Ray: A Distributed Framework for Emerging AI Applications

Saber Malekmohammadi

November 2021

Computer Science Department

UNIVERSITY OF WATERLOO

Background - AI applications and system requirements

- Different applications of AI have their specific computational tasks
- Based on these tasks, they impose some system requirements
- Example: supervised Learning application:

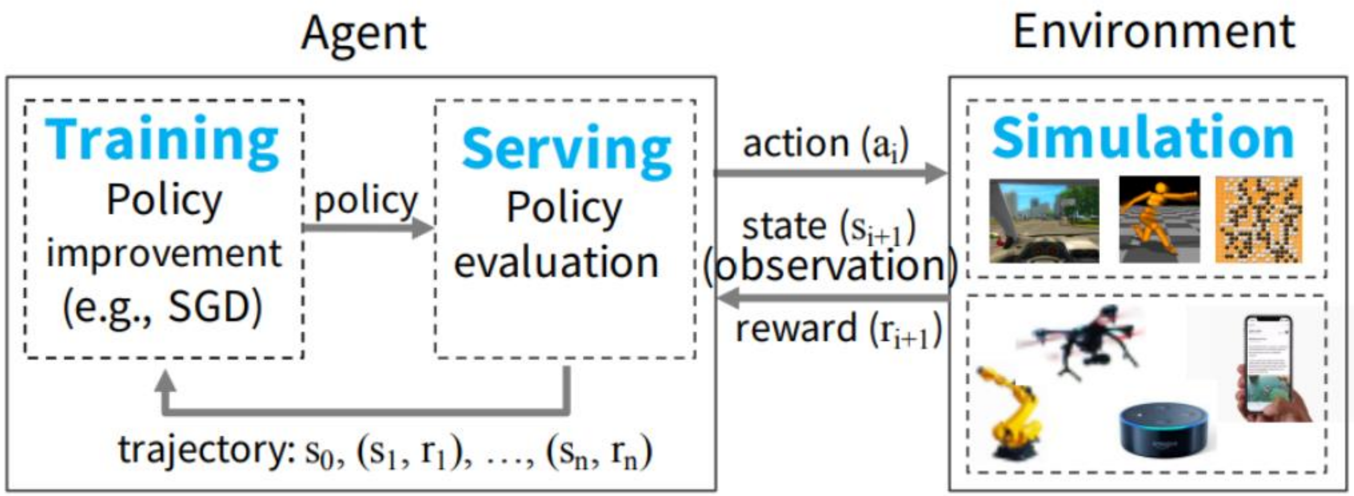
The stateful training task

The stateless prediction task

Impose system requirements (training stage):
Tensorflow, MXNet and Pytorch

Research Problem

- The scope of AI applications encompasses more complex applications
- Three simultaneously required capabilities in Reinforcement Learning (RL):
  - Distributed training: fine-grained computations, heterogeneous computations
  - Serving: latency-sensitive, fine-grained computations, heterogeneous computations
  - Simulations: dynamic execution



Philipp Moritz, et al. **"Ray: A Distributed Framework for Emerging AI Applications"**. In: OSDI. 2018

Prior Work

- The existing frameworks developed for supervised learning and Big Data workloads can not meet these requirements

- Bulk-synchronous parallel systems such as MapReduce, Apache Spark and Dryad:

  Do not support fine-grained action rendering and simulation computations in RL

- Task-parallel systems such as CIEL and DASK:

  Do not completely support distributed training and serving

- Distributed deep learning frameworks such as Ten-sorflow and MXNet:

  Do not support simulation and serving naturally

Solution

- Considering the system requirements of more complex applications such as RL:
  - Ray, a distributed framework, was proposed for RL applications requirements
  - Provides a general programming model supporting task-parallel and actor-based computations
  - Supports a range of computations: from lightweight and stateless computations (simulations) to long and stateful computations (training)
  - Provides low latency, high scalability and failure tolerance

- This enables us to do all the tasks of training, serving and simulation together by a single framework.

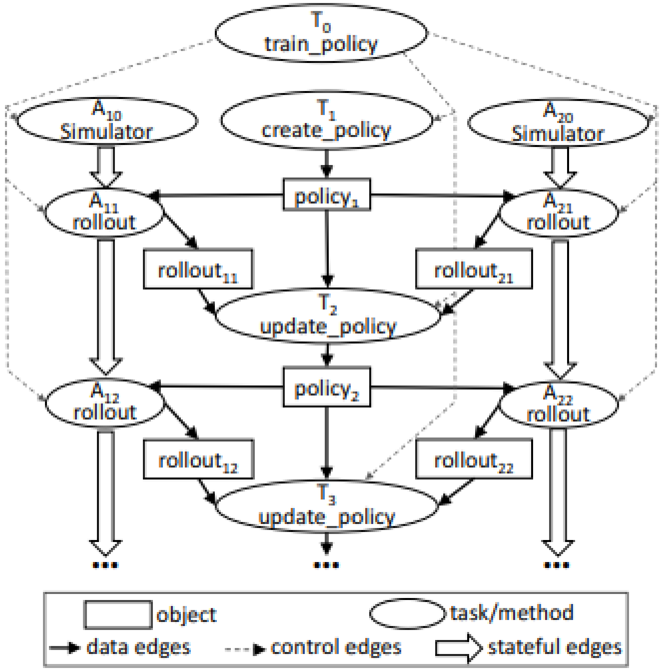
Technical Details – Dynamic Computation Graph

```
@ray.remote
def create_policy():
    # Initialize the policy randomly.
    return policy

@ray.remote(num_gpus=1)
class Simulator(object):
    def __init__(self):
        # Initialize the environment.
        self.env = Environment()
    def rollout(self, policy, num_steps):
        observations = []
        observation = self.env.current_state()
        for _ in range(num_steps):
            action = compute(policy, observation)
            observation = self.env.step(action)
            observations.append(observation)
        return observations
```

```
@ray.remote(num_gpus=2)
def update_policy(policy, *rollouts):
    # Update the policy.
    return policy

@ray.remote
def train_policy():
    # Create a policy.
    policy_id = create_policy.remote()
    # Create 10 actors.
    simulators = [Simulator.remote() for _ in range(10)]
    # Do 100 steps of training.
    for _ in range(100):
        # Perform one rollout on each actor.
        rollout_ids = [s.rollout.remote(policy) for s in simulators]
        # Update the policy with the rollouts.
        policy_id = update_policy.remote(policy_id,
                                         *rollout_ids)
    return ray.get(policy_id)
```



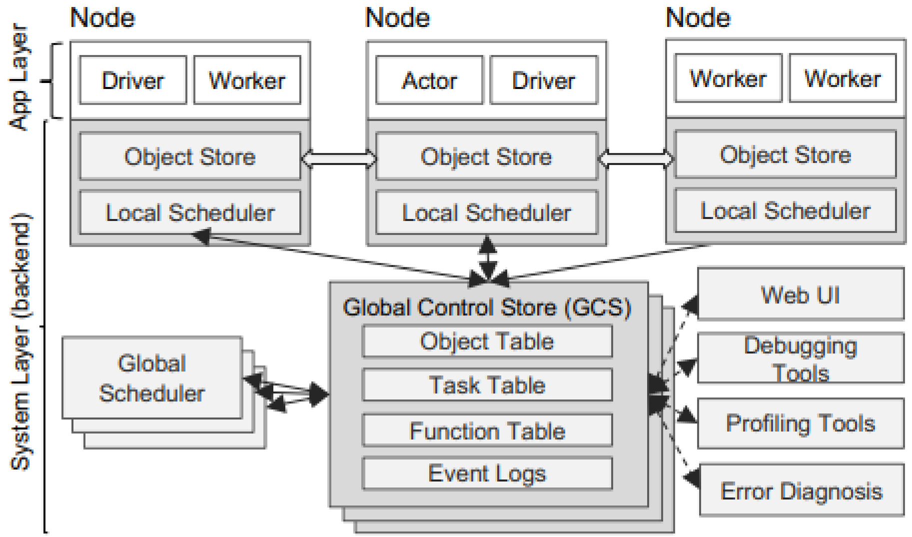
Tasks (stateless)	Actors (stateful)
Fine-grained load balancing	Coarse-grained load balancing
Support for object locality	Poor locality support
High overhead for small updates	Low overhead for small updates
Efficient failure handling	Overhead from checkpointing

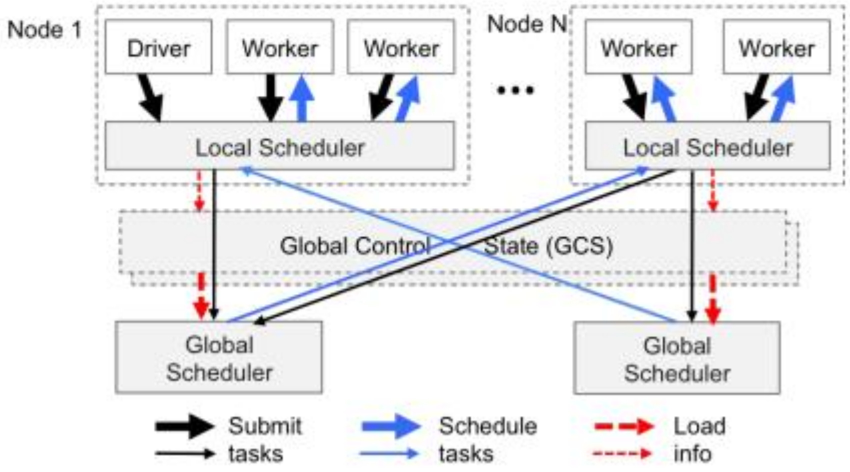
Name	Description
$futures = \mathbf{f.remote}(args)$	Execute function f remotely. **f.remote**() can take objects or futures as inputs and returns one or more futures. This is non-blocking.
$objects = \mathbf{ray.get}(futures)$	Return the values associated with one or more futures. This is blocking.
$ready\_futures = \mathbf{ray.wait}(futures, k, timeout)$	Return the futures whose corresponding tasks have completed as soon as either k have completed or the timeout expires.
$actor = \mathbf{Class.remote}(args)$ $futures = actor.\mathbf{method.remote}(args)$	Instantiate class *Class* as a remote actor, and return a handle to it. Call a method on the remote actor and return one or more futures. Both are non-blocking.

Philipp Moritz, et al. **"Ray: A Distributed Framework for Emerging AI Applications"**. In: OSDI. 2018

Technical Details – Architecture

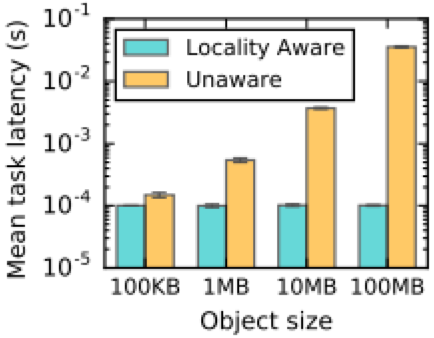
- Locality aware task scheduling: load balancing and locality-aware scheduling
- Global control state: scalability, task/actor failure tolerance
- Separating task scheduling from task dispatch: scalability, high throughput of fine-grained tasks, low latency
- Distributed object store: unifying actors and tasks on nodes



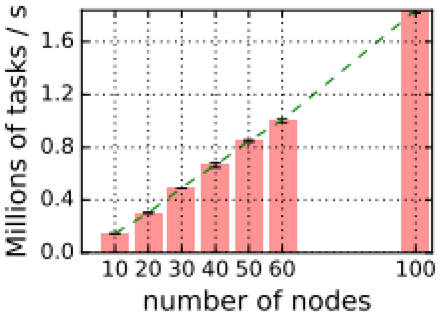


Philipp Moritz, et al. **“Ray: A Distributed Framework for Emerging AI Applications”**. In: OSDI. 2018

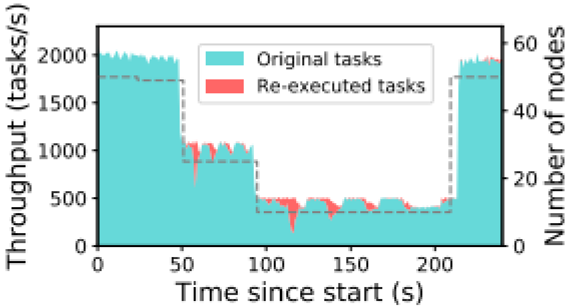
Performance



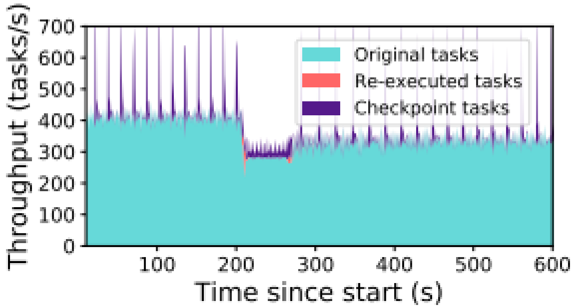
(a) Ray locality scheduling



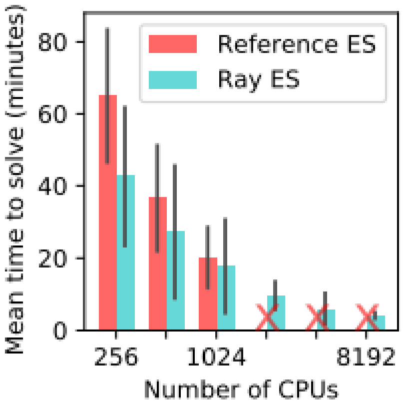
(b) Ray scalability



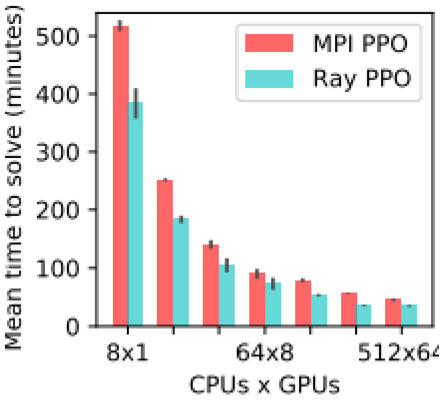
(a) Task reconstruction



(b) Actor reconstruction



(a) Evolution Strategies



(b) PPO

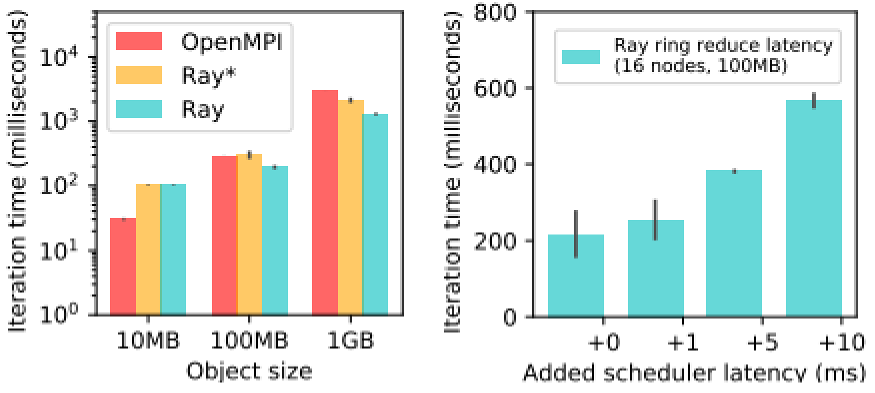
System	Small Input	Larger Input
Clipper	4400 ± 15 states/sec	290 ± 1.3 states/sec
Ray	6200 ± 21 states/sec	6900 ± 150 states/sec

Philipp Moritz, et al. **“Ray: A Distributed Framework for Emerging AI Applications”**. In: OSDI. 2018

Strengths and Weaknesses

- Strengths:
  - Supporting task-parallel and actor-based computations: supporting training, serving, simulation
  - Locality aware task scheduling: load balancing and locality-aware scheduling
  - Global control state: scalability, task/actor failure tolerance
  - Separating task scheduling from task dispatch: scalability, high task throughput, low latency
  - Distributed object store: unifying actors and tasks on nodes
  - Scalable architecture: scheduler
- Weaknesses:
  - Higher overhead imposed on distributed primitives:

    For small objects, Ray can not outperform dedicated systems like OpenMP



(a) Ray vs OpenMPI (b) Ray scheduler ablation

Philipp Moritz, et al. **"Ray: A Distributed Framework for Emerging AI Applications"**. In: OSDI. 2018

Discussion

- The Comparison with dedicated systems is just based on RL workloads
- No comparison with other workloads
- Object store does not support distributed objects storage: distributed objects like large matrices or trees that do not fit on a single node. Would that impose large latencies?

References

- Joseph E. Gonzalez, et al. **“GraphX: Graph Processing in a Distributed Dataflow Framework”**. In: OSDI. 2014.
- Derek Gordon Murray, et al. **“Naiad: a timely dataflow system”**. In: Proceedings of the ACM Symposium on Operating Systems Principles (2013).
- Jeffrey Dean, et al. **“MapReduce: simplified data processing on large clusters”**. In: Commun. ACM 51 (2008), pp. 107–113.
- Mart´ın Abadi, et al. **“TensorFlow: A system for large-scale machine learning”**. In: OSDI. 2016.
- Tianqi Chen, et al. **“MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems”**. In: ArXiv abs/1512.01274 (2015).
- Adam Paszke, et al. **“Automatic differentiation in PyTorch”**. In: 2017.
- Philipp Moritz, et al. **“Ray: A Distributed Framework for Emerging AI Applications”**. In: OSDI. 2018.
- Derek Gordon Murray, et al. **“CIEL: A Universal Execution Engine for Distributed Data-Flow Computing”**. In: NSDI. 2011.
- Matthew Rocklin. **“Dask: Parallel Computation with Blocked algorithms and Task Scheduling”**. In: 2015.