Ray: A Distributed Framework for Emerging AI Applications

R. Nishihara, R. Moritz, et al.

Presented by Harri Bell-Thomas

22/10/2019

Reinforcement Learning: Agents

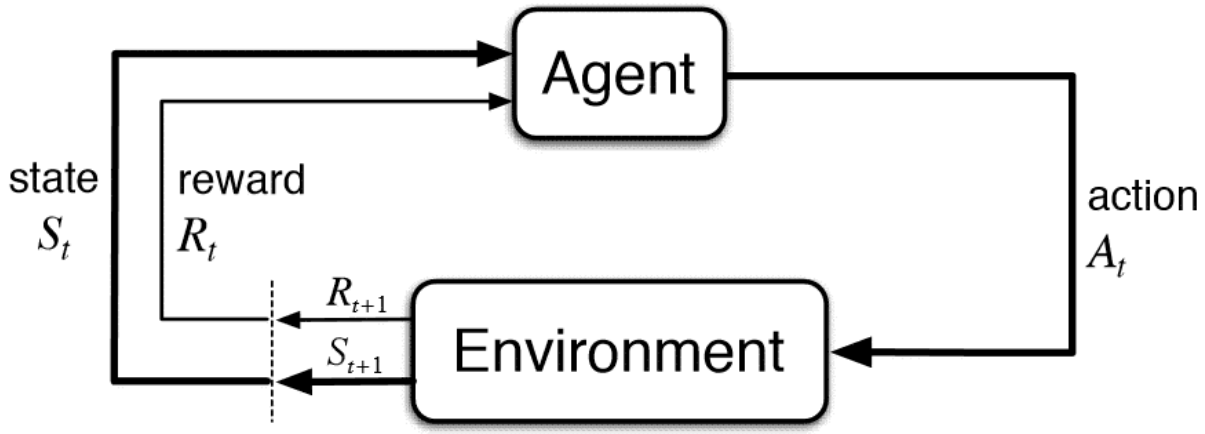


Figure: A traditional RL simiulation loop.

Reinforcement Learning: Policy Training

```
train_policy(environment):
  policy ← initial_policy()
  while (policy has not converged):
    trajectories ← []
    // generate k rollouts and use them to update policy
    for i from 1 to k:
      trajectories.append(rollout(policy, environment))
    policy = policy.update(trajectories)
  return policy
```

Figure: Simplified RL policy training pseudocode.

Requirement Statement

A system optimised for real-time Reinforcement Learning must;

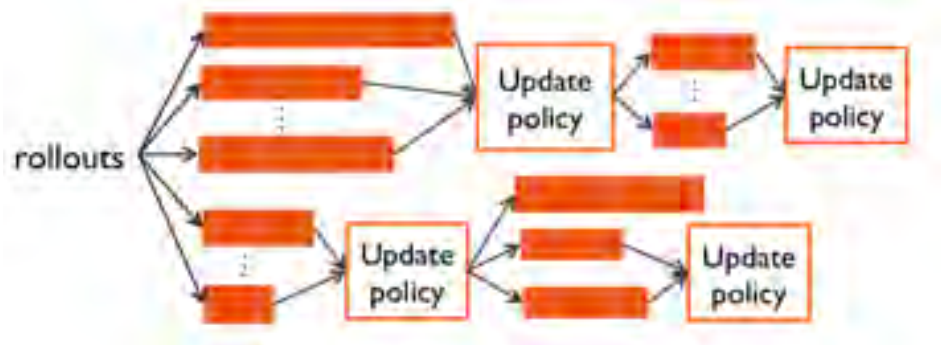
- Support heterogeneous tasks...
- ... within dynamically changing computation graphs...
- ... at a scale of more than a million tasks per second...
- ... with sub millisecond level latencies.

Ray: Overview

Distributed RL computation framework for Python, offering;

1. A dynamic, highly concurrent, task graph.
2. Low latency distributed task scheduling.
3. Heterogeneous computations.
4. Supports both the task-parallel and actor programming models.
5. Horizontal scalability (scale-out).
6. Transparent fault tolerance to aid debugging.

Computation Graph



Key takeaway points;

- Dynamic graphs can't be properly batched.
- Enables *nested remote functions*.
- Requires a number of world states to maintained simultaneously.

Architecture

Architecture: Overview

- Application Layer

  Consisting of; Drivers; Workers; and Actors.

- System Layer
  1. Global Control Store (GCS)
  2. Distributed Scheduler
  3. Apache Arrow [1], an in-memory object store

Architecture: Global Control Store (GCS)

- Repository for all shared system state, results, and metadata.
  1. Every task specification.
  2. The code for every remote function.
  3. The current computation graph.
  4. The current locations of all copies of objects.
  5. Every scheduling event.

- Powerful as this allows the rest of the system to be stateless.
- Scales via sharding (cf. Redis Cluster Sharding [2]).
- Important to note the separation of the data (GCS) and control plane (distributed scheduler).

Architecture: Schedulers

Described as a *bottom-up* decentralised scheduling system in two parts; the global scheduler and per-node local schedulers.

Tasks are submitted to their node's local scheduler first, but passed on to the global scheduler if;

- The node is overloaded.
- The task inputs are not held locally.
- The node can't satisfy the resource requirements.

Aside: Stateful 3^{rd} Party Libraries in a Stateless System

Alongside typical task-parallel execution, Ray supports the Erlang *Actor* model.

- Wrap each stateful process in an *Actor* object.
- Use additional *stateful edges* in the computation graph to track changes.
- This makes task replay deterministic as the versions of state are tracked and accounted for.

Architecture: Diagram

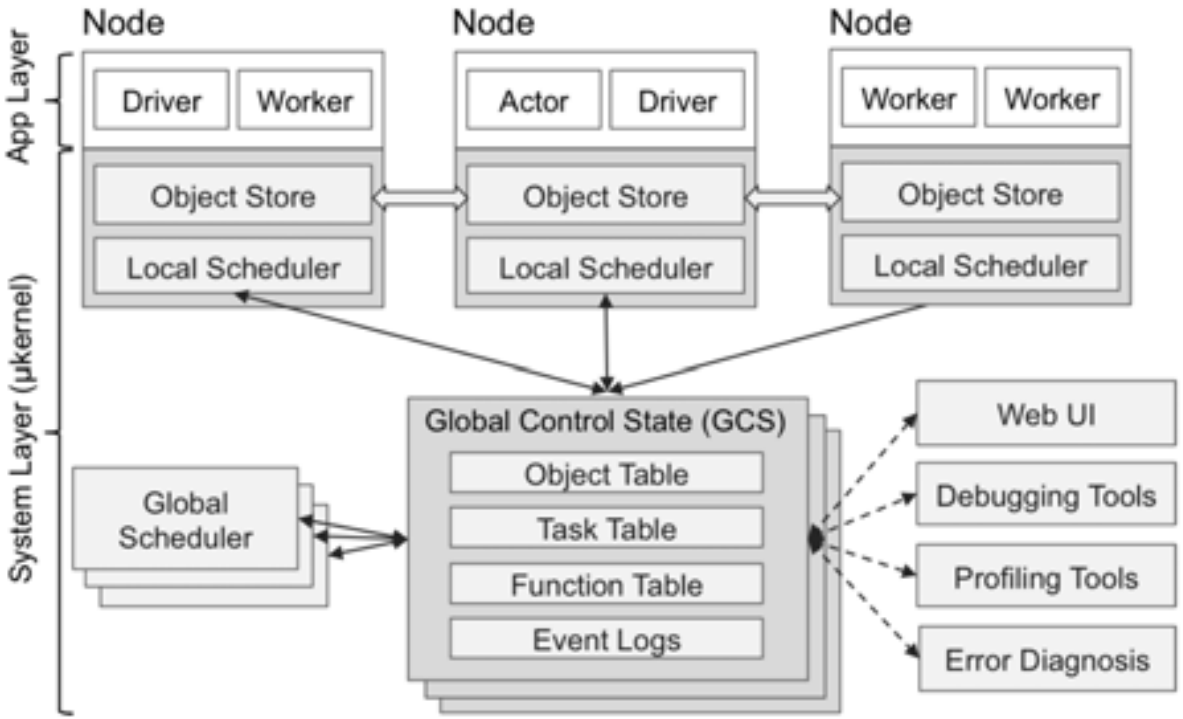


Figure: Ray's high-level system design.

Architecture: Example

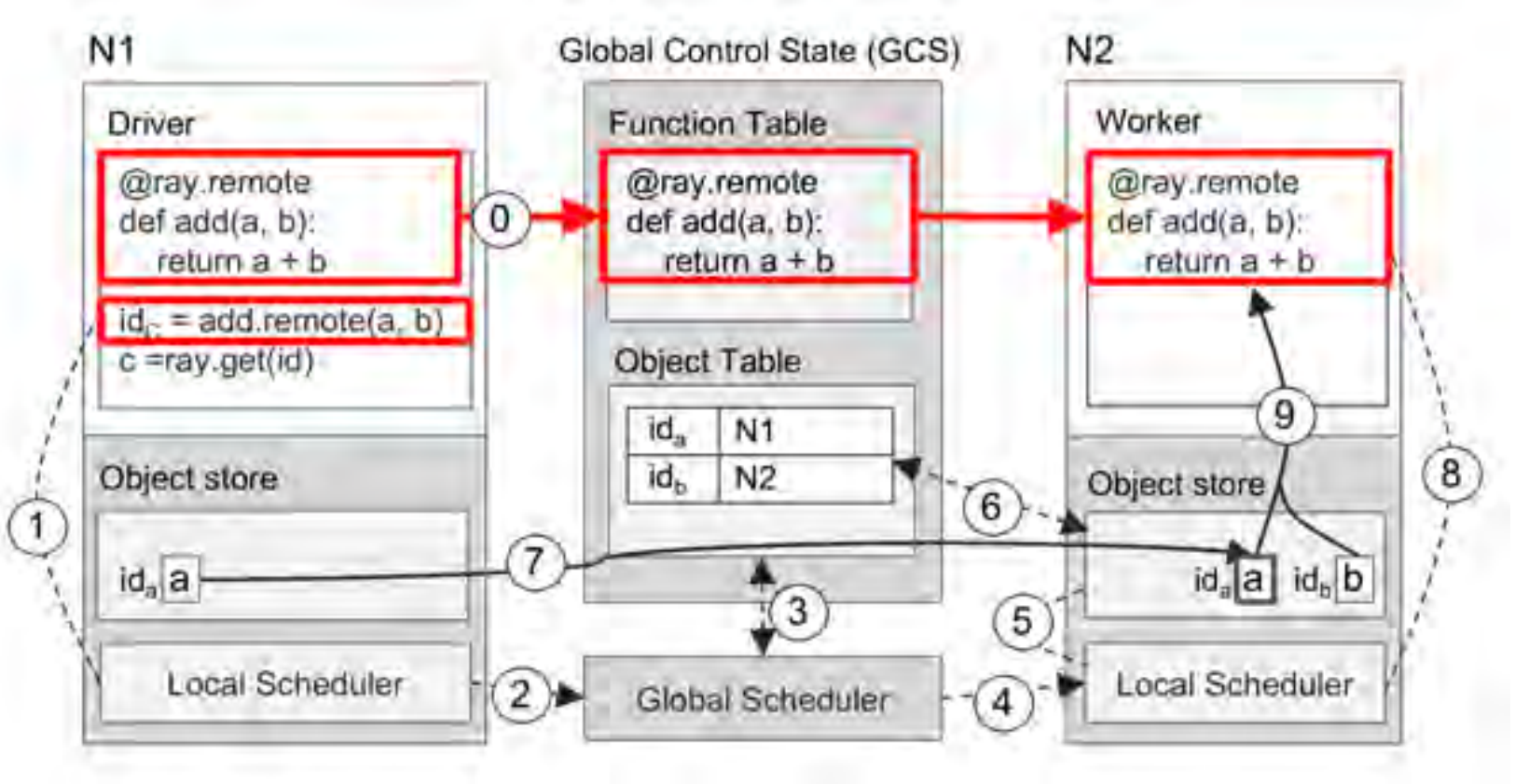


Figure: A stepped through example of executing a remote function in Ray.

System Evaluation

Performance

- Demonstrably linear scaling as number of nodes increases.
  Up to 1.8 million tasks per seconds.
- Peak object store throughput $>$ 15 GB/s.
  Peak IOPS 18k, giving $\sim 56\mu s$/operation.
- Tests against real-world RL workloads extremely positive.
  - Ray scales much better than other solutions.
  - In one test, Ray beats the previous industry best time by a factor of 3.
  - Highly granular optimisation has a large impact; the scheduler allocates resources, such as GPUs, in a more efficient manner.

Fault Tolerance

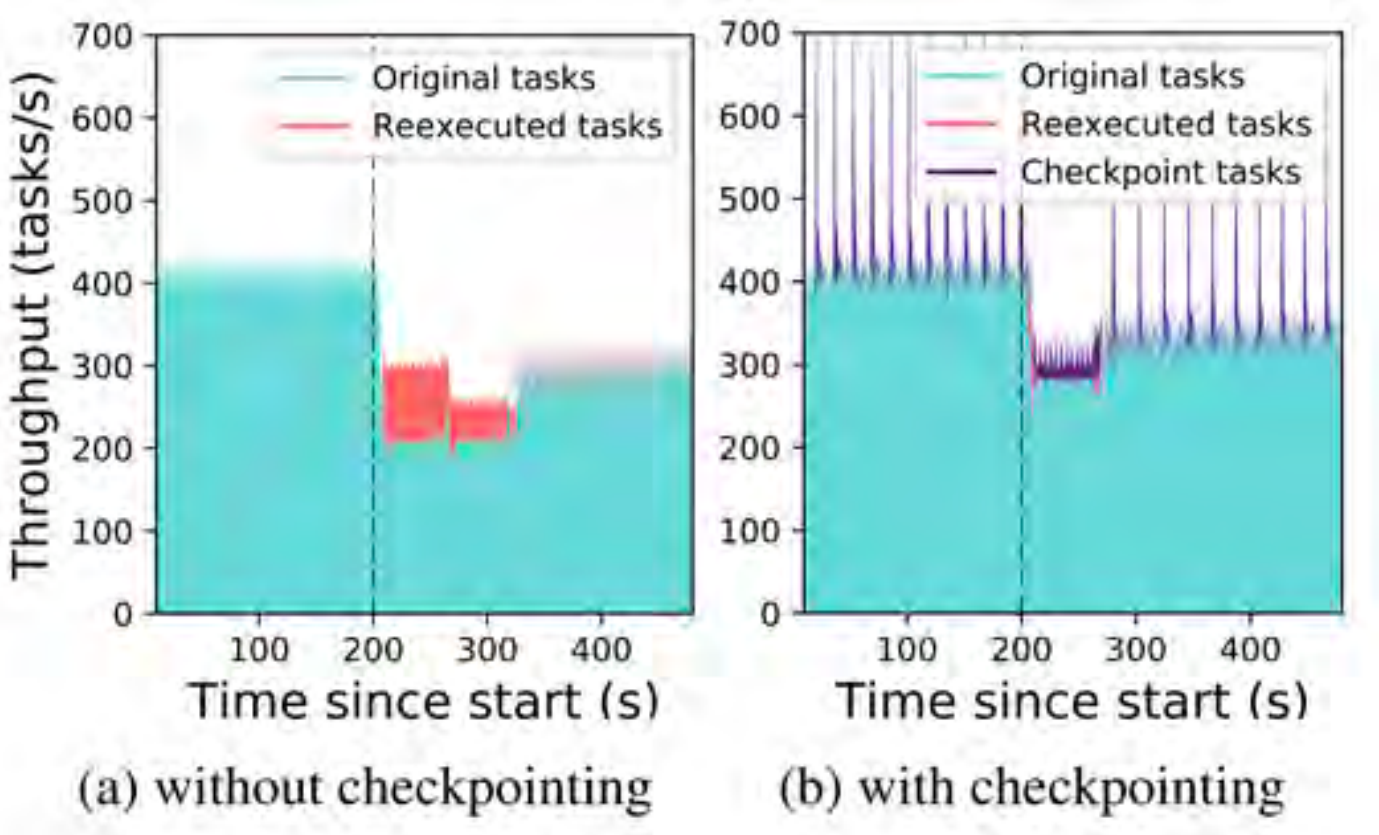


Figure: Ray recovering from node failures. Throughput remains maximal throughout, and using task replay the computation still completed successfully.

Related Work and Discussion Points

Related Works

- MapReduce [3], Spark [4]
  - Centralised scheduler on a master node.
  - BSP execution model, without the abstraction of Actors.
- Dask [5], Ciel [6]
  - Centralised scheduling.
  - Dynamic task graphs.
  - Lacks the Actor abstraction.
- OpenMPI [7]
  - Comparatively hard to program.
  - Requires explicit coordination for dynamic, heterogeneous task graphs.
  - No fault tolerance by default.

Discussion Points / Critique

- Claims, but does not show, it is designed for 'emergent' and 'next generation' AI applications – just good at RL.
- Trusts the user to make important allocation decisions. `@ray.remote(num_gpus=2)`
- Ignore issues around decentralising the data plane. Is it susceptible to split-brain failures when the network fails?
- Appear to be vulnerable to straggler nodes (cf. MapReduce [3]). There is no mechanism for detecting or handling this.
- No mention of how the system identifies what resources it has, and how the global scheduler interfaces with the infrastructure. Essential to the quoted performance metrics.

Bibliography I

Apache Arrow. https://arrow.apache.org/. Accessed: 2019-10-20.

Redis Cluster Sharding. https://redis.io/topics/partitioning. Accessed: 2019-10-20.

Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *OSDI'04: Sixth Symposium on Operating System Design and Implementation*. San Francisco, CA, 2004, pp. 137–150.

Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. "Spark: Cluster Computing with Working Sets". In: *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*. HotCloud'10. Boston, MA: USENIX Association, 2010, pp. 10–10. URL: http://dl.acm.org/citation.cfm?id=1863103.1863113.

Dask Development Team. *Dask: Library for dynamic task scheduling*. 2016. URL: https://dask.org.

Bibliography II

Derek Gordon Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. "CIEL: A Universal Execution Engine for Distributed Data-Flow Computing". In: *NSDI*. 2011.

Richard L. Graham, Timothy S. Woodall, and Jeffrey M. Squyres. "Open MPI: A Flexible High Performance MPI". In: *Proceedings of the 6th International Conference on Parallel Processing and Applied Mathematics*. PPAM'05. Poznań, Poland: Springer-Verlag, 2006, pp. 228–239. ISBN: 3-540-34141-2, 978-3-540-34141-3. DOI: 10.1007/11752578_29. URL: http://dx.doi.org/10.1007/11752578_29.