Ray:
A Unified Distributed Framework for Emerging AI Applications

CS675: Distributed Systems (Spring 2020)

Lecture 11

Yue Cheng

Supervised Learning

- One prediction

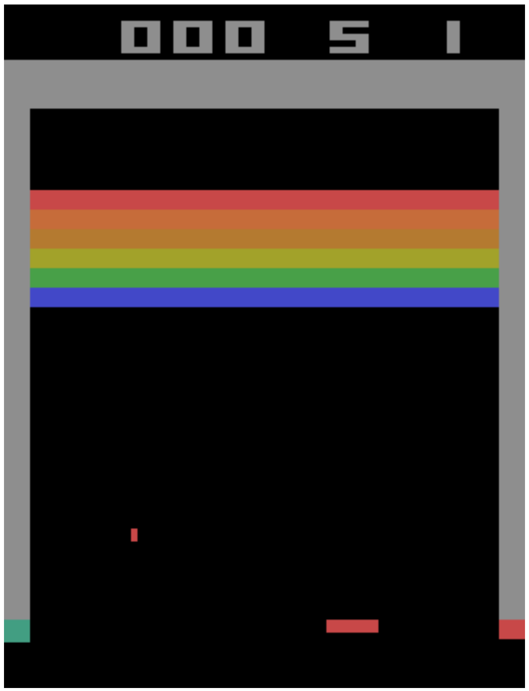
- Static environment

- Immediate feedback

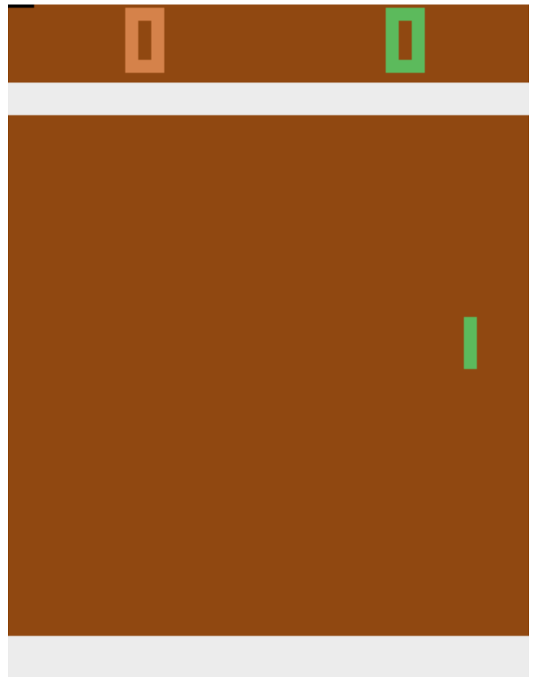
Supervised Learning → Reinforcement Learning (RL)

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

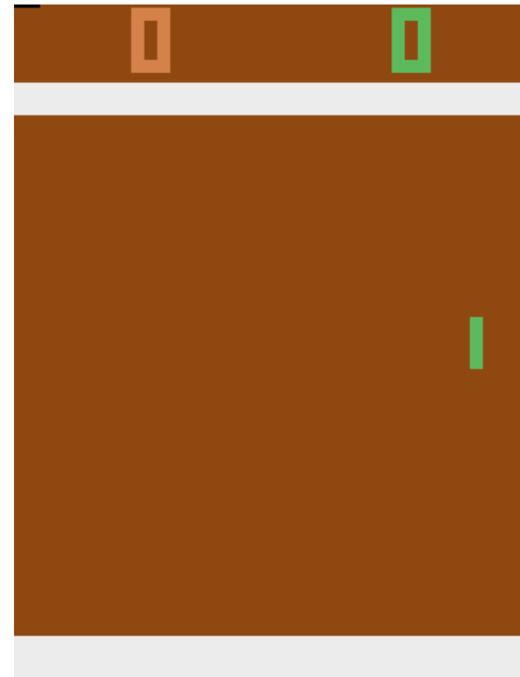
Reinforcement learning



Atari breakout



Pong: after 30 mins of training



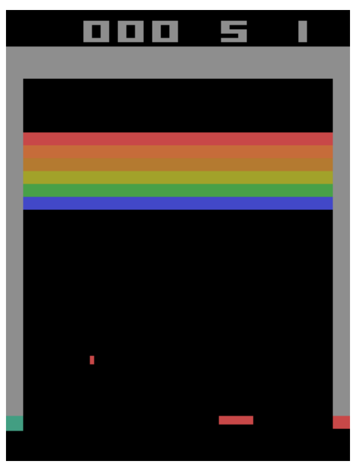
Pong: DQN wins like a boss

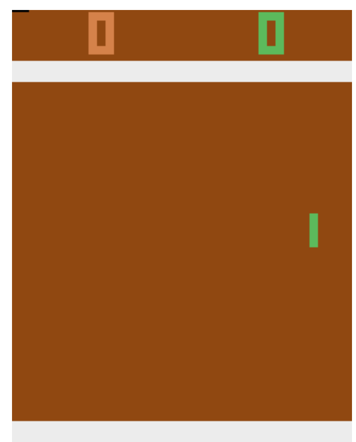
*: Playing Atari with Deep Reinforcement Learning: https://arxiv.org/abs/1312.5602

RL application pattern

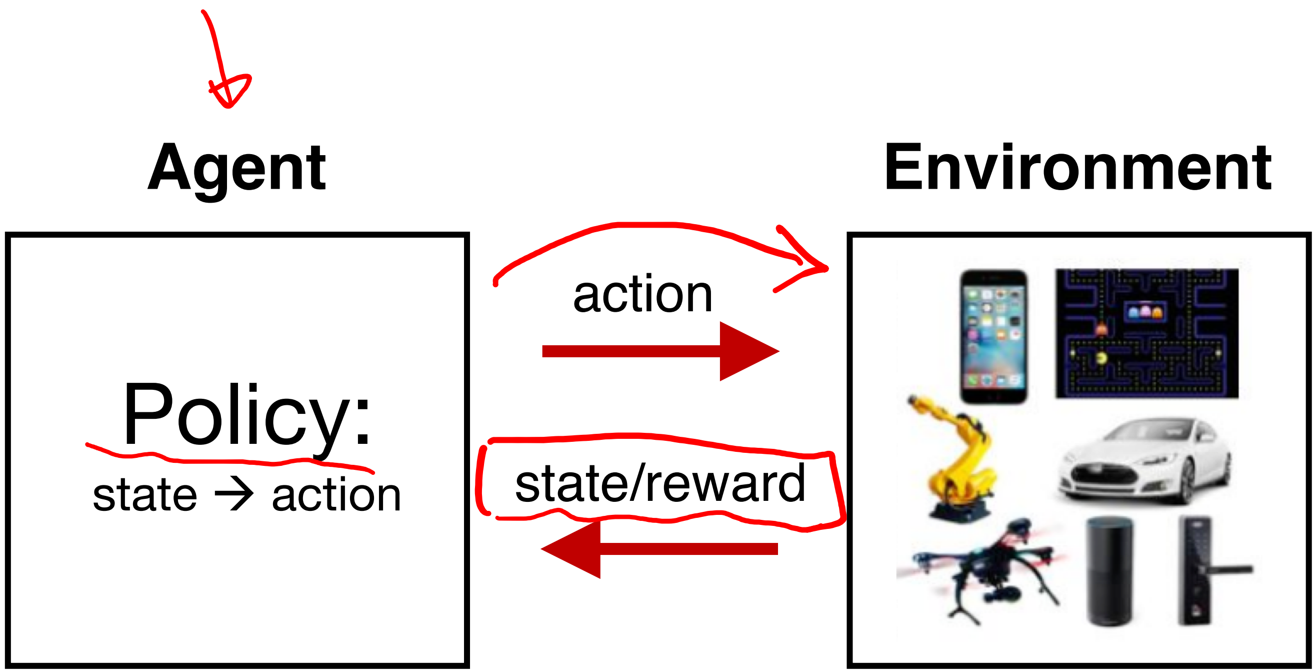
- Process inputs from different sensors in parallel & real-time

- Execute large number of simulations, e.g., up to 100s of millions

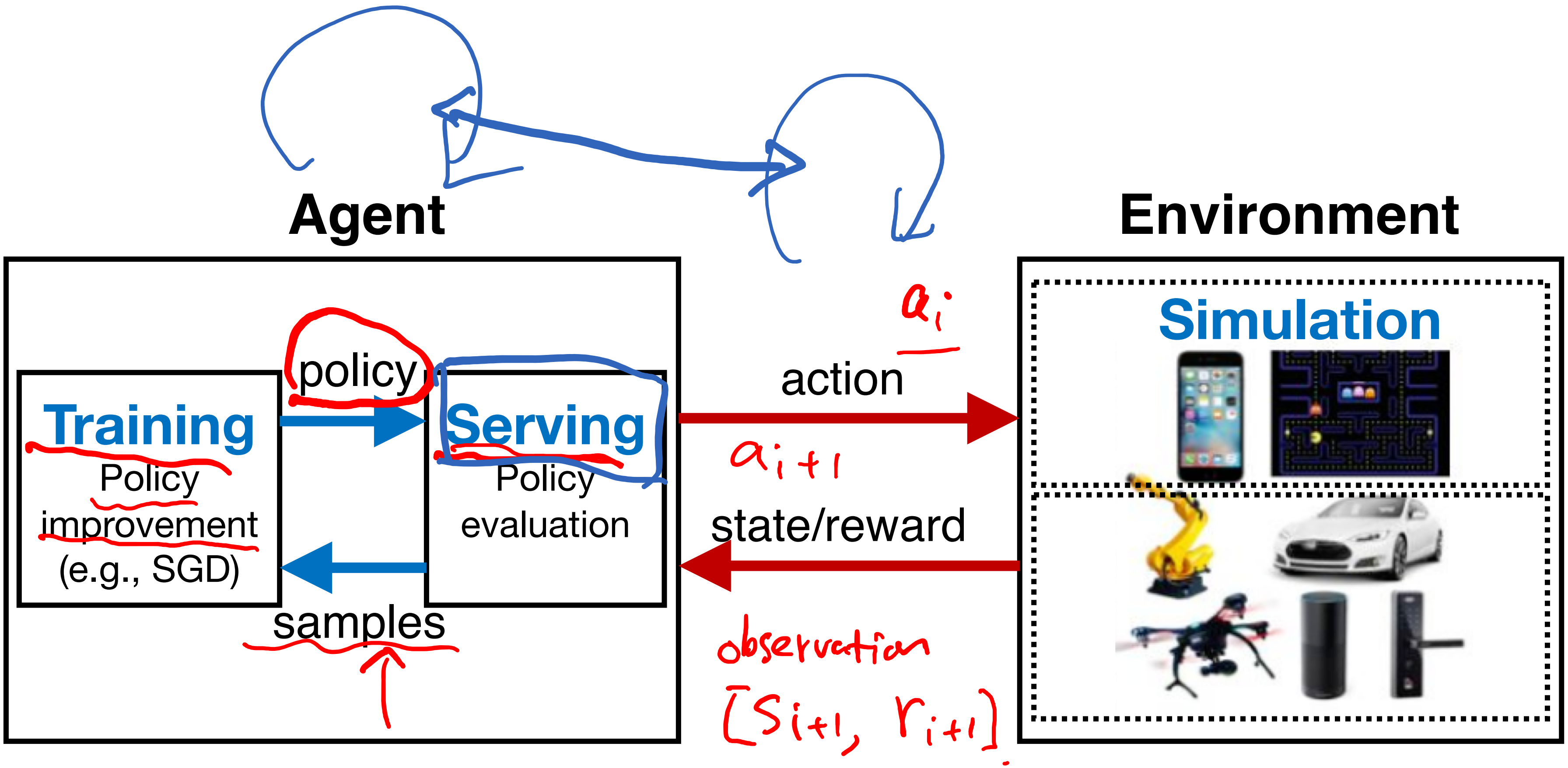




RL setup


Agent
Environment
action
Policy:
state → action
state/reward

RL setup in more detail



trajectory: $s_0, (s_1, r_1), (s_2, r_2) \ldots$

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

Simulations.

length $\begin{cases} \text{ms} \\ \downarrow \\ \text{mins.} \end{cases}$

- Need to handle dynamic task graphs, where tasks have
  - Heterogeneous durations
  - Heterogenous computations

Training: Compute-intensive.

throughput requirement

- Schedule millions of tasks / sec

Serving: high-throughput $\boxed{\text{low-latency}}$

- Make it easy to parallelize ML algorithms (often written in Python)

Challenges for cross-cutting Apps.

The ML/AI ecosystems today

RL

action

chess/go

states/rewards

Distributed systems

Distributed training

TensorFlow, PyTorch, MXNet

Distributed systems

Model serving

Clipper, TensorFlow serving

Distributed systems

Data processing

Spark, Hadoop, Dask

Distributed systems

Simulation

MPI, simulators, custom tools

Distributed systems

Data streaming

Flink, many others

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

Ad hoc integrations are difficult to manage and program!

Ray API

handle to the result objs.

Stateless

Tasks

Python func.

execute remotely

①

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

in-memory obj store.

Stateful

Actors

Python

Erlang.

②

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

manipulate state

blocking.

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

vector of multiple objs.

Ray API examples

- See separate notes

Computation model

model serving component

1,000 steps

Vertices:

: Tasks/Actors.

: Data

Edge:

: Control edge

: Data edge

: Stateful edge

Task

Actor

Actor

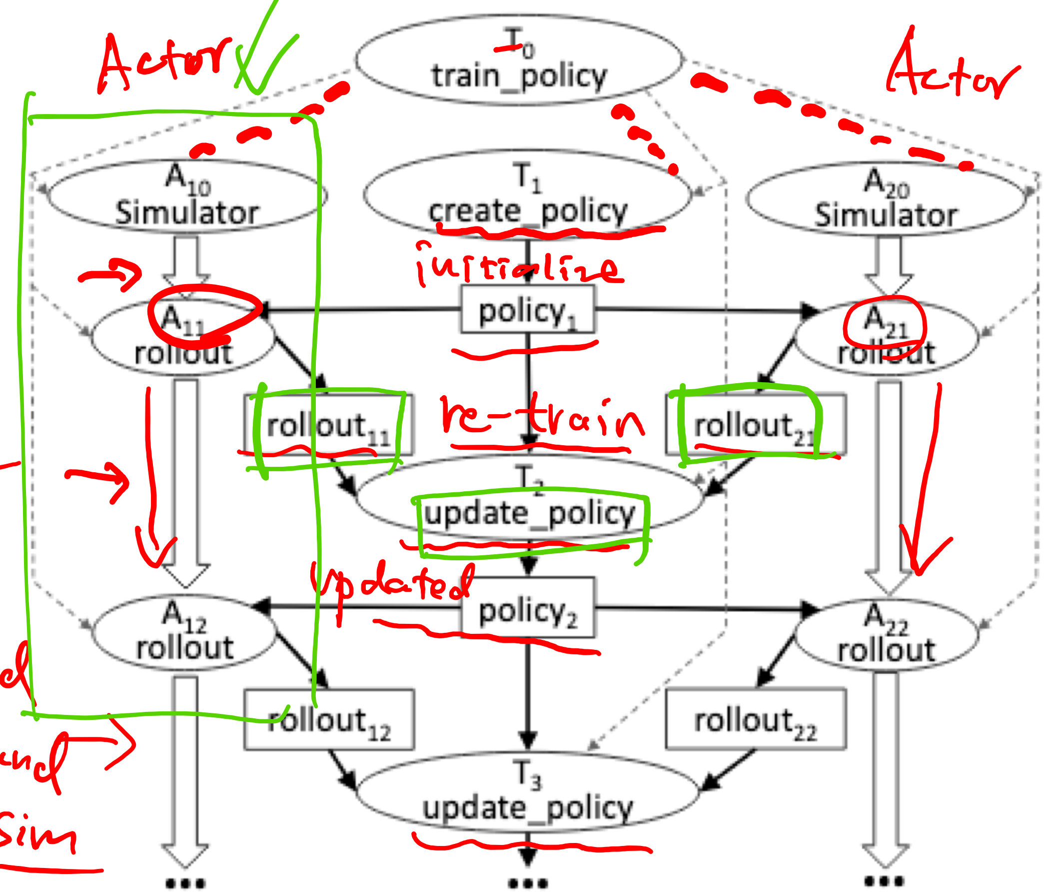
1st round of sim

initialize

re-train

updated

2nd round sim



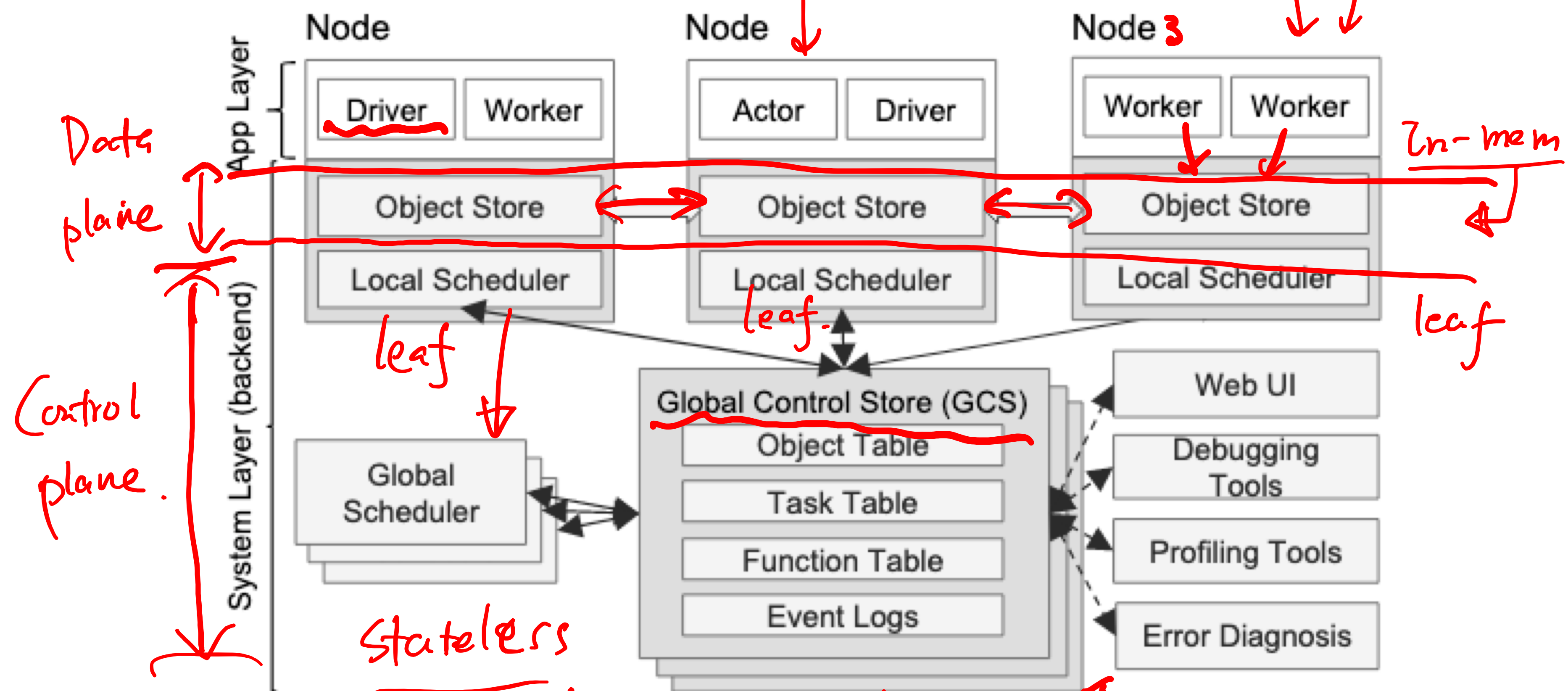
Ray architecture

Plasma (Apache Arrow)

shared mem

Actor running in

Actors Tasks



bottom-up scheduler.

Metadata store

pub/sub

Global control store (GCS)

- Object table

  → lists of objects
  their locations.

  Hadoop (namenode)

- Task table

  → lineage graph. { tasks created
  edges.

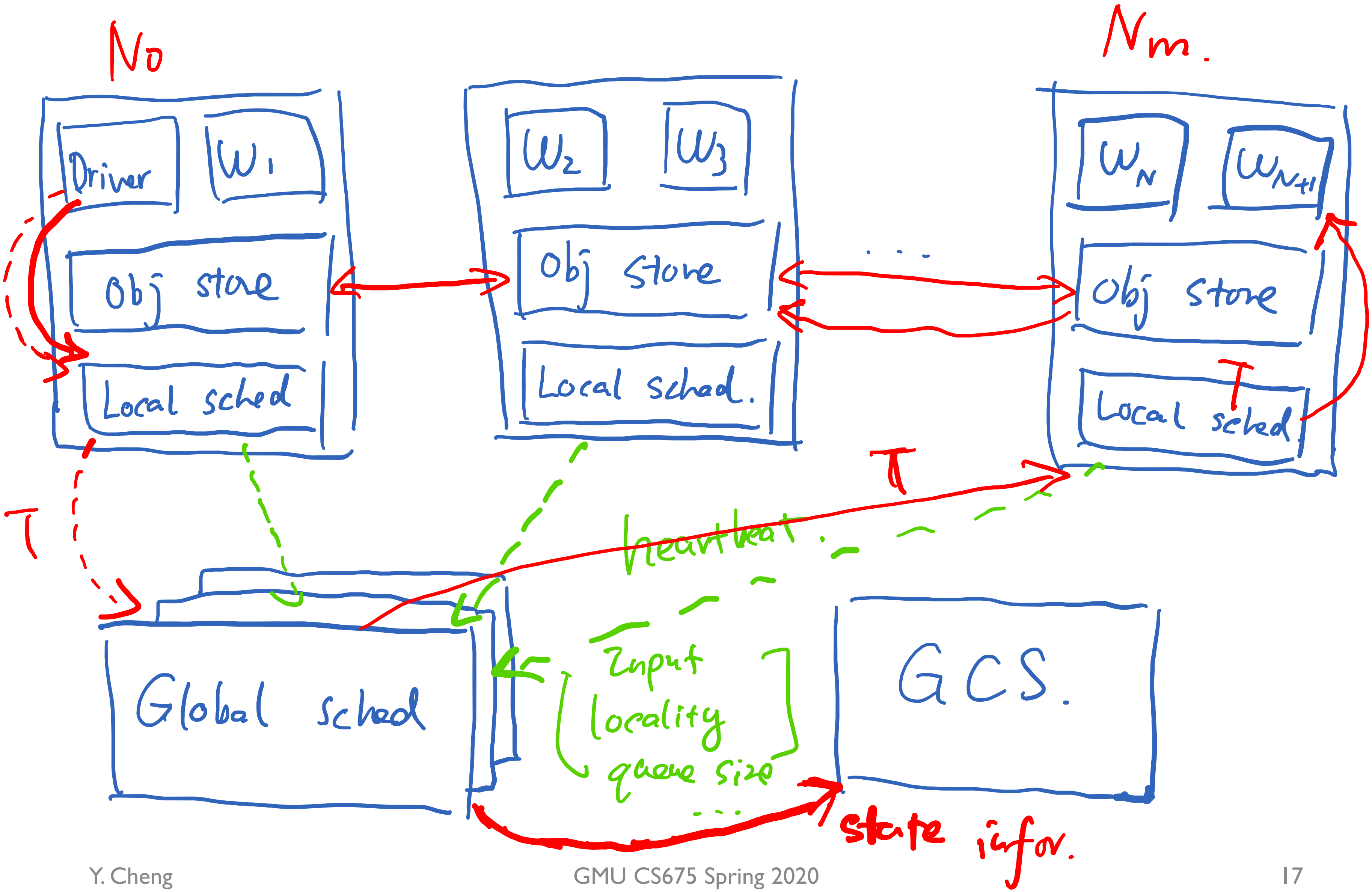
  Spark.

- Function table

  → Code blocks. { Tasks.
  Actors.

  Intermediate objects
  ↓
  Obj store.

Ray scheduler



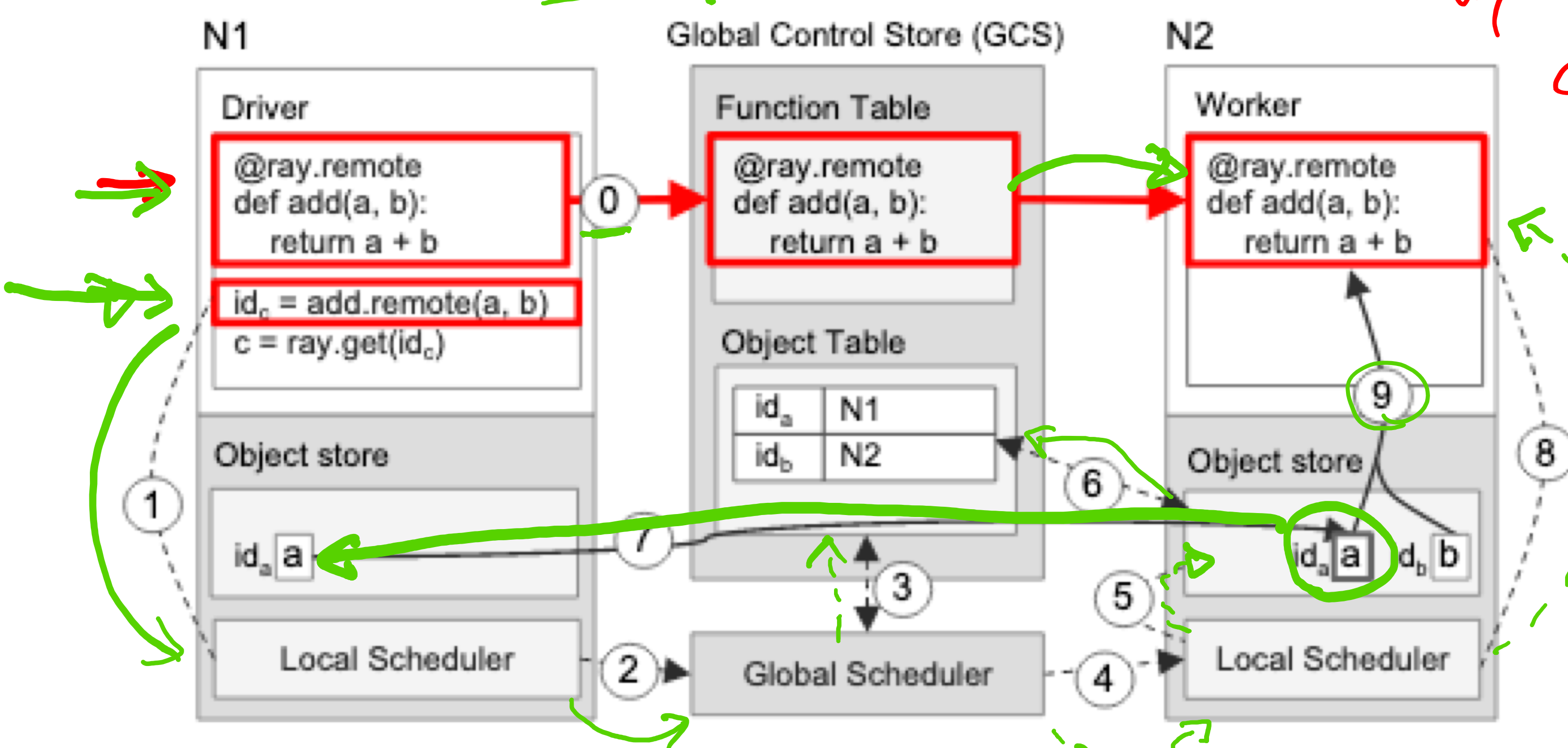
Fault tolerance

- Tasks Stateless.
  - ↳ Lineage graph (GCS).
- Actors
  - ↳ User-defined checkpointing.
- GCS
  - ↳ shards.
  - Replica: (Master : slaves.) Primary-Backup. CR.
- Scheduler Stateless.

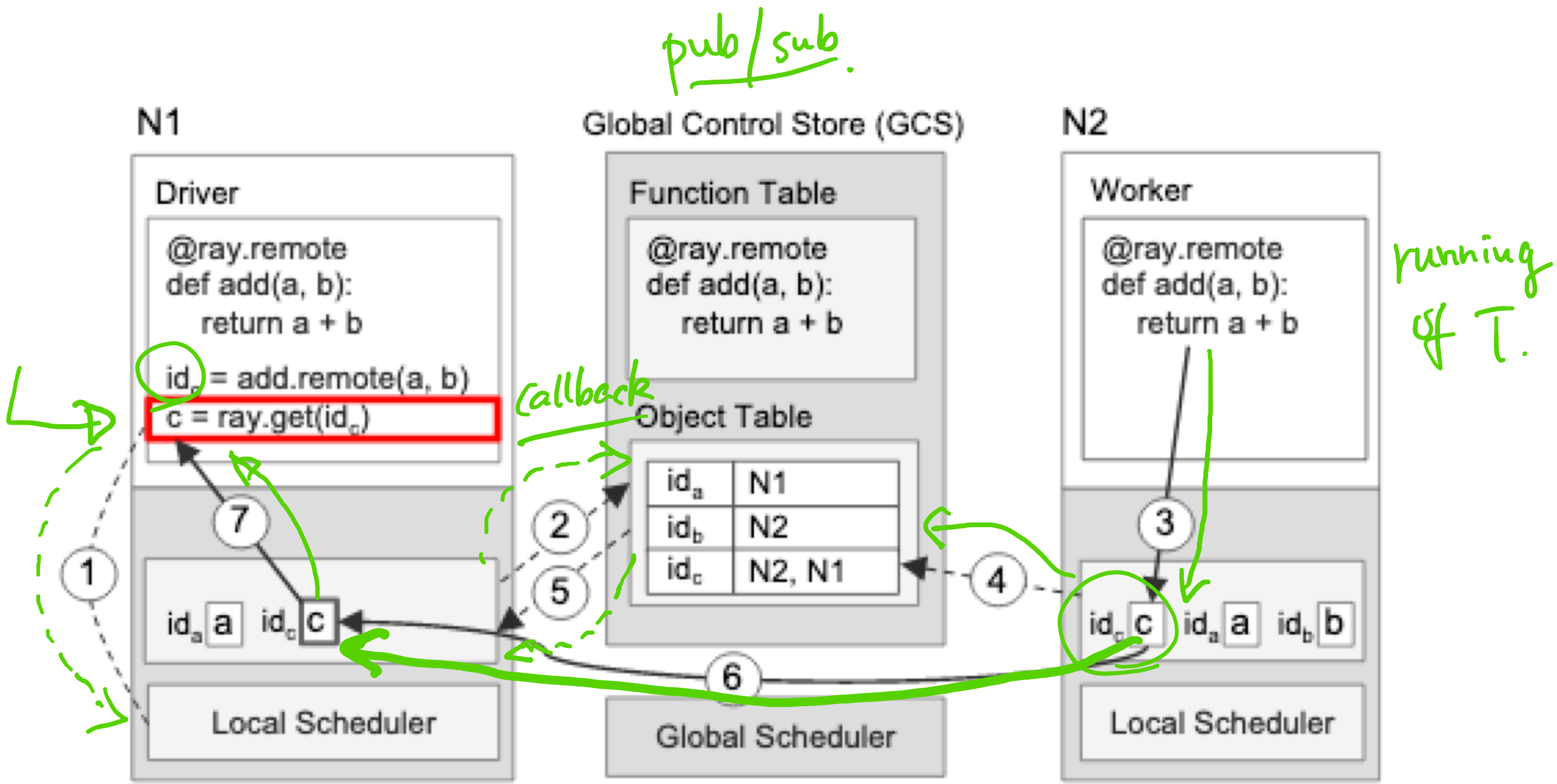
Executing a task remotely

much runtime overhead w/ RT comm.

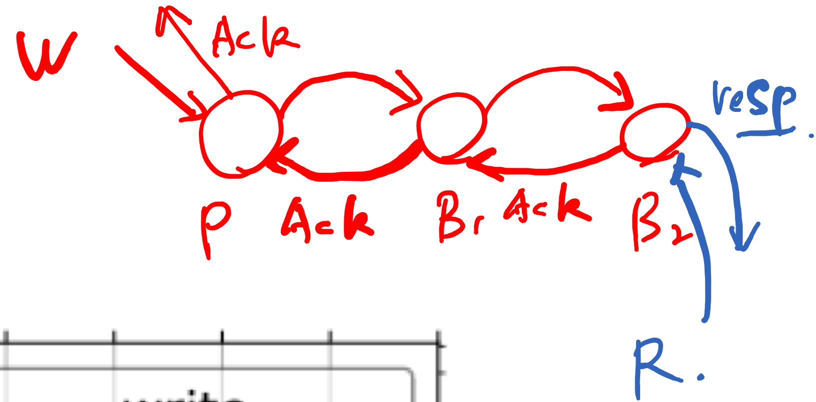
Register pub/sub.

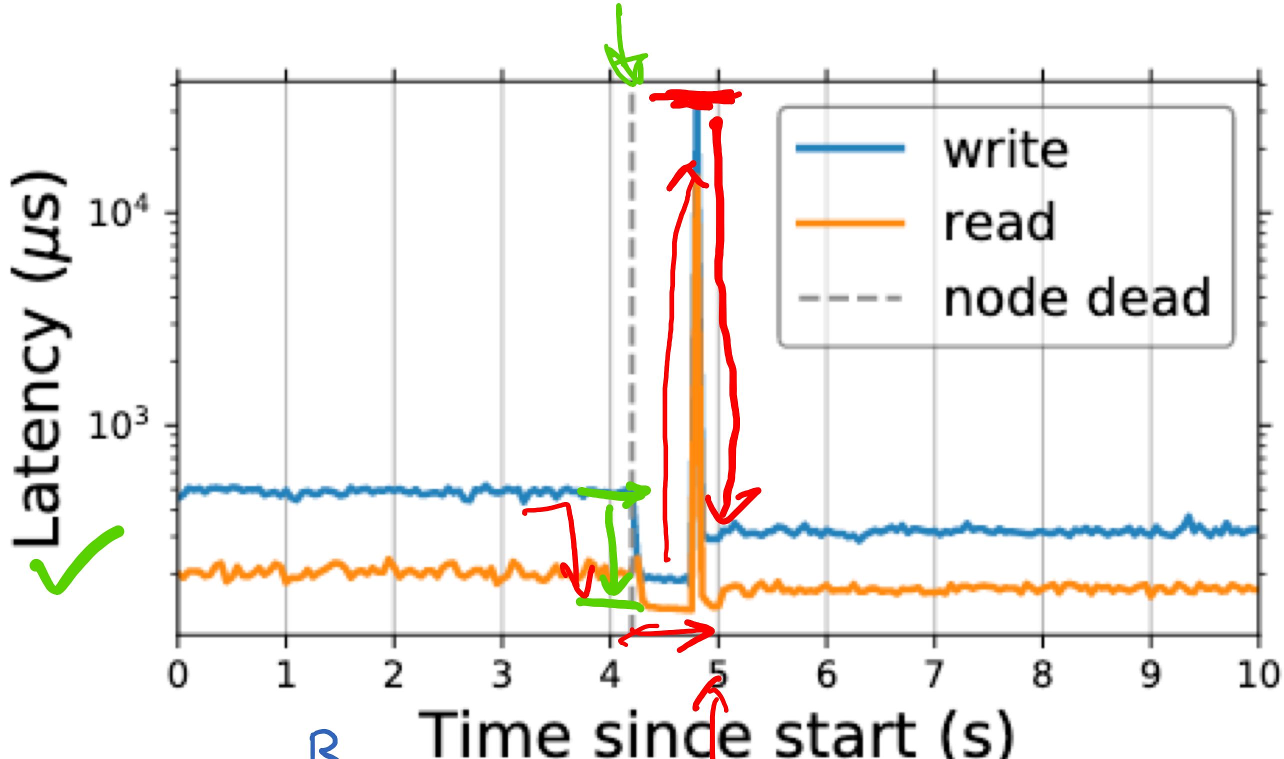


Returning the results of a remote task



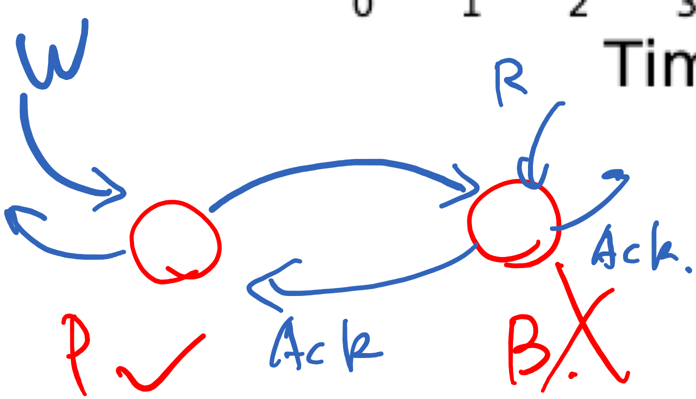
GCS fault tolerance



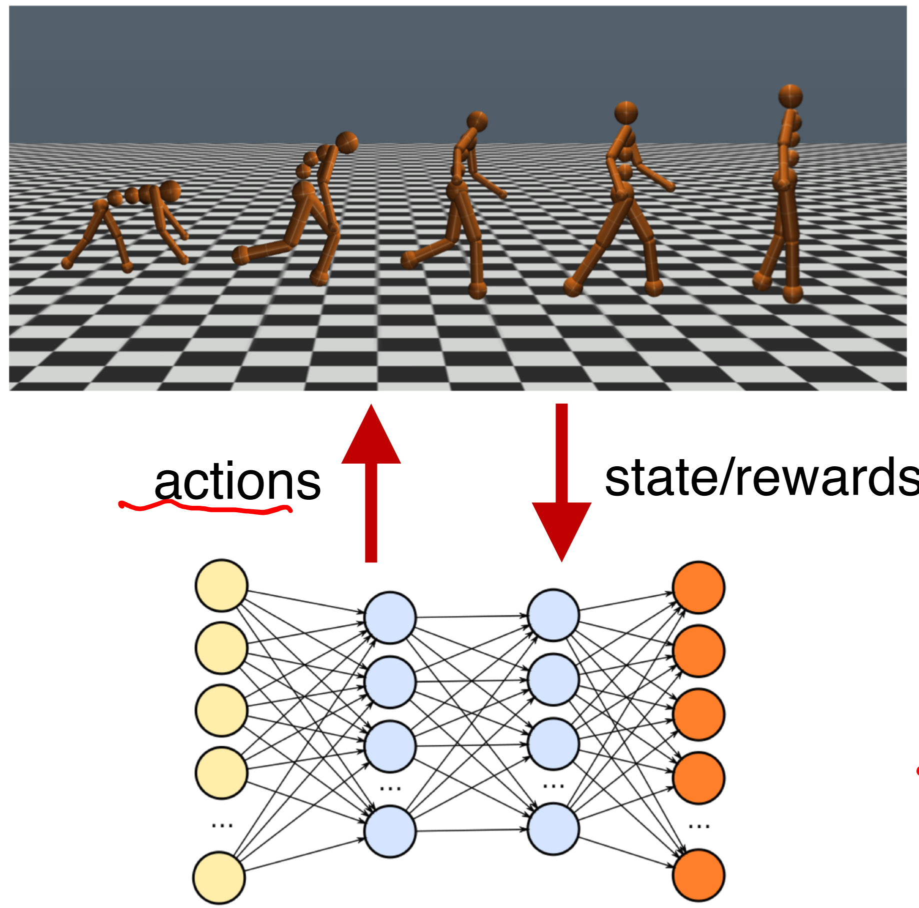


new node joins

Chain Replication.



Evolution strategies (ES)



Simulator

Policy

Brain

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)
```

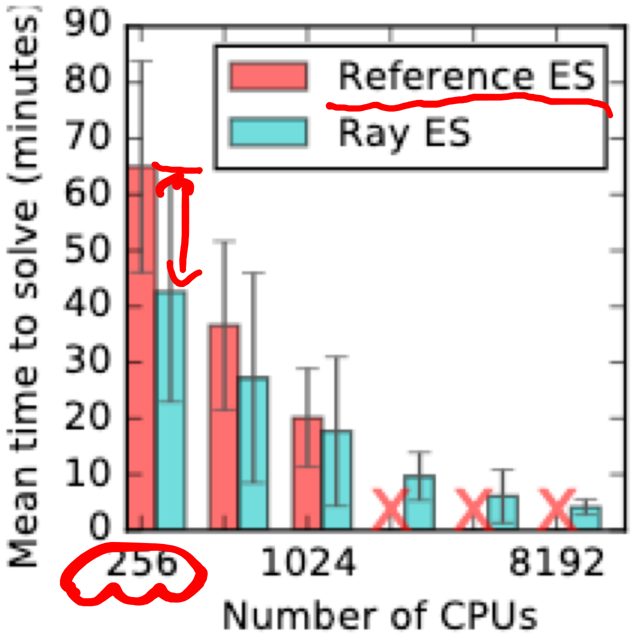
200 steps.

old

Pseudocode

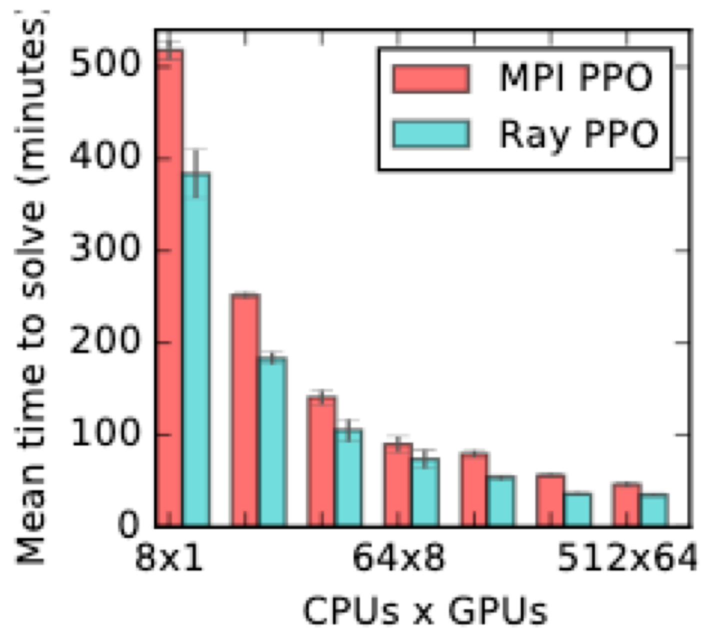
```
@ray.remote
class Worker(object):
  def do_simulation(policy, seed):
    # perform simulation and return reward

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

ES



(b) PPO

Further discussion

- What part of the Ray paper excites you and disappoints you the most?