Ray: A Distributed Execution Framework for Emerging AI Applications

Presenters: Philipp Moritz, Robert Nishihara

Spark Summit West
June 6, 2017


Berkeley
UNIVERSITY OF CALIFORNIA
BERKELEY

Why build a new system?

Supervised Learning

Data point

Model

Label

“CAT”

Emerging AI Applications

I need a hotel in San Francisco next week.

AI: What are the dates you want to go?

Next Monday through Thursday.

AI: Do you need to rent a car? I don't see a reservation.

No, taking BART and Uber.

AI: In that case, stay in SoMa since your meetings are all in that area. What's the budget?

Probably like $400 a night.

AI: The W gets good reviews from other software developers on TripAdvisor and has a promotion for $369/night.

Ok, book it.

AI: All set, I have you booked into a deluxe king bed room from Monday the 1st through Thursday the 4th. Your Starwood number has been linked to the reservation and I charged it to your corporate credit card.

Emerging AI Applications

I need a hotel in San Francisco next week.

AI: What are the dates you want to go?

Next Monday through Thursday.

AI: Do you need to rent a car? I don't see a reservation.

No, taking BART and Uber.

AI: In that case, stay in SoMa since your meetings are all in that area. What's the budget?

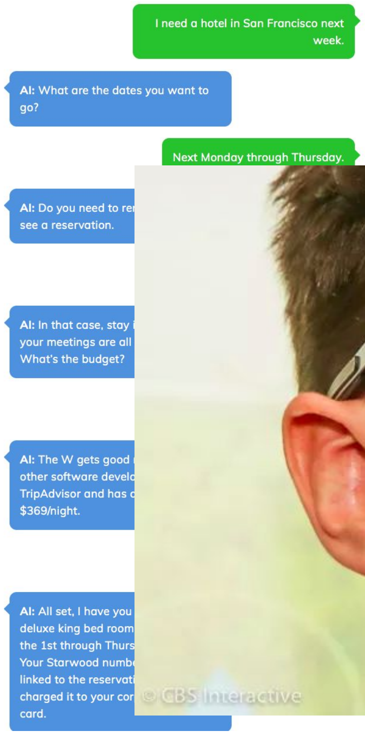
Probably like $400 a night.

AI: The W gets good reviews from other software developers on TripAdvisor and has a promotion for $369/night.

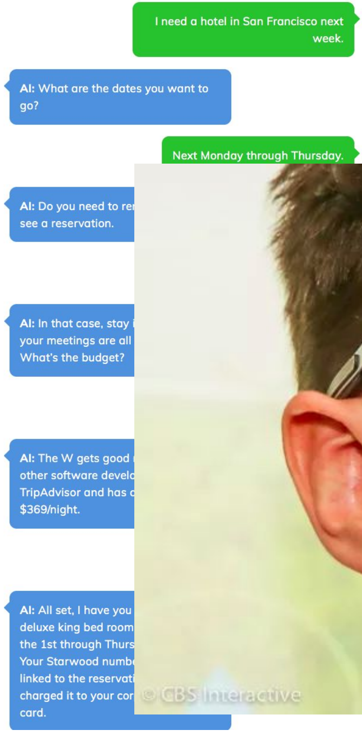
Ok, book it.

AI: All set, I have you booked into a deluxe king bed room from Monday the 1st through Thursday the 4th. Your Starwood number has been linked to the reservation and I charged it to your corporate credit card.

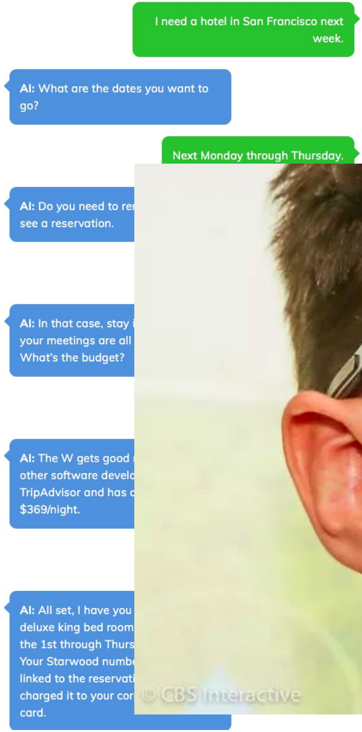
Emerging AI Applications



Emerging AI Applications



Emerging AI Applications



Supervised Learning → Reinforcement Learning

Supervised Learning → Reinforcement Learning

- One prediction →
- Sequences of actions

Supervised Learning → Reinforcement Learning

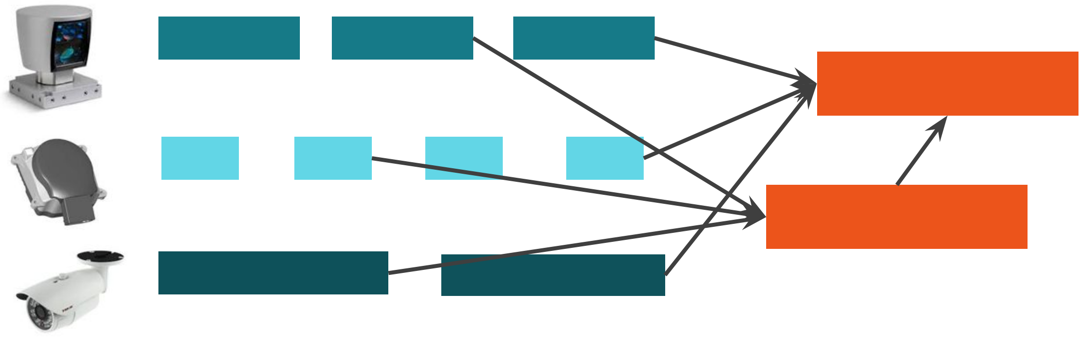
● One prediction	→	● Sequences of actions
● Static environments	→	● Dynamic environments

Supervised Learning → Reinforcement Learning

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

RL Application Pattern

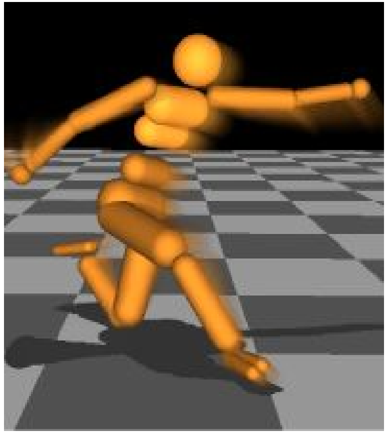
Process inputs from different sensors in parallel & real-time

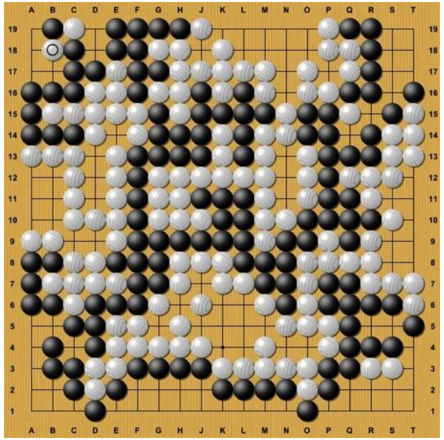


RL Application Pattern

Process inputs from different sensors in parallel & real-time

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



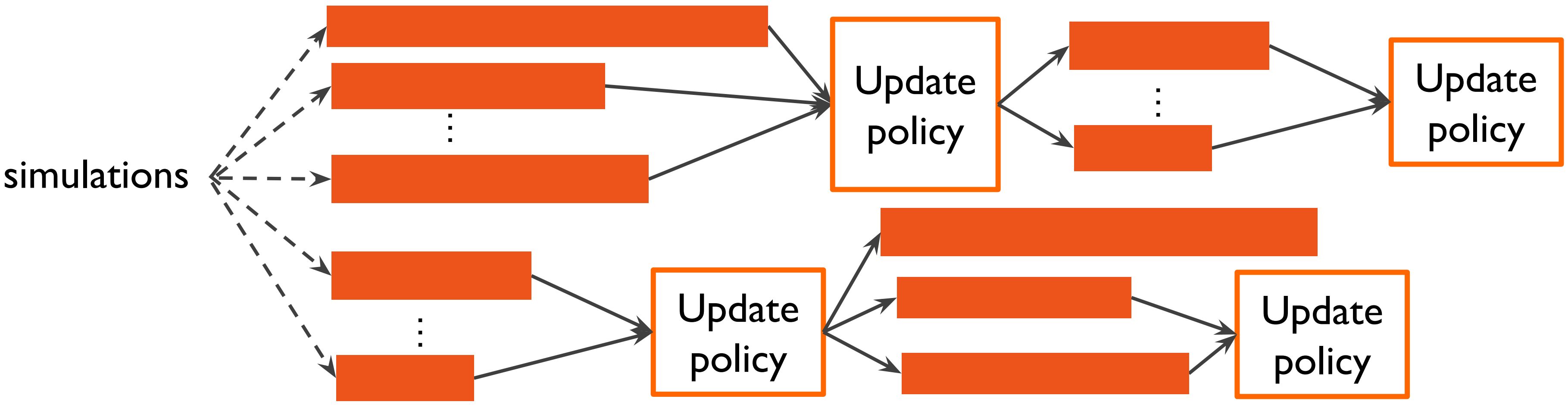


RL Application Pattern

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 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)

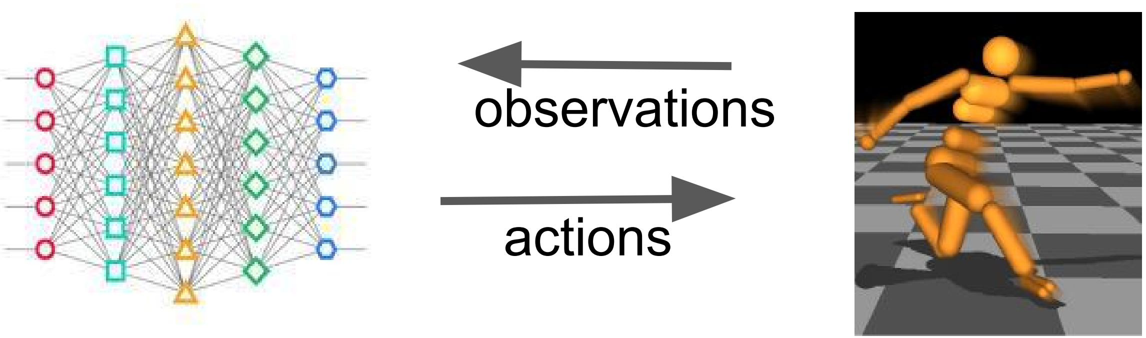
RL Application Pattern

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)

Often policies implemented by DNNs



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)

Often policies implemented by DNNs

Most RL algorithms developed in Python

RL Application Requirements

Need to handle dynamic task graphs, where tasks have

- Heterogeneous durations
- Heterogeneous computations

Schedule millions of tasks/sec

Make it easy to parallelize ML algorithms written in Python

Ray API - remote functions

```python
def zeros(shape):
    return np.zeros(shape)


def dot(a, b):
  return np.dot(a, b)
```

Ray API - remote functions

```
@ray.remote
def zeros(shape):
    return np.zeros(shape)

@ray.remote
def dot(a, b):
  return np.dot(a, b)
```

Ray API - remote functions

```python
@ray.remote
def zeros(shape):
    return np.zeros(shape)

@ray.remote
def dot(a, b):
  return np.dot(a, b)

id1 = zeros.remote([5, 5])
id2 = zeros.remote([5, 5])
id3 = dot.remote(id1, id2)
ray.get(id3)
```

Ray API - remote functions

```python
@ray.remote
def zeros(shape):
    return np.zeros(shape)

@ray.remote
def dot(a, b):
  return np.dot(a, b)

id1 = zeros.remote([5, 5])
id2 = zeros.remote([5, 5])
id3 = dot.remote(id1, id2)
ray.get(id3)
```

- **Blue** variables are Object IDs.

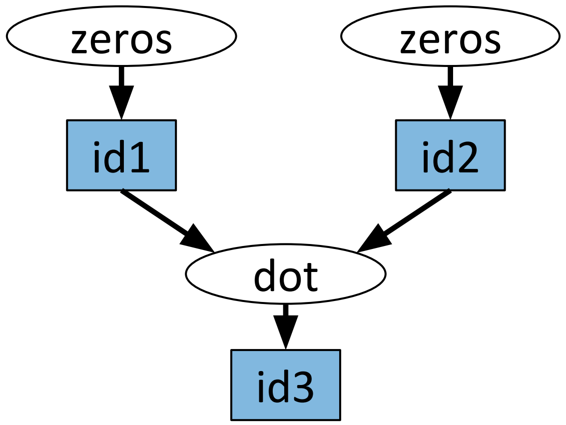
Ray API - remote functions

```python
@ray.remote
def zeros(shape):
    return np.zeros(shape)

@ray.remote
def dot(a, b):
  return np.dot(a, b)

id1 = zeros.remote([5, 5])
id2 = zeros.remote([5, 5])
id3 = dot.remote(id1, id2)
ray.get(id3)
```

- **Blue** variables are Object IDs.



Ray API - actors

```python
class Counter(object):
    def __init__(self):
        self.value = 0
    def inc(self):
        self.value += 1
        return self.value

c = Counter()
c.inc()  # This returns 1
c.inc()  # This returns 2
c.inc()  # This returns 3
```

Ray API - actors

```python
@ray.remote
class Counter(object):
    def __init__(self):
        self.value = 0
    def inc(self):
        self.value += 1
        return self.value

c = Counter.remote()
id1 = c.inc.remote()
id2 = c.inc.remote()
id3 = c.inc.remote()
ray.get([id1, id2, id3])  # This returns [1, 2, 3]
```

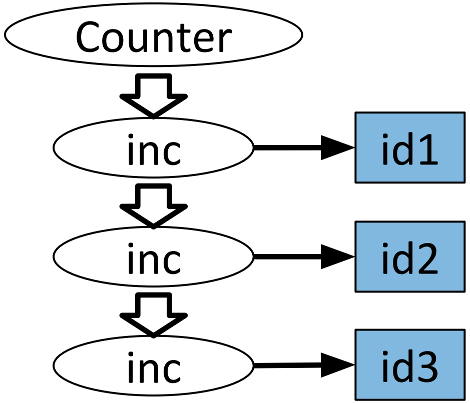
- State is shared between actor methods.
- Actor methods return **Object IDs**.

Ray API - actors

```python
@ray.remote
class Counter(object):
    def __init__(self):
        self.value = 0
    def inc(self):
        self.value += 1
        return self.value

c = Counter.remote()
id1 = c.inc.remote()
id2 = c.inc.remote()
id3 = c.inc.remote()
ray.get([id1, id2, id3])  # This returns [1, 2, 3]
```

- State is shared between actor methods.
- Actor methods return **Object IDs**.

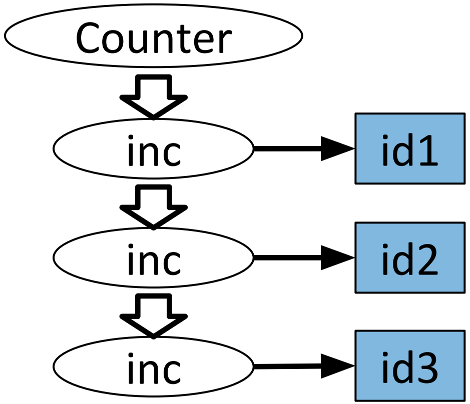


Ray API - actors

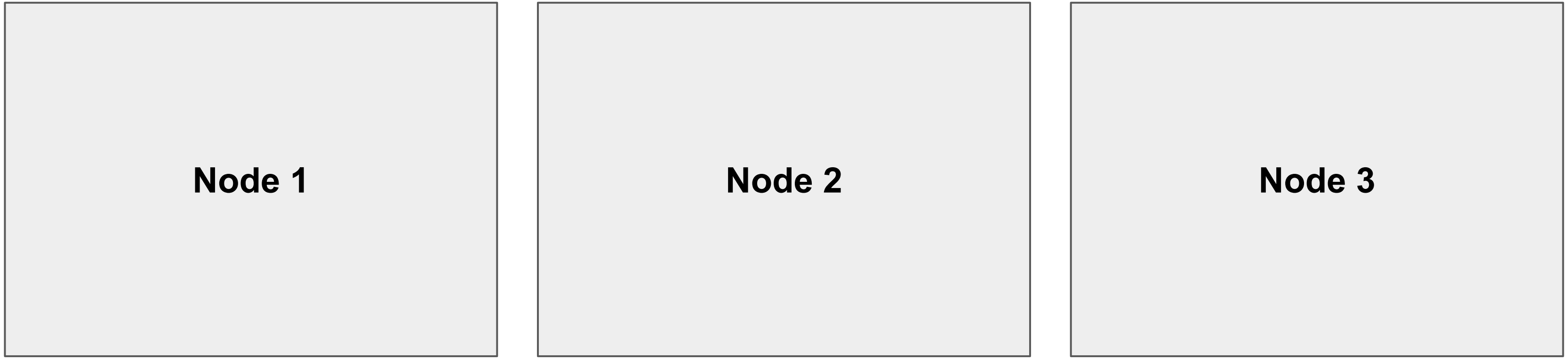
```
@ray.remote(num_gpus=1)
class Counter(object):
    def __init__(self):
        self.value = 0
    def inc(self):
        self.value += 1
        return self.value

c = Counter.remote()
id1 = c.inc.remote()
id2 = c.inc.remote()
id3 = c.inc.remote()
ray.get([id1, id2, id3])  # This returns [1, 2, 3]
```

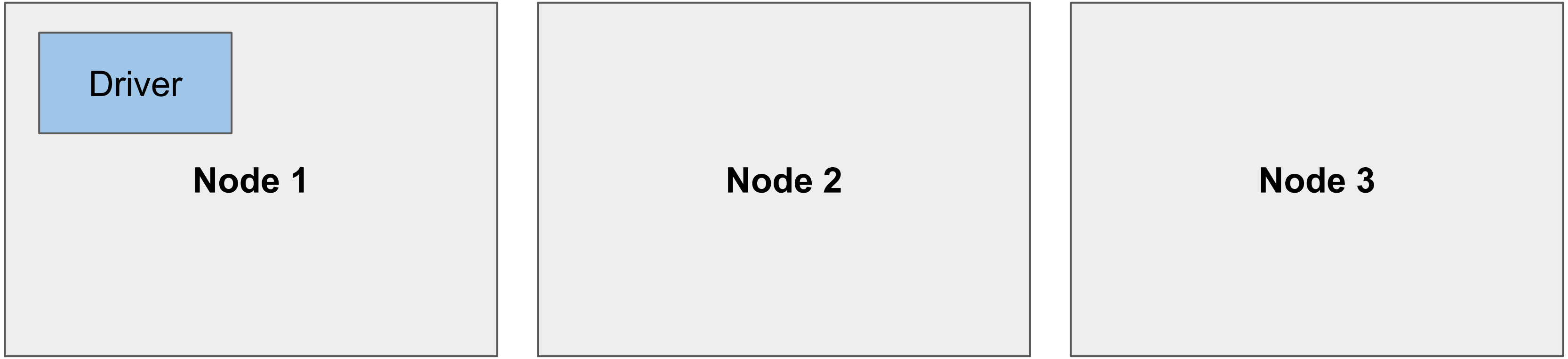
- State is shared between actor methods.
- Actor methods return **Object IDs**.
- Can specify **GPU** requirements



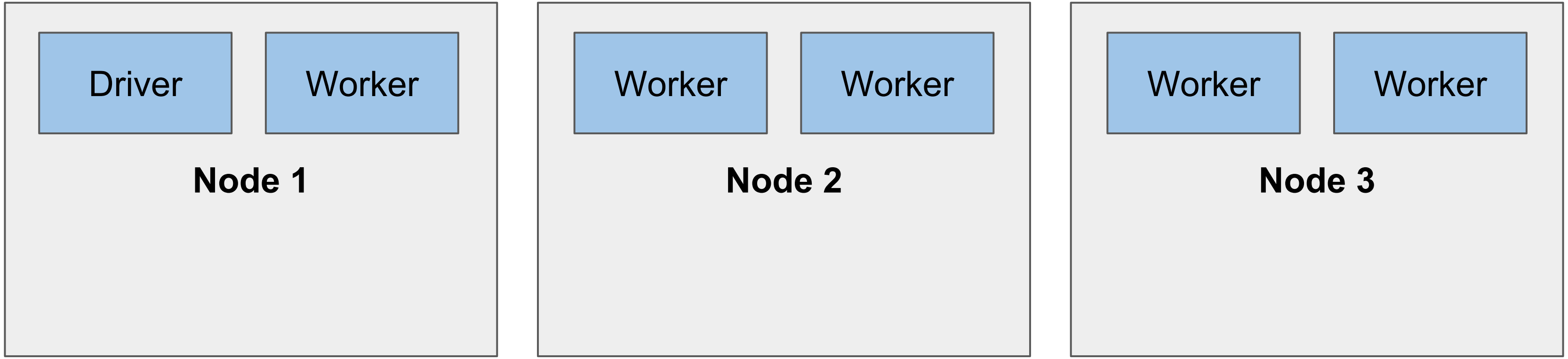
Ray architecture



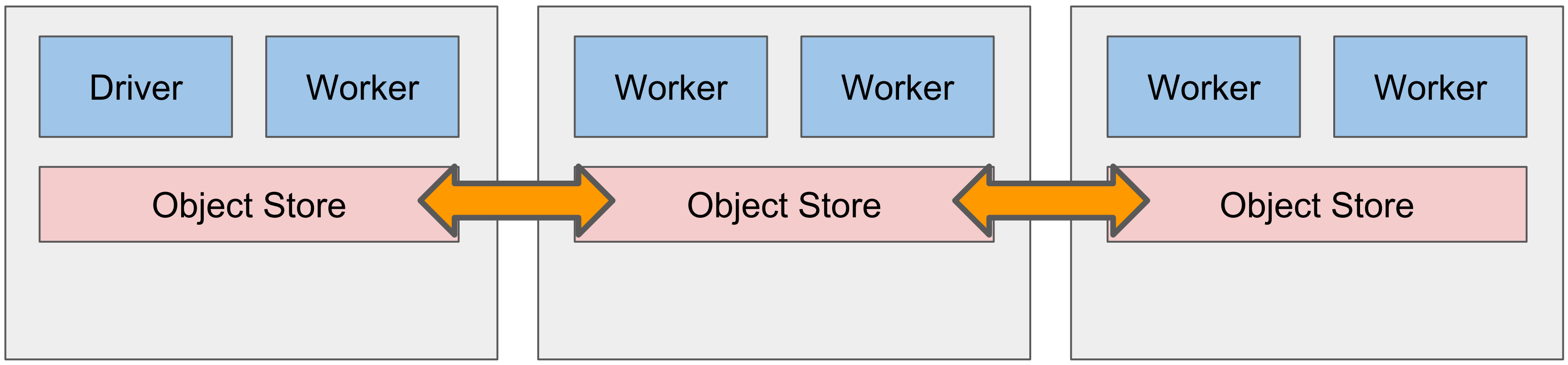
Ray architecture



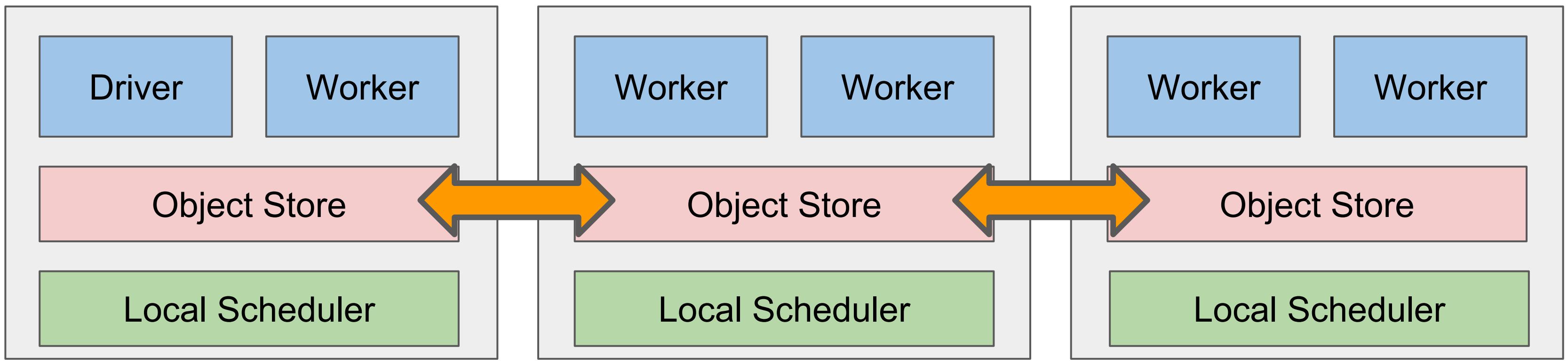
Ray architecture



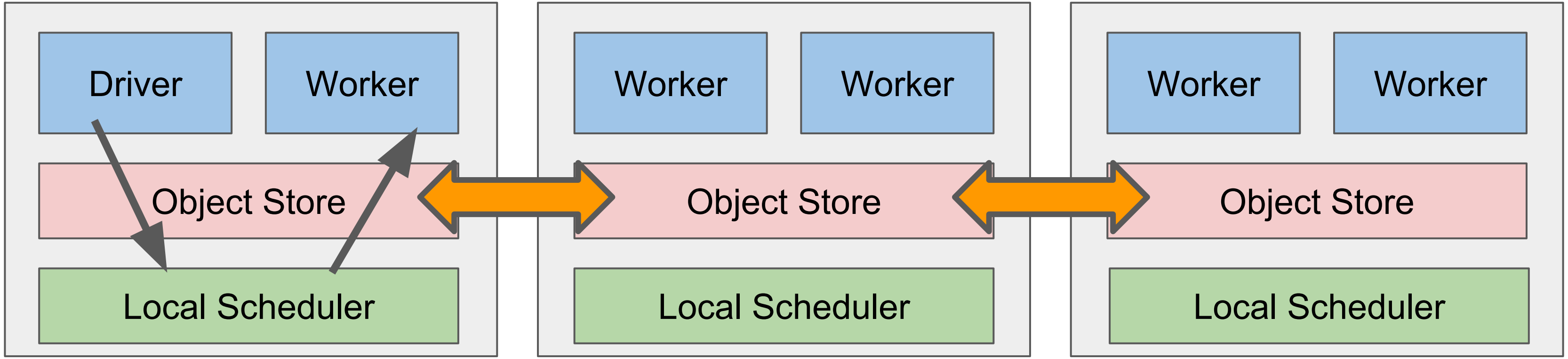
Ray architecture



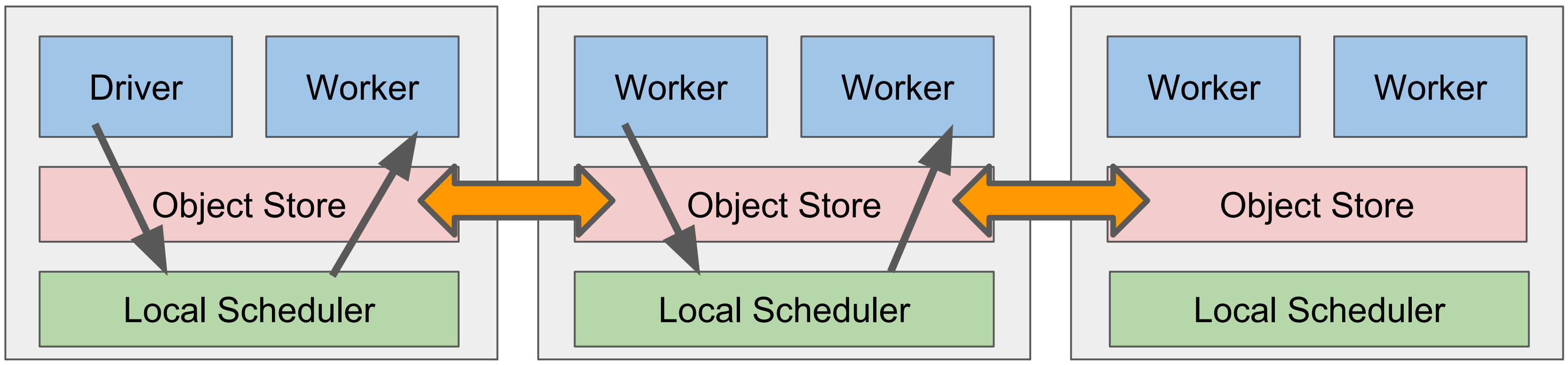
Ray architecture



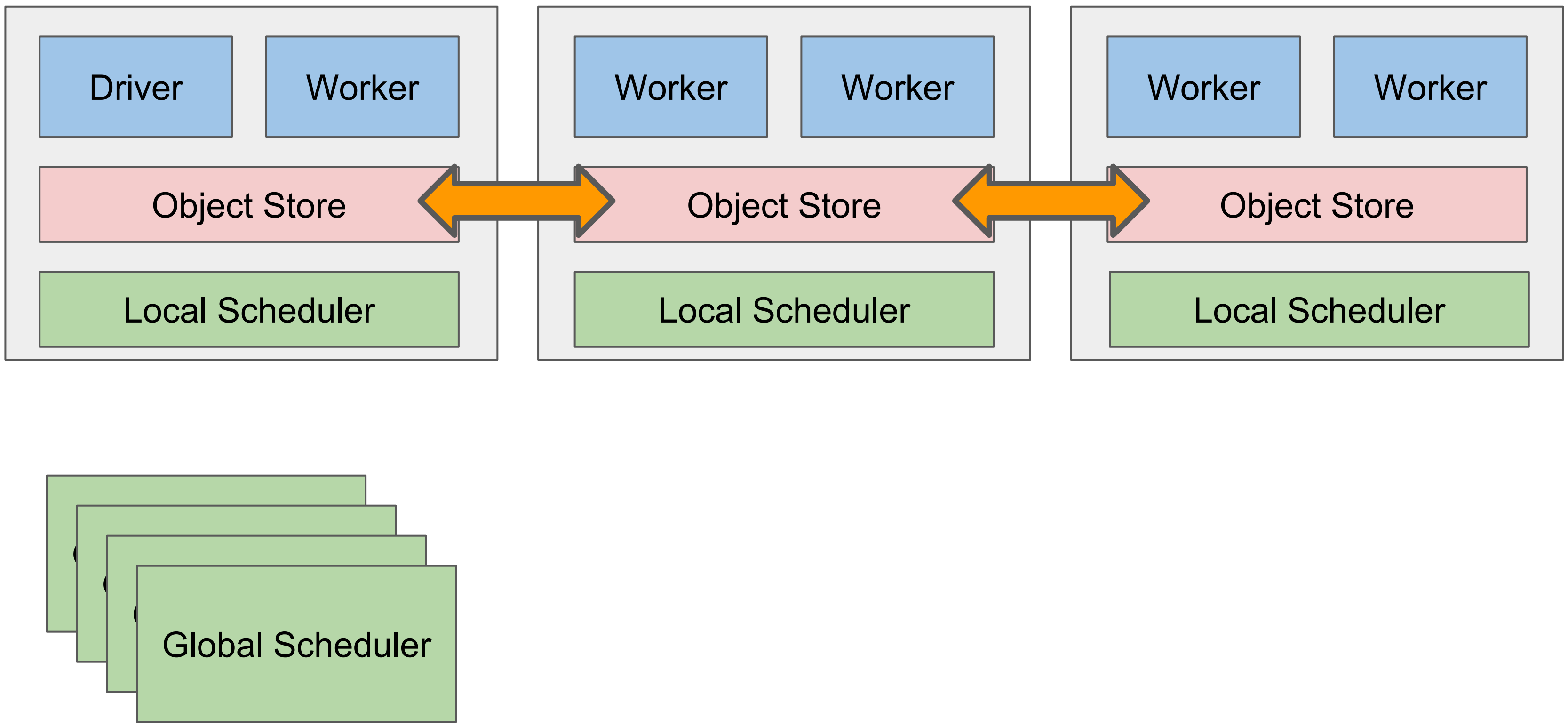
Ray architecture



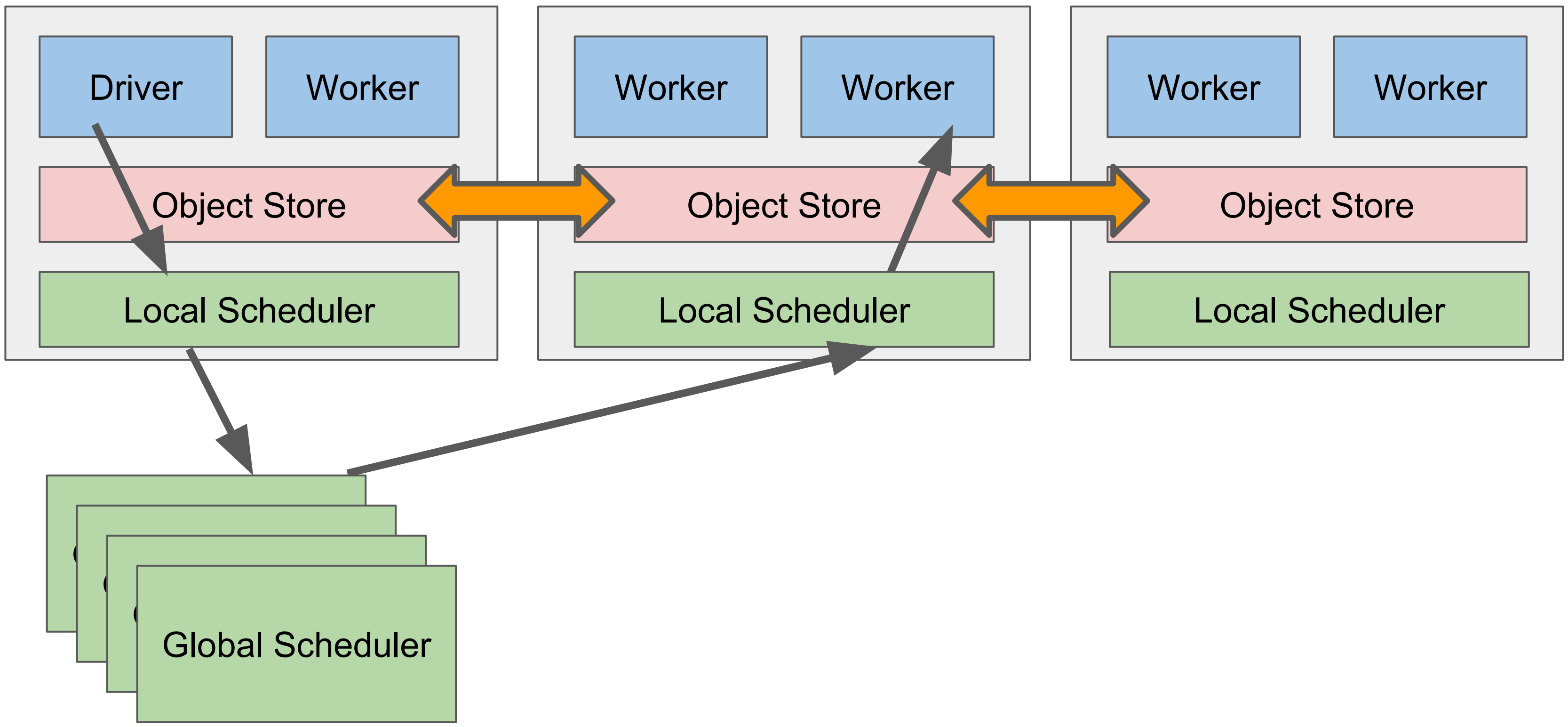
Ray architecture



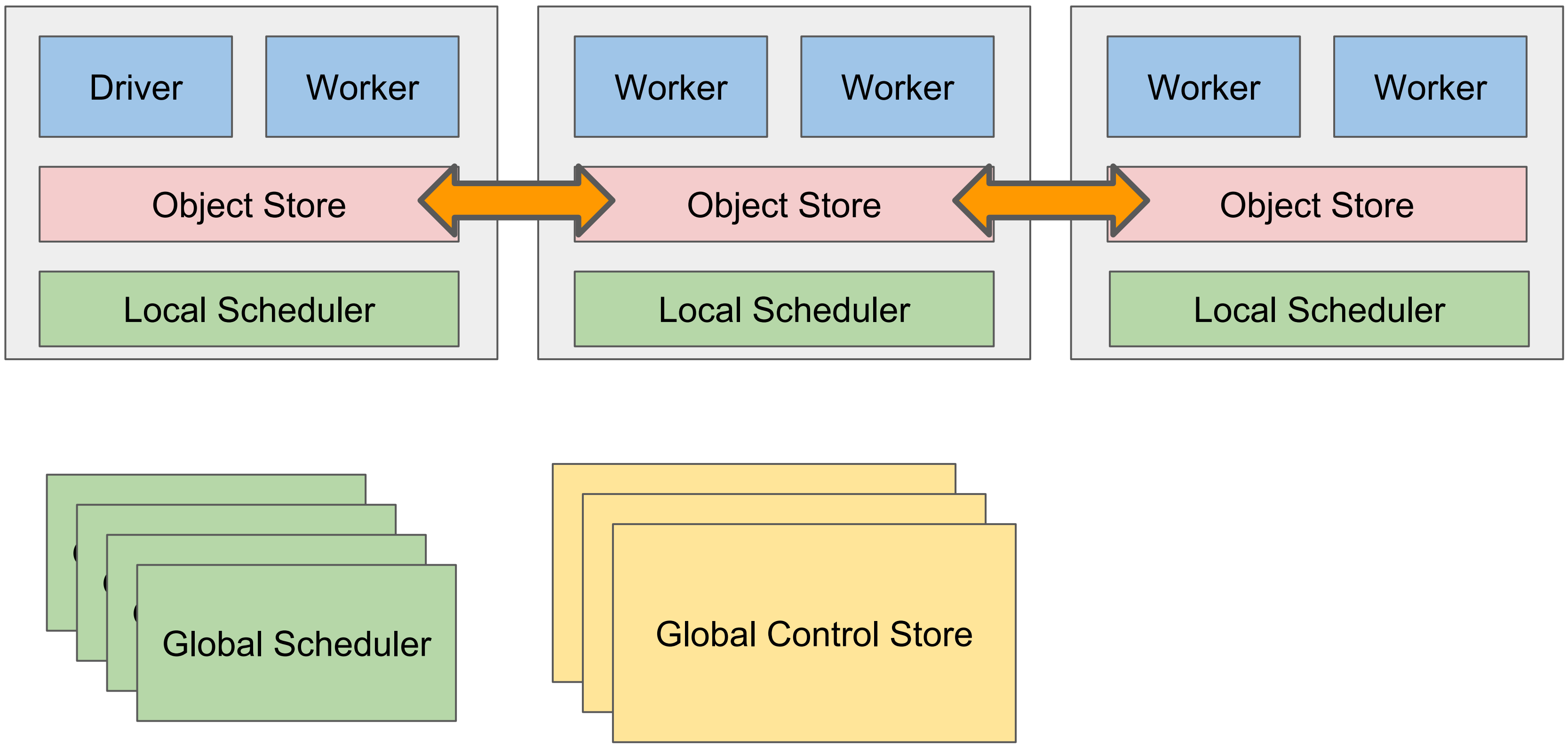
Ray architecture



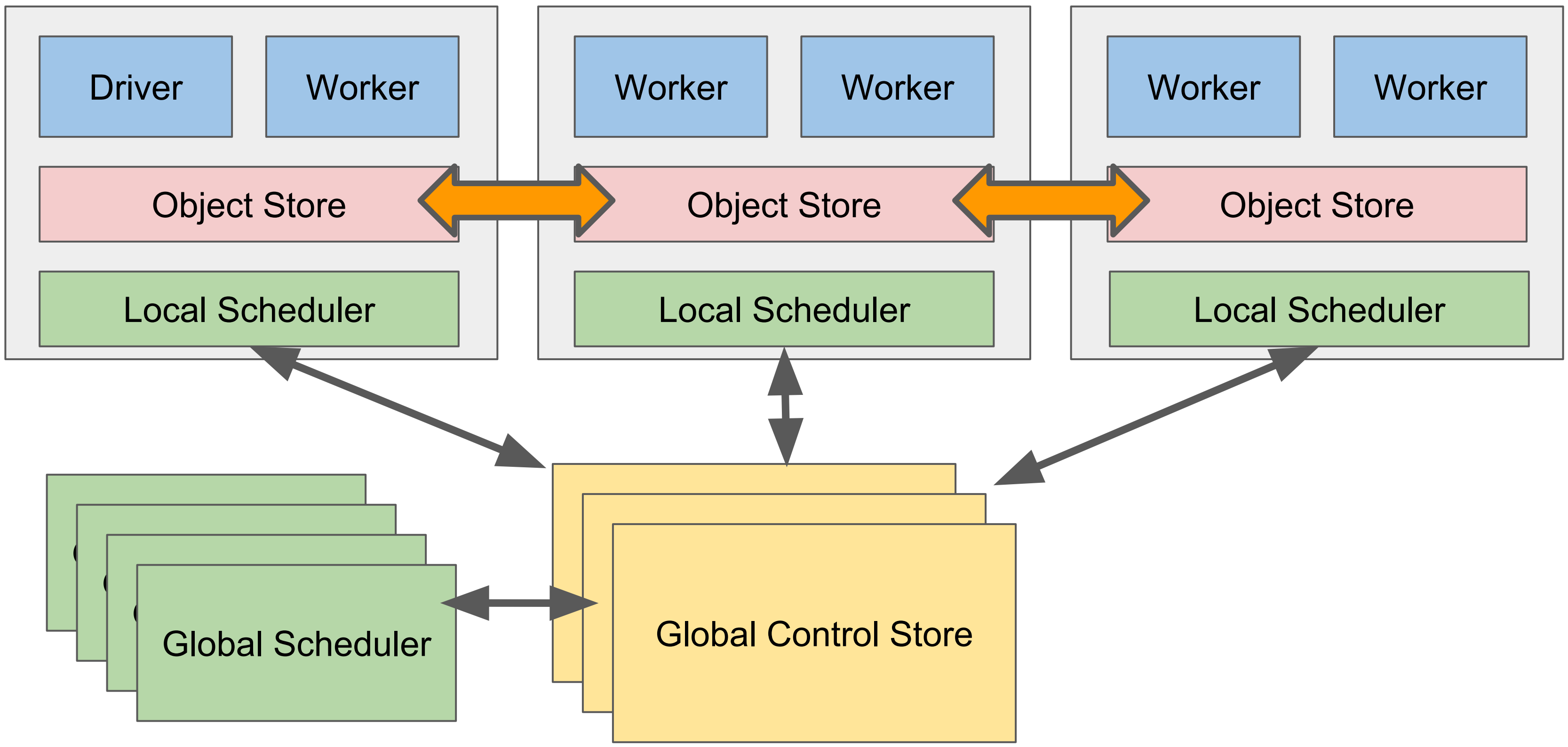
Ray architecture



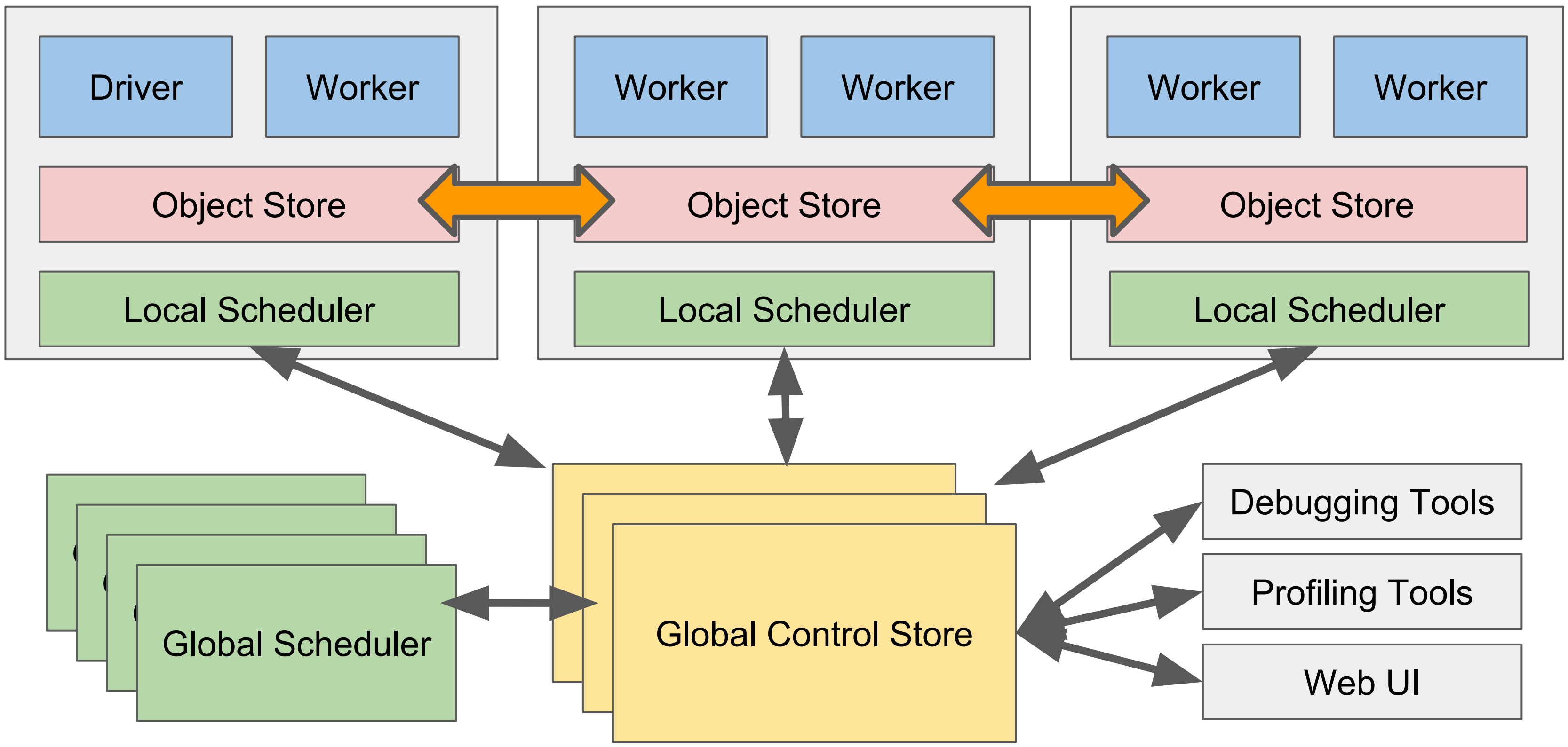
Ray architecture



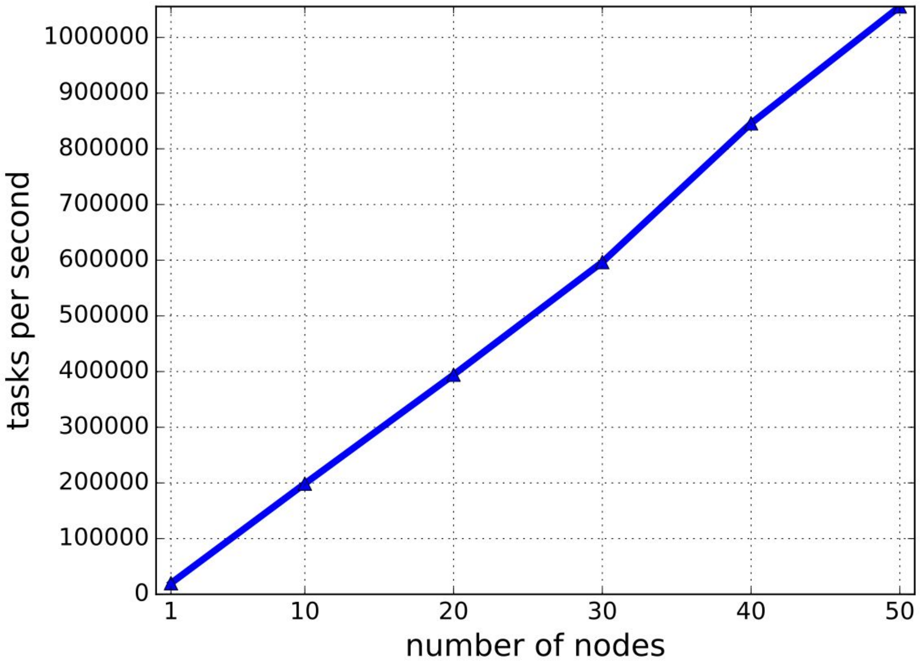
Ray architecture



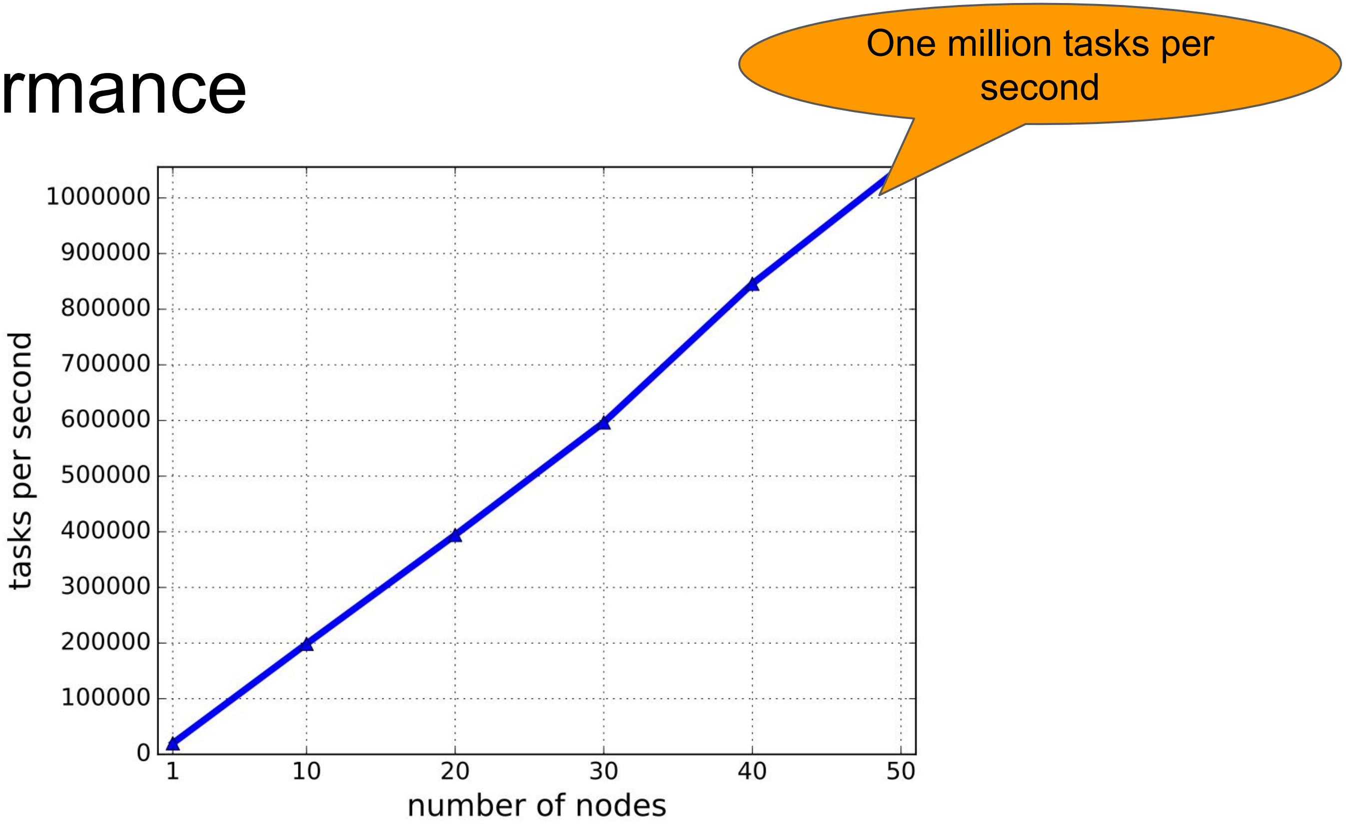
Ray architecture



Ray performance

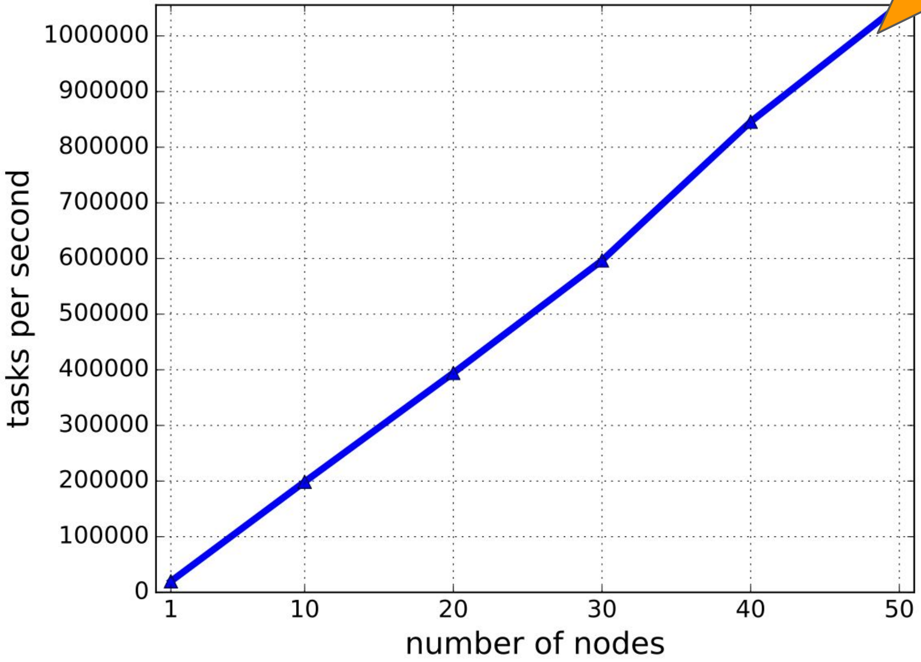


Ray performance



Ray performance

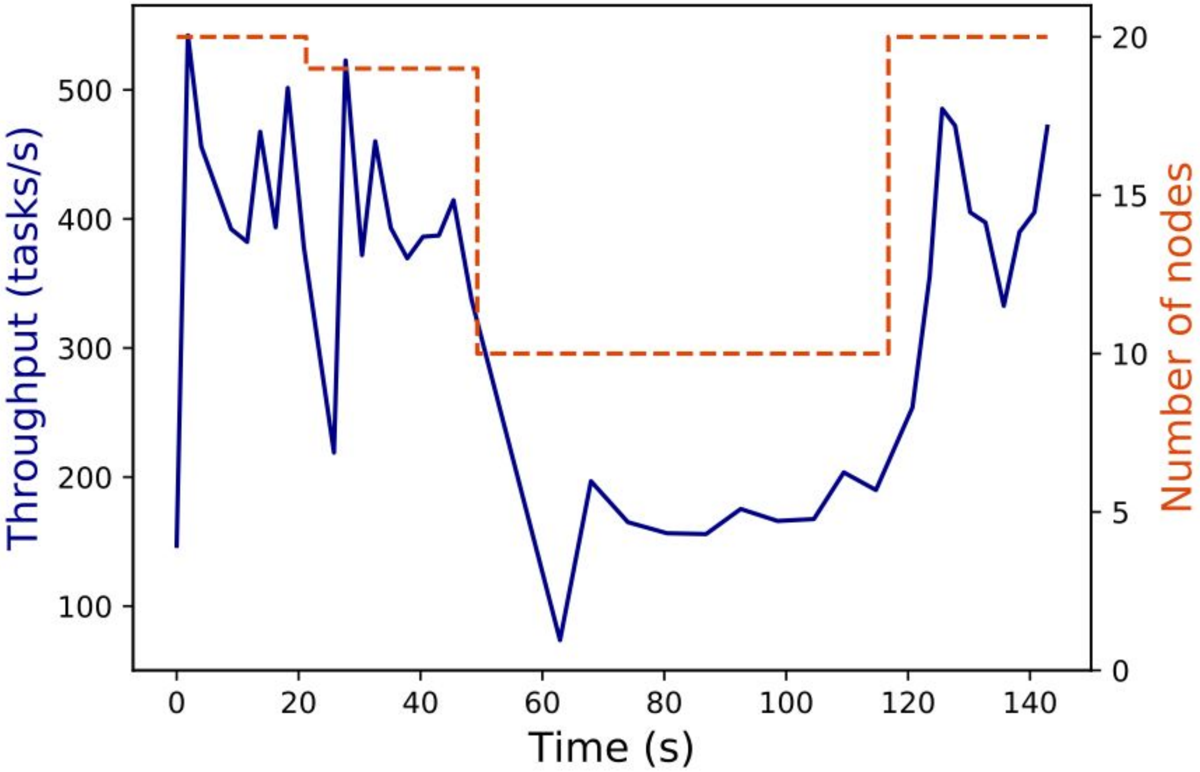
One million tasks per second



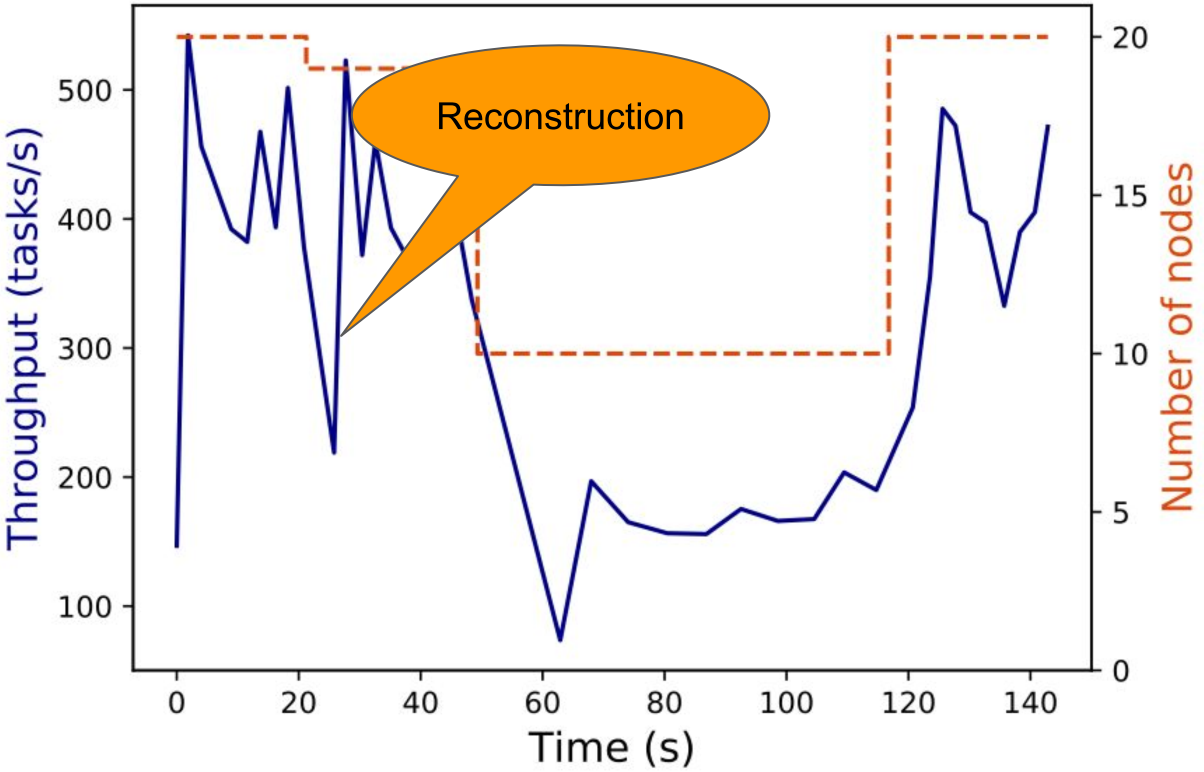
Latency of local task execution: ~300 us

Latency of remote task execution: ~1ms

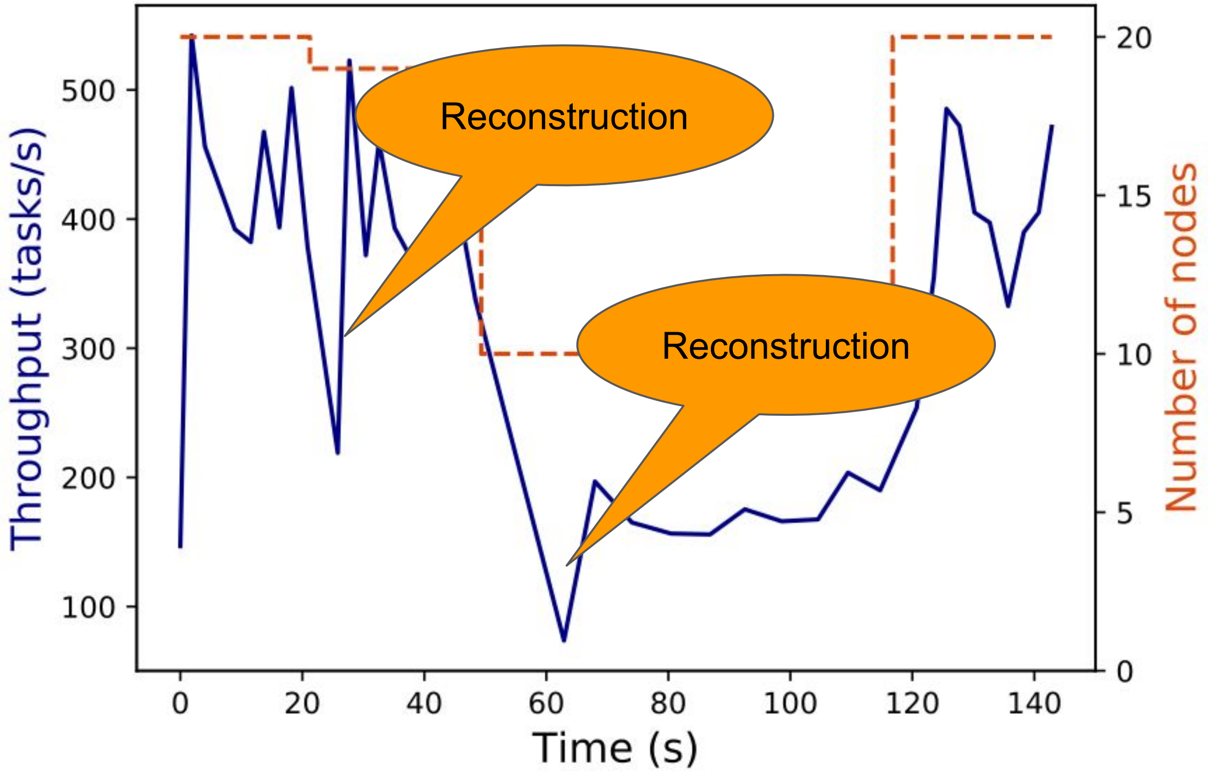
Ray fault tolerance



Ray fault tolerance



Ray fault tolerance



Evolution Strategies as a Scalable Alternative to Reinforcement Learning

Tim Salimans [1] Jonathan Ho [1] Xi Chen [1] Ilya Sutskever [1]

Abstract

We explore the use of Evolution Strategies, a class of black box optimization algorithms, as an alternative to popular RL techniques such as Q-learning and Policy Gradients. Experiments on MuJoCo and Atari show that ES is a viable solution strategy that scales extremely well with the number of CPUs available: By using hundreds to thousands of parallel workers, ES can solve 3D humanoid walking in 10 minutes and obtain competitive results on most Atari games after one hour of training time. In addition, we highlight several advantages of ES as a black box optimization technique: it is invariant to action frequency and delayed rewards, tolerant of extremely long horizons, and does not need temporal discounting or value function approximation.

1. Introduction

Developing agents that can accomplish challenging tasks

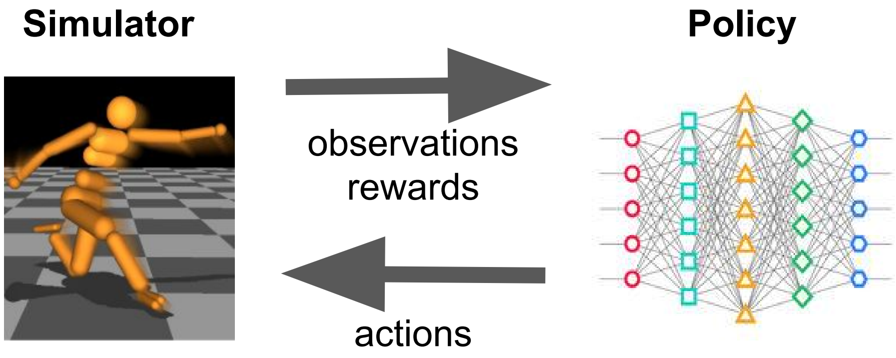
In this paper, we investigate the effectiveness of evolution strategies in the context of controlling robots in the MuJoCo physics simulator (Todorov et al., 2012) and playing Atari games with pixel inputs (Mnih et al., 2015). Our key findings are as follows:

1. We found specific network parameterizations that cause evolution strategies to reliably succeed, which we elaborate on in section 2.2.

2. We found the evolution strategies method to be highly parallelizable: we observe linear speedups in run time even when using over a thousand workers. In particular, using 1,440 workers, we have been able to solve the MuJoCo 3D humanoid task in under 10 minutes.

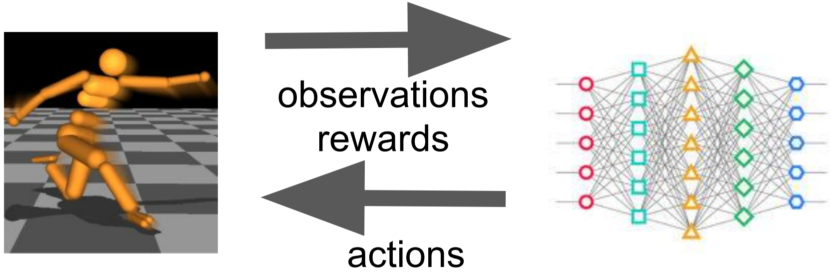
3. The data efficiency of the evolution strategies method was surprisingly good: we were able to match the final performance of a good A3C implementation (Mnih et al., 2016) on most Atari environments while using between 3x and 10x as much data. The slight decrease in data efficiency is partly offset by a reduction

Evolution Strategies



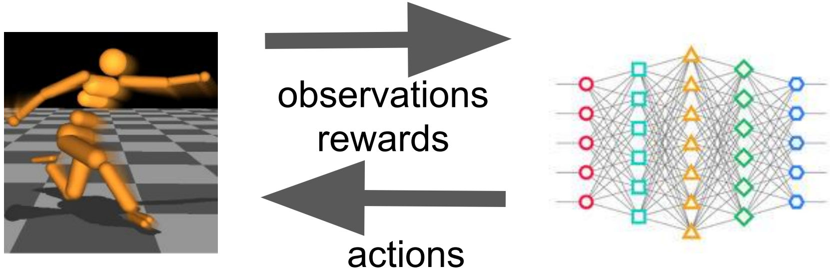
Try lots of different policies and see which one works best!

Pseudocode




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

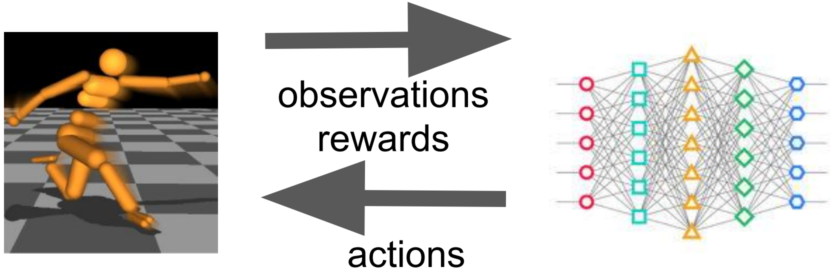
Pseudocode




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

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

Pseudocode

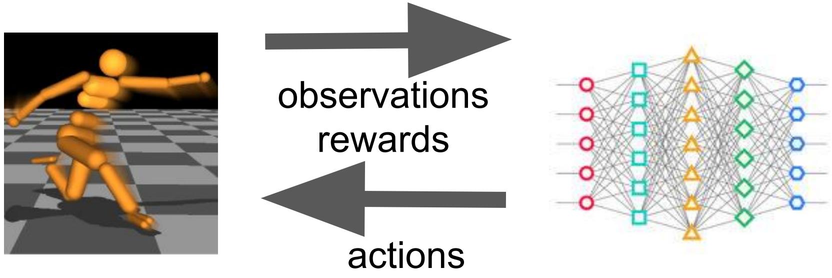



```
class Worker(object):
  def do_simulation(policy, seed):
    # perform simulation and return reward

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

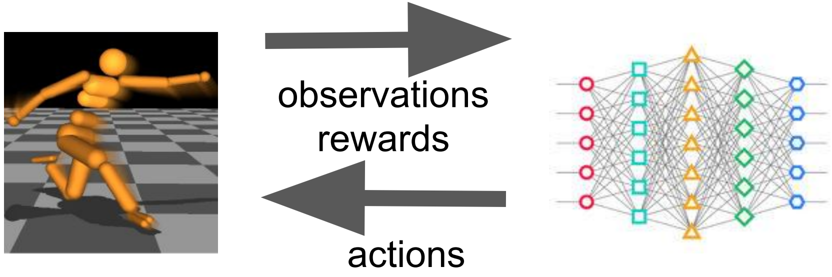



```
class Worker(object):
  def do_simulation(policy, seed):
    # perform simulation and return reward

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

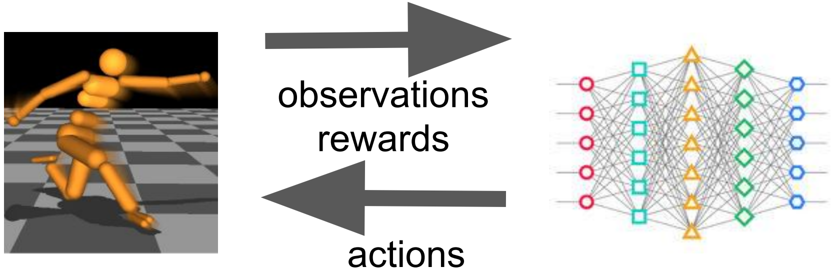



```
@ray.remote
class Worker(object):
  def do_simulation(policy, seed):
    # perform simulation and return reward

workers = [Worker() 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

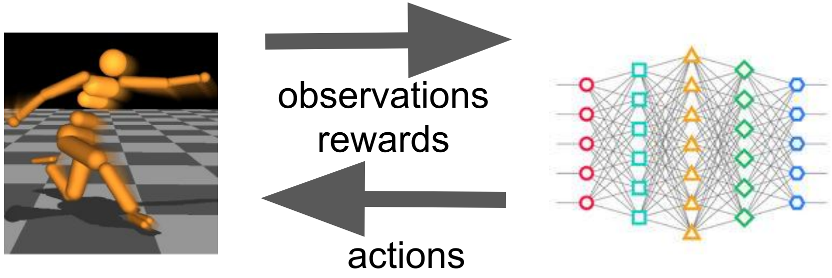



```
@ray.remote
class Worker(object):
  def do_simulation(policy, seed):
    # perform simulation and return reward

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(policy, seeds[j])
             for j in range(20)]
  policy = compute_update(policy, rewards, seeds)
```

Pseudocode

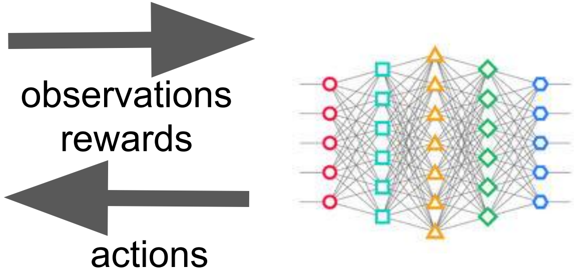



```
@ray.remote
class Worker(object):
  def do_simulation(policy, seed):
    # perform simulation and return reward

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, 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)
```

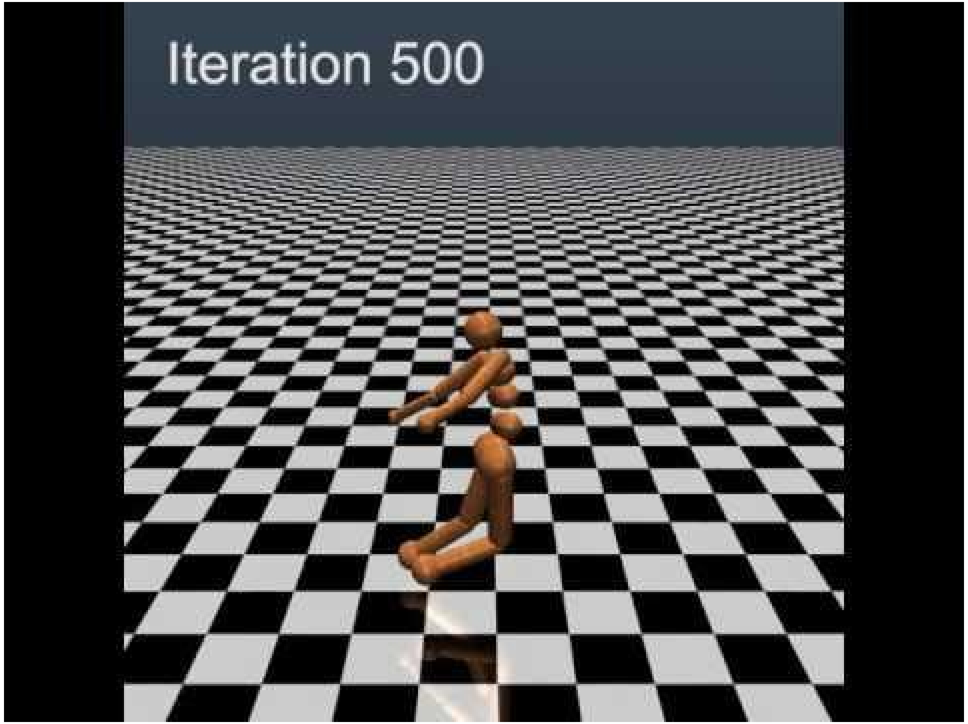
Evolution strategies on Ray

Simulator steps per second:

	10 nodes	**20 nodes**	**30 nodes**	**40 nodes**	**50 nodes**
Reference	97K	215K	202K	N/A	N/A
Ray	152K	285K	323K	476K	571K

The Ray implementation takes **half the amount of code** and was **implemented in a couple of hours**

Policy Gradients

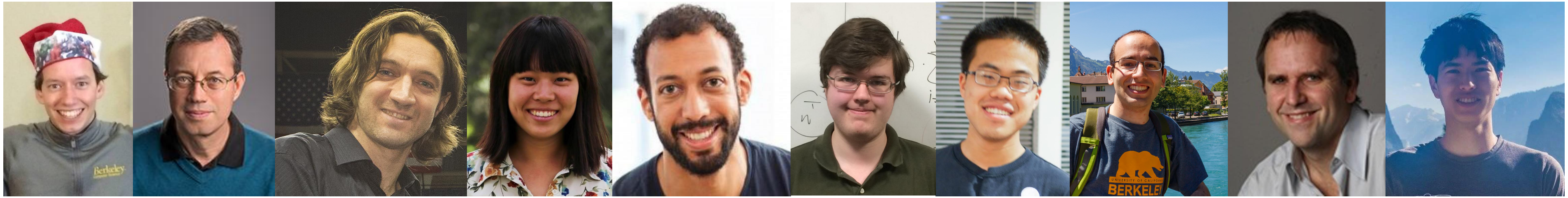


Ray + Apache Spark

- Complementary
  - Spark handles data processing, “classic” ML algorithms
  - Ray handles emerging AI algos., e.g. reinforcement learning (RL)
- Interoperability through object store based on Apache Arrow
  - Common data layout
  - Supports multiple languages

Ray is a system for AI Applications

- Ray is open source! https://github.com/ray-project/ray
- We have a v0.1 release!
  pip install ray
- We'd love your feedback



Philipp Ion Alexey Stephanie Johann William Richard Mehrdad Mike Robert