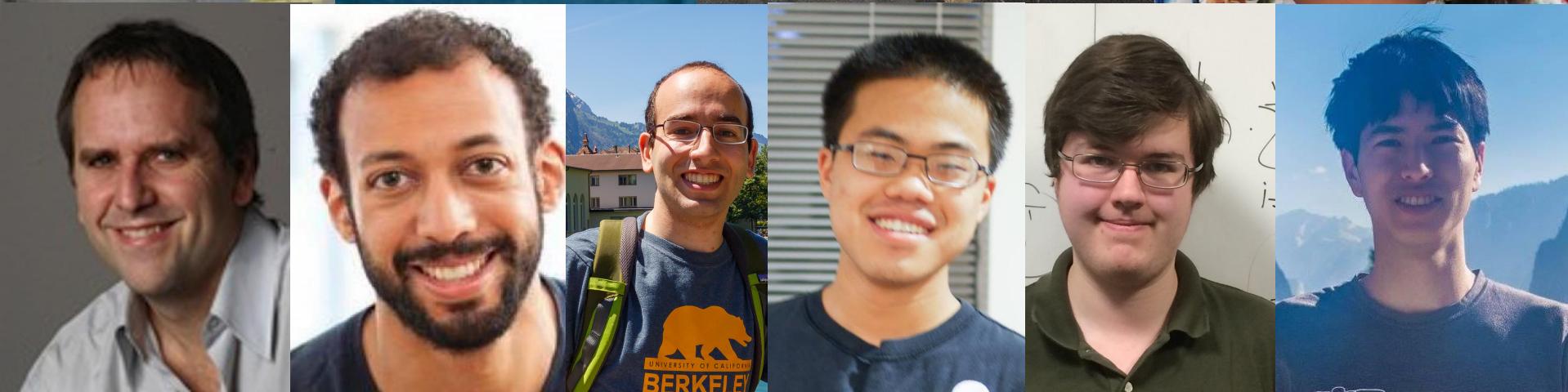


# Ray

## A Distributed Execution Framework for Emerging AI Applications

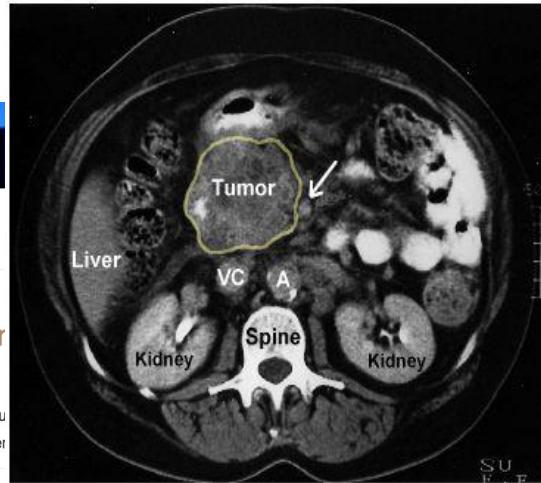
BDD/RISE mini-retreat

Philipp Moritz



# Emerging AI applications

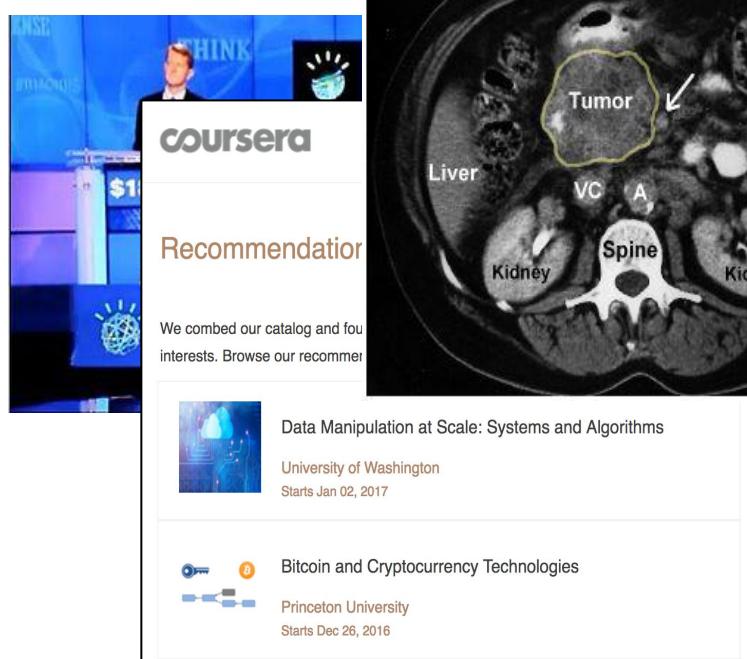
# Emerging AI applications



The screenshot shows a 'Recommendations' section on the Coursera homepage. It features a banner with a person speaking and the Coursera logo. Below the banner, text says: 'We combed our catalog and found courses you might like based on your interests. Browse our recommended courses.' It lists two courses:

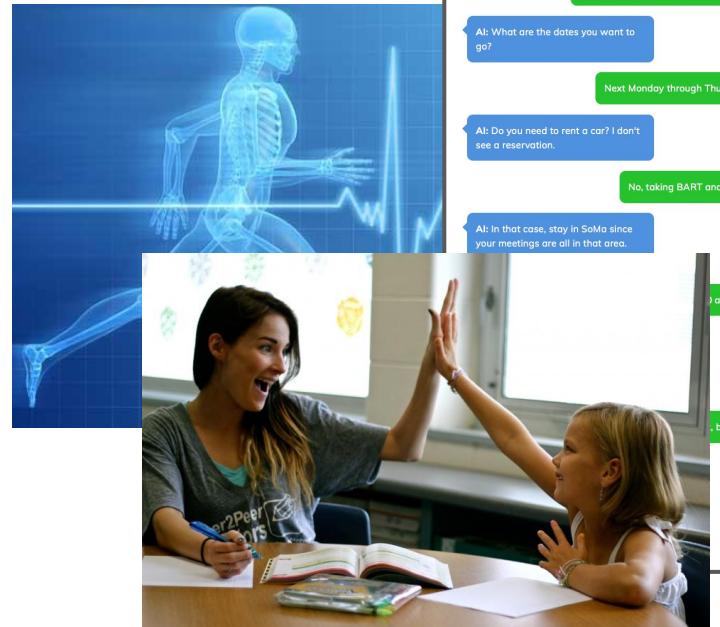
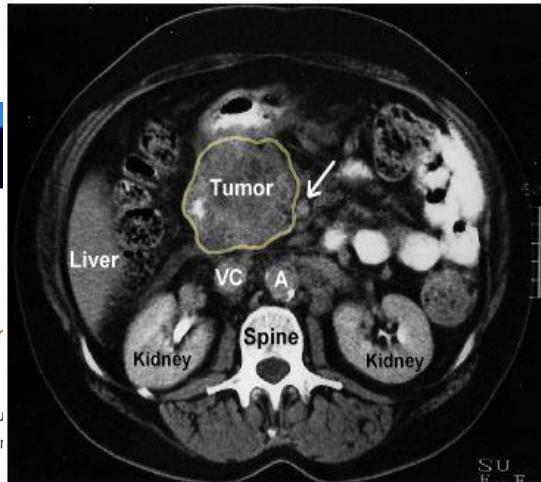
- Data Manipulation at Scale: Systems and Algorithms**  
University of Washington  
Starts Jan 02, 2017
- Bitcoin and Cryptocurrency Technologies**  
Princeton University  
Starts Dec 26, 2016

# Emerging AI applications

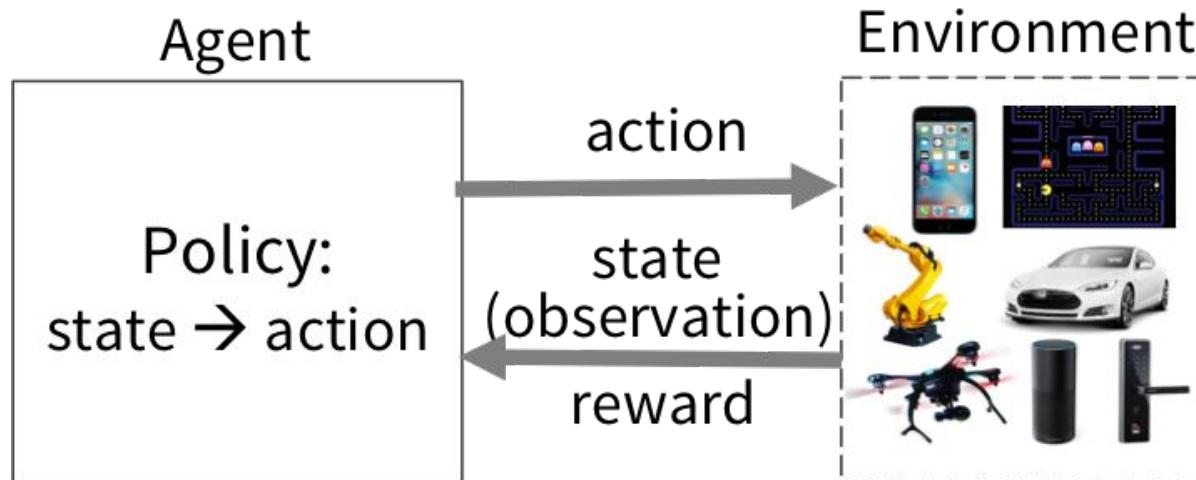


The image shows a screenshot of the Coursera website. At the top, there's a banner with a person speaking and the word "THINK". Below it, the Coursera logo is displayed. The main content area has a heading "Recommendations" in orange. It says: "We combed our catalog and found courses based on your interests. Browse our recommendations." Below this, two course cards are visible:

- Data Manipulation at Scale: Systems and Algorithms**  
University of Washington  
Starts Jan 02, 2017
- Bitcoin and Cryptocurrency Technologies**  
Princeton University  
Starts Dec 26, 2016



# Interacting with an environment



# Things that are hard with current distributed systems

# Things that are hard with current distributed systems

- Reinforcement learning training
- Fine-grained task parallelism with heterogeneous tasks
- Planning in real-time for a robot

# Things that are hard with current distributed systems

- Reinforcement learning training
- Fine-grained task parallelism with heterogeneous tasks
- Planning in real-time for a robot

## Requirements

- Low latency tasks
- High throughput tasks
- Adapt computation based on task progress
- Complex task dependencies
- Nested parallelism, dynamic task graph construction
- Tolerance of machine failures
- Seamless usage of GPUs and other accelerators

# Example: RL training

```
def train(env, hyperparameters):
    policy = initial_policy()
    for _ in range(1000):
        trajectories = [rollout(policy, env) for _ in range(K)]
        policy.update(trajectories)
    return policy

def rollout(policy, env):
    # alternately evaluate policy and simulate env
```

# Example: RL training

```
def train(env, hyperparameters):
    policy = initial_policy()
    for _ in range(1000):
        trajectories = [rollout(policy, env) for _ in range(K)]
        policy.update(trajectories)
    return policy

def rollout(policy, env):
    # alternately evaluate policy and simulate env

while True:
    train(env, random_hyperparameters()):
```

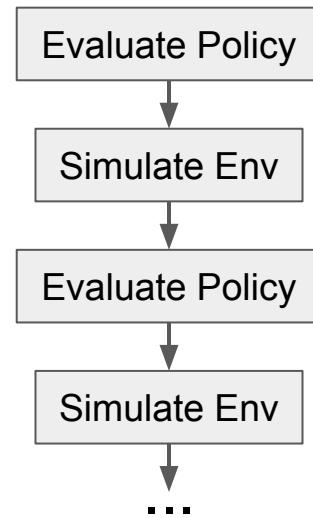
# Example: RL training

```
@ray.remote
def train(env, hyperparameters):
    policy = initial_policy()
    for _ in range(1000):
        trajectories = ray.get([rollout.remote(policy, env) for _ in range(K)])
        policy.update(trajectories)
    return policy

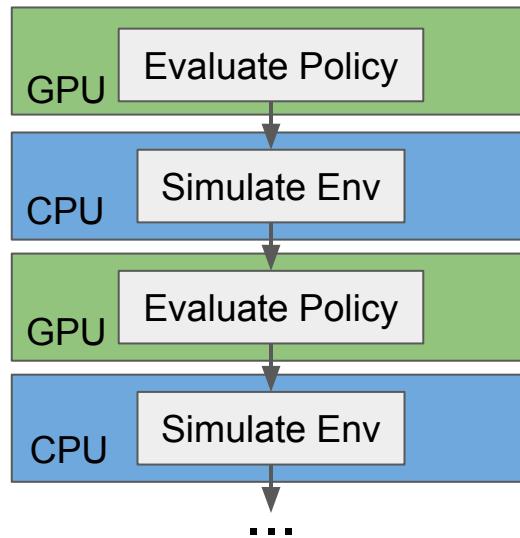
@ray.remote
def rollout(policy, env):
    # alternately evaluate policy and simulate env

while True:
    train.remote(env, random_hyperparameters()):
```

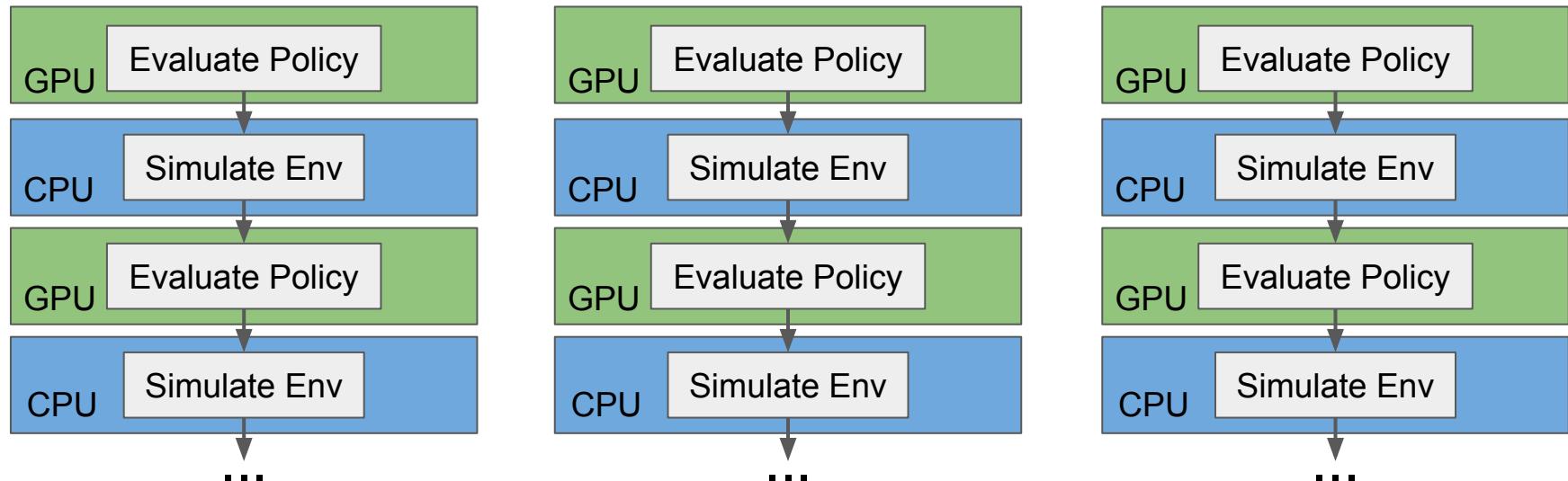
# Example: simulators on CPUs, policies on GPUs



# Example: simulators on CPUs, policies on GPUs



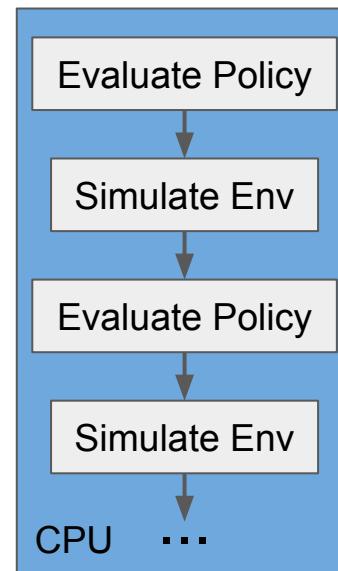
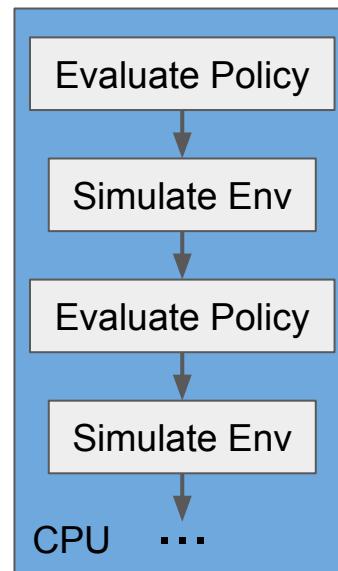
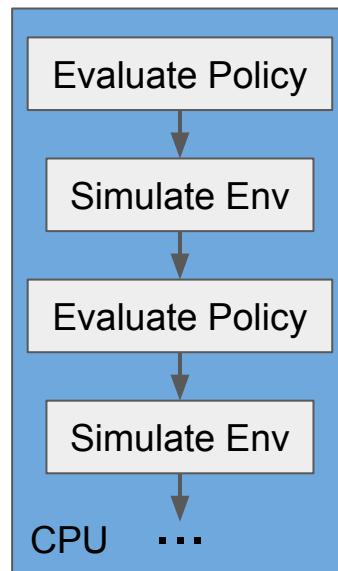
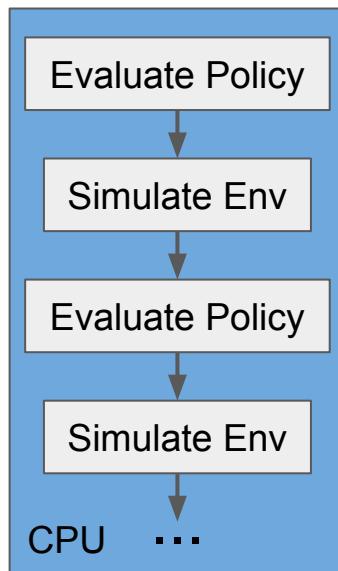
# Example: simulators on CPUs, policies on GPUs



# Example: simulators on CPUs, policies on GPUs

Options:

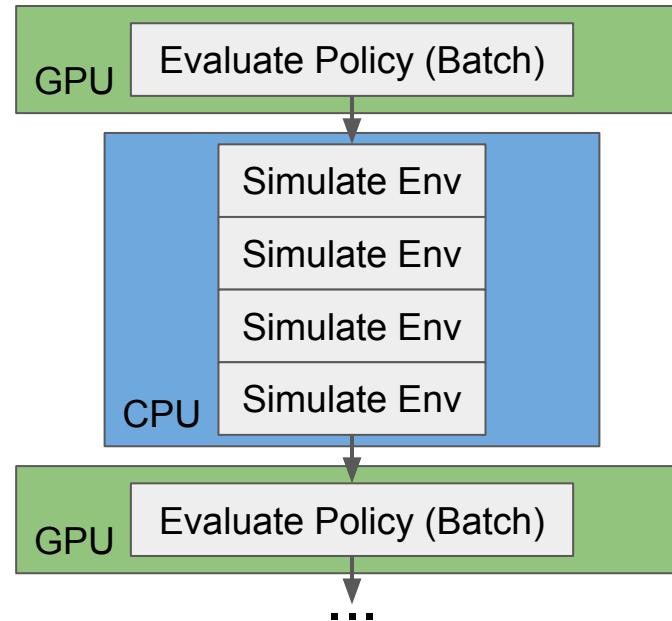
- 1) One task per CPU, do everything on CPUs.
- 2) One task per GPU (batch policy evaluation on GPU).
- 3) Many tasks. Policy evaluation on GPUs and simulator on CPUs.



# Example: simulators on CPUs, policies on GPUs

Options:

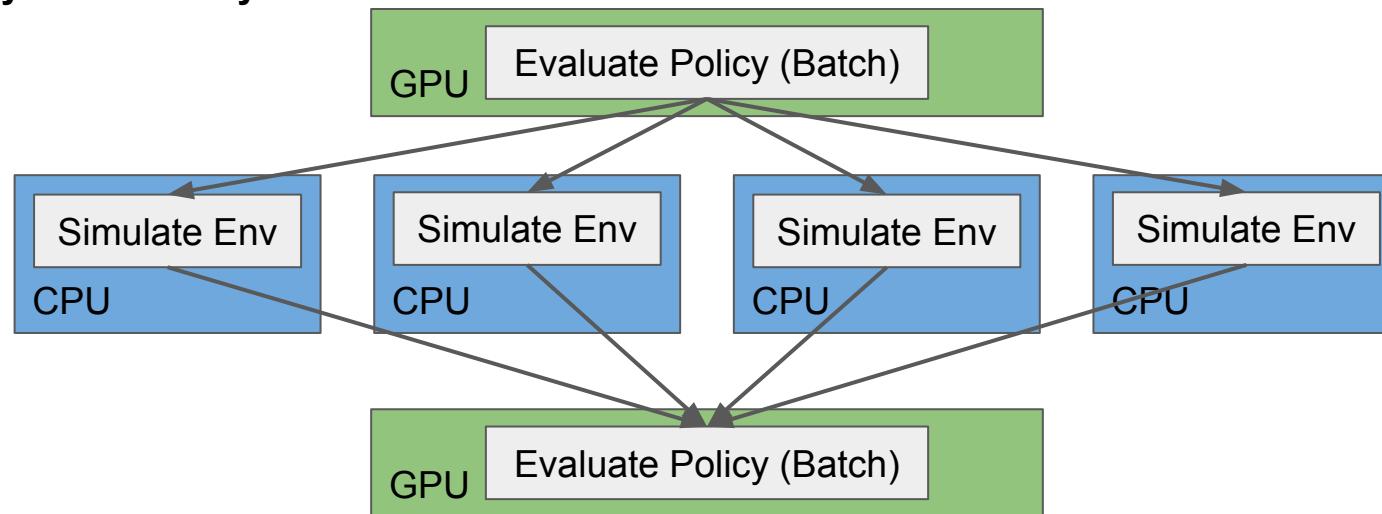
- 1) One task per CPU, do everything on CPUs.
- 2) **One task per GPU (batch policy evaluation on GPU).**
- 3) Many tasks. Policy evaluation on GPUs and simulator on CPUs.



# Example: simulators on CPUs, policies on GPUs

Options:

- 1) One task per CPU, do everything on CPUs.
- 2) One task per GPU (batch policy evaluation on GPU).
- 3) **Many tasks. Policy evaluation on GPUs and simulator on CPUs.**



# Ray API

# Ray API - remote functions

```
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

```
@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

```
@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

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

```
@ray.remote(num_gpus=2)  
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.
- Can specify **GPU** requirements

# Ray API - actors

```
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

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

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

# Ray API - actors

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

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

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

# Ray API - actors

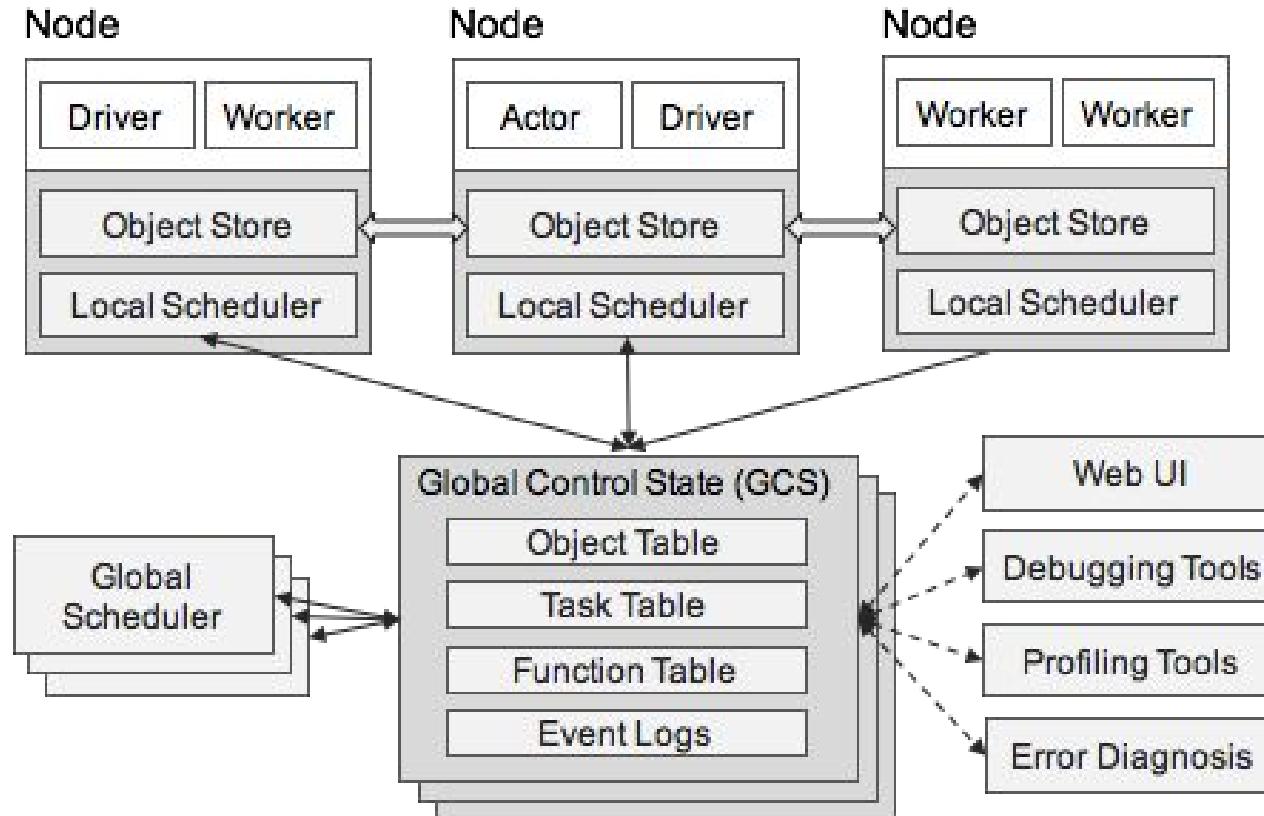
```
@ray.actor(num_gpus=1)
class Counter(object):
    def __init__(self):
        self.value = 0
    def inc(self):
        self.value += 1
        return self.value
```

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

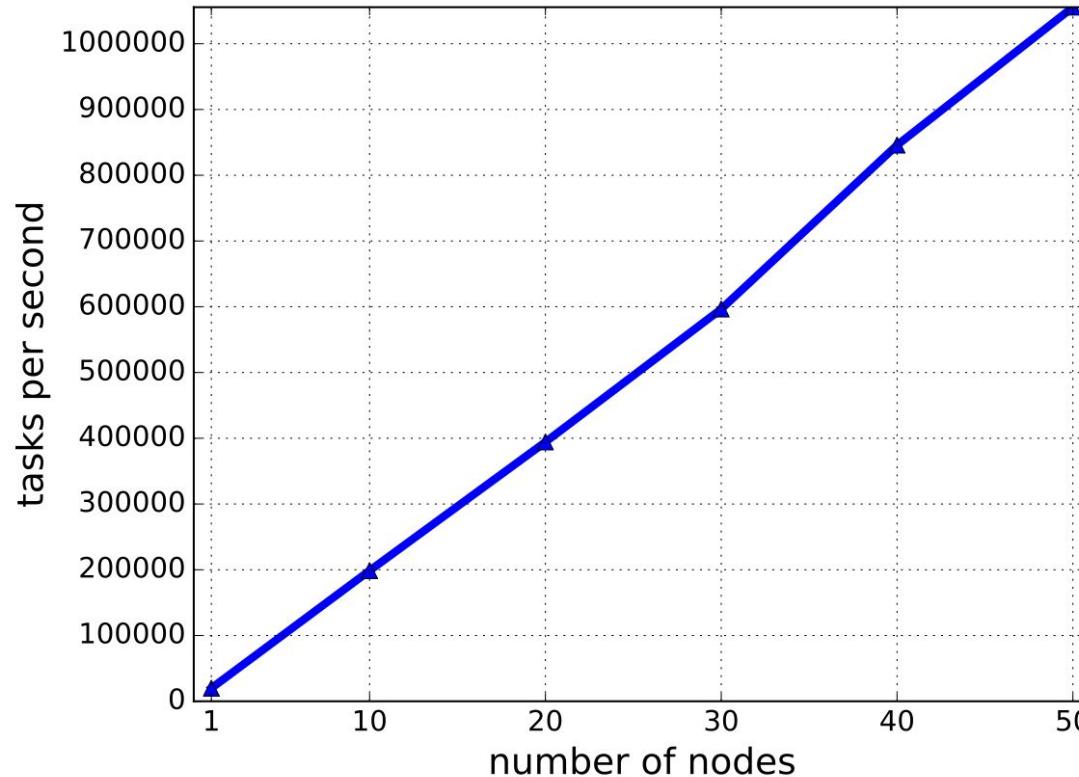
- State is shared between actor methods.
- Actor methods return **Object IDs**.
- Can specify **GPU** requirements

# Ray Architecture

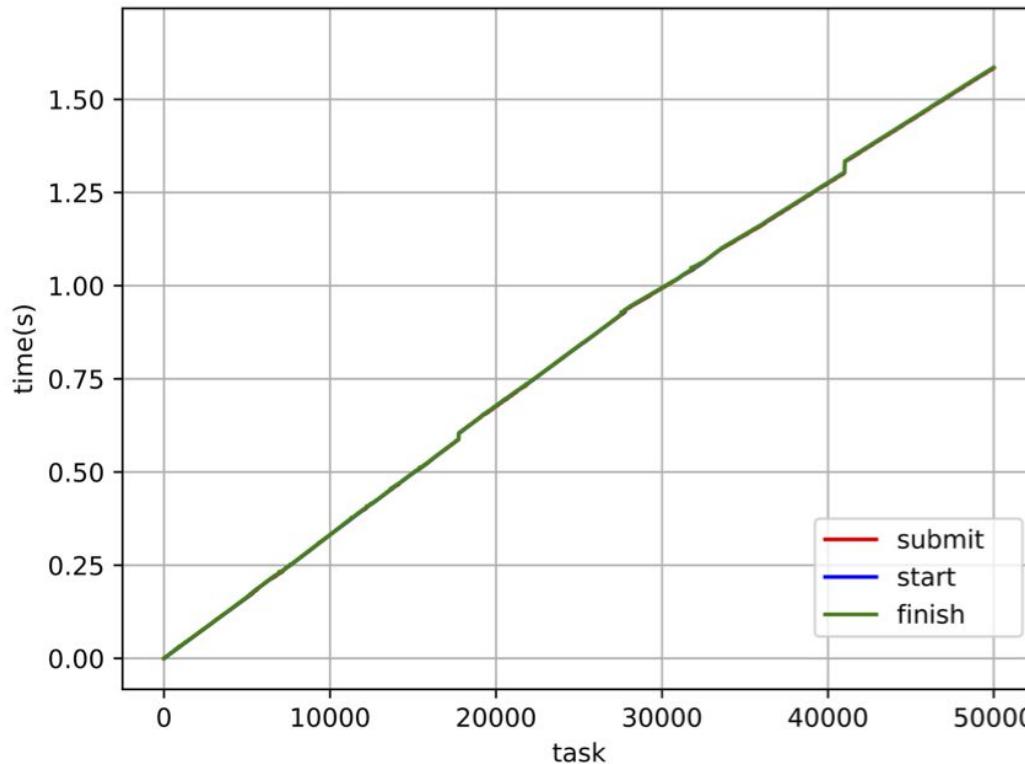
# System Architecture



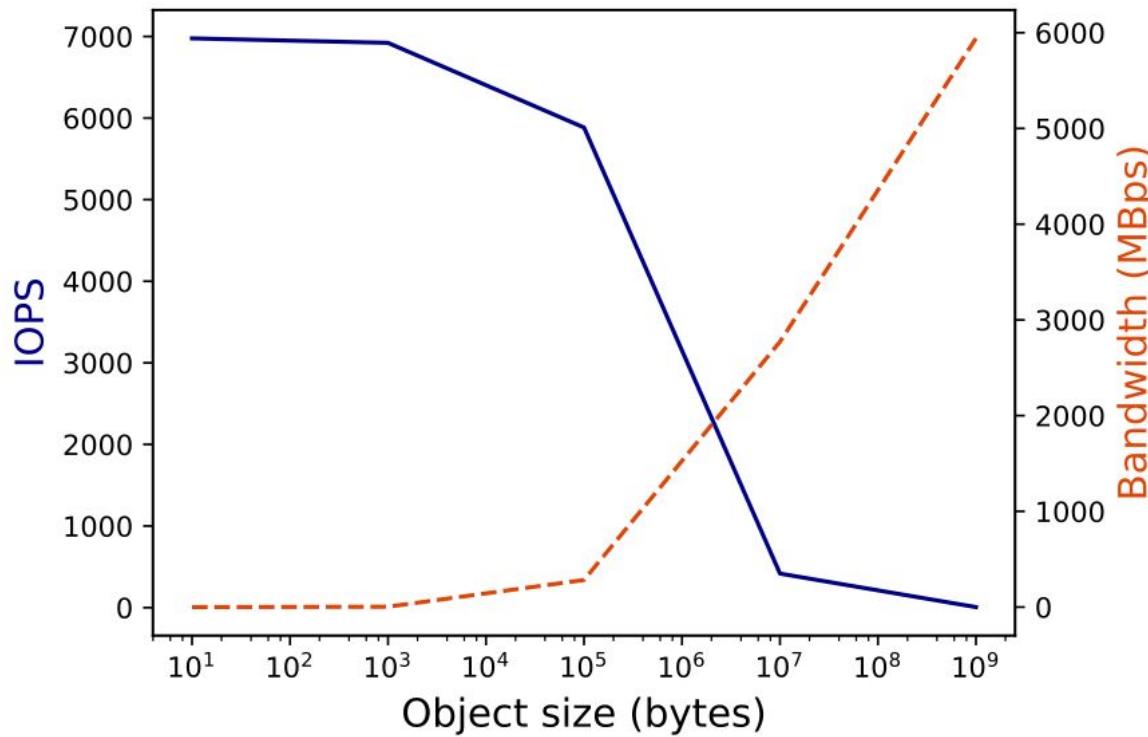
# System throughput



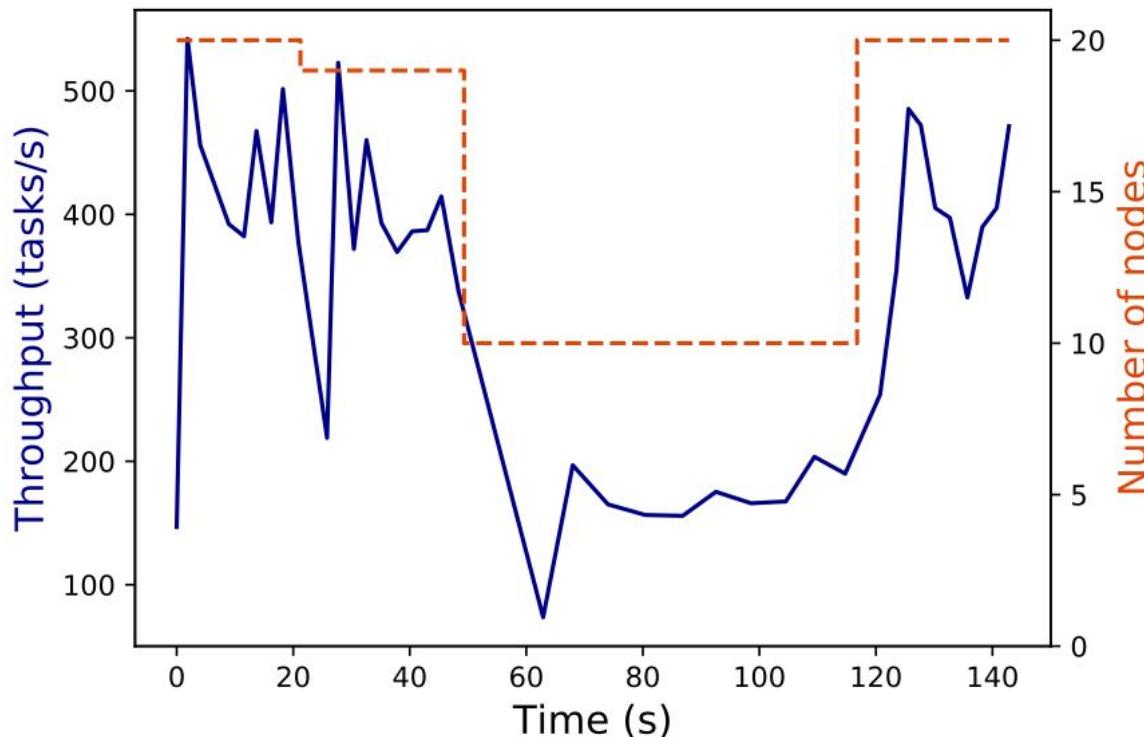
# Single machine throughput



# Object store performance

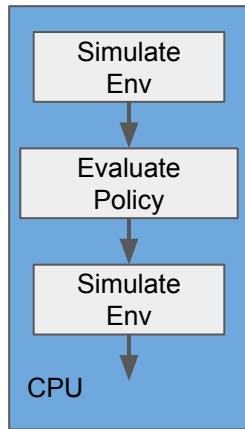


# Robustness to node failure



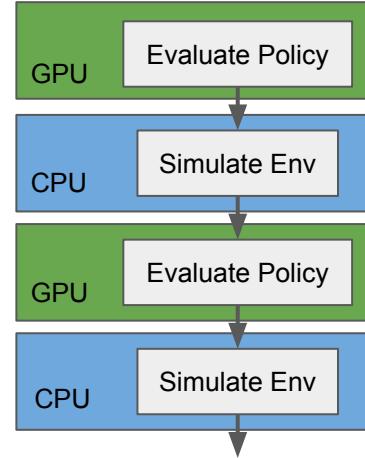
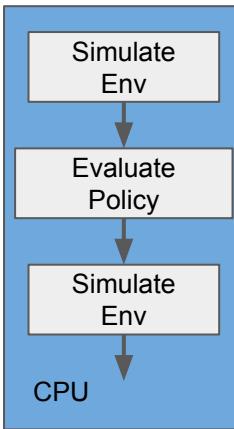
# Experiments

# Speeding up rollouts for policy gradients



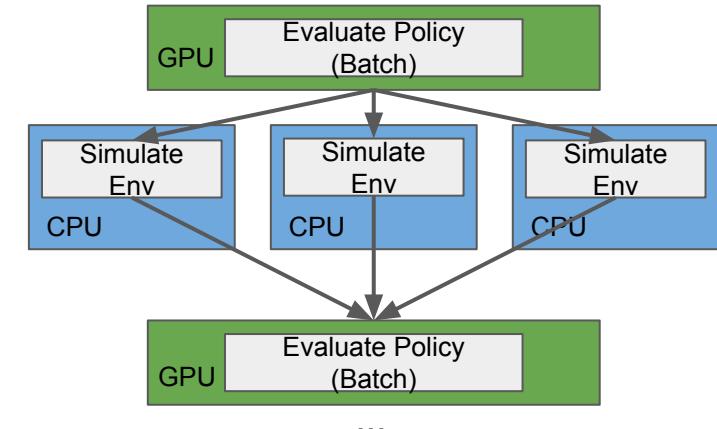
Parallel Rollouts on CPU

**1.0x**



Policy Evaluation on GPU

**1.3x**



Fine grained rollouts

**4.1x**

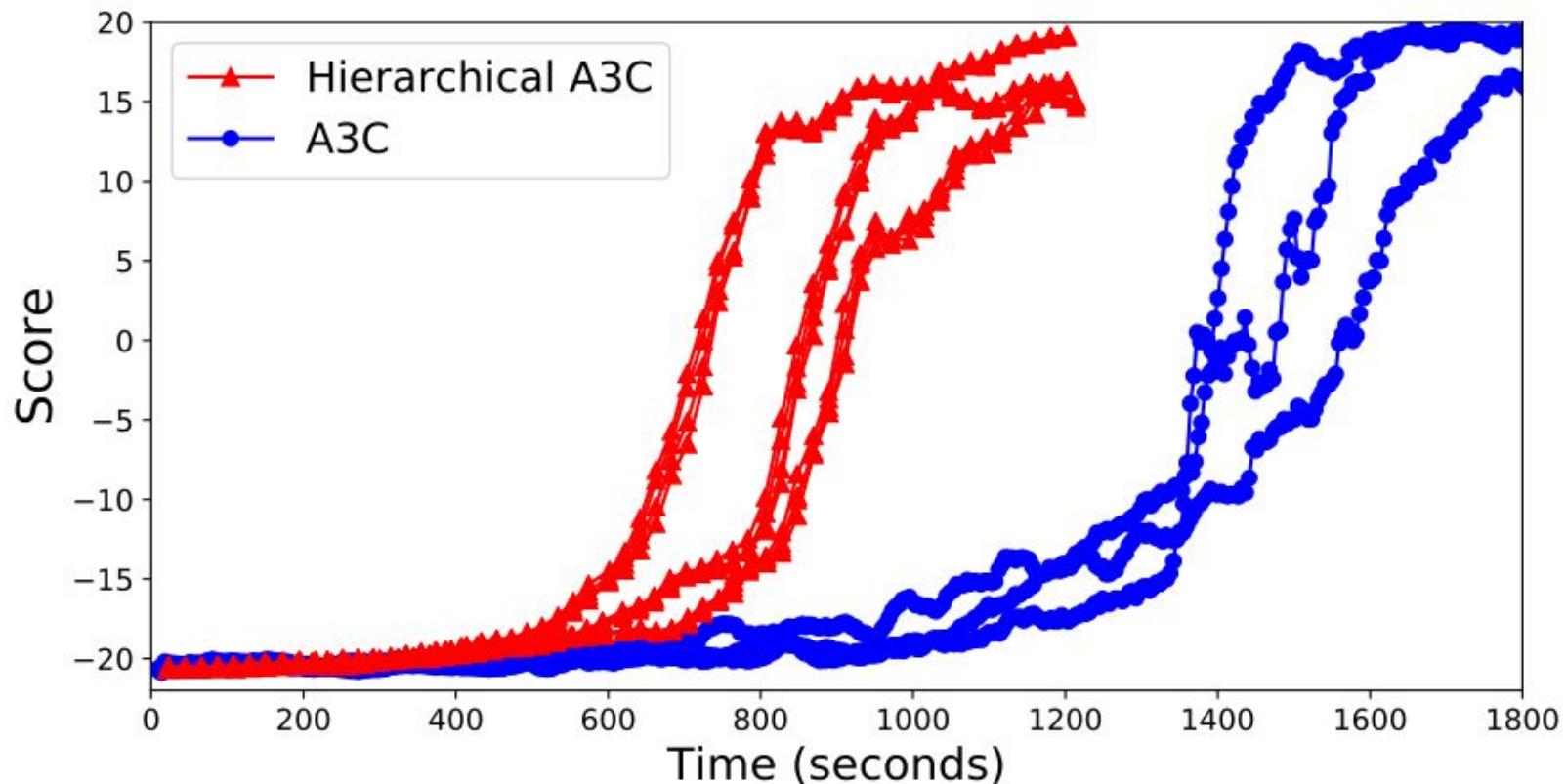
Speedup:

# Evolution strategies

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

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

# Hierarchical A3C



# Ray is a system for AI Applications

- Ray is open source! <https://github.com/ray-project/ray>
- We have a pre-release!
- We'd love your feedback.



Philipp

Ion

Alexey

Stephanie

Johann

William

Richard

Mehrdad

Mike

Robert