# Summary of “Ray: A Distributed Framework for Emerging AI Applications”

Diego Rojas Salvador (darojas)

## Problem and Motivation

The last few years have seen a large growth in the applications of machine learning (ML) techniques and algorithms to tackle large scale problems. With this growth has come a need for flexible and robust systems that have the computational capability required to handle the workloads of these problems in a timely fashion. The development of Ray was particularly motivated by the growth in applications of reinforcement learning (RL), which is the area of ML which deals with learning to operate continuously within an uncertain environment based on delayed and limited feedback. RL requires systems that can allow for the training and serving of policies to an agent and the simulation of an environment for the agent. The cyclic and interactive nature of the process poses challenges for most existing systems used to tackle ML problems.

## Hypothesis

In order to solve the above problem, the authors proposed Ray, a distributed framework that enables simulation to evaluate policies, training in a distributed fashion, and serving policies in interactive closed/open loop scenarios.

The requirements for this system respond to some of the needs include fine grained, heterogeneous computations with duration range from milliseconds to hours; a flexible computation model for both stateless and stateful computations; and dynamic execution, which is necessary because the order in which computations finish is not always known in advance, and the result of a computation can determine future computations.

## Solution Overview

Ray was designed to implement a dynamic task graph computational model, on top of which Ray provides an actor and a task-parallel programming abstraction.

The programming model is defined in terms of two constructs: tasks and actors. A task represents the execution of a remote function on a stateless worker. An actor represents a stateful computation. Actors expose methods that can be invoked remotely

and are executed serially. Similarly to tasks, methods return a future, but they differ in that they execute on stateful workers. It is important to mention that data stored separately in a store and actors only remember what data they care about.

The computational model models an application as a graph of dependent tasks that evolves during execution. In this model, the graph is comprised of *nodes* representing data objects, remote function invocations (tasks) and actor method invocations, and *edges* (data, control and stateful edges) that capture dependencies between nodes.

Ray's *architecture* is organized in an application layer and a system layer. The *application layer* implements an API, and it consists of three types of processes: a driver, which is a process executing user program; a worker, which is a stateless process that executes tasks; and an actor, which is a  stateful process that executes the methods it exposes. The *system layer* provides the system's high scalability and fault tolerance, and it consists of three main components: global control store (GCS), a distributed scheduler, and a distributed object store. The GCS is key-value store that maintains the entire control state of the system. The reason for the GCS is to maintain fault tolerance and low latency for a system that can dynamically spawn millions of tasks per second. This unique feature of Ray differentiates it from other systems used for RL and other ML tasks, which require the usage of other types of distributed storage systems running in parallel with the computing systems. The bottom-up distributed scheduler is a two-level hierarchical scheduler consisting of a global scheduler and per node schedulers. The scheduling policy consists of first scheduling locally and if not possible, then move request to global scheduling. Finally, the in-memory distributed object store is a per node store for the inputs and outputs of every task or stateless computation. This part of the system is in place to allow for zero-copy data sharing between tasks running on the same node.

## Limitations and Possible Improvements

The structure of the scheduler seems to be a limitation of this system. Having a single global scheduler can lead to it being the bottleneck for certain workloads requiring extremely high throughput. An improvement could be having a global scheduler for the specific job. This limitation was addressed in a later version of Ray.

Another limitation comes from the generality in the system's design. Certain workloads and tasks could benefit with specific optimizations. The system as presented in the paper did not provide an easy way of implementing these system layer improvements.

# Summary of “Dynamic Control Flow in Large-Scale Machine Learning”

## Problem and Motivation

Advances in ML and its large scale applications have led to a growing need for high performance systems that are scalable, expressive and flexible enough to support both production and research. One of the characteristics required by some of these systems is fine-grained dynamic control flow. Reinforcement learning and recurrent neural networks are two examples of current techniques that require this for training and inference because they rely on recurrence relations, data-dependent conditional execution, and other features that call for dynamic control flow. The needs expressed above led the authors to tackle the problem of designing and implementing a system that would allow for rapid control-flow decisions across a set of computing devices in a distributed system.

## Hypothesis

In order to provide a system better equipped to perform these kinds of tasks, the authors present a programming model for machine learning that includes dynamic control flow, and an implementation of this model as an extension to TensorFlow. A benefit of this work is that it provided additional functionality and flexibility to an already robust system, allowing for parallel and distributed execution across CPUs, GPUs, and custom machine learning accelerators.

## Solution Overview

In order to extend TensorFlow to allow dynamic control flow, the programming interface that was developed focused on constructs that would allow conditional and iterative computation. These interfaces are important because they allow for computations that are common in recursive neural networks and reinforcement learning.

- Conditional: the result is a tuple of tensors, representing the result of the branch that executes.
- Iterative: returns a boolean tensor, and tuple of updated loop variables.

From the architecture perspective, this paper maintains TensorFlow’s execution of a dataflow graph, but in order to introduce dynamic control flow some changes were needed. Before the addition of control flow, each operation in the graph executed exactly once, but with control flow an operation in a loop can execute zero or more

times. This means that rendezvous keys must be generated dynamically to distinguish multiple invocations of the same operations.

In addition, due to the possible execution of the same operation many times, the local executor in a node now manages the (possibly concurrent) execution of multiple instances of the same operation, and is in charge of determining the completion of the entire execution.

Whenever the subgraph of a conditional branch or loop body is partitioned across devices the local executors communicate only via Send and Recv operations, and a centralized coordinator is involved only in the event of completion or failure.

## Limitations and Possible Improvements

This paper presented a well designed solution to add functionality (dynamic control flow) to an already robust and reliable system (TensorFlow.) It achieved its goal with a strong approach. One possible limitation of the paper refers to the specific example of distributed execution of a loop. As mentioned in class, it seemed like the partition presented did not provide many benefits given that the operations of the loop body were all executed on a single node while the other device executes the loop predicate, which poses the question of how distributed computing of a loop is defined for the authors.

## Discussion

Global Control Store in charge of storing the location of each object. If we lost a node we could use the Global Control Store for fault tolerance. Having the Global Control Store also takes away the worries of having dependencies across multiple systems running together. With the inclusion of the Global Control Store in the system, changes can be implemented in a single place instead of having to propagate changes in dependencies.

**Update to Ray's scheduling**
Whatever machine runs the driver, it works as a driving scheduler for the whole job. No single global scheduler, but a global driver/scheduler for the specific job. With this change, the driver and scheduler fail together. Fail sharing: they all fail or they all survive.

If tasks are too small, global scheduler becomes bottleneck because there is not enough throughput. Important to have flexibility for the global scheduling. Need good load balancing so that schedulers don't hog the same nodes. Distributed scheduling.

**Are there any specific size of tasks?**
Assumption is that the tasks should last in the range of several milliseconds. Not tens.

**Evaluation**
They show that they are as good in the tasks provided in others, but they also added a new functionality: Reinforcement Learning and the simulation of an environment.

The system was doing everything well enough compared to other solutions, but its greatest advantage is that it is easier to use due to the lack of need to focus on dependencies (not having to deal with all the plumbing etc.)

**Dynamic Control Flow**
System supports distributed execution, but for while loops we only allow a specific kind of partition. When we partition any edge (model partition), the system can reap the benefits of processing some layers in some GPU, and send to another one where some other part of the model resides.

**Closing Discussion thoughts**

How can we make a better system? Perhaps doing better in GAN and then for everything else comparably. Emerging workload that might be very hot. In order to develop a new paper or product, focus on one thing you can’t do very well. Try to achieve a point at which you are the best, or the only ones at some specific task.