Ray:
A Unified Distributed Framework for Emerging AI Applications

CS675: Distributed Systems (Spring 2020)

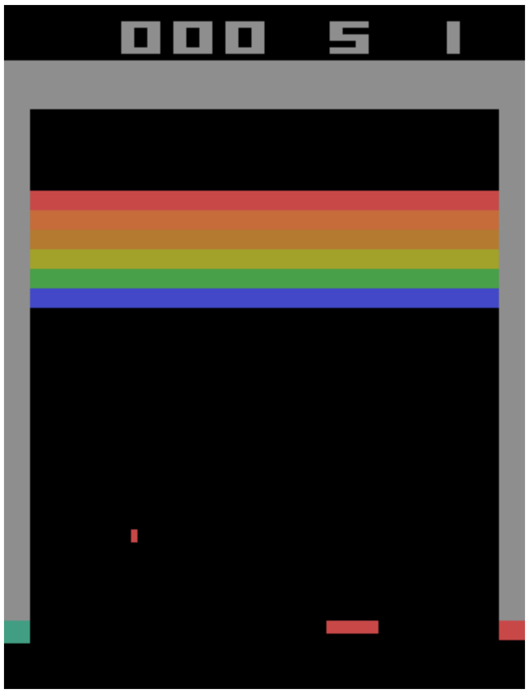
Lecture 11

Yue Cheng

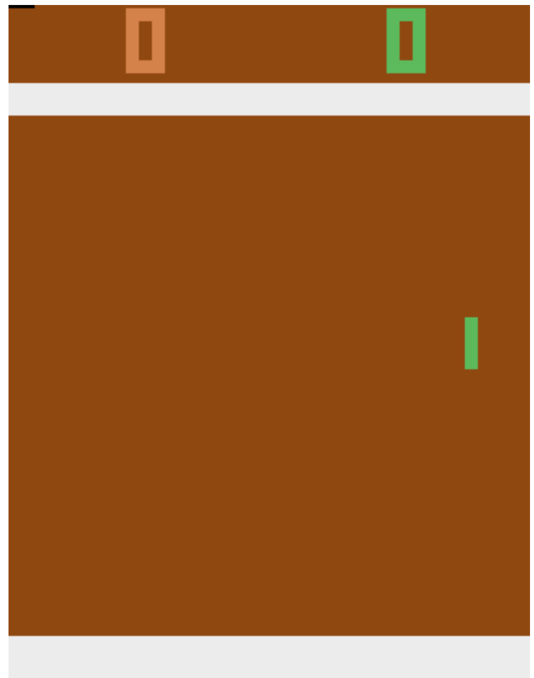
Supervised Learning → Reinforcement Learning (RL)

• One prediction →	• Sequences of actions
• Static environment →	• Dynamic environments
• Immediate feedback →	• Delayed rewards

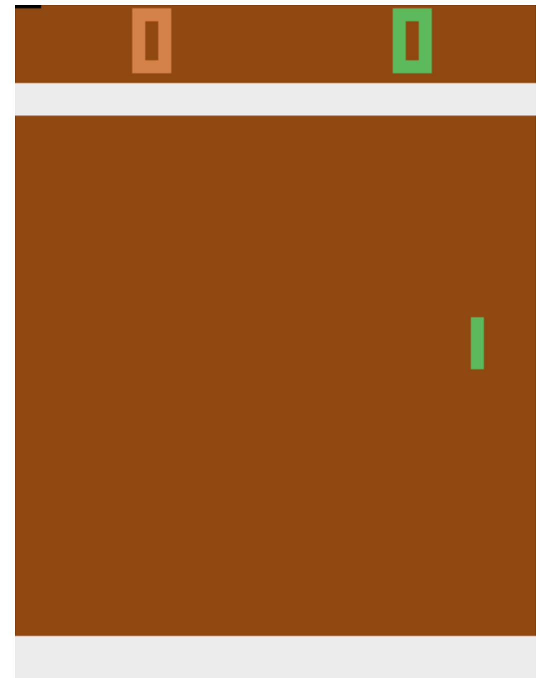
Reinforcement learning



Atari breakout



Pong: after 30 mins of training

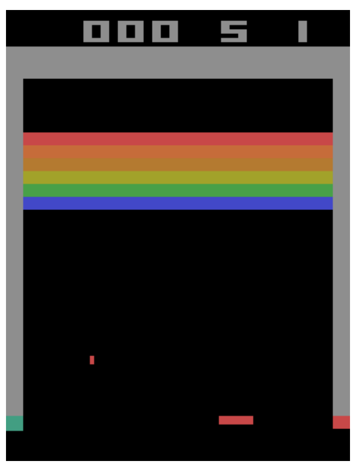


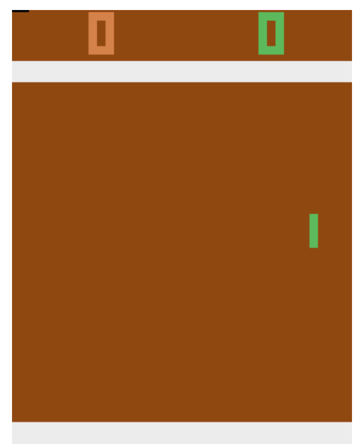
Pong: DQN wins like a boss

*: Playing Atari with Deep Reinforcement Learning: https://arxiv.org/abs/1312.5602

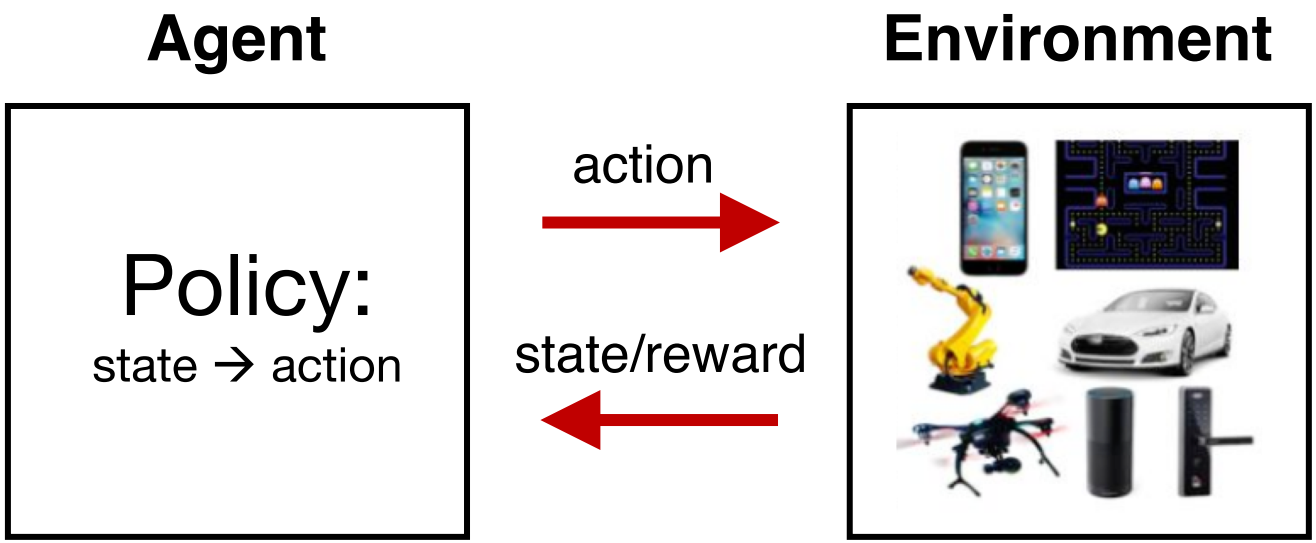
RL application pattern

- Process inputs from different sensors in parallel & real-time
- Execute large number of simulations, e.g., up to 100s of millions

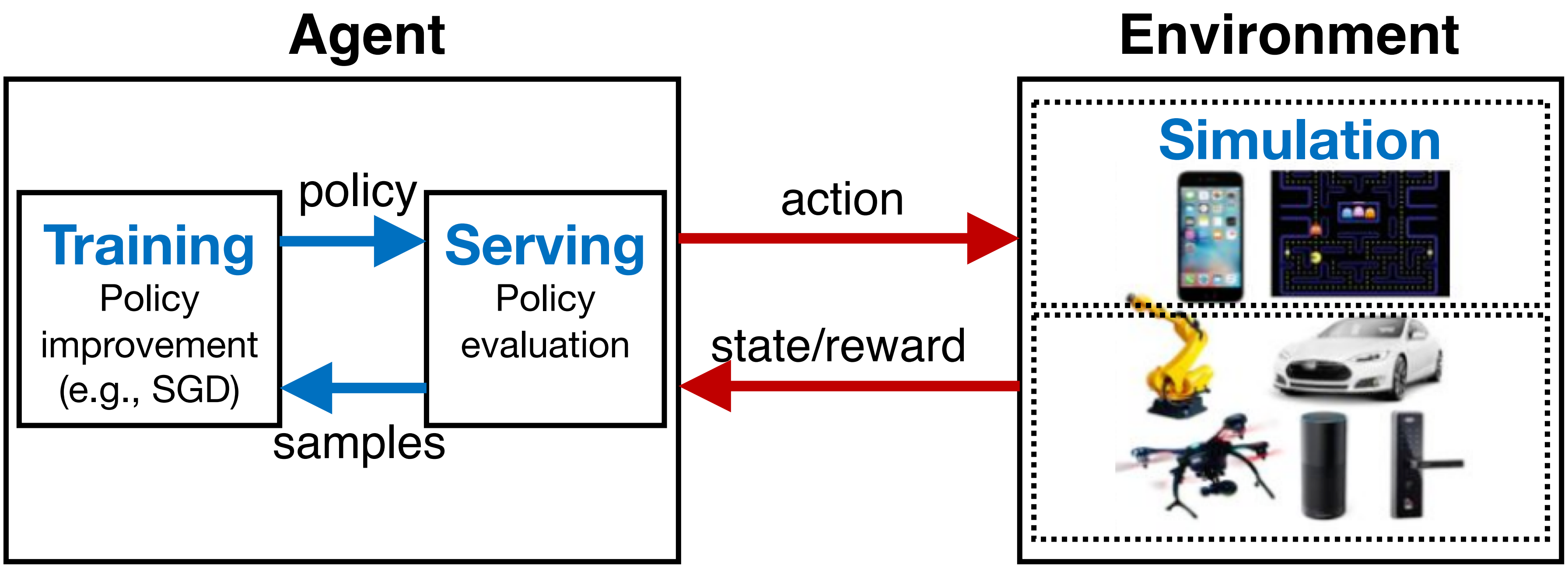




RL setup


Agent
Environment
Policy:
state → action
action
state/reward

RL setup in more detail



RL application pattern

- Process inputs from different sensors in parallel & real-time
- Execute large number of simulations, e.g., up to 100s of millions
- Rollouts outcomes are used to update policy (e.g., SGD)

RL application requirements

- Need to handle dynamic task graphs, where tasks have
  - Heterogeneous durations
  - Heterogenous computations
- Schedule millions of tasks / sec
- Make it easy to parallelize ML algorithms (often written in Python)

The ML/AI ecosystems today

Distributed systems

Distributed training

TensorFlow, PyTorch, MXNet

Distributed systems

Model serving

Clipper, TensorFlow serving

Distributed systems

Data streaming

Flink, many others

Distributed systems

Data processing

Spark, Hadoop, Dask

Distributed systems

Simulation

MPI, simulators, custom tools

Emerging AI applications require **stitching** together **multiple** disparate systems

Ad hoc integrations are difficult to manage and program!

Ray API

Tasks

```
futures = f.remote(args)
```

Actors

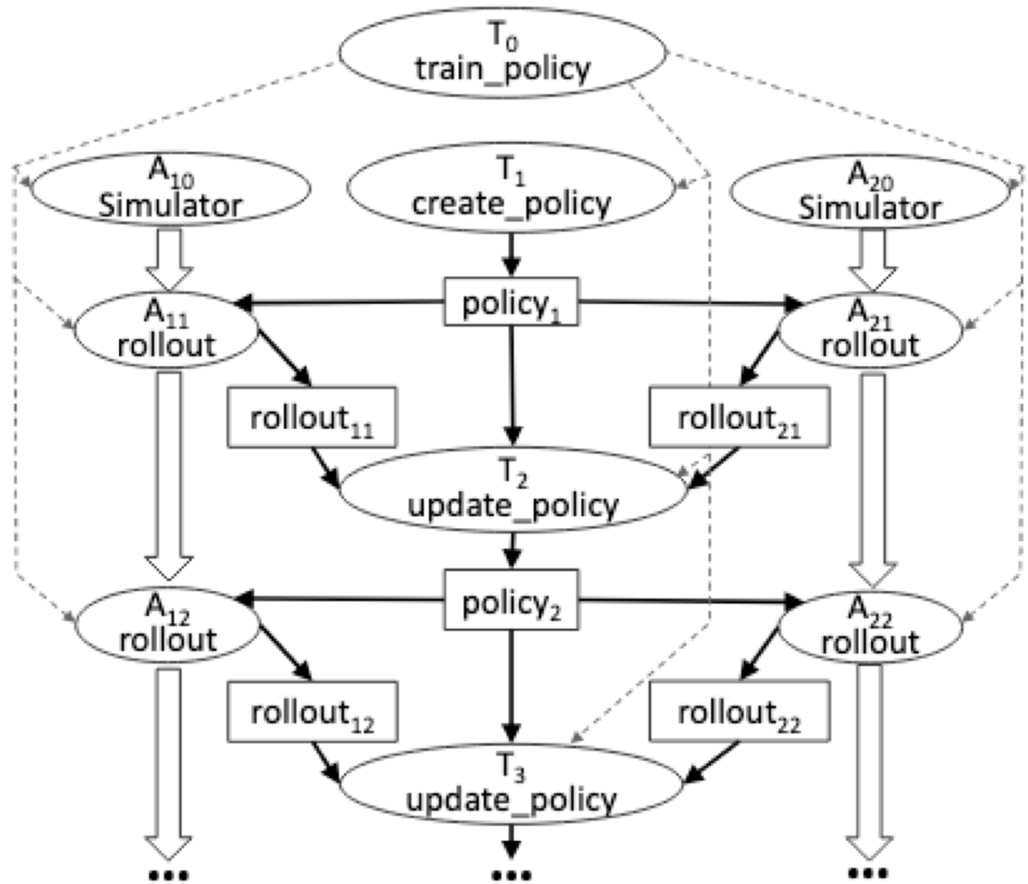
```
actor = Class.remote(args)
futures = actor.method.remote(args)
```

```
objects = ray.get(futures)
ready_futures = ray.wait(futures, k, timeout)
```

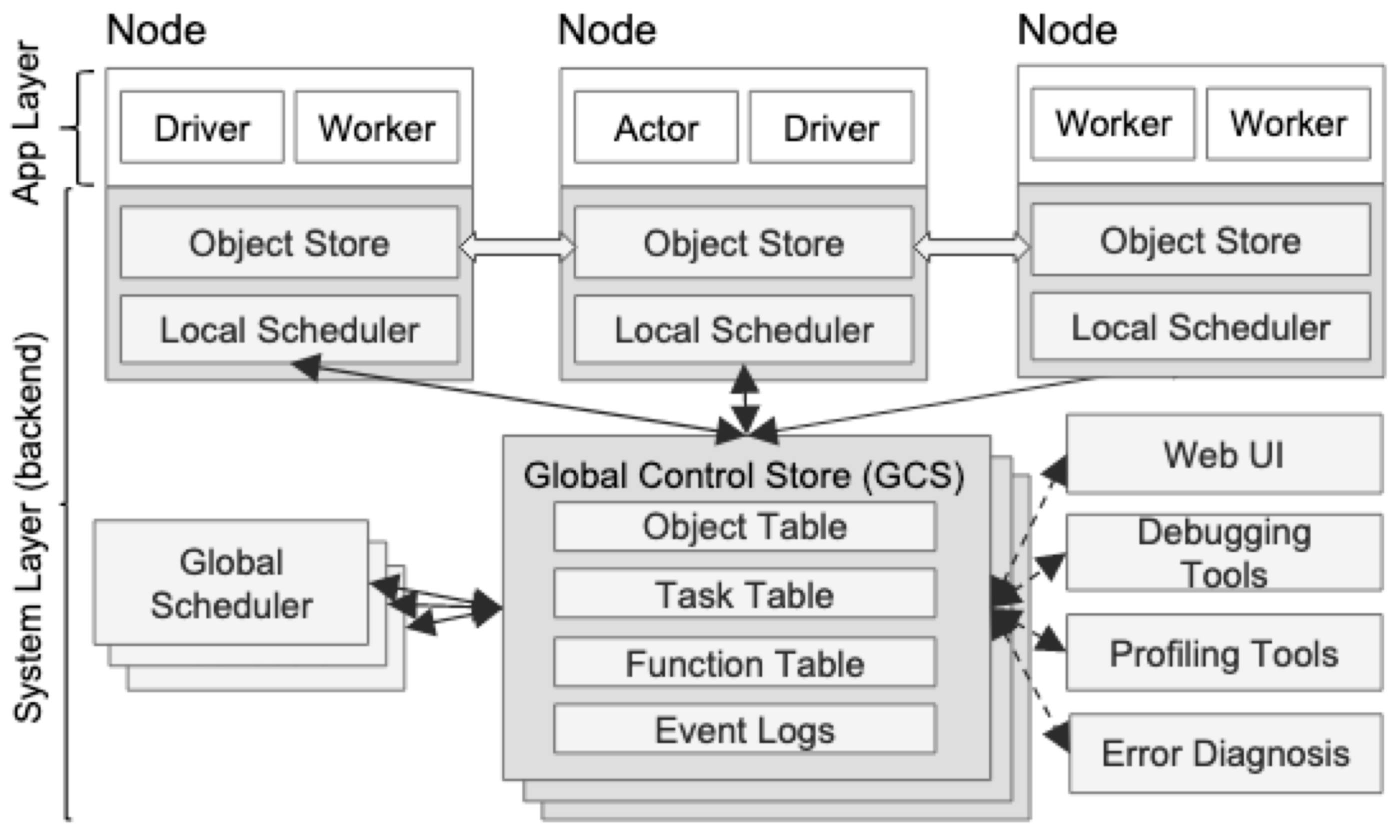
Ray API examples

- See separate notes

Computation model



Ray architecture



Global control store (GCS)

- Object table

- Task table

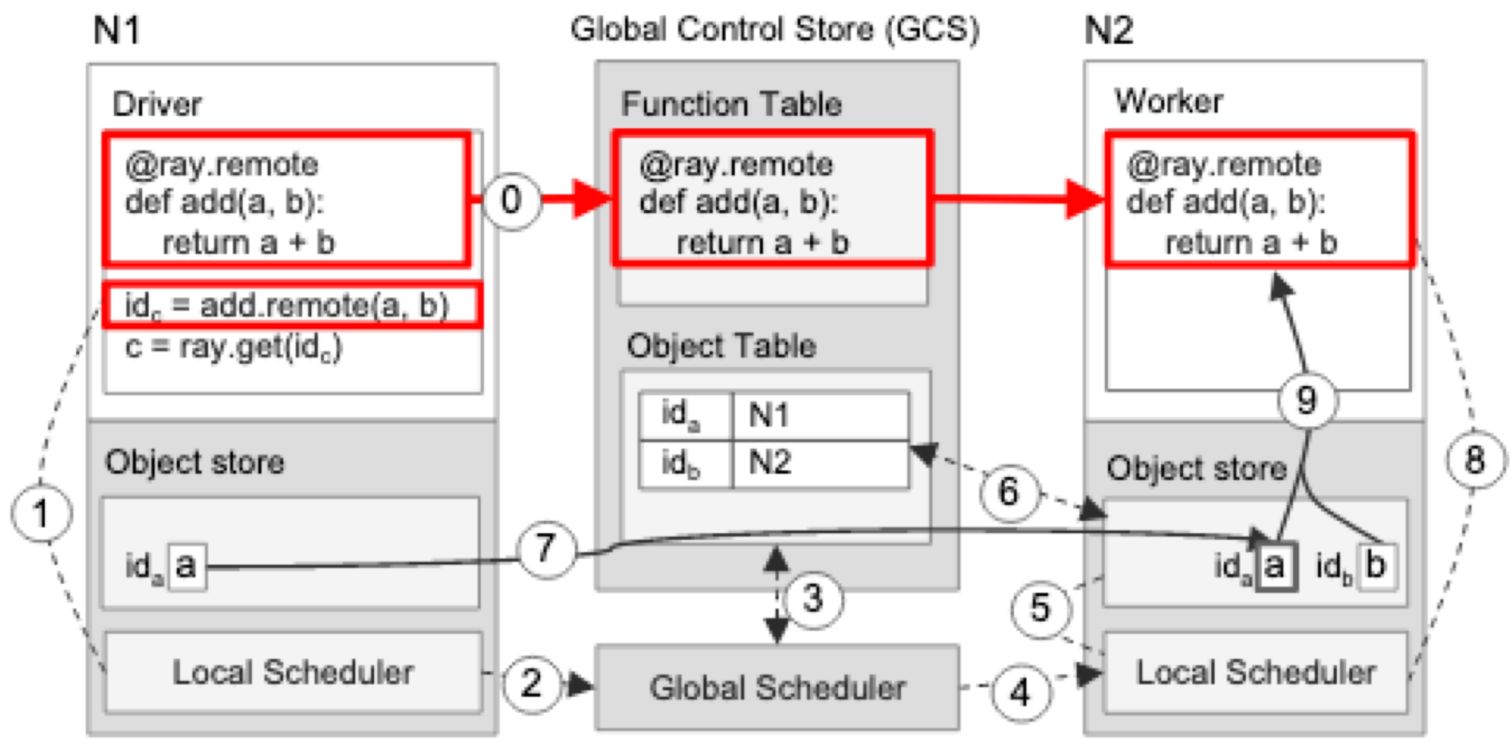
- Function table

Ray scheduler

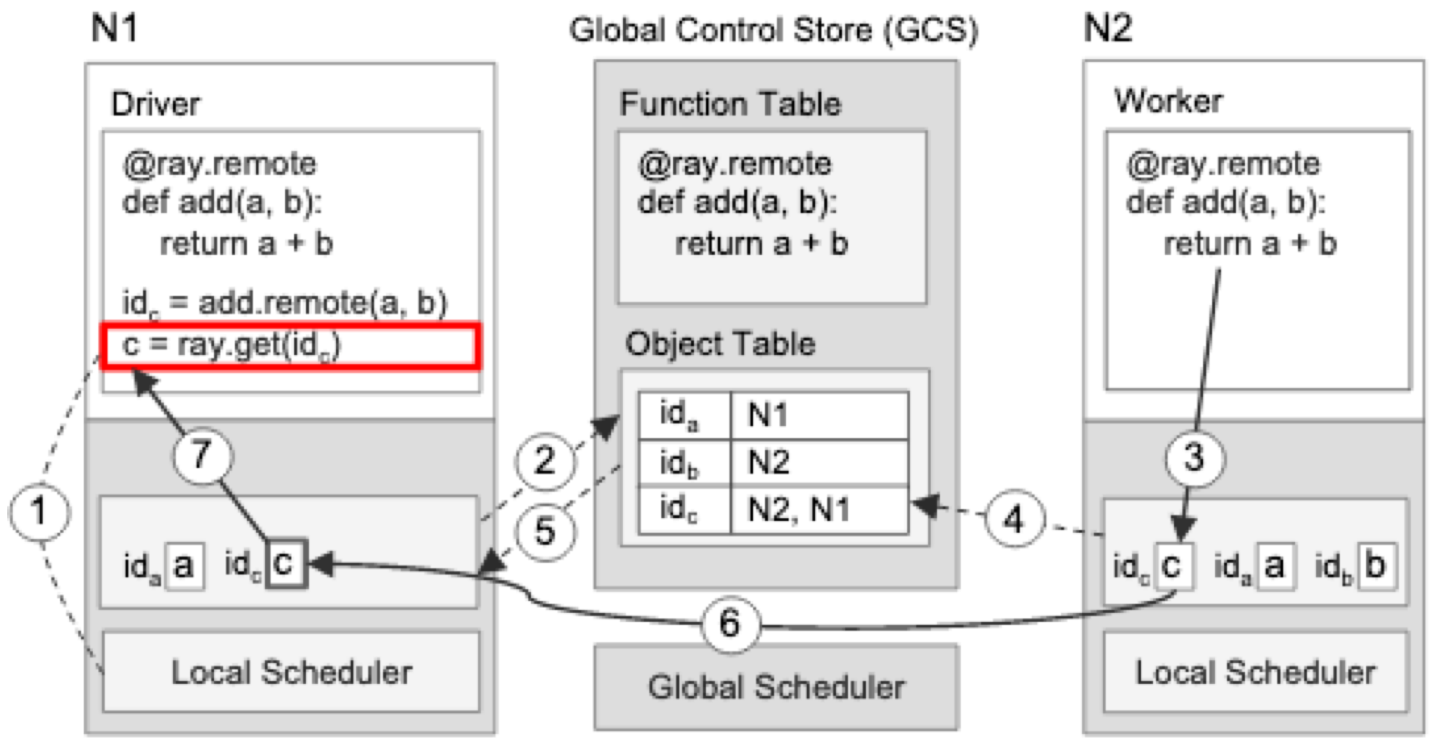
Fault tolerance

- Tasks
- Actors
- GCS
- Scheduler

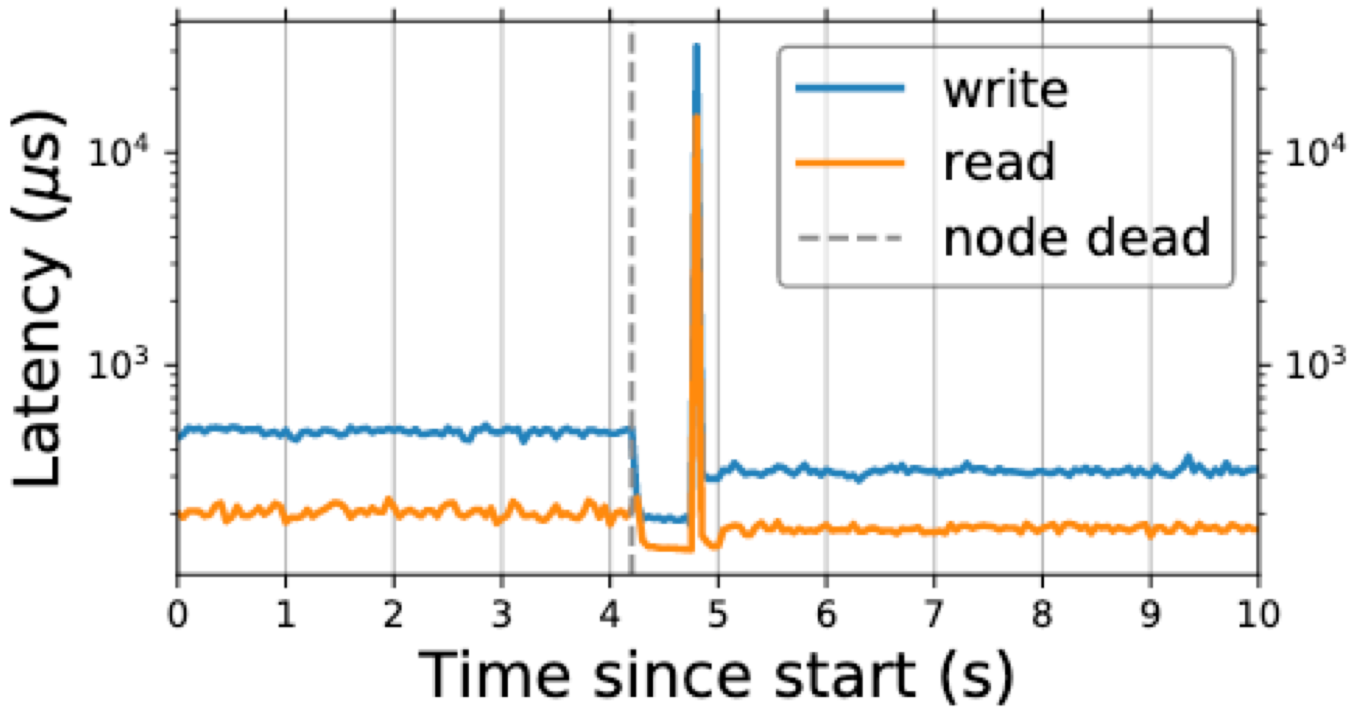
Executing a task remotely



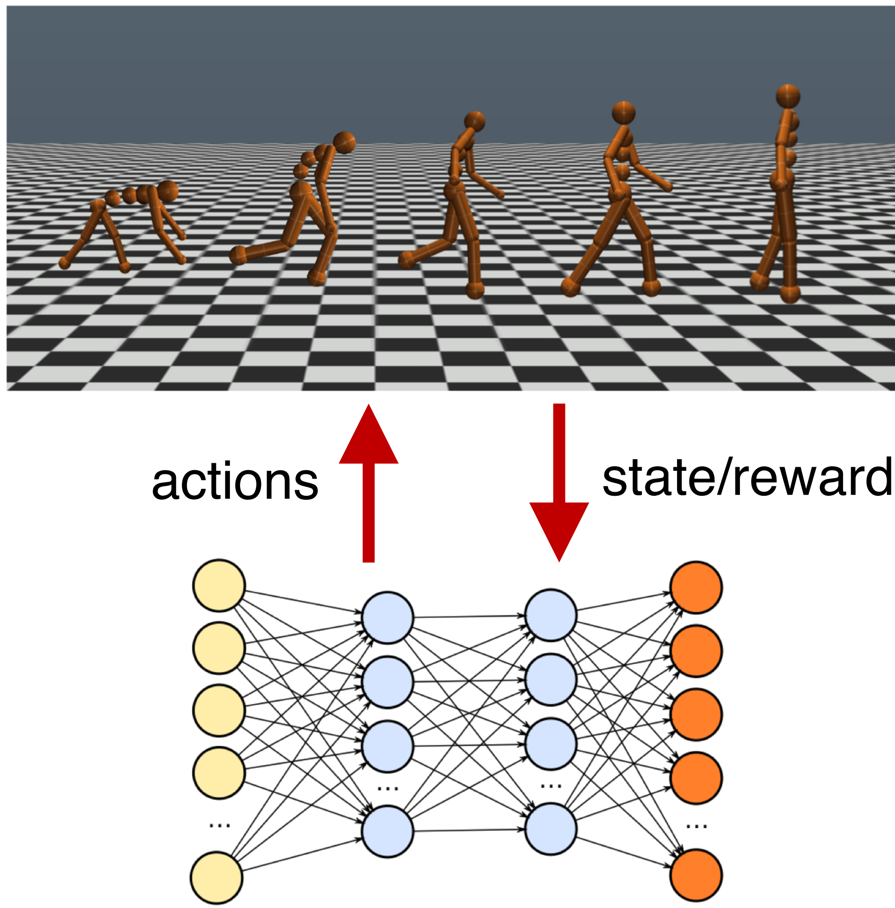
Returning the results of a remote task



GCS fault tolerance



Evolution strategies (ES)



Simulator

Policy

Try lots of different policies and see which one works best!

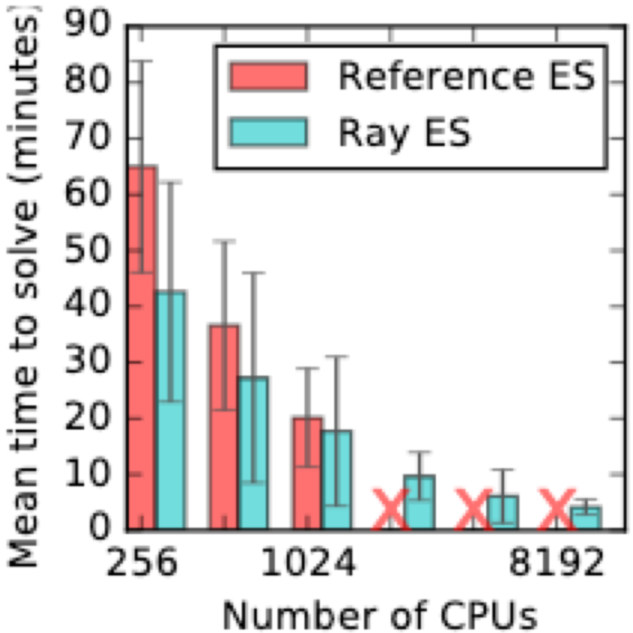
Pseudocode

```
class Worker(object):
  def do_simulation(policy, seed):
    # perform simulation and return reward

workers = [Worker() for i in range(20)]
policy = initial_policy()


for i in range(200):
  seeds = generate_seeds(i)
  rewards = [workers[j].do_simulation(policy, seeds[j])
             for j in range(20)]
  policy = compute_update(policy, rewards, seeds)
```

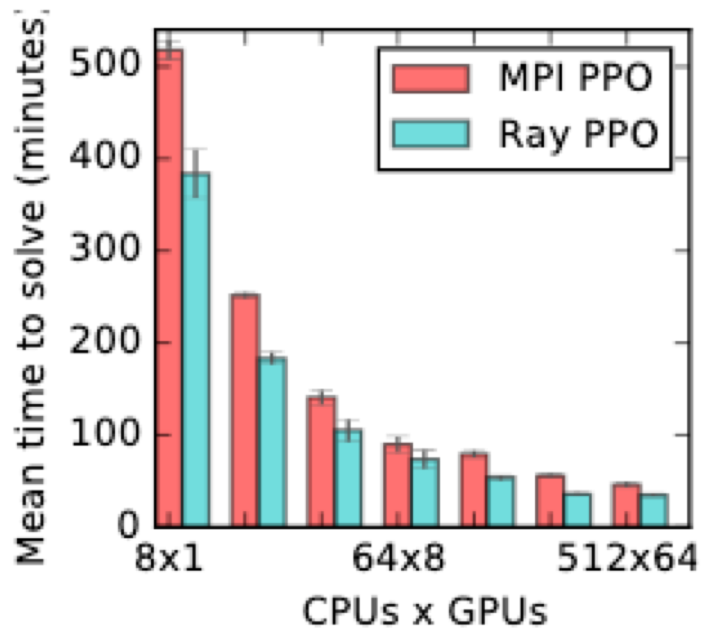
Pseudocode

```
@ray.remote
class Worker(object):
  def do_simulation(policy, seed):
    # perform simulation and return reward

workers = [Worker.remote() for i in range(20)]
policy = initial_policy()


for i in range(200):
  seeds = generate_seeds(i)
  rewards = [workers[j].do_simulation.remote(policy, seeds[j])
              for j in range(20)]
  policy = compute_update(policy, ray.get(rewards), seeds)
```

Performance of RL applications



(a) Evolution Strategies



(b) PPO