BERKELEY ARTIFICIAL INTELLIGENCE RESEARCH

Ray: A Distributed System for AI

Robert Nishihara and Philipp Moritz *Jan 9, 2018*

As machine learning algorithms and techniques have advanced, more and more machine learning applications require multiple machines and must exploit parallelism. However, the infrastructure for doing machine learning on clusters remains ad-hoc. While good solutions for specific use cases (e.g., parameter servers or hyperparameter search) and high-quality distributed systems outside of AI do exist (e.g., Hadoop or Spark), practitioners developing algorithms at the frontier often build their own systems infrastructure from scratch. This amounts to a lot of redundant effort.

As an example, take a conceptually simple algorithm like Evolution Strategies for reinforcement learning. The algorithm is about a dozen lines of pseudocode, and its Python implementation doesn't take much more than that. However, running the algorithm efficiently on a larger machine or cluster requires significantly more software engineering. The authors' implementation involves thousands of lines of code and must define communication protocols, message serialization and deserialization strategies, and various data handling strategies.

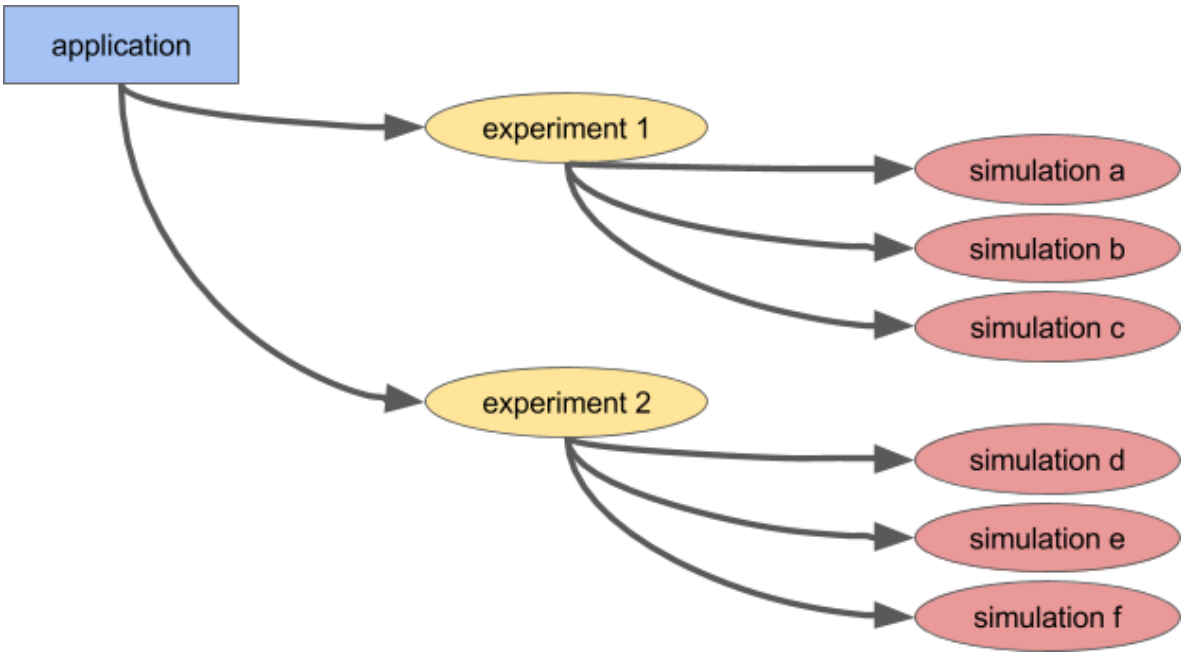
One of Ray's goals is to enable practitioners to turn a prototype algorithm that runs on a laptop into a high-performance distributed application that runs efficiently on a cluster (or on a single multi-core machine) with relatively few additional lines of code. Such a framework should include the performance benefits of a hand-optimized system without requiring the user to reason about scheduling, data transfers, and machine failures.

An Open-Source Framework for AI

Relation to deep learning frameworks: Ray is fully compatible with deep learning frameworks like TensorFlow, PyTorch, and MXNet, and it is natural to use one or more deep learning frameworks along with Ray in many applications (for example, our reinforcement learning libraries use TensorFlow and PyTorch heavily).

Relation to other distributed systems: Many popular distributed systems are used today, but most of them were not built with AI applications in mind and lack the required performance for supporting and the APIs for expressing AI applications. The following features are missing (in various combinations) from today's distributed systems:

- Support for millisecond level tasks and millions of tasks per second
- Nested parallelism (parallelizing tasks within tasks, e.g., parallel simulations inside of a hyperparameter search) (see the figure below)
- Arbitrary task dependencies determined dynamically at runtime (e.g., to avoid waiting for slow workers)
- Tasks operating on shared mutable state (e.g., neural net weights or a simulator)
- Support for heterogeneous resources (CPUs, GPUs, etc)



A simple example of nested parallelism. One application runs two parallel experiments (each of which is a long-running task), and each experiment runs a number of parallel simulations (each of which is also a task).

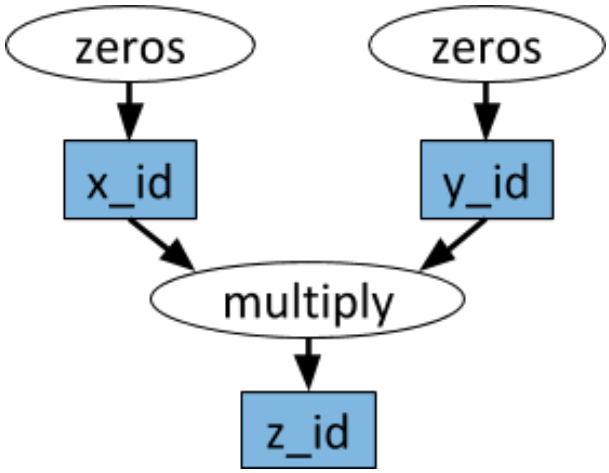
There are two main ways of using Ray: through its lower-level APIs and higher-level libraries. The higher-level libraries are built on top of the lower-level APIs. Currently these include Ray RLlib, a scalable reinforcement learning library and Ray.tune, an efficient distributed hyperparameter search library.

Ray Lower-Level APIs

The goal of the Ray API is to make it natural to express very general computational patterns and applications without being restricted to fixed patterns like MapReduce.

Dynamic Task Graphs

The underlying primitive in a Ray application or job is a *dynamic task graph*. This is very different from the computation graph in TensorFlow. Whereas in TensorFlow, a computation graph represents a neural network and is executed many times in a single application, in Ray, the task graph represents the entire application and is only executed a single time. The task graph is not known up front. It is constructed dynamically as an application runs, and the execution of one task may trigger the creation of more tasks.



An example computation graph. The white ovals refer to tasks, and the blue boxes refer to objects. Arrows indicate that a task depends on an object or that a task creates an object.

Arbitrary Python functions can be executed as tasks, and they can depend arbitrarily on the outputs of other tasks. This is illustrated in the example below.

```
# Define two remote functions. Invocations of these functions create
# that are executed remotely.

@ray.remote
def multiply(x, y):
    return np.dot(x, y)

@ray.remote
def zeros(size):
    return np.zeros(size)

# Start two tasks in parallel. These immediately return futures and
# tasks are executed in the background.
x_id = zeros.remote((100, 100))
y_id = zeros.remote((100, 100))

# Start a third task. This will not be scheduled until the first two
# tasks have completed.
z_id = multiply.remote(x_id, y_id)

# Get the result. This will block until the third task completes.
z = ray.get(z_id)
```

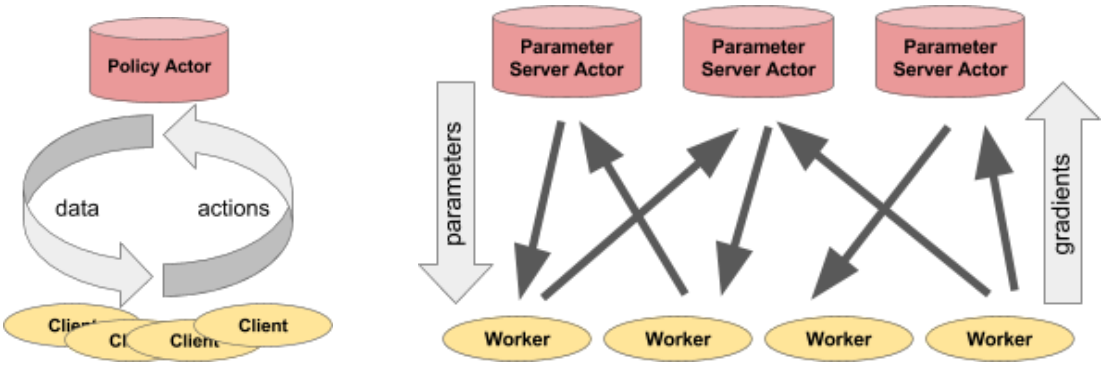
Actors

One thing that can't be done with just the remote functions and tasks described above is to have multiple tasks operating on the same shared mutable state. This comes up in multiple contexts in machine learning where the shared state may be the state of a simulator, the weights of a neural network, or something else entirely. Ray uses an actor abstraction to encapsulate

mutable state shared between multiple tasks. Here's a toy example of how to do this with an Atari simulator.

```
import gym

@ray.remote
class Simulator(object):
    def __init__(self):
        self.env = gym.make("Pong-v0")
        self.env.reset()

    def step(self, action):
        return self.env.step(action)

# Create a simulator, this will start a remote process that will run
# all methods for this actor.
simulator = Simulator.remote()

observations = []
for _ in range(4):
    # Take action 0 in the simulator. This call does not block and
    # it returns a future.
    observations.append(simulator.step.remote(0))
```

Though simple, an actor can be used in very flexible ways. For example, and actor can encapsulate a simulator or a neural network policy, and it can be used for distributed training (as with a parameter server) or for policy serving in a live application.



Left: *An actor serving predictions/actions to a number of client processes.* ***Right:*** *Multiple parameter server actors performing distributed training with multiple worker processes.*

Parameter server example

A parameter server can be implemented as a Ray actor as follows:

```
@ray.remote
class ParameterServer(object):
    def __init__(self, keys, values):
        # These values will be mutated, so we must create a local co
        values = [value.copy() for value in values]
        self.parameters = dict(zip(keys, values))

    def get(self, keys):
        return [self.parameters[key] for key in keys]

    def update(self, keys, values):
```

```
        # This update function adds to the existing values, but the
        # function can be defined arbitrarily.
        for key, value in zip(keys, values):
            self.parameters[key] += value
```

See a more complete example.

To instantiate the parameter server, do the following.

```
parameter_server = ParameterServer.remote(initial_keys, initial_valu
```

To create four long-running workers that continuously retrieve and update the parameters, do the following.

```
@ray.remote
def worker_task(parameter_server):
    while True:
        keys = ['key1', 'key2', 'key3']
        # Get the latest parameters.
        values = ray.get(parameter_server.get.remote(keys))
        # Compute some parameter updates.
        updates = …
        # Update the parameters.
        parameter_server.update.remote(keys, updates)

# Start 4 long-running tasks.
for _ in range(4):
    worker_task.remote(parameter_server)
```

Ray High-Level Libraries

Ray RLlib is a scalable reinforcement learning library built to run on many machines. It can be used through example training scripts as well as through a Python API. It currently includes implementations of:

- A3C
- DQN
- Evolution Strategies
- PPO

We are working on adding more algorithms. RLlib is fully compatible with the OpenAI gym.

Ray.tune is an efficient distributed hyperparameter search library. It provides a Python API for use with deep learning, reinforcement learning, and other compute-intensive tasks. Here is a toy example illustrating usage:

```
from ray.tune import register_trainable, grid_search, run_experiment

# The function to optimize. The hyperparameters are in the config
# argument.
def my_func(config, reporter):
    import time, numpy as np
    i = 0
    while True:
```

```
            reporter(timesteps_total=i, mean_accuracy=(i ** config['alph
            i += config['beta']
            time.sleep(0.01)

    register_trainable('my_func', my_func)

    run_experiments({
        'my_experiment': {
            'run': 'my_func',
            'resources': {'cpu': 1, 'gpu': 0},
            'stop': {'mean_accuracy': 100},
            'config': {
                'alpha': grid_search([0.2, 0.4, 0.6]),
                'beta': grid_search([1, 2]),
            },
        }
    })
```

In-progress results can be visualized live using tools such as Tensorboard and rllab's VisKit (or you can read the JSON logs directly). Ray.tune supports grid search, random search, and more sophisticated early stopping algorithms like HyperBand.

More Information

For more information about Ray, please take a look at the following links.

- Our paper
- Our codebase
- Our documentation

Ray can be installed with `pip install ray`. We encourage you to try out Ray and see what you think. If you have any feedback for us, please let us know, e.g., through our mailing list ray-dev@googlegroups.com.

Subscribe to our RSS feed.
Spread the word:

Comments

ALSO ON **BAIR BLOG**

Four Novel Approaches to ...

3 years ago

The BAIR Blog

Exploring Exploration: ...

3 years ago • 1 comment

The BAIR Blog

Unsupervised Me Learning: ...

3 years ago • 2 comments

The BAIR Blog

BAIR Blog Comment Policy

Please be civil and only engage in academic, professional discussions that are relevant to the blog post.

Please read our Comment Policy before commenting.

Got it

7 Comments

Login

Join the discussion...

LOG IN WITH OR SIGN UP WITH DISQUS

Name

4 Share

Best Newest Oldest

Ben

6 years ago edited

How does that compare to a relatively mature Dask project? I have no offilation with it, just noticed that their workload balancing seems to be pretty smart.

1 0 Reply • Share ›

Robert Ben

6 years ago

Dask also aims to improve the ecosystem for parallel/distributed Python. There are lots of practical differences and design decisions.

- Dask supports something very similar to our remote functions. The actor abstraction doesn't have an equivalent and is pretty important for building stateful services like parameter servers.
- Ray uses a distributed scheduling scheme to allow high task throughput (e.g., millions of tasks per second), whereas Dask uses a centralized scheduler.
- Ray focuses a lot on latency, so the latency to submit a task and get the result is about 30x lower in Ray than in Dask.
- Ray has focused more on libraries for machine learning and reinforcement learning, whereas Dask has built more distributed collections libraries (distributed arrays, DataFrames)

collections libraries (distributed arrays, DataFrames).
- We handle data differently from Dask (using shared memory and zero-copy serialization). Dask does serialization with pickle (with some optimizations).
- Ray handles failures (e.g., transparent recovery from machine failures).

1 0 Reply • Share ›

Saman Biook

3 years ago

How is this different from something like Spark?

0 0 Reply • Share ›

B

Bln C

5 years ago

Is it possible to install it on Windows (10)? Seems error arises: "error: command 'x86_64-linux-gnu-gcc' failed with exit status 1". Thanks!!

0 0 Reply • Share ›

Hafeez

5 years ago

Very well written article. Kudos!

0 0 Reply • Share ›

A

Andrew Czeizler

6 years ago

Could this application be used in an end to end streaming machine learning product. Serving a model to a front end visualisation.

0 0 Reply • Share ›

Robert Andrew Czeizler