Ray/RLLib Walkthrough

- Ray enables arbitrary Python functions to be executed asynchronously
- Communication through TCP/IP (and not MPI)
- A simple decorator can “parallelize” python functions

```python
# A regular Python function.
def regular_function():
    return 1

# A Ray remote function.
@ray.remote
def remote_function():
    return 1
```

```python
# These happen serially.
for _ in range(4):
    regular_function()
```

```python
# These happen in parallel.
for _ in range(4):
    remote_function.remote()
```

Non-blocking

```python
@ray.remote
def remote_chain_function(value):
    return value + 1


y1_id = remote_function.remote()
assert ray.get(y1_id) == 1

chained_id = remote_chain_function.remote(y1_id)
assert ray.get(chained_id) == 2
```

Calls can be chained: Second function will not be evaluated until first one is complete

If first and second function are scheduled on different machines, the output of the first task is communicated over the network to the second

```python
@ray.remote(num_cpus=4, num_gpus=2)
def f():
    return 1
```

Easy heterogeneity

```python
# Ray also supports fractional resource requirements
@ray.remote(num_gpus=0.5)
def h():
    return 1
```

Starting Ray (on Theta)

```python
def run_ray_head(head_ip):
    with open('ray.log.head', 'wb') as fp:
        subprocess.run(
            f'ray start --head \
                    --num-cpus 8 \
                    --node-ip-address={head_ip} \
                    --redis-port={REDIS_PORT}',
            shell=True,
            check=True,
            stdout=fp,
            stderr=subprocess.STDOUT
        )

def run_ray_worker(head_redis_address):
    with open(f'ray.log.{rank}', 'wb') as fp:
        subprocess.run(
            f'ray start --redis-address={head_redis_address} \
                    --num-cpus 8',
            shell=True,
            check=True,
            stdout=fp,
            stderr=subprocess.STDOUT
        )
```

Head node

Worker node

What you'll really use Ray like

- For the most part – you'll end up using one of the multiple ML applications built on top of Ray.
- Distributed hyperparameter search using Tune. (But please use DeepHyper instead)
- Reinforcement learning with RLLib.
- Distributed training with RaySGD.
- All one needs to do is start Ray on multiple head and worker nodes and call an RLLib code from the head.

```
if rank == 0:
    head_redis_address = master()
else:
    worker()

comm.barrier()

if rank == 0:
    # Run the python script to do RL
    exec_string = "python train_ppo.py --ray-address='"+str(head_redis_address)+r"'"
    subprocess.run(exec_string,shell=True,check=True)
    logging.info("RL LIB invoked successfully. Exiting.")

comm.barrier()
```

On head node

```
ray.init(redis_address=args.ray_address)
config = appo.DEFAULT_CONFIG.copy()
config["log_level"] = "WARN"
config["num_gpus"] = 0
config["num_workers"] = int(ray.available_resources()['CPU'])
config["lr"] = 1e-4

trainer = appo.APPOTrainer(config=config, env="CartPole-v0")

# Can optionally call trainer.restore(path) to load a checkpoint.
with open('Training_iterations.txt','wb',0) as f:
    for i in range(10):
        result = trainer.train()
```

I lost a lot of hair trying to figure out what dependencies worked with each other for running on Theta. Finally, found a combination:
Ray[rllib]==0.7.6
Tensorflow==1.14
Numpy==1.16.1
Mpi4py 3.1.0 (built from source)

Multiple RL algorithms with Rllib

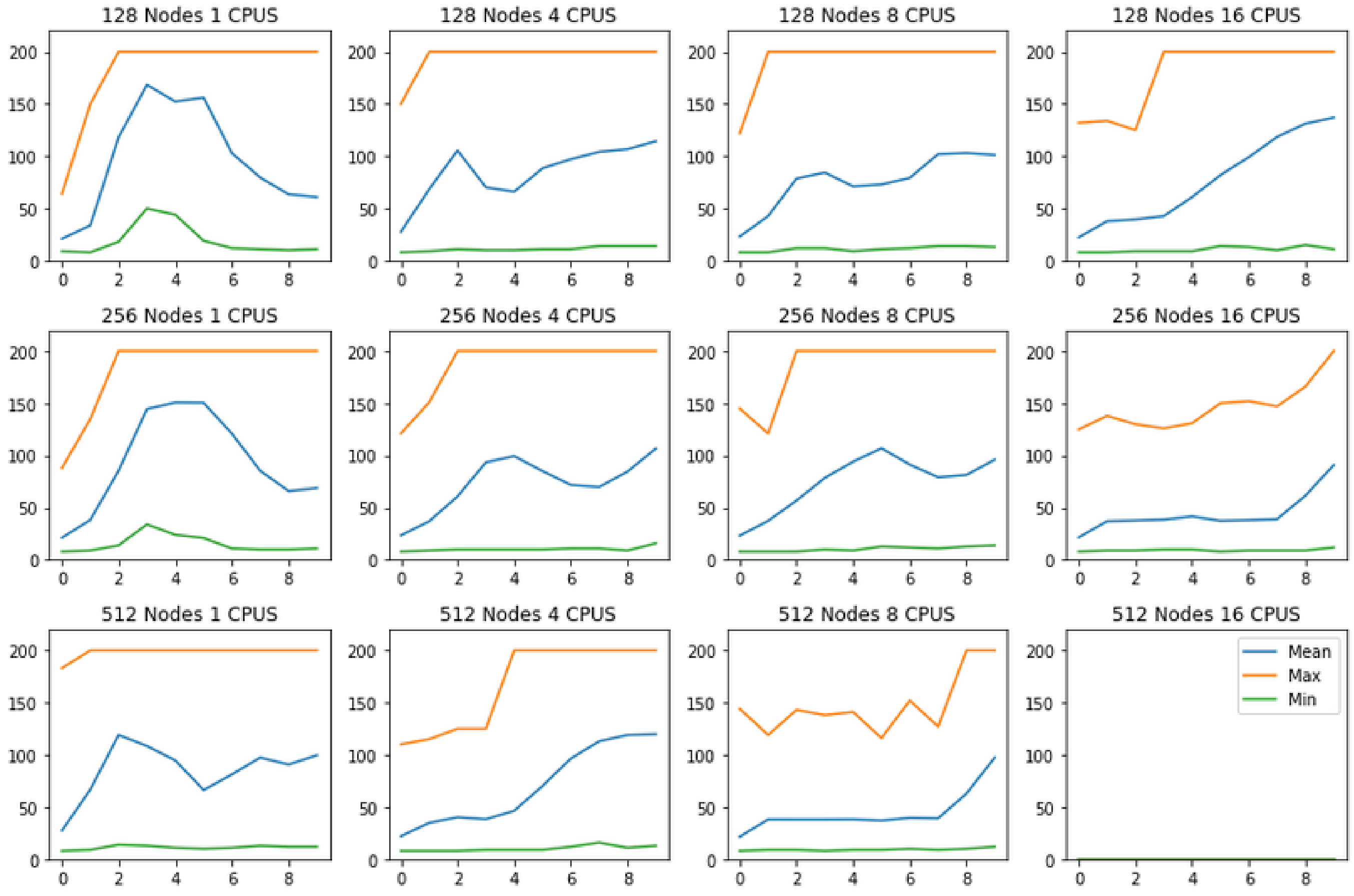
- Lots of algorithmic choices (Value based, policy based, synchronous, asynchronous)

Algorithm	Frameworks	Discrete Actions	Continuous Actions	Multi-Agent
A2C, A3C	tf + torch	Yes +parametric	Yes	Yes
ARS	tf + torch	Yes	Yes	No
ES	tf + torch	Yes	Yes	No
DDPG, TD3	tf + torch	No	Yes	Yes
APEX-DDPG	tf	No	Yes	Yes
DQN, Rainbow	tf + torch	Yes +parametric	No	Yes
APEX-DQN	tf + torch	Yes +parametric	No	Yes
IMPALA	tf	Yes +parametric	Yes	Yes
MARWIL	tf + torch	Yes +parametric	Yes	Yes
PG	tf + torch	Yes +parametric	Yes	Yes
PPO, APPO	tf + torch	Yes +parametric	Yes	Yes
QMIX	torch	Yes	No	Yes
SAC	tf + torch	Yes	Yes	Yes
AlphaZero	torch	Yes +parametric	No	No
LinUCB, LinTS	torch	Yes +parametric	No	Yes
MADDPG	tf	No	Yes	Yes

- Easy integration for custom agent models and environments

```
class my_environment(gym.Env):

    def __init__(self, config):
        self.Scalar = config['Scalar']
        print('Scalar value : ', self.Scalar)
        self.observation_space = spaces.MultiDiscrete([ 4, 49, 49, 49, 49 ])
        self.action_space = spaces.Discrete(49)
        self.current_step = 0
        self.intvector = np.asarray([0,0,0,0,0], dtype=np.int64)

    def reset(self):
        self.current_step = 0
        self.intvector = np.asarray([0,0,0,0,0], dtype=np.int64)

        return self.intvector

    def _take_action(self, action):
        self.intvector[self.current_step +1] = action
        self.intvector[0] += 1

    def step(self, action):
         # Need to call a simulation here using a further subprocess
        self._take_action(action)
```

The cartpole experiment at scale



Suraj has the following to-do:

1. Custom gym-env that uses subprocess to call OpenFOAM/Fenics
2. Replicate some DRL based control experiments (previously serial/O(10) nodes parallel) at scale
3. Hopefully be able to take requests for other RL tasks at the end of the summer.

```
aprun -n $COBALT_JOBSIZE -N 1 --cc none python start_ray.py
```

First iteration did not finish in 180 minutes of wall time