A Summary of *Ray: A Distributed Framework for Emerging AI Applications*

Jonah Rosenblum (jonaher), Yiran Si (yiransi), Wei-Chung Liao (wchliao)

Problem and Motivation

Multiple groups and organizations are trying to develop new data-intensive applications in AI, namely in reinforcement learning. There is a high demand for these RL algorithms as they are able to solve extremely complex problems such as robotic manipulation and autonomous aerial vehicles. Simulations are used to figure out how well an RL algorithm can make choices based on the policies, which feeds its data to a distributed training network which can help improve the RL algorithm's policies, and after the algorithm must be served so that it can again interact with the simulation or a live environment. This must all be done by a framework that can support dynamic execution at the scale of millions of tasks per second, each one executing at ms latency. Most existing frameworks only meet several of these requirements, and none of them support a "full-stack" framework that supports simulation, training, and serving all in one.

Hypothesis

There are currently too many instances of "one-off" implementations of RL systems for specialized applications. In order to save development time in the future and improve on past systems we need a general-purpose framework that supports simulation, training, and serving that can also operate at a massive scale with fault tolerance and on heterogeneous task sizes/hardware.

Solution Overview

Programming Model

The two primary programming abstractions offered by Ray are *tasks*, which are stateless functions simply taking input and returning an output, and *actors*, which are stateful methods best thought of as a "class" in a traditional programming language. Actors can contain state and have different methods invoked on it.

Computation Model

Ray turns the source program into a graph representing the computation to be completed. This graph shows dependencies between data and tasks which has the benefit of allowing us to keep track of lineage much like how we did in Spark/RRD.

Global Control Store

At the center of Ray is a large key-value store used to manage fault tolerance, handle global scheduling, and task dispatch all independently of one another to ensure scalability. It takes advantage of sharding to help avoid becoming a bottleneck.

Scheduling

Using only a global centralized scheduler would not work at a massive scale, so an alternative approach is taken. Each node has a local scheduler that can schedule tasks without consulting any other node or central authority. If the node determines that it does not have adequate resources to schedule a task it will send it to a global scheduler. The global scheduler receives heartbeats from all nodes containing information about resource availability and schedules tasks it receives accordingly.

Limitations and Possible Improvements

The main limitation listed in the paper relates to the scheduling. Because of the strict latency requirements and the "bottom-up scheduling," it seems as though non-optimal scheduling decisions can be made without full knowledge of the computation graph. This is a general framework, and so applying specific optimizations to help some cases is difficult without hurting the performance of other cases. We thought of two potential workarounds, the first being certain optimization classes that can be opted in to. For example, if you (the developer) have an RL algorithm that follows a specific commonly known paradigm, certain optimizations could be enabled in the program through a compilation flag or adding a specific argument to a function call. Another strategy to overcome this scheduling issue is to have the system adapt to the specific RL algorithm over the course of the program runtime - this is to say that the system can try and learn/infer characteristics about your algorithm as it executes it and then make optimization predictions in real-time. Obviously, both of these ideas have potential issues but might be good places to start looking at ways of overcoming this downside of Ray.

Summary of Class Discussion

If a lot of workers are asking things to do at the same time, what should the master do? GCS decouples task queuing and dispatch making it capable of dealing with this issue at scale.

Wouldn't replica and sharding cause overhead? Yes, there is a bit, but it is necessary. If you don't, the master will overflow eventually. There is some overhead for writes, but not so much for reads.

What factors are taken into consideration for schedulers to make the decision to run a function locally vs remotely? It depends on the strategy of the local scheduler. If no available local resources are available to run the task, it chooses the remote scheduler.

Do we have schedulers at every node or is the scheduler like a separate service layer that has knowledge of all nodes and does the assignment? We have a scheduler at every node, and each node knows how to access the global scheduler if it needs to offload a task.

Is fault tolerance really necessary? They don't give a reason to directly justify the overhead. With the second paper (Lineage Stash) this overhead becomes a lot less of an issue. Everything is a tradeoff, you don't get anything for free so while fault tolerance is a large benefit in most systems and we enjoy the debugging bonus it gives us we cannot get this without paying some (probably small) overhead.

A Summary of *Lineage Stash: Fault Tolerance Off the Critical Path*

Jonah Rosenblum (jonaher), Yiran Si (yiransi), Wei-Chung Liao (wchliao)

Problem and Motivation

Fault tolerance is growing in importance for cluster computing frameworks. Fault tolerance strategies used by current systems have tradeoffs. Checkpointing exhibits low overhead during normal operation but high overhead during recovery, while lineage-based solutions have higher overheads during execution but are much faster in recovery. The motivation is to develop some techniques that would significantly reduce the runtime overhead of lineage-based approaches without impacting recovery efficiency.

Hypothesis

If we send the lineage along with the task we can make updating logging dependencies asynchronous. This will greatly reduce the previous overhead of traditional lineage-based approaches to fault tolerance while still allowing for fast recovery in a large-scale system.

Solution Overview

The solution to the problem is lineage stash - a decentralized causal logging technique. The main idea is to extend the ideas of both lineage reconstruction and causal logging by identifying the nondeterministic events that must be logged for application correctness and design an efficient protocol to store this information off the critical path of execution.
Each worker asynchronously stores the lineage and forward only the most recent part which has not been durably stored yet. In particular, Each worker keeps a lineage stash in local memory containing all tasks that it has seen recently. Then each worker runs a local protocol to flush its stash to a global store, a reliable key-value store that maps task ID to specification.
To ensure global consistency (recovery consistency), rules are defined for global orderings. The lineage of a task consists of the task itself and the lineage of all its dependencies. Lineage consistency after failure is ensured by the following rule: each worker would forward uncommitted lineage with each submitted task. However, only nondeterministic events need to be forwarded to receiving nodes. Deterministic events only need to remember tasks that have been submitted so far.

Limitations and Possible Improvements

When there is nondeterministic behavior and unlimited forwarding allowed the uncommitted lineage can grow too large and cause serious latency issues. If, however, the lineage can only be forwarded to at most 8 nodes, the latency will stabilize. Therefore, one possible improvement we suggest would be to allow for asynchronous logging until a certain number in the forwarding chain, i.e. 8 nodes, at which point the logging must be done synchronously with an ack. In a real system we can't guarantee that forwarding will be bounded so this seems like a good feature to add to a live system to prevent the lineage from growing out of control.

The recovery protocol can be further optimized by leveraging the property common to decentralized data processing applications: that most processes only send tasks to a small number of other processes.

Summary of Class Discussion

When reading systems papers, think about how it might break. It is possible that two nodes sending each other tasks back and forth could create a situation which breaks this because of nondeterminism. Not clear if this is the case but usually in systems papers we will see a paper come out 2-3 years later pointing out a case where this one fails and will introduce some solution to fix it, and so on and so on.

This paper directly addresses one of the main concerns from Ray that there might be a significant drawback from overhead in the fault recovery approach.