Ray: A Distributed Framework for Emerging AI Applications

R244: Large-Scale Data Processing and Optimisation

Kian Cross

Moritz, P., Nishihara, R., Wang, S., Tumanov, A., Liaw, R., Liang, E., ... & Stoica, I. (2018). Ray: A distributed framework for emerging AI applications. In 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18) (pp. 561-577).

Background

- Reinforcement learning applications "rely heavily on simulations"
- "This generally requires massive amounts of computation"
- "The computation graph of an RL application is *heterogeneous* and evolves *dynamically*"
- Some RL-based applications require low-latency
- Is there a cluster computing framework that satisfies these requirements?

Existing Solutions

Map-Reduce

APACHE Spark™

dask/**dask**

Parallel computing with task scheduling

CIEL

These don't support the throughputs or latencies required

Requirements for a New Framework

- Flexible
  - Execution of concurrent, heterogenous tasks
  - Support dynamic task graphs
- Performant
  - Schedule tasks in less than a millisecond
  - Schedule millions of tasks per second
- Easy development
  - Deterministic replay and fault tolerance
  - Easy parallelization of existing algorithms

RAY

What is Ray?

- Published in 2017
- A Python library
- For distributed computing
- Motivated by the needs of reinforcement learning applications

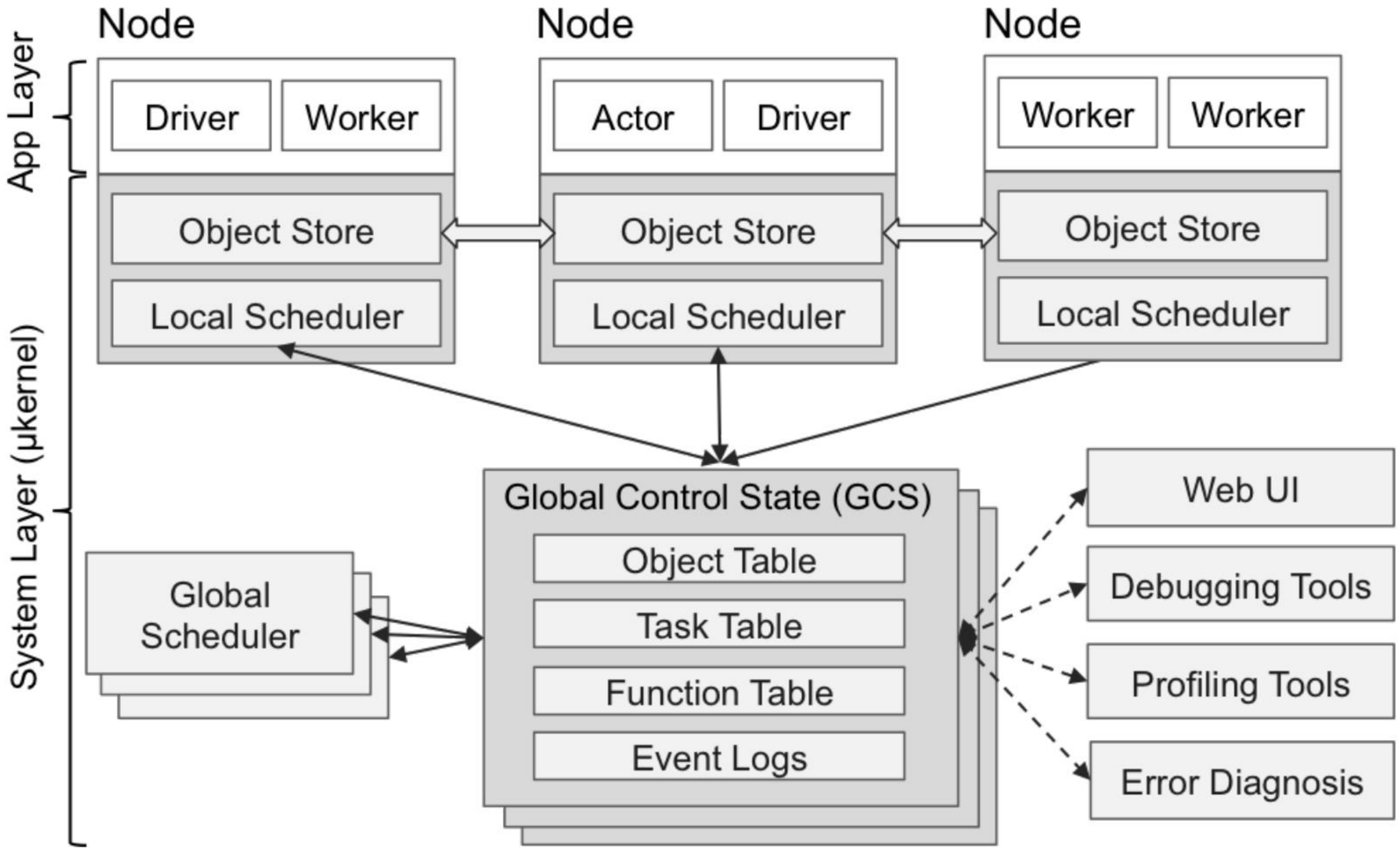
Application Layer

- ***Driver***: A process executing the user program.
- ***Worke***r: A stateless process that remote functions invoke by a driver or another worker.
- ***Actor***: A stateful process that executes, when invoked, the methods it exposes.

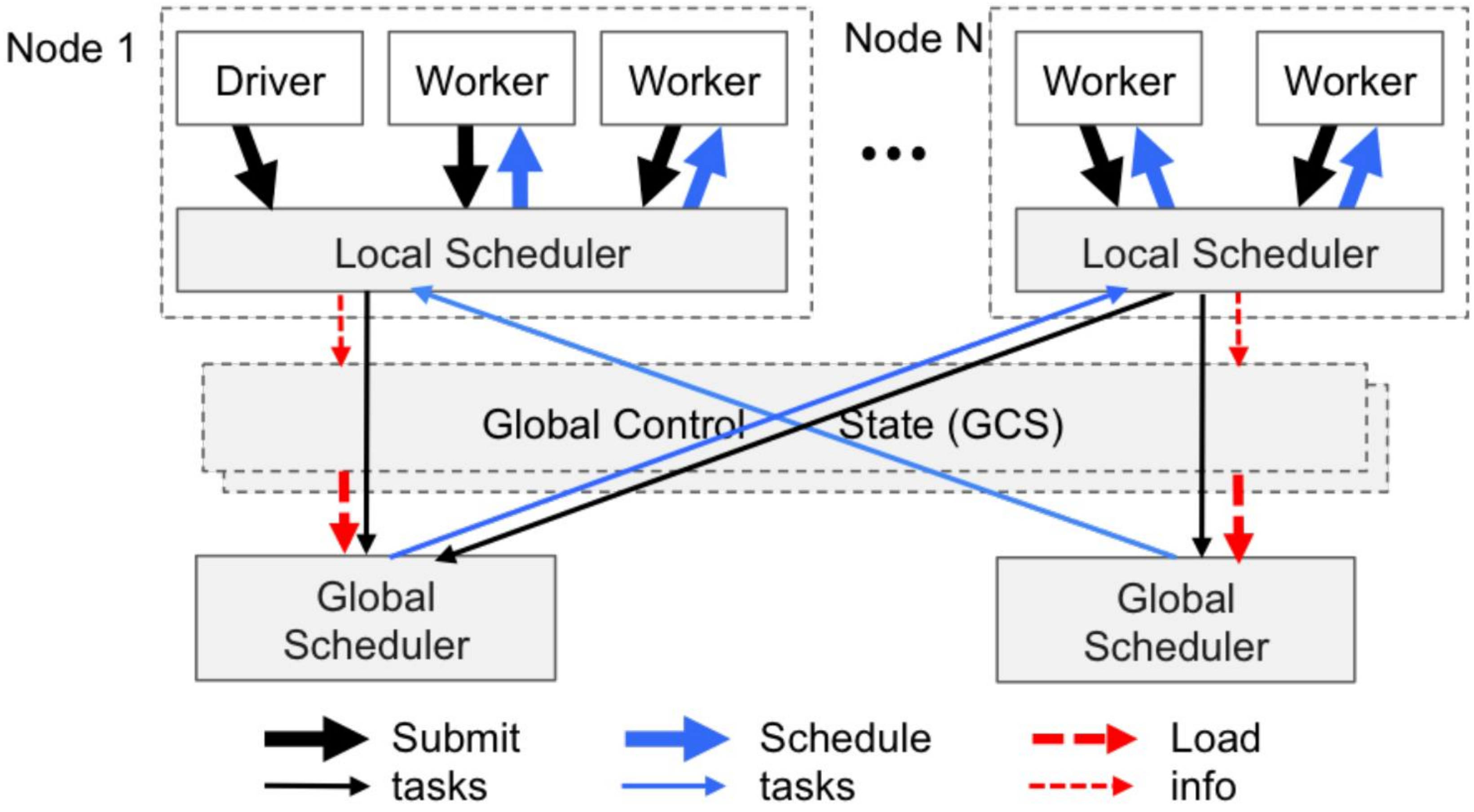
System Layer

- ***Global Control Store***: Stores all up-to-date metadata and control state information in the system.
- ***Bottom-Up Distributed Scheduler***: Tasks are submitted to the local scheduler first, which delegates to the global scheduler if necessary.
- ***In-Memory Distributed Object Store***: Shared memory on workers and actors to share data efficiently.
  - Object reconstruction by ‘replaying’ computation subgraphs with all inputs available.

Architecture



Bottom-up Distributed Scheduler



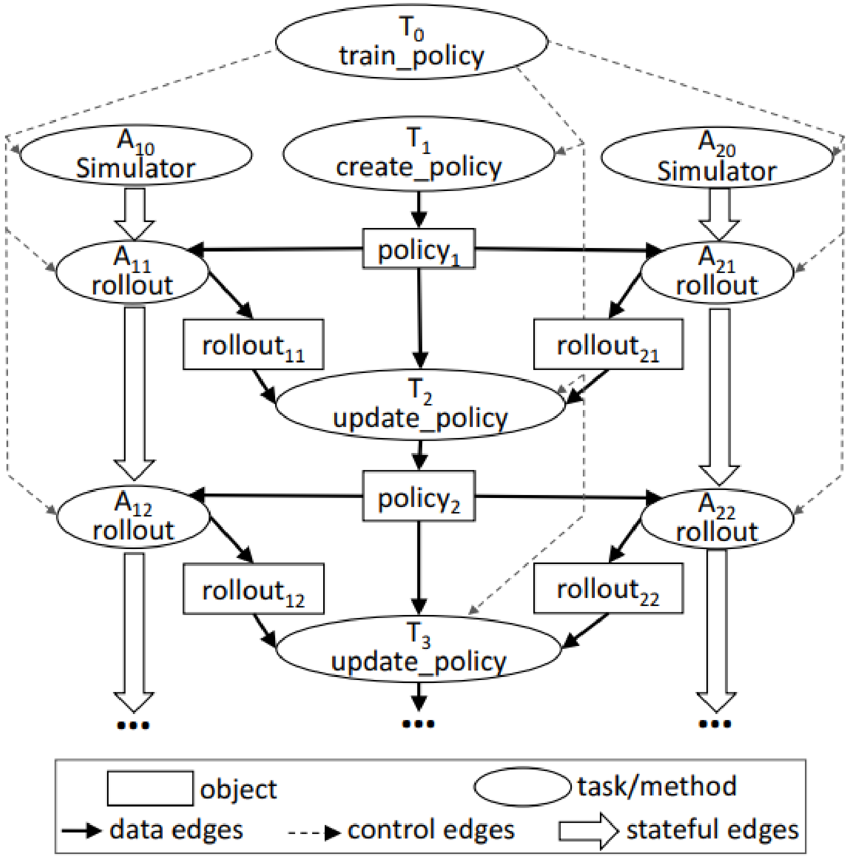
Task Graph

```
@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)
```



Evaluation of Performance

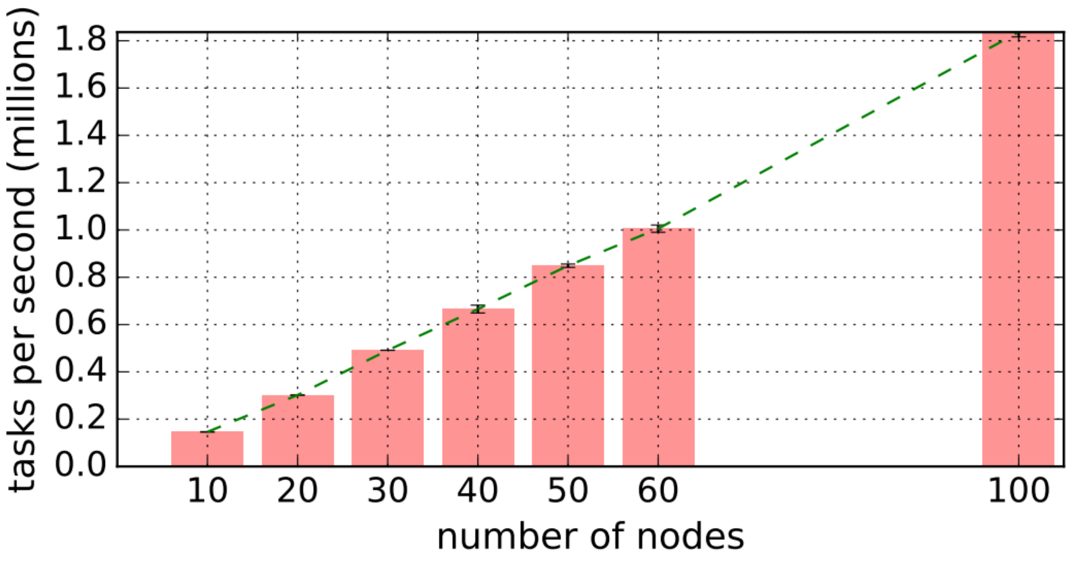


Figure 7: End-to-end scalability of the system is achieved in a linear fashion, leveraging the GCS and bottom-up distributed scheduler. Ray reaches 1 million tasks per second throughput with 60 m4.16xlarge nodes and processes 100 million tasks in under a minute. We omit $x \in \{70, 80, 90\}$ due to cost.

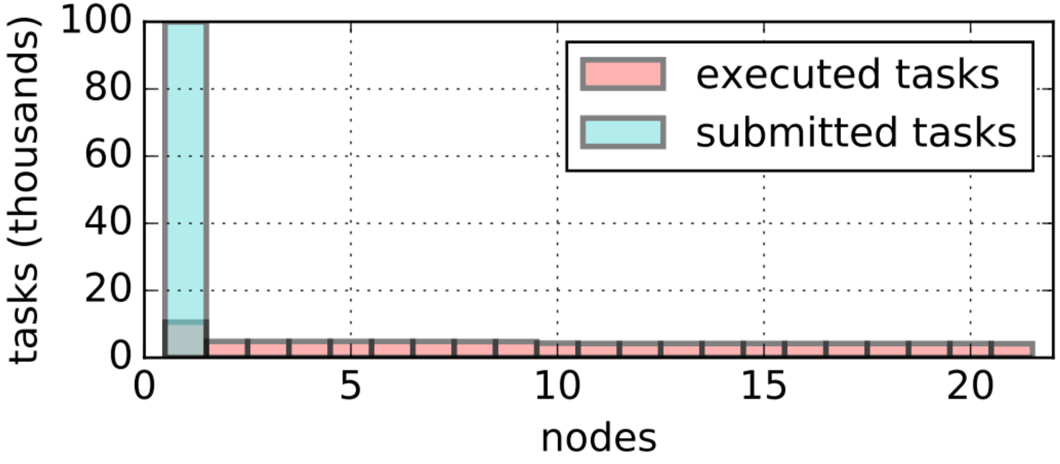
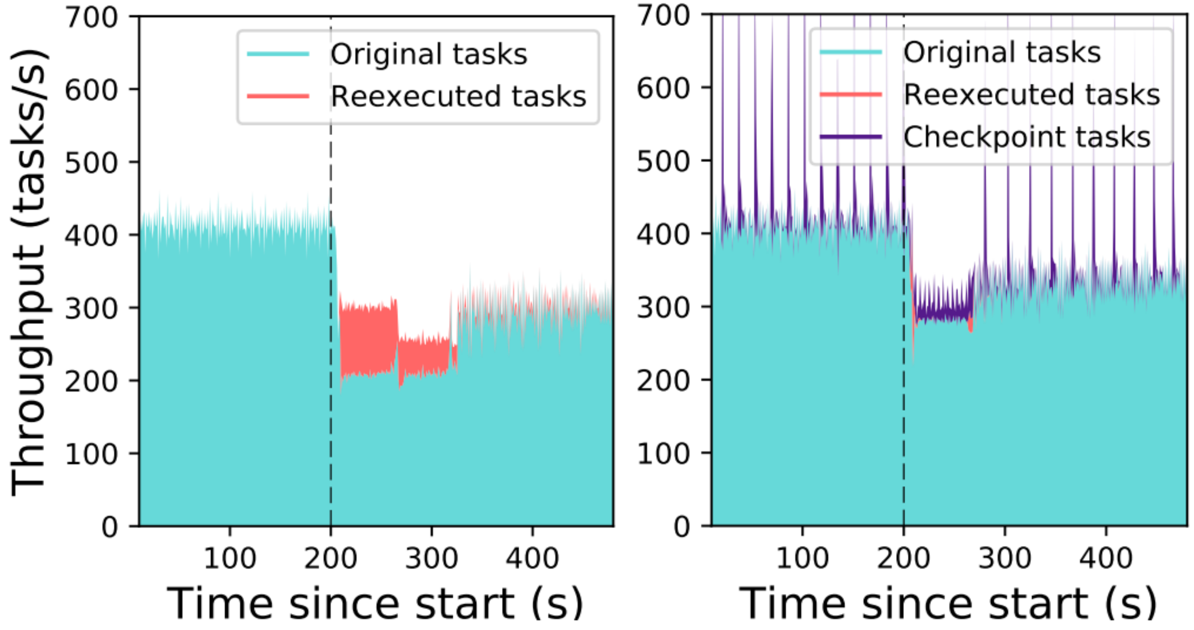


Figure 8: Ray maintains balanced load. A driver on the first node submits 100K tasks, which are rebalanced by the global scheduler across the 21 available nodes.

Evaluation of Performance (Checkpointing)



(a) without checkpointing (b) with checkpointing

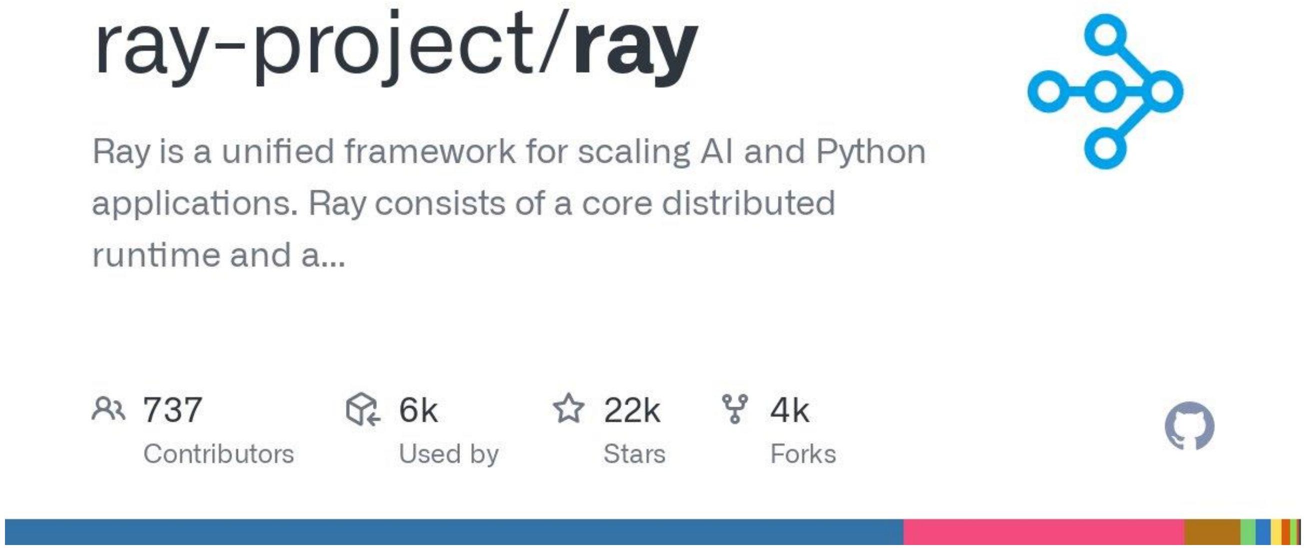
Figure 11: Fully transparent fault tolerance for actor methods. The driver continually submits tasks to the actors in the cluster. At $t = 200$s, we kill 2 of the 10 nodes, causing 400 of the 2000 actors in the cluster to be recovered on the remaining nodes.

Problems

- Very simple API
- Requires manual configuration of Global Control Store shards and global schedulers

Where is Ray today?

- Successful open source project
- RLlib: Abstractions for Distributed Reinforcement Learning
- Tune: A Research Platform for Distributed Model Selection and Training



Questions and discussion…