. All the required libraries, for example *ray*, *gym*, and *tensorflow* are then installed into this environment.

```
$conda create -n ray pip
$source activate ray
$pip install ray
$pip install gym
$pip install tensorflow
```

B. Head node

The first step in setting up a Ray cluster is to initialize a head node. A head node comprises of a single redis server or multiple redis servers. The RL algorithms utilize head node resources to perform global synchronization operations like policy optimization. The redis service helps in letting workers register and connect to the head node. On a SLURM-based scheduler, first we allocate a single node in Cluster Compatibility Mode (CCM). We recommend it to be allocated as a ccm_queue node in order to run interactive jobs on the node. The IP address of the head node is written to either an NFS location visible by other nodes or to lustre file system so that workers can utilize this information.

For example, the following commands are run on the XC login node of one of our internal development systems.

```
$salloc -N 1 -p ccm_queue -C P100 --gres=gpu --
    exclusive
$module load ccm
$ccmlogin -V
```

This will launch into shell running on the compute node. Once on the compute node, the following script can be executed to activate the conda environment and start up the Ray head node.

```
#!/bin/bash
source activate ray
IP=$(ip -oneline -family inet addr list ipogif0 \
| head --lines 1 | grep --perl-regexp \
--only-matching 'inet \K[\d.]+')
echo $IP:6380 > $HOME/ray_head_node

ray start --head --node-ip-address=$IP --redis-port
    =6380
```

The Ray cluster can also be initialized by providing number of CPU cores, GPUs, and amount of memory available to the workload. If no value is provided, Ray performs a node-level system query and the maximum available resources found are assumed to be available to the workload. The policy optimizers in RLLib are designed to leverage wide range of resource capabilities which include GPUs and high speed interconnect for all-reduce and broadcast operations. Although most of the small Atari game workloads that we experimented with were over-provisioned in terms of number of CPUs available, we believe the CPU-hosted GPU nodes on XC provide a reasonable setting to train Atari games.

C. Worker nodes

The second step is to start the worker nodes. The worker nodes can be allocated without CCM. The worker script which starts Ray on each of the worker and connect to the head node can be launched using srun. The workers nodes can also be initialized with the number of CPUs and GPUs. Hence

these nodes can be utilized by other workloads in parallel. In the RLLib application, the workers usually evaluate a policy, sample experiences, and can run multiple environments. Thus provisioning multiple CPUs and significant memory can enable better performance.

The following commands are run on the worker node with the script below. This will enable nodes to join a Ray cluster as worker nodes. The Ray cluster can be scaled according to the requirement without tearing down the cluster.

```
$salloc -N 1 -C P100 --gres=gpu --exclusive
$srun sh worker.sh
```

```
#!/bin/bash
source activate ray
HEAD_IP=$(head -n 1 ray_head_node)
ray start --redis-address $HEAD_IP
while [ 1 ];
do
        sleep 1
done
```

D. Connectivity across the nodes

On an XC system, the connectivity across the nodes is TCP/IP socket based over Aries interconnect. We use the TCP/IP address of the head node and workers use it to become members of a Ray cluster. Ray run-time comprises of a local scheduler on each node and a global scheduler on the head node. A object store is initialized on each node and also globally. The scheduler and object store are used by RLlib optimizers to send and receive the data between head node and worker nodes. The schedulers and object stores are used to manage different components like experience replay buffers, parameter servers, and broadcasting gradients.

E. Ray entities on XC

One of the main features of Ray is to execute python functions remotely and asynchronously. Ray introduces actors that are like remote services which are executed on different nodes in parallel. The actor objects, along with local scheduler, global scheduler and object store provide a right framework for RL applications. For example, the RL environments which are stateful are implemented as actors.

Here is ab example of running a Ray remote function on a two node XC system. The total number of CPUs are 72.

```
def f1():
    time.sleep(1)
@ray.remote
def f():
    time.sleep(1)
    return 1
start=time.time()
[f1() for _ in range(72)]
end=time.time()
print("Local task", end-start)
ray.init(redis_address="10.128.0.231:6380")
start=time.time()
results = ray.get([f.remote() for i in range(72)])
end=time.time()
print("Remote Ray task", end-start)
```

```
Local task 72.0748701095581
Remote Ray task 1.0508031845092773
```

The output is as shown below

It can be seem from the above code, that Ray enables execution of remote functions asynchronously on a 2 node, 72 CPU XC configuration.

F. Resource scaling using a workload manager

Ray provides auto-scaling capability by dynamically controlling the number of nodes used for training. During run-time, workers join the cluster without restarting the cluster. On XC, node allocation happens through a workload manager like Slurm or PBS. We explored a possibility of users allocating nodes and changing the number of workers or resources without tearing down the cluster. We trained a PPO agent playing Pong. We vary the number of workers every few time steps. The checkpoint state is used to initialize a new run with new set of workers. By combining the centralized scheduler capability of Ray and Slurm node allocation and launching new workers we can achieve resource scaling.

Figure 3 shows the trend of episode reward means of a PPO agent trained on the game of Atari Qbert for 10M time-steps. In our experiment, we set the number of workers to be 128 for first 2M time-steps, then for 5M time-steps the number of workers is set to be 64, and for the final 5M time-steps the number of workers is set to be 32. The other 3 plots indicate the number of workers not changed from 0M to 10M time-steps.

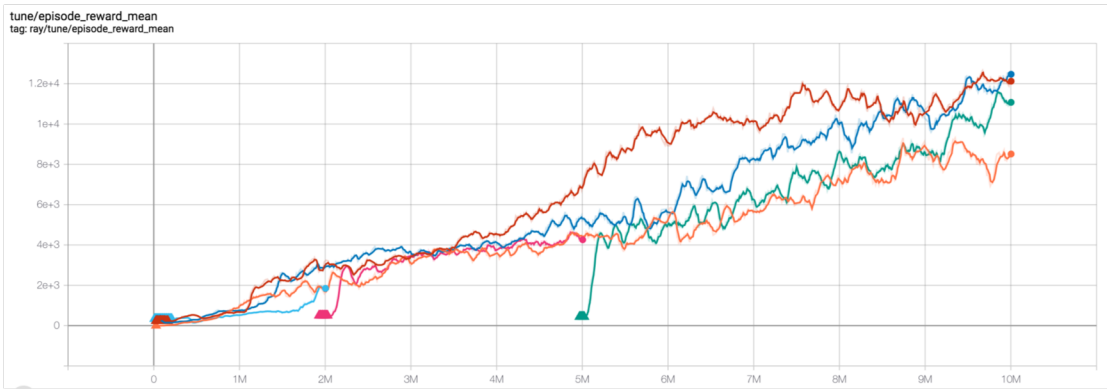
As it can be noted from the figure that the training time when varying the number of workers is 44m, 24m, and 15m for 32 worker phase, 64 worker phase, and 128 worker phase respectively. Hence the total time is 1 hour 24 min, which is close to the other non-varying runs. Also, the episode reward mean is close to a maximum value which was achieved by a complete run with 64 workers. Hence, it can be noted that Ray's efficient way of scaling resources an help achieve the an efficient way to achieve a good reward mean.

G. Resource selection for applications

In order to achieve optimal performance and resource allocation to RL applications, Ray provides ways to set the available CPU cores and GPUs during initialization of the Ray cluster. These settings enable partial allocation of resources to workloads. This provides a significant flexibility. For example some of the policy optimizers - Async, Shared Parameter server and AllReduce may not use all the GPUs available on the nodes. Similarly some of the policies like ApeX doesn't require all the GPUs on the worker nodes. The following section on launching RL agents will discuss the configuration that were used for this paper.

IV. Launching RL agents

Here is an example of the a training function using Ray, RLlib and OpenAI gym [10]. In this program, we created



Configuration	Reward Mean	Timestep	Walltime
0M_10M_128W	1.2125e+4	10M	1h 16m 15s
0M_10M_32W	8515	10M	1h 18m 57s
0M_10M_64W	1.247e+4	10M	1h 24m 13s
0M_2M_128W	1847	2M	15m 13s
2M_5M_64W	4260	5M	24m 59s
5M_10M_32W	1.1077e+4	10M	44m 45s

Fig. 3. Episode reward means of Qbert game by changing the number of workers during the training by scaling the number of nodes required. Each configuration indicates the range of time-steps for which the number of workers has been set

a PPO agent and trained to learn CartPole environment for 100000 time steps. The state is check-pointed. The Ray cluster is initialized with the head node redis server and hence the application will utilize all the available resources on the cluster, for example, here we specified 64 workers.

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import ray
from ray.rllib.agents.ppo import PPOAgent
from ray.tune import run_experiments


def train_fn(config, reporter):
    agent1 = PPOAgent(env="CartPole-v0", config=
        config)
    for _ in range(100000):
        result = agent1.train()
        result["phase"] = 1
        reporter(**result)
        phase1_time = result["timesteps_total"]
    state = agent1.save()
    agent1.stop()
if __name__ == "__main__":
    ray.init(redis_address="10.128.0.225:6380")
    run_experiments({
        "demo": {
            "run": train_fn,
            "local_dir": "/lus/scratch/user/
                ray_results/custom/",
            "config": {
                "lr": 0.01,
                "num_workers": 64,
            },
        },
    })
```

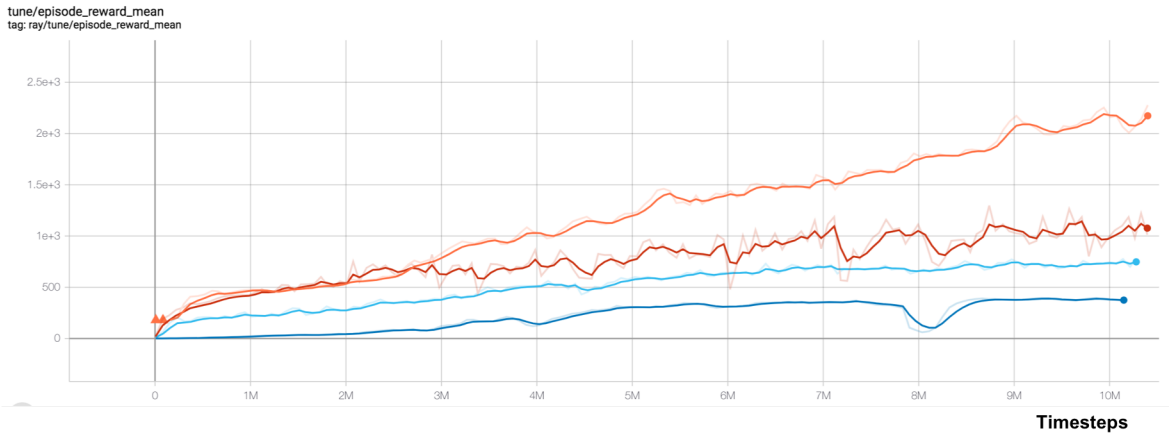
V. High Throughput RL Algorithms

In this section, we demonstrate that Cray XC systems can be used to launch two state of the art RL algorithms to learn Atari games with multiple different configurations by utilizing resources at different levels. We observed that high throughput RL techniques like IMPALA, Ape-X scale well with availability of resources.

In Ape-X, many actors use a trained policy to collect experiences on multiple different instances of environment. These experiences are stored in a buffer and a central policy learns from these experiences through a prioritized experience replay. Hence, it can be noted that Ape-X scales well with multiple actors.

Similarly, IMPALA also launches multiple actors and collects experiences with a previously trained model. Unlike, Ape-X, the centrally located target model is trained by collecting the experiences asynchronously. IMPALA also differs from other techniques like PPO and A2C, as the sampled experiences are transferred to a centralized policy rather than the gradients. The centrally located learner or multiple learners train a target policy by collected the sampled experiences. Our results indicate that the data flow architecture of IMPALA and Ape-X scale very well with number of worker nodes and reduces the training times significantly. We trained IMPALA and Ape-X agents to learn Atari games - Breakout, BeamRider, Qbert, and SpaceInvaders. The configuration is as shown in Table I. These training configuration indicate that we launch multiple workers, 32 for IMPALA and 16 for ApeX.

Figure 4 shows the episode reward means over the 10M time-steps of training with IMPALA. It can be noted that



Configuration	Reward Mean	Timestep	Walltime
IMPALA_Beamrider	2173	10M	28m 18s
IMPALA_Breakout	374	10M	28m 14s
IMPALA_Qbert	1077	10M	28m 18s
IMPALA_SpaceInvader	747	10M	28m 9s

Fig. 4. Episode reward means of Atari games learned by IMPALA agents with 32 workers

TABLE I
CONFIGURATIONS OF IMPALA AND APEX AGENTS TO TRAIN ATARI GAMES

Name	IMPALA	Apex
num_workers	32	16
num_gpus	1	1
num_cpus_per_worker	1	8
num_envs_per_worker	5	8
sample_batch_size	50	20
train_batch_size	500	512
GPU	P100	P100
CPU	36 Xeon Cores	36 Xeon Cores

IMPALA agents learned is less than 30 min on a 2 node configuration of XC.

Figure 5 shows that the Ape-X agents are slower than IMPALA as they reached the 10M timesteps in an hour. Although Ape-X agents are slow, the reward mean is higher than the IMPALA agents.

We also studied the performance of these agents across multiple nodes and we observed that for these games the agents reach a better reward mean with fewer number of CPU cores.

A. Optimally configuring agents on XC

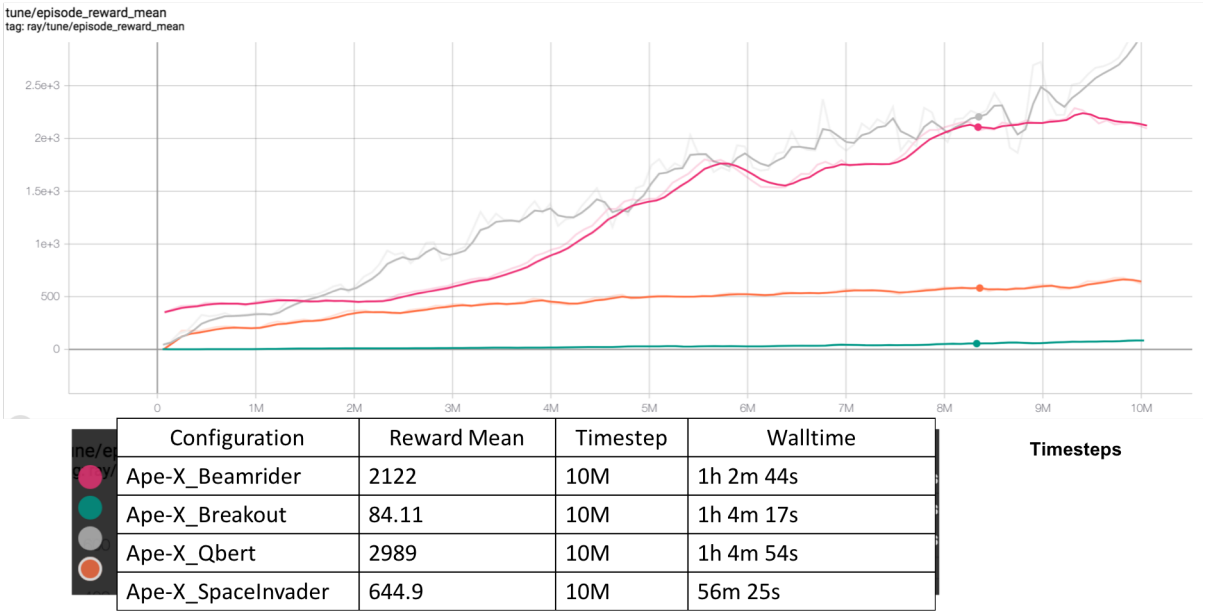
In addition to the algorithmic configuration of agents, we can configure the agents to utilize the resources in multiple different ways. RLlib provides vectorization by assigning multiple environments to a single worker. In most of the XC nodes, the Xeon CPU cores provide hyper-threading capability. By selecting the right number of CPU cores we can achieve better training times. Hence each worker can be mapped to multiple CPUs. The table I shows the configuration parameters used to select the number of CPUs assigned to each worker. In our experiments, we configured head node to avail a P100 GPU and worker nodes to use K40 or K20 GPU nodes. The underlying Tensorflow [11] models can also be configured to launch optimally on these nodes by tuning inter-thread and intra-thread level parallelism.

B. Using Tune for hyper-parameter optimization

Ray, RLlib provide hyper-parameter optimization using Tune [12]. Tune can be configured to run in parallel on multiple CPU cores provided the cluster has the availability of the resources. For example, in most of the experiments we trained the agents to perform grid execution of multiple games with multiple learning rates. These trials are launched in parallel across the cluster.

VI. CRAY XC VS DENSE GPU NODE PERFORMANCE COMPARISON

Most of the RL algorithms leverage availability of GPUs. For example, PPO algorithm uses multi-GPU optimization when training localized SGDs of multiple samples that are collected locally on each worker. Hence, we considered comparing these algorithms between a dense GPU Cray CS cluster node and multiple Cray XC nodes. The A2C and PPO algorithms can be scaled by increasing the number of workers and assigning GPUs to the workers for local computations inside on the locally policies. We compared the performance of PPO and A2C agents on Atari games. It can be noted that on XC by allocating 7 worker nodes (and one node for head), the workers can avail more number of CPU cores. In order



Configuration	Reward Mean	Timestep	Walltime
Ape-X_Beamrider	2122	10M	1h 2m 44s
Ape-X_Breakout	84.11	10M	1h 4m 17s
Ape-X_Qbert	2989	10M	1h 4m 54s
Ape-X_SpaceInvader	644.9	10M	56m 25s

Fig. 5. Episode reward means of Atari games learned by Ape-X agents with 16 workers

TABLE II
CONFIGURATIONS OF PPO AGENTS RUNNING ON XC AND CS

Name	CS	XC
num_nodes	1	8
num_gpus_on_node	8 (P100)	1 P(100)
num_cpus_available	72	36
num_workers	7	7
num_gpus	1	1

TABLE III
CONFIGURATIONS OF IMPALA AND APEX AGENTS TO TRAIN ATARI GAMES ON A SINGLE NODE

num_workers	num_cpus_per_worker
1	32
2	16
4	8
8	4
16	2

to compare the performance, we assigned equal number i.e. 9 CPU cores to each worker. We also assign 9 environments to each worker to avail the 9 CPU cores. The sample batch size - 100 and train batch size - 7K remain the same between two systems. The dense CS node consists of 8 GPUs. We assign 1 GPU to head and 7 to workers on the same node. Overall the number of CPU cores is 64 and GPUs is 8.

Figures 6 and 7 compares Qbert and SpaceInvader games trained by A2C and PPO agents. In both the cases, for a 10M time-step run, agents running on CS tend to have taken more time. For SpaceInvader, the mean episode reward is similar across the configuration. But for Qbert, the mean episode reward for slower runs on CS are significantly higher than the runs on XC.

VII. SINGLE NODE PERFORMANCE

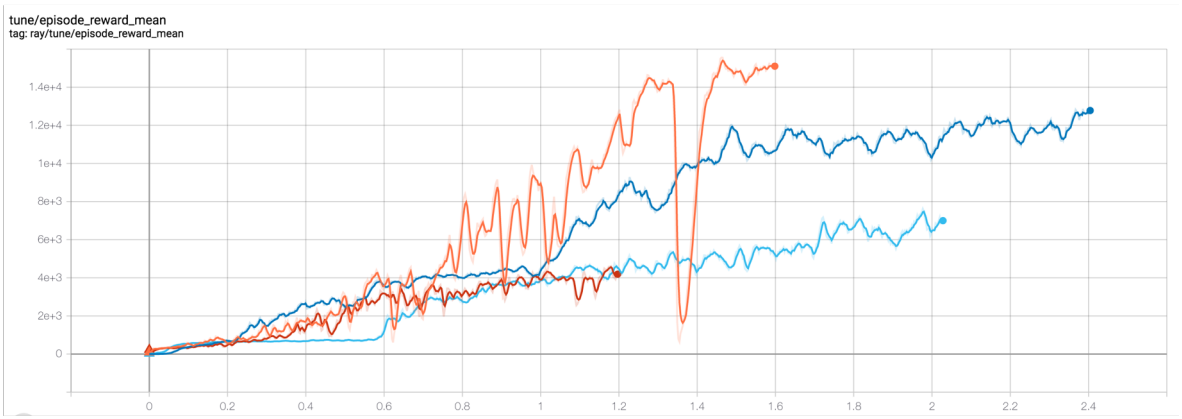
Given the flexibility of scaling the RL algorithms, a single node on a XC system is sufficient to train a simple model effectively. In this section we explore different configuration on a single node. A typical XC node contains a GPU and multiple CPU cores.

Figure 8 shows the training performance of a PPO agent on Atari SpaceInvaders game for multiple different combinations of workers and CPU cores available per worker. The number of environments per workers is same as the number of CPU cores available to the worker, hence we exploit the vectorized parallelism of the RLlib execution. The data shows that a 1 worker configuration is as expected is slow although 32 environments are run in parallel. The 16 worker and 8 worker configurations perform relatively similar indicating that launching more number of environments per workers can cause significant performance loss.

We experimented with using A2C agents and other Atari games and observed a similar trend indicating that these algorithms scale well with number of workers.

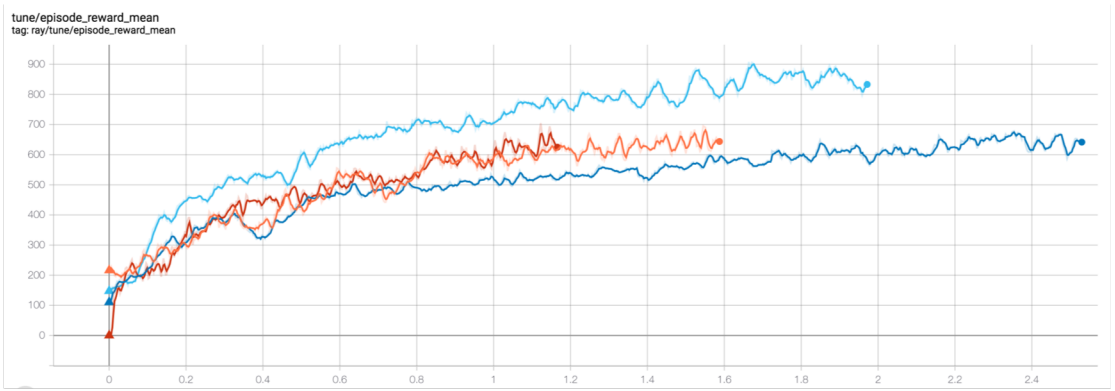
VIII. SCALING ON A MULTI-NODE CONFIGURATION

In this section, we discuss the experiment with training RL agents on a multi-node configuration of XC systems. The configuration is shown in Table IV. Each worker is assigned a single node to utilize the P100 GPU available on each single node. The goal is to utilize the maximum resources available in the allocation. Therefore, we assigned as many available CPU cores as possible to each of the worker. We also vectorized each worker by instantiating multiple environments.



Configuration	Reward Mean	Timestep	Walltime
Qbert_CS_A2C	1.5101e+4	10M	1h 35m 54s
Qbert_CS_PPO	1.2775e+4	10M	2h 24m 15s
Qbert_XC_A2C	4193	10M	1h 11m 48s
Qbert_XC_Ppo	6998	10M	2h 1m 40s

Fig. 6. Comparing the performance of Atari Qbert on XC vs dense 8 GPU node of CS.



Configuration	Reward Mean	Timestep	Walltime
SpaceInvaders_CS_A2C	643	10M	1h 35m 15s
SpaceInvaders_CS_PPO	641	10M	2h 32m 47s
SpaceInvaders_XC_A2C	624	10M	1h 9m 57s
SpaceInvaders_XC_PPO	833	10M	1h 58m 18s

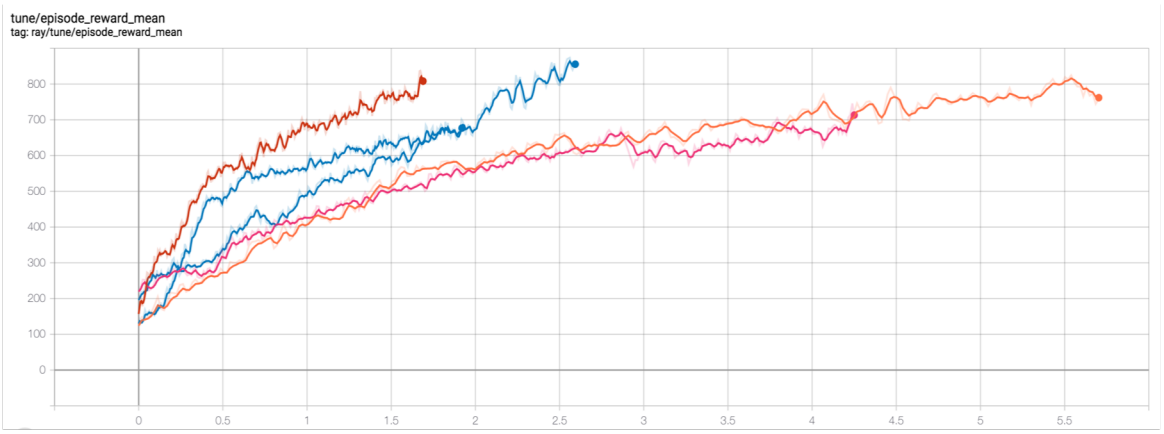
Fig. 7. Comparing the performance of Atari SpaceInvader on XC vs dense 8 GPU node of CS.

This enabled us to increase the training batch size substantially - in some cases close to 32000. The same Atari games were run with this configuration.

We analyzed the scaling by training A2C and PPO agents on Atari games. We trained these agents with large batch sizes for 10M time-steps. Figure 9 shows the scaling of Atari-Qbert across different node configurations. For A2C, the best reward mean is achieved by single worker configuration running for close to 5 hours. Compared to A2C, PPO performed extremely well at 1,2, and 4 worker configurations. The correlation between the training time and the episode reward mean score can be observed for the figures.

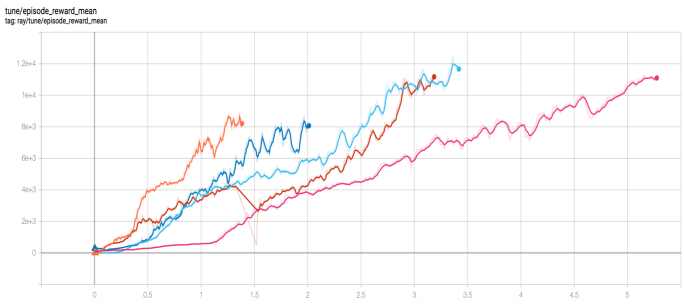
Figure 10 shows a similar trend with respect to the wall time for Atari-SpaceInvader. Although time to train an agent with less number of workers is slow, the overall reward mean is almost the same in both A2C and PPO scenarios.

The scaling trend is shown in Figure 11. The figure shows that the trend is almost linear till 4 workers, but slowly tapers off beyond 8 workers configuration. We have observed such a trend is most of the games. This could indicate that the policy

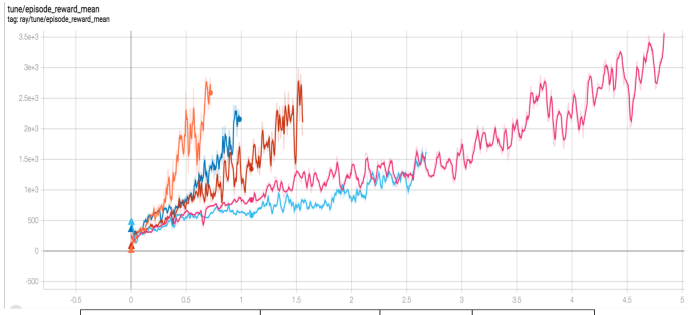


SpaceInvaders	Reward Mean	Timestep	Walltime
PPO_16w_0gpu_2cpu	808.3	10M	1h 41m 21s
PPO_1w_0gpu_32cpu	761.9	10M	5h 42m 10s
PPO_2w_0gpu_16cpu	713.4	10M	4h 15m 1s
PPO_4w_0gpu_8cpu	855.7	10M	2h 35m 35s
PPO_8w_0gpu_4cpu	676.9	10M	1h 55m 11s

Fig. 8. Single node performance of Atari-SpaceInvaders using PPO agent with different configurations



Qbert	Reward Mean	Timestep	Walltime
PPO_16w_1gpu_20cpu	8211	10M	1h 23m 1s
PPO_1w_1gpu_36cpu	1.1095e+4	10M	5h 16m 27s
PPO_2w_1gpu_36cpu	1.1662e+4	10M	3h 25m 3s
PPO_4w_1gpu_36cpu	1.117e+4	10M	3h 11m 9s
PPO_8w_1gpu_36cpu	8068	10M	2h 0m 41s



Qbert	Reward Mean	Timestep	Walltime
A2C_16w_1gpu_20cpu	2589	10M	43m 21s
A2C_1w_1gpu_36cpu	3561	10M	4h 50m 8s
A2C_2w_1gpu_36cpu	1621	10M	2h 40m 39s
A2C_4w_1gpu_36cpu	2106	10M	1h 33m 25s
A2C_8w_1gpu_36cpu	2155	10M	59m 6s

Fig. 9. Comparing multi-node performance of Atari - Qbert game across 1, 2, 4, 8 and 16 node configuration. Each worker is assigned multiple CPUs. The x-axis indicates the wall time to train for 10M steps

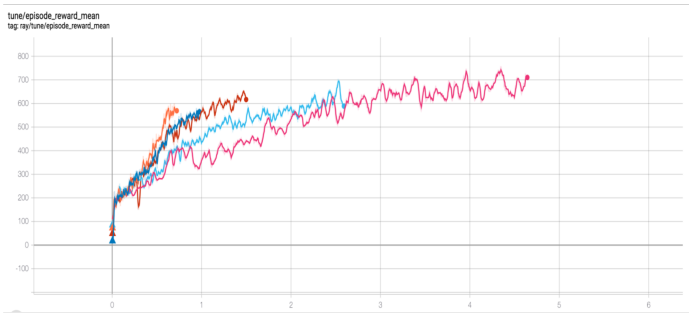
optimizers are susceptible to network latency. In both these policies experiences or locally computed gradients are sent to the a global optimizer. Both the policies employ synchronous updates of gradients. We explored other potential causes for such trend, we believe in order to answer such interaction further research might be required.

IX. Conclusion

State of the art reinforcement algorithms impose significant demand on system resources like memory, storage, compute, and network. These algorithms are designed to scale across multiple nodes and thereby perform all-reduce operations to train and optimize decision making policies. This motivated us to study these algorithms in depth and consider them as

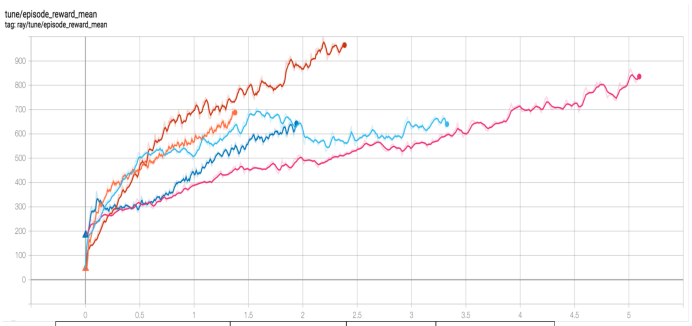
TABLE IV
Configurations of Multi-Node scaling with P100 GPUs and Xeon CPUs

workers	cpus/worker	gpus/worker	num/worker
1	36	1	36
2	36	1	36
4	36	1	36
8	36	1	36
16	20	1	20



SpaceInvaders	Reward Mean	Timestep	Walltime
A2C_8w_1gpu_36cpu	564	10M	58m 34s
A2C_16w_1gpu_20cpu	569	10M	43m 21s
A2C_1w_1gpu_36cpu	710.3	10M	4h 38m 36s
A2C_2w_1gpu_36cpu	588	10M	2h 35m 27s
A2C_4w_1gpu_36cpu	616.7	10M	1h 29m 52s

Time in hrs



SpaceInvaders	Reward Mean	Timestep	Walltime
PPO_2w_1gpu_36cpu	640.1	10M	3h 19m 35s
PPO_16w_1gpu_20cpu	687.8	10M	1h 22m 40s
PPO_1w_1gpu_36cpu	852.6	10M	5h 5m 42s
PPO_4w_1gpu_36cpu	965.4	10M	2h 23m 4s
PPO_8w_1gpu_36cpu	643.3	10M	1h 56m 31s

Time in hrs

Fig. 10. Comparing multi-node performance of Atari - SpaceInvader game across 1, 2, 4, 8 and 16 node configuration.

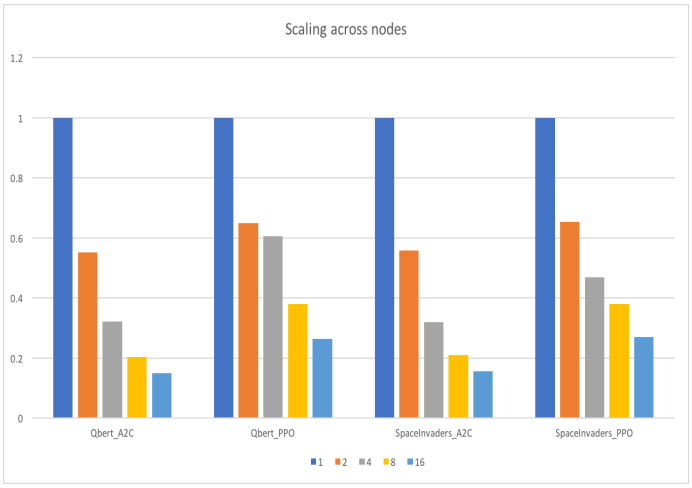


Fig. 11. Scaling Qbert with a PPO agent across 1, 2, 4, 8, and 16 nodes. Each worker is assigned a single node and node level parallelism is utilized by assigning multiple CPUs to each of the worker

HPC workloads.

In this paper, we trained some of these algorithms on Atari games on Cray XC systems to analyze their resource usage and scalability. We demonstrated that these applications are suitable workloads to exploit the computation, communication, and memory capacities of XC systems. We demonstrated that distributed execution framework like Ray can be deployed on these systems. This paved a way to execute multiple RL environments and algorithms with multiple different configurations using RLlib. We also analyzed comparative performance study between XC and dense GPU node of a Cray CS cluster system. Further, we explored single node performance and single node scalability of these algorithms on an XC system. Finally, we discussed scalability of these techniques on multi-node configuration spanning 2 to 16 nodes. We analyzed Atari games with various agent algorithms and demonstrated that these algorithms scale with the number of nodes.

This work provides us with a strong foundation to experiment with complex environments and use cases in scientific discovery, self-driving cars, and game playing. Further, this work motives us to explore RL algorithms on an exascale system.

ACKNOWLEDGMENT

The authors would like to thank the reviewers of the paper, Alexey Tumanov, Kai Rothauge for providing feedback on the work carried out in this paper.

REFERENCES

[1] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. R. Baker, M. Lai, A. Bolton, Y. Chen, T. P. Lillicrap, F. F. C. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, "Mastering the game of go without human knowledge," *Nature*, vol. 550, pp. 354–359, 2017.

[2] R.Evans, J.Jumper, J.Kirkpatrick, L.Sifre, T.F.G.Green, C.Qin, A.Zidek, A.Nelson, A.Bridgland, H.Penedones, S.Petersen, K.Simonyan, S.Crossan, D.T.Jones, D.Silver, K.Kavukcuoglu, D.Hassabis, and A.W.Senior, "De novo structure prediction with deep-learning based scoring," *Thirteenth Critical Assessment of Techniques for Protein Structure Prediction*, 2018.

[3] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica, "Ray: A distributed framework for emerging ai applications," in *OSDI*, 2018.

[4] E. Liang, R. Liaw, R. Nishihara, P. Moritz, R. Fox, K. Y. Goldberg, J. Gonzalez, M. I. Jordan, and I. Stoica, "Rllib: Abstractions for distributed reinforcement learning," in *ICML*, 2018.

[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, "Playing atari with deep reinforcement learning," *CoRR*, vol. abs/1312.5602, 2013.

[6] L. Espeholt, H. Soyer, R. Munos, K. Simonyan, V. Mnih, T. Ward, Y. Doron, V. Firoiu, T. Harley, I. Dunning, S. Legg, and K. Kavukcuoglu, "Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures," in *ICML*, 2018.

[7] D. Horgan, J. Quan, D. Budden, G. Barth-Maron, M. Hessel, H. P. van Hasselt, and D. Silver, "Distributed prioritized experience replay," *CoRR*, vol. abs/1803.00933, 2018.

[8] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *CoRR*, vol. abs/1707.06347, 2017.

[9] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *ICML*, 2016.

[10] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.

[11] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. J. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Józefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. G. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. A. Tucker, V. Vanhoucke, V. Vasudevan, F. B. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *CoRR*, vol. abs/1603.04467, 2015.

[12] R. Liaw, E. Liang, R. Nishihara, P. Moritz, J. E. Gonzalez, and I. Stoica, "Tune: A research platform for distributed model selection and training," *arXiv preprint arXiv:1807.05118*, 2018.