Ray Documentation

Release 0.1.2

The Ray Team

Aug 20, 2017

Installation

Ray is a flexible, high-performance distributed execution framework.

CHAPTER 1

Installation on Ubuntu

Ray should work with Python 2 and Python 3. We have tested Ray on Ubuntu 14.04 and Ubuntu 16.04.

You can install Ray as follows.

```
pip install ray
```

Building Ray from source

If you want to use the latest version of Ray, you can build it from source.

Dependencies

To build Ray, first install the following dependencies. We recommend using Anaconda.

```
sudo apt-get update
sudo apt-get install -y cmake pkg-config build-essential autoconf curl libtool␣
↪libboost-dev libboost-filesystem-dev libboost-system-dev unzip

# If you are not using Anaconda, you need the following.
sudo apt-get install python-dev  # For Python 2.
sudo apt-get install python3-dev  # For Python 3.

# If you are on Ubuntu 14.04, you need the following.
pip install cmake

pip install numpy cloudpickle funcsigs click colorama psutil redis flatbuffers
```

If you are using Anaconda, you may also need to run the following.

```
conda install libgcc
```

Install Ray

Ray can be built from the repository as follows.

```
git clone https://github.com/ray-project/ray.git
cd ray/python
python setup.py install
```

Test if the installation succeeded

To test if the installation was successful, try running some tests. This assumes that you've cloned the git repository.

```
python test/runtest.py
```

CHAPTER 2

Installation on Mac OS X

Ray should work with Python 2 and Python 3. We have tested Ray on OS X 10.11 and 10.12.

You can install Ray as follows.

```
pip install ray
```

Building Ray from source

If you want to use the latest version of Ray, you can build it from source.

Dependencies

To build Ray, first install the following dependencies. We recommend using Anaconda.

```
brew update
brew install cmake pkg-config automake autoconf libtool boost wget

pip install numpy cloudpickle funcsigs click colorama psutil redis flatbuffers --
↪ignore-installed six
```

If you are using Anaconda, you may also need to run the following.

```
conda install libgcc
```

Install Ray

Ray can be built from the repository as follows.

```
git clone https://github.com/ray-project/ray.git
cd ray/python
python setup.py install
```

Test if the installation succeeded

To test if the installation was successful, try running some tests. This assumes that you've cloned the git repository.

```
python test/runtest.py
```

CHAPTER 3

Installation on Docker

You can install Ray on any platform that runs Docker. We do not presently publish Docker images for Ray, but you can build them yourself using the Ray distribution.

Using Docker can streamline the build process and provide a reliable way to get up and running quickly.

Install Docker

Mac, Linux, Windows platforms

The Docker Platform release is available for Mac, Windows, and Linux platforms. Please download the appropriate version from the Docker website and follow the corresponding installation instructions. Linux user may find these alternate instructions helpful.

Docker installation on EC2 with Ubuntu

The instructions below show in detail how to prepare an Amazon EC2 instance running Ubuntu 16.04 for use with Docker.

Apply initialize the package repository and apply system updates:

```
sudo apt-get update
sudo apt-get -y dist-upgrade
```

Install Docker and start the service:

```
sudo apt-get install -y docker.io
sudo service docker start
```

Add the `ubuntu` user to the `docker` group to allow running Docker commands without sudo:

```
sudo usermod -a -G docker ubuntu
```

Initiate a new login to gain group permissions (alternatively, log out and log back in again):

```
exec sudo su -l ubuntu
```

Confirm that docker is running:

```
docker images
```

Should produce an empty table similar to the following:

```
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
```

Clone the Ray repository

```
git clone https://github.com/ray-project/ray.git
```

Build Docker images

Run the script to create Docker images.

```
cd ray
./build-docker.sh
```

This script creates several Docker images:

- The `ray-project/deploy` image is a self-contained copy of code and binaries suitable for end users.
- The `ray-project/examples` adds additional libraries for running examples.
- The `ray-project/base-deps` image builds from Ubuntu Xenial and includes Anaconda and other basic dependencies and can serve as a starting point for developers.

Review images by listing them:

```
docker images
```

Output should look something like the following:

```
REPOSITORY                          TAG                 IMAGE ID            CREATED␣
↪           SIZE
ray-project/examples                latest              7584bde65894        4 days␣
↪ago           3.257 GB
ray-project/deploy                  latest              970966166c71        4 days␣
↪ago           2.899 GB
ray-project/base-deps               latest              f45d66963151        4 days␣
↪ago           2.649 GB
ubuntu                              xenial              f49eec89601e        3 weeks␣
↪ago          129.5 MB
```

Launch Ray in Docker

Start out by launching the deployment container.

```
docker run --shm-size=<shm-size> -t -i ray-project/deploy
```

Replace `<shm-size>` with a limit appropriate for your system, for example `512M` or `2G`. The `-t` and `-i` options here are required to support interactive use of the container.

Note: Ray requires a **large** amount of shared memory because each object store keeps all of its objects in shared memory, so the amount of shared memory will limit the size of the object store.

You should now see a prompt that looks something like:

```
root@ebc78f68d100:/ray#
```

Test if the installation succeeded

To test if the installation was successful, try running some tests. Within the container shell enter the following commands:

```
python test/runtest.py # This tests basic functionality.
```

You are now ready to continue with the tutorial.

Running examples in Docker

Ray includes a Docker image that includes dependencies necessary for running some of the examples. This can be an easy way to see Ray in action on a variety of workloads.

Launch the examples container.

```
docker run --shm-size=1024m -t -i ray-project/examples
```

Hyperparameter optimization

```
cd /ray/examples/hyperopt/
python /ray/examples/hyperopt/hyperopt_simple.py
```

Batch L-BFGS

```
python /ray/examples/lbfgs/driver.py
```

Learning to play Pong

```
python /ray/examples/rl_pong/driver.py
```

CHAPTER 4

Installation Troubleshooting

Trouble installing Numbuf

If the installation of Numbuf fails, chances are there was a problem building Arrow. Some candidate possibilities.

You have a different version of Flatbuffers installed

Arrow pulls and builds its own copy of Flatbuffers, but if you already have Flatbuffers installed, Arrow may find the wrong version. If a directory like `/usr/local/include/flatbuffers` shows up in the output when installing Numbuf, this may be the problem. To solve it, get rid of the old version of flatbuffers.

There is some problem with Boost

If a message like `Unable to find the requested Boost libraries` appears when installing Numbuf, there may be a problem with Boost. This can happen if you installed Boost using MacPorts. This is sometimes solved by using Brew instead.

Trouble installing or running Ray

One of the Ray libraries is compiled against the wrong version of Python

If there is a segfault or a sigabort immediately upon importing Ray, one of the components may have been compiled against the wrong Python libraries. CMake should normally find the right version of Python, but this process is not completely reliable. In this case, check the CMake output from installation and make sure that the version of the Python libraries that were found match the version of Python that you're using.

Note that it's common to have multiple versions of Python on your machine (for example both Python 2 and Python 3). Ray will be compiled against whichever version of Python is found when you run the `python` command from the command line, so make sure this is the version you wish to use.

CHAPTER 5

Tutorial

To use Ray, you need to understand the following:

- How Ray executes tasks asynchronously to achieve parallelism.
- How Ray uses object IDs to represent immutable remote objects.

Overview

Ray is a Python-based distributed execution engine. The same code can be run on a single machine to achieve efficient multiprocessing, and it can be used on a cluster for large computations.

When using Ray, several processes are involved.

- Multiple **worker** processes execute tasks and store results in object stores. Each worker is a separate process.
- One **object store** per node stores immutable objects in shared memory and allows workers to efficiently share objects on the same node with minimal copying and deserialization.
- One **local scheduler** per node assigns tasks to workers on the same node.
- A **global scheduler** receives tasks from local schedulers and assigns them to other local schedulers.
- A **driver** is the Python process that the user controls. For example, if the user is running a script or using a Python shell, then the driver is the Python process that runs the script or the shell. A driver is similar to a worker in that it can submit tasks to its local scheduler and get objects from the object store, but it is different in that the local scheduler will not assign tasks to the driver to be executed.
- A **Redis server** maintains much of the system's state. For example, it keeps track of which objects live on which machines and of the task specifications (but not data). It can also be queried directly for debugging purposes.

Starting Ray

To start Ray, start Python and run the following commands.

```
import ray
ray.init()
```

This starts Ray.

Immutable remote objects

In Ray, we can create and compute on objects. We refer to these objects as **remote objects**, and we use **object IDs** to refer to them. Remote objects are stored in **object stores**, and there is one object store per node in the cluster. In the cluster setting, we may not actually know which machine each object lives on.

An **object ID** is essentially a unique ID that can be used to refer to a remote object. If you're familiar with Futures, our object IDs are conceptually similar.

We assume that remote objects are immutable. That is, their values cannot be changed after creation. This allows remote objects to be replicated in multiple object stores without needing to synchronize the copies.

Put and Get

The commands `ray.get` and `ray.put` can be used to convert between Python objects and object IDs, as shown in the example below.

```
x = "example"
ray.put(x)  # ObjectID(b49a32d72057bdcfc4dda35584b3d838aad89f5d)
```

The command `ray.put(x)` would be run by a worker process or by the driver process (the driver process is the one running your script). It takes a Python object and copies it to the local object store (here *local* means *on the same node*). Once the object has been stored in the object store, its value cannot be changed.

In addition, `ray.put(x)` returns an object ID, which is essentially an ID that can be used to refer to the newly created remote object. If we save the object ID in a variable with `x_id = ray.put(x)`, then we can pass `x_id` into remote functions, and those remote functions will operate on the corresponding remote object.

The command `ray.get(x_id)` takes an object ID and creates a Python object from the corresponding remote object. For some objects like arrays, we can use shared memory and avoid copying the object. For other objects, this copies the object from the object store to the worker process's heap. If the remote object corresponding to the object ID `x_id` does not live on the same node as the worker that calls `ray.get(x_id)`, then the remote object will first be transferred from an object store that has it to the object store that needs it.

```
x_id = ray.put("example")
ray.get(x_id)  # "example"
```

If the remote object corresponding to the object ID `x_id` has not been created yet, the command `ray.get(x_id)` will wait until the remote object has been created.

A very common use case of `ray.get` is to get a list of object IDs. In this case, you can call `ray.get(object_ids)` where `object_ids` is a list of object IDs.

```
result_ids = [ray.put(i) for i in range(10)]
ray.get(result_ids)  # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Asynchronous Computation in Ray

Ray enables arbitrary Python functions to be executed asynchronously. This is done by designating a Python function as a **remote function**.

For example, a normal Python function looks like this.

```
def add1(a, b):
    return a + b
```

A remote function looks like this.

```
@ray.remote
def add2(a, b):
    return a + b
```

Remote functions

Whereas calling `add1(1, 2)` returns `3` and causes the Python interpreter to block until the computation has finished, calling `add2.remote(1, 2)` immediately returns an object ID and creates a **task**. The task will be scheduled by the system and executed asynchronously (potentially on a different machine). When the task finishes executing, its return value will be stored in the object store.

```
x_id = add2.remote(1, 2)
ray.get(x_id)  # 3
```

The following simple example demonstrates how asynchronous tasks can be used to parallelize computation.

```
import time

def f1():
    time.sleep(1)

@ray.remote
def f2():
    time.sleep(1)

# The following takes ten seconds.
[f1() for _ in range(10)]

# The following takes one second (assuming the system has at least ten CPUs).
ray.get([f2.remote() for _ in range(10)])
```

There is a sharp distinction between *submitting a task* and *executing the task*. When a remote function is called, the task of executing that function is submitted to a local scheduler, and object IDs for the outputs of the task are immediately returned. However, the task will not be executed until the system actually schedules the task on a worker. Task execution is **not** done lazily. The system moves the input data to the task, and the task will execute as soon as its input dependencies are available and there are enough resources for the computation.

When a task is submitted, each argument may be passed in by value or by object ID. For example, these lines have the same behavior.

```
add2.remote(1, 2)
add2.remote(1, ray.put(2))
add2.remote(ray.put(1), ray.put(2))
```

Remote functions never return actual values, they always return object IDs.

When the remote function is actually executed, it operates on Python objects. That is, if the remote function was called with any object IDs, the system will retrieve the corresponding objects from the object store.

Note that a remote function can return multiple object IDs.

```
@ray.remote(num_return_vals=3)
def return_multiple():
    return 1, 2, 3

a_id, b_id, c_id = return_multiple.remote()
```

Expressing dependencies between tasks

Programmers can express dependencies between tasks by passing the object ID output of one task as an argument to another task. For example, we can launch three tasks as follows, each of which depends on the previous task.

```
@ray.remote
def f(x):
    return x + 1

x = f.remote(0)
y = f.remote(x)
z = f.remote(y)
ray.get(z) # 3
```

The second task above will not execute until the first has finished, and the third will not execute until the second has finished. In this example, there are no opportunities for parallelism.

The ability to compose tasks makes it easy to express interesting dependencies. Consider the following implementation of a tree reduce.

```
import numpy as np

@ray.remote
def generate_data():
    return np.random.normal(size=1000)

@ray.remote
def aggregate_data(x, y):
    return x + y

# Generate some random data. This launches 100 tasks that will be scheduled on
# various nodes. The resulting data will be distributed around the cluster.
data = [generate_data.remote() for _ in range(100)]

# Perform a tree reduce.
while len(data) > 1:
    data.append(aggregate_data.remote(data.pop(0), data.pop(0)))

# Fetch the result.
ray.get(data)
```

Remote Functions Within Remote Functions

So far, we have been calling remote functions only from the driver. But worker processes can also call remote functions. To illustrate this, consider the following example.

```
@ray.remote
def sub_experiment(i, j):
    # Run the jth sub-experiment for the ith experiment.
    return i + j

@ray.remote
def run_experiment(i):
    sub_results = []
    # Launch tasks to perform 10 sub-experiments in parallel.
    for j in range(10):
        sub_results.append(sub_experiment.remote(i, j))
    # Return the sum of the results of the sub-experiments.
    return sum(ray.get(sub_results))

results = [run_experiment.remote(i) for i in range(5)]
ray.get(results) # [45, 55, 65, 75, 85]
```

When the remote function `run_experiment` is executed on a worker, it calls the remote function `sub_experiment` a number of times. This is an example of how multiple experiments, each of which takes advantage of parallelism internally, can all be run in parallel.

CHAPTER 6

The Ray API

Starting Ray

There are two main ways in which Ray can be used. First, you can start all of the relevant Ray processes and shut them all down within the scope of a single script. Second, you can connect to and use an existing Ray cluster.

Starting and stopping a cluster within a script

One use case is to start all of the relevant Ray processes when you call `ray.init` and shut them down when the script exits. These processes include local and global schedulers, an object store and an object manager, a redis server, and more.

Note: this approach is limited to a single machine.

This can be done as follows.

```
ray.init()
```

If there are GPUs available on the machine, you should specify this with the `num_gpus` argument. Similarly, you can also specify the number of CPUs with `num_cpus`.

```
ray.init(num_cpus=20, num_gpus=2)
```

By default, Ray will use `psutil.cpu_count()` to determine the number of CPUs, and by default the number of GPUs will be zero.

Instead of thinking about the number of "worker" processes on each node, we prefer to think in terms of the quantities of CPU and GPU resources on each node and to provide the illusion of an infinite pool of workers. Tasks will be assigned to workers based on the availability of resources so as to avoid contention and not based on the number of available worker processes.

Connecting to an existing cluster

Once a Ray cluster has been started, the only thing you need in order to connect to it is the address of the Redis server in the cluster. In this case, your script will not start up or shut down any processes. The cluster and all of its processes may be shared between multiple scripts and multiple users. To do this, you simply need to know the address of the cluster's Redis server. This can be done with a command like the following.

```
ray.init(redis_address="12.345.67.89:6379")
```

In this case, you cannot specify `num_cpus` or `num_gpus` in `ray.init` because that information is passed into the cluster when the cluster is started, not when your script is started.

View the instructions for how to start a Ray cluster on multiple nodes.

`ray.`**`init`**(*redis_address=None*, *node_ip_address=None*, *object_id_seed=None*, *num_workers=None*, *driver_mode=0*, *redirect_output=False*, *num_cpus=None*, *num_gpus=None*, *num_custom_resource=None*, *num_redis_shards=None*)

Connect to an existing Ray cluster or start one and connect to it.

This method handles two cases. Either a Ray cluster already exists and we just attach this driver to it, or we start all of the processes associated with a Ray cluster and attach to the newly started cluster.

Parameters

- **`node_ip_address`** (*str*) – The IP address of the node that we are on.
- **`redis_address`** (*str*) – The address of the Redis server to connect to. If this address is not provided, then this command will start Redis, a global scheduler, a local scheduler, a plasma store, a plasma manager, and some workers. It will also kill these processes when Python exits.
- **`object_id_seed`** (*int*) – Used to seed the deterministic generation of object IDs. The same value can be used across multiple runs of the same job in order to generate the object IDs in a consistent manner. However, the same ID should not be used for different jobs.
- **`num_workers`** (*int*) – The number of workers to start. This is only provided if redis_address is not provided.
- **`driver_mode`** (*bool*) – The mode in which to start the driver. This should be one of ray.SCRIPT_MODE, ray.PYTHON_MODE, and ray.SILENT_MODE.
- **`redirect_output`** (*bool*) – True if stdout and stderr for all the processes should be redirected to files and false otherwise.
- **`num_cpus`** (*int*) – Number of cpus the user wishes all local schedulers to be configured with.
- **`num_gpus`** (*int*) – Number of gpus the user wishes all local schedulers to be configured with.
- **`num_custom_resource`** (*int*) – The quantity of a user-defined custom resource that the local scheduler should be configured with. This flag is experimental and is subject to changes in the future.
- **`num_redis_shards`** – The number of Redis shards to start in addition to the primary Redis shard.

Returns Address information about the started processes.

Raises `Exception` – An exception is raised if an inappropriate combination of arguments is passed in.

Defining remote functions

Remote functions are used to create tasks. To define a remote function, the `@ray.remote` decorator is placed over the function definition.

The function can then be invoked with `f.remote`. Invoking the function creates a **task** which will be scheduled on and executed by some worker process in the Ray cluster. The call will return an **object ID** (essentially a future) representing the eventual return value of the task. Anyone with the object ID can retrieve its value, regardless of where the task was executed (see *Getting values from object IDs*).

When a task executes, its outputs will be serialized into a string of bytes and stored in the object store.

Note that arguments to remote functions can be values or object IDs.

```
@ray.remote
def f(x):
    return x + 1

x_id = f.remote(0)
ray.get(x_id)  # 1

y_id = f.remote(x_id)
ray.get(y_id)  # 2
```

If you want a remote function to return multiple object IDs, you can do that by passing the `num_return_vals` argument into the remote decorator.

```
@ray.remote(num_return_vals=2)
def f():
    return 1, 2

x_id, y_id = f.remote()
ray.get(x_id)  # 1
ray.get(y_id)  # 2
```

`ray.remote`(**args*, ***kwargs*)

This decorator is used to define remote functions and to define actors.

Parameters

- **num_return_vals** (*int*) – The number of object IDs that a call to this function should return.
- **num_cpus** (*int*) – The number of CPUs needed to execute this function.
- **num_gpus** (*int*) – The number of GPUs needed to execute this function.
- **num_custom_resource** (*int*) – The quantity of a user-defined custom resource that is needed to execute this function. This flag is experimental and is subject to changes in the future.
- **max_calls** (*int*) – The maximum number of tasks of this kind that can be run on a worker before the worker needs to be restarted.
- **checkpoint_interval** (*int*) – The number of tasks to run between checkpoints of the actor state.

Getting values from object IDs

Object IDs can be converted into objects by calling `ray.get` on the object ID. Note that `ray.get` accepts either a single object ID or a list of object IDs.

```
@ray.remote
def f():
    return {'key1': ['value']}

# Get one object ID.
ray.get(f.remote())  # {'key1': ['value']}

# Get a list of object IDs.
ray.get([f.remote() for _ in range(2)])  # [{'key1': ['value']}, {'key1': ['value']}]
```

Numpy arrays

Numpy arrays are handled more efficiently than other data types, so **use numpy arrays whenever possible**.

Any numpy arrays that are part of the serialized object will not be copied out of the object store. They will remain in the object store and the resulting deserialized object will simply have a pointer to the relevant place in the object store's memory.

Since objects in the object store are immutable, this means that if you want to mutate a numpy array that was returned by a remote function, you will have to first copy it.

`ray.get`(*object_ids*, *worker=<ray.worker.Worker object>*)

Get a remote object or a list of remote objects from the object store.

This method blocks until the object corresponding to the object ID is available in the local object store. If this object is not in the local object store, it will be shipped from an object store that has it (once the object has been created). If object_ids is a list, then the objects corresponding to each object in the list will be returned.

Parameters **object_ids** – Object ID of the object to get or a list of object IDs to get.

Returns A Python object or a list of Python objects.

Putting objects in the object store

The primary way that objects are placed in the object store is by being returned by a task. However, it is also possible to directly place objects in the object store using `ray.put`.

```
x_id = ray.put(1)
ray.get(x_id)  # 1
```

The main reason to use `ray.put` is that you want to pass the same large object into a number of tasks. By first doing `ray.put` and then passing the resulting object ID into each of the tasks, the large object is copied into the object store only once, whereas when we directly pass the object in, it is copied multiple times.

```
import numpy as np

@ray.remote
def f(x):
    pass
```

```
x = np.zeros(10 ** 6)

# Alternative 1: Here, x is copied into the object store 10 times.
[f.remote(x) for _ in range(10)]

# Alternative 2: Here, x is copied into the object store once.
x_id = ray.put(x)
[f.remote(x_id) for _ in range(10)]
```

Note that `ray.put` is called under the hood in a couple situations.

- It is called on the values returned by a task.
- It is called on the arguments to a task, unless the arguments are Python primitives like integers or short strings, lists, tuples, or dictionaries.

`ray.put`(*value*, *worker=<ray.worker.Worker object>*)
Store an object in the object store.

Parameters **value** – The Python object to be stored.

Returns The object ID assigned to this value.

Waiting for a subset of tasks to finish

It is often desirable to adapt the computation being done based on when different tasks finish. For example, if a bunch of tasks each take a variable length of time, and their results can be processed in any order, then it makes sense to simply process the results in the order that they finish. In other settings, it makes sense to discard straggler tasks whose results may not be needed.

To do this, we introduce the `ray.wait` primitive, which takes a list of object IDs and returns when a subset of them are available. By default it blocks until a single object is available, but the `num_returns` value can be specified to wait for a different number. If a `timeout` argument is passed in, it will block for at most that many milliseconds and may return a list with fewer than `num_returns` elements.

The `ray.wait` function returns two lists. The first list is a list of object IDs of available objects (of length at most `num_returns`), and the second list is a list of the remaining object IDs, so the combination of these two lists is equal to the list passed in to `ray.wait` (up to ordering).

```
import time
import numpy as np

@ray.remote
def f(n):
    time.sleep(n)
    return n

# Start 3 tasks with different durations.
results = [f.remote(i) for i in range(3)]
# Block until 2 of them have finished.
ready_ids, remaining_ids = ray.wait(results, num_returns=2)

# Start 5 tasks with different durations.
results = [f.remote(i) for i in range(3)]
# Block until 4 of them have finished or 2.5 seconds pass.
ready_ids, remaining_ids = ray.wait(results, num_returns=4, timeout=2500)
```

It is easy to use this construct to create an infinite loop in which multiple tasks are executing, and whenever one task finishes, a new one is launched.

```
@ray.remote
def f():
    return 1

# Start 5 tasks.
remaining_ids = [f.remote() for i in range(5)]
# Whenever one task finishes, start a new one.
for _ in range(100):
    ready_ids, remaining_ids = ray.wait(remaining_ids)
    # Get the available object and do something with it.
    print(ray.get(ready_ids))
    # Start a new task.
    remaining_ids.append(f.remote())
```

ray.**wait**(*object_ids*, *num_returns=1*, *timeout=None*, *worker=<ray.worker.Worker object>*)

Return a list of IDs that are ready and a list of IDs that are not.

If timeout is set, the function returns either when the requested number of IDs are ready or when the timeout is reached, whichever occurs first. If it is not set, the function simply waits until that number of objects is ready and returns that exact number of objectids.

This method returns two lists. The first list consists of object IDs that correspond to objects that are stored in the object store. The second list corresponds to the rest of the object IDs (which may or may not be ready).

Parameters

- **object_ids** (*List[ObjectID]*) – List of object IDs for objects that may or may not be ready. Note that these IDs must be unique.
- **num_returns** (*int*) – The number of object IDs that should be returned.
- **timeout** (*int*) – The maximum amount of time in milliseconds to wait before returning.

Returns

A list of object IDs that are ready and a list of the remaining object IDs.

Viewing errors

Keeping track of errors that occur in different processes throughout a cluster can be challenging. There are a couple mechanisms to help with this.

1. If a task throws an exception, that exception will be printed in the background of the driver process.
2. If `ray.get` is called on an object ID whose parent task threw an exception before creating the object, the exception will be re-raised by `ray.get`.

The errors will also be accumulated in Redis and can be accessed with `ray.error_info`. Normally, you shouldn't need to do this, but it is possible.

```
@ray.remote
def f():
    raise Exception("This task failed!!")

f.remote()  # An error message will be printed in the background.

# Wait for the error to propagate to Redis.
```

```
import time
time.sleep(1)

ray.error_info()  # This returns a list containing the error message.
```

ray.**error_info**(*worker=<ray.worker.Worker object>*)
Return information about failed tasks.

CHAPTER 7

Actors

Remote functions in Ray should be thought of as functional and side-effect free. Restricting ourselves only to remote functions gives us distributed functional programming, which is great for many use cases, but in practice is a bit limited.

Ray extends the dataflow model with **actors**. An actor is essentially a stateful worker (or a service). When a new actor is instantiated, a new worker is created, and methods of the actor are scheduled on that specific worker and can access and mutate the state of that worker.

Suppose we've already started Ray.

```
import ray
ray.init()
```

Defining and creating an actor

Consider the following simple example. The `ray.remote` decorator indicates that instances of the `Counter` class will be actors.

```
@ray.remote
class Counter(object):
    def __init__(self):
        self.value = 0

    def increment(self):
        self.value += 1
        return self.value
```

To actually create an actor, we can instantiate this class by calling `Counter.remote()`.

```
a1 = Counter.remote()
a2 = Counter.remote()
```

When an actor is instantiated, the following events happen.

1. A node in the cluster is chosen and a worker process is created on that node (by the local scheduler on that node) for the purpose of running methods called on the actor.
2. A `Counter` object is created on that worker and the `Counter` constructor is run.

Using an actor

We can schedule tasks on the actor by calling its methods.

```
a1.increment.remote()  # ray.get returns 1
a2.increment.remote()  # ray.get returns 1
```

When `a1.increment.remote()` is called, the following events happens.

1. A task is created.
2. The task is assigned directly to the local scheduler responsible for the actor by the driver's local scheduler. Thus, this scheduling procedure bypasses the global scheduler.
3. An object ID is returned.

We can then call `ray.get` on the object ID to retrieve the actual value.

Similarly, the call to `a2.increment.remote()` generates a task that is scheduled on the second `Counter` actor. Since these two tasks run on different actors, they can be executed in parallel (note that only actor methods will be scheduled on actor workers, regular remote functions will not be).

On the other hand, methods called on the same `Counter` actor are executed serially in the order that they are called. They can thus share state with one another, as shown below.

```
# Create ten Counter actors.
counters = [Counter.remote() for _ in range(10)]

# Increment each Counter once and get the results. These tasks all happen in
# parallel.
results = ray.get([c.increment.remote() for c in counters])
print(results)  # prints [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

# Increment the first Counter five times. These tasks are executed serially
# and share state.
results = ray.get([counters[0].increment.remote() for _ in range(5)])
print(results)  # prints [2, 3, 4, 5, 6]
```

A More Interesting Actor Example

A common pattern is to use actors to encapsulate the mutable state managed by an external library or service.

Gym provides an interface to a number of simulated environments for testing and training reinforcement learning agents. These simulators are stateful, and tasks that use these simulators must mutate their state. We can use actors to encapsulate the state of these simulators.

```
import gym

@ray.remote
class GymEnvironment(object):
    def __init__(self, name):
```

```
        self.env = gym.make(name)
        self.env.reset()

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

    def reset(self):
        self.env.reset()
```

We can then instantiate an actor and schedule a task on that actor as follows.

```
pong = GymEnvironment.remote("Pong-v0")
pong.step.remote(0)  # Take action 0 in the simulator.
```

Using GPUs on actors

A common use case is for an actor to contain a neural network. For example, suppose we have imported Tensorflow and have created a method for constructing a neural net.

```
import tensorflow as tf

def construct_network():
    x = tf.placeholder(tf.float32, [None, 784])
    y_ = tf.placeholder(tf.float32, [None, 10])

    W = tf.Variable(tf.zeros([784, 10]))
    b = tf.Variable(tf.zeros([10]))
    y = tf.nn.softmax(tf.matmul(x, W) + b)

    cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_
↪indices=[1]))
    train_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross_entropy)
    correct_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y_,1))
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

    return x, y_, train_step, accuracy
```

We can then define an actor for this network as follows.

```
import os

# Define an actor that runs on GPUs. If there are no GPUs, then simply use
# ray.remote without any arguments and no parentheses.
@ray.remote(num_gpus=1)
class NeuralNetOnGPU(object):
    def __init__(self):
        # Set an environment variable to tell TensorFlow which GPUs to use. Note
        # that this must be done before the call to tf.Session.
        os.environ["CUDA_VISIBLE_DEVICES"] = ",".join([str(i) for i in ray.get_gpu_
↪ids()])
        with tf.Graph().as_default():
            with tf.device("/gpu:0"):
                self.x, self.y_, self.train_step, self.accuracy = construct_network()
                # Allow this to run on CPUs if there aren't any GPUs.
                config = tf.ConfigProto(allow_soft_placement=True)
```

```
                self.sess = tf.Session(config=config)
                # Initialize the network.
                init = tf.global_variables_initializer()
                self.sess.run(init)
```

To indicate that an actor requires one GPU, we pass in `num_gpus=1` to `ray.remote`. Note that in order for this to work, Ray must have been started with some GPUs, e.g., via `ray.init(num_gpus=2)`. Otherwise, when you try to instantiate the GPU version with `NeuralNetOnGPU.remote()`, an exception will be thrown saying that there aren't enough GPUs in the system.

When the actor is created, it will have access to a list of the IDs of the GPUs that it is allowed to use via `ray.get_gpu_ids()`. This is a list of integers, like `[]`, or `[1]`, or `[2, 5, 6]`. Since we passed in `ray.remote(num_gpus=1)`, this list will have length one.

We can put this all together as follows.

```
import os
import ray
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data

ray.init(num_gpus=8)

def construct_network():
    x = tf.placeholder(tf.float32, [None, 784])
    y_ = tf.placeholder(tf.float32, [None, 10])

    W = tf.Variable(tf.zeros([784, 10]))
    b = tf.Variable(tf.zeros([10]))
    y = tf.nn.softmax(tf.matmul(x, W) + b)

    cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_
↪indices=[1]))
    train_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross_entropy)
    correct_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y_,1))
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

    return x, y_, train_step, accuracy

@ray.remote(num_gpus=1)
class NeuralNetOnGPU(object):
    def __init__(self, mnist_data):
        self.mnist = mnist_data
        # Set an environment variable to tell TensorFlow which GPUs to use. Note
        # that this must be done before the call to tf.Session.
        os.environ["CUDA_VISIBLE_DEVICES"] = ",".join([str(i) for i in ray.get_gpu_
↪ids()])
        with tf.Graph().as_default():
            with tf.device("/gpu:0"):
                self.x, self.y_, self.train_step, self.accuracy = construct_network()
                # Allow this to run on CPUs if there aren't any GPUs.
                config = tf.ConfigProto(allow_soft_placement=True)
                self.sess = tf.Session(config=config)
                # Initialize the network.
                init = tf.global_variables_initializer()
                self.sess.run(init)

    def train(self, num_steps):
```

```
        for _ in range(num_steps):
            batch_xs, batch_ys = self.mnist.train.next_batch(100)
            self.sess.run(self.train_step, feed_dict={self.x: batch_xs, self.y_:␣
↪batch_ys})

    def get_accuracy(self):
        return self.sess.run(self.accuracy, feed_dict={self.x: self.mnist.test.images,
                                                       self.y_: self.mnist.test.
↪labels})


# Load the MNIST dataset and tell Ray how to serialize the custom classes.
mnist = input_data.read_data_sets("MNIST_data", one_hot=True)

# Create the actor.
nn = NeuralNetOnGPU.remote(mnist)

# Run a few steps of training and print the accuracy.
nn.train.remote(100)
accuracy = ray.get(nn.get_accuracy.remote())
print("Accuracy is {}.".format(accuracy))
```

CHAPTER 8

Using Ray with GPUs

GPUs are critical for many machine learning applications. Ray enables remote functions and actors to specify their GPU requirements in the `ray.remote` decorator.

Starting Ray with GPUs

In order for remote functions and actors to use GPUs, Ray must know how many GPUs are available. If you are starting Ray on a single machine, you can specify the number of GPUs as follows.

```
ray.init(num_gpus=4)
```

If you don't pass in the `num_gpus` argument, Ray will assume that there are 0 GPUs on the machine.

If you are starting Ray with the `ray start` command, you can indicate the number of GPUs on the machine with the `--num-gpus` argument.

```
ray start --head --num-gpus=4
```

Note: There is nothing preventing you from passing in a larger value of `num_gpus` than the true number of GPUs on the machine. In this case, Ray will act as if the machine has the number of GPUs you specified for the purposes of scheduling tasks that require GPUs. Trouble will only occur if those tasks attempt to actually use GPUs that don't exist.

Using Remote Functions with GPUs

If a remote function requires GPUs, indicate the number of required GPUs in the remote decorator.

```
@ray.remote(num_gpus=1)
def gpu_method():
    return "This function is allowed to use GPUs {}.".format(ray.get_gpu_ids())
```

Inside of the remote function, a call to `ray.get_gpu_ids()` will return a list of integers indicating which GPUs the remote function is allowed to use.

Note: The function `gpu_method` defined above doesn't actually use any GPUs. Ray will schedule it on a machine which has at least one GPU, and will reserve one GPU for it while it is being executed, however it is up to the function to actually make use of the GPU. This is typically done through an external library like TensorFlow. Here is an example that actually uses GPUs. Note that for this example to work, you will need to install the GPU version of TensorFlow.

```
import os
import tensorflow as tf

@ray.remote(num_gpus=1)
def gpu_method():
    os.environ["CUDA_VISIBLE_DEVICES"] = ",".join(map(str, ray.get_gpu_ids()))
    # Create a TensorFlow session. TensorFlow will restrict itself to use the
    # GPUs specified by the CUDA_VISIBLE_DEVICES environment variable.
    tf.Session()
```

Note: It is certainly possible for the person implementing `gpu_method` to ignore `ray.get_gpu_ids` and to use all of the GPUs on the machine. Ray does not prevent this from happening, and this can lead to too many workers using the same GPU at the same time. For example, if the `CUDA_VISIBLE_DEVICES` environment variable is not set, then TensorFlow will attempt to use all of the GPUs on the machine.

Using Actors with GPUs

When defining an actor that uses GPUs, indicate the number of GPUs an actor instance requires in the `ray.remote` decorator.

```
@ray.remote(num_gpus=1)
class GPUActor(object):
    def __init__(self):
        return "This actor is allowed to use GPUs {}.".format(ray.get_gpu_ids())
```

When the actor is created, GPUs will be reserved for that actor for the lifetime of the actor.

Note that Ray must have been started with at least as many GPUs as the number of GPUs you pass into the `ray.remote` decorator. Otherwise, if you pass in a number greater than what was passed into `ray.init`, an exception will be thrown when instantiating the actor.

The following is an example of how to use GPUs in an actor through TensorFlow.

```
@ray.remote(num_gpus=1)
class GPUActor(object):
    def __init__(self):
        self.gpu_ids = ray.get_gpu_ids()
        os.environ["CUDA_VISIBLE_DEVICES"] = ",".join(map(str, self.gpu_ids))
        # The call to tf.Session() will restrict TensorFlow to use the GPUs
        # specified in the CUDA_VISIBLE_DEVICES environment variable.
        self.sess = tf.Session()
```

Troubleshooting

Note: Currently, when a worker executes a task that uses a GPU, the task may allocate memory on the GPU and may not release it when the task finishes executing. This can lead to problems. See this issue.

CHAPTER 9

Hyperparameter Optimization

This document provides a walkthrough of the hyperparameter optimization example. To run the application, first install some dependencies.

```
pip install tensorflow
```

You can view the code for this example.

The simple script that processes results as they become available and launches new experiments can be run as follows.

```
python ray/examples/hyperopt/hyperopt_simple.py --trials=5 --steps=10
```

The variant that divides training into multiple segments and aggressively terminates poorly performing models can be run as follows.

```
python ray/examples/hyperopt/hyperopt_adaptive.py --num-starting-segments=5 \
                                                  --num-segments=10 \
                                                  --steps-per-segment=20
```

Machine learning algorithms often have a number of *hyperparameters* whose values must be chosen by the practitioner. For example, an optimization algorithm may have a step size, a decay rate, and a regularization coefficient. In a deep network, the network parameterization itself (e.g., the number of layers and the number of units per layer) can be considered a hyperparameter.

Choosing these parameters can be challenging, and so a common practice is to search over the space of hyperparameters. One approach that works surprisingly well is to randomly sample different options.

Problem Setup

Suppose that we want to train a convolutional network, but we aren't sure how to choose the following hyperparameters:

- the learning rate
- the batch size

- the dropout probability
- the standard deviation of the distribution from which to initialize the network weights

Suppose that we've defined a remote function `train_cnn_and_compute_accuracy`, which takes values for these hyperparameters as its input (along with the dataset), trains a convolutional network using those hyperparameters, and returns the accuracy of the trained model on a validation set.

```
import numpy as np
import ray

@ray.remote
def train_cnn_and_compute_accuracy(hyperparameters,
                                   train_images,
                                   train_labels,
                                   validation_images,
                                   validation_labels):
    # Construct a deep network, train it, and return the accuracy on the
    # validation data.
    return np.random.uniform(0, 1)
```

Basic random search

Something that works surprisingly well is to try random values for the hyperparameters. For example, we can write a function that randomly generates hyperparameter configurations.

```
def generate_hyperparameters():
    # Randomly choose values for the hyperparameters.
    return {"learning_rate": 10 ** np.random.uniform(-5, 5),
            "batch_size": np.random.randint(1, 100),
            "dropout": np.random.uniform(0, 1),
            "stddev": 10 ** np.random.uniform(-5, 5)}
```

In addition, let's assume that we've started Ray and loaded some data.

```
import ray

ray.init()

from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data", one_hot=True)
train_images = ray.put(mnist.train.images)
train_labels = ray.put(mnist.train.labels)
validation_images = ray.put(mnist.validation.images)
validation_labels = ray.put(mnist.validation.labels)
```

Then basic random hyperparameter search looks something like this. We launch a bunch of experiments, and we get the results.

```
# Generate a bunch of hyperparameter configurations.
hyperparameter_configurations = [generate_hyperparameters() for _ in range(20)]

# Launch some experiments.
results = []
for hyperparameters in hyperparameter_configurations:
    results.append(train_cnn_and_compute_accuracy.remote(hyperparameters,
```

```
                                                          train_images,
                                                          train_labels,
                                                          validation_images,
                                                          validation_labels))

# Get the results.
accuracies = ray.get(results)
```

Then we can inspect the contents of *accuracies* and see which set of hyperparameters worked the best. Note that in the above example, the for loop will run instantaneously and the program will block in the call to `ray.get`, which will wait until all of the experiments have finished.

Processing results as they become available

One problem with the above approach is that you have to wait for all of the experiments to finish before you can process the results. Instead, you may want to process the results as they become available, perhaps in order to adaptively choose new experiments to run, or perhaps simply so you know how well the experiments are doing. To process the results as they become available, we can use the `ray.wait` primitive.

The most simple usage is the following. This example is implemented in more detail in driver.py.

```
# Launch some experiments.
remaining_ids = []
for hyperparameters in hyperparameter_configurations:
    remaining_ids.append(train_cnn_and_compute_accuracy.remote(hyperparameters,
                                                               train_images,
                                                               train_labels,
                                                               validation_images,
                                                               validation_labels))

# Whenever a new experiment finishes, print the value and start a new
# experiment.
for i in range(100):
    ready_ids, remaining_ids = ray.wait(remaining_ids, num_returns=1)
    accuracy = ray.get(ready_ids[0])
    print("Accuracy is {}".format(accuracy))
    # Start a new experiment.
    new_hyperparameters = generate_hyperparameters()
    remaining_ids.append(train_cnn_and_compute_accuracy.remote(new_hyperparameters,
                                                               train_images,
                                                               train_labels,
                                                               validation_images,
                                                               validation_labels))
```

More sophisticated hyperparameter search

Hyperparameter search algorithms can get much more sophisticated. So far, we've been treating the function `train_cnn_and_compute_accuracy` as a black box, that we can choose its inputs and inspect its outputs, but once we decide to run it, we have to run it until it finishes.

However, there is often more structure to be exploited. For example, if the training procedure is going poorly, we can end the session early and invest more resources in the more promising hyperparameter experiments. And if we've saved the state of the training procedure, we can always restart it again later.

This is one of the ideas of the Hyperband algorithm. Start with a huge number of hyperparameter configurations, aggressively stop the bad ones, and invest more resources in the promising experiments.

To implement this, we can first adapt our training method to optionally take a model and to return the updated model.

```
@ray.remote
def train_cnn_and_compute_accuracy(hyperparameters, model=None):
    # Construct a deep network, train it, and return the accuracy on the
    # validation data as well as the latest version of the model. If the model
    # argument is not None, this will continue training an existing model.
    validation_accuracy = np.random.uniform(0, 1)
    new_model = model
    return validation_accuracy, new_model
```

Here's a different variant that uses the same principles. Divide each training session into a series of shorter training sessions. Whenever a short session finishes, if it still looks promising, then continue running it. If it isn't doing well, then terminate it and start a new experiment.

```
import numpy as np

def is_promising(model):
    # Return true if the model is doing well and false otherwise. In practice,
    # this function will want more information than just the model.
    return np.random.choice([True, False])

# Start 10 experiments.
remaining_ids = []
for _ in range(10):
    experiment_id = train_cnn_and_compute_accuracy.remote(hyperparameters, model=None)
    remaining_ids.append(experiment_id)

accuracies = []
for i in range(100):
    # Whenever a segment of an experiment finishes, decide if it looks promising
    # or not.
    ready_ids, remaining_ids = ray.wait(remaining_ids, num_returns=1)
    experiment_id = ready_ids[0]
    current_accuracy, current_model = ray.get(experiment_id)
    accuracies.append(current_accuracy)

    if is_promising(experiment_id):
        # Continue running the experiment.
        experiment_id = train_cnn_and_compute_accuracy.remote(hyperparameters,
                                                              model=current_model)
    else:
        # Start a new experiment.
        experiment_id = train_cnn_and_compute_accuracy.remote(hyperparameters)

    remaining_ids.append(experiment_id)
```

CHAPTER 10

Learning to Play Pong

In this example, we'll train a **very simple** neural network to play Pong using the OpenAI Gym. This application is adapted, with minimal modifications, from Andrej Karpathy's code (see the accompanying blog post).

You can view the code for this example.

To run the application, first install some dependencies.

```
pip install gym[atari]
```

Then you can run the example as follows.

```
python ray/examples/rl_pong/driver.py --batch-size=10
```

To run the example on a cluster, simply pass in the flag `--redis-address=<redis-address>`.

At the moment, on a large machine with 64 physical cores, computing an update with a batch of size 1 takes about 1 second, a batch of size 10 takes about 2.5 seconds. A batch of size 60 takes about 3 seconds. On a cluster with 11 nodes, each with 18 physical cores, a batch of size 300 takes about 10 seconds. If the numbers you see differ from these by much, take a look at the **Troubleshooting** section at the bottom of this page and consider submitting an issue.

Note that these times depend on how long the rollouts take, which in turn depends on how well the policy is doing. For example, a really bad policy will lose very quickly. As the policy learns, we should expect these numbers to increase.

The distributed version

At the core of Andrej's code, a neural network is used to define a "policy" for playing Pong (that is, a function that chooses an action given a state). In the loop, the network repeatedly plays games of Pong and records a gradient from each game. Every ten games, the gradients are combined together and used to update the network.

This example is easy to parallelize because the network can play ten games in parallel and no information needs to be shared between the games.

We define an **actor** for the Pong environment, which includes a method for performing a rollout and computing a gradient update. Below is pseudocode for the actor.

```
@ray.remote
class PongEnv(object):
    def __init__(self):
        # Tell numpy to only use one core. If we don't do this, each actor may try
        # to use all of the cores and the resulting contention may result in no
        # speedup over the serial version. Note that if numpy is using OpenBLAS,
        # then you need to set OPENBLAS_NUM_THREADS=1, and you probably need to do
        # it from the command line (so it happens before numpy is imported).
        os.environ["MKL_NUM_THREADS"] = "1"
        self.env = gym.make("Pong-v0")

    def compute_gradient(self, model):
        # Reset the game.
        observation = self.env.reset()
        while not done:
            # Choose an action using policy_forward.
            # Take the action and observe the new state of the world.
        # Compute a gradient using policy_backward. Return the gradient and reward.
        return [gradient, reward_sum]
```

We then create a number of actors, so that we can perform rollouts in parallel.

```
actors = [PongEnv() for _ in range(batch_size)]
```

Calling this remote function inside of a for loop, we launch multiple tasks to perform rollouts and compute gradients in parallel.

```
model_id = ray.put(model)
actions = []
# Launch tasks to compute gradients from multiple rollouts in parallel.
for i in range(batch_size):
    action_id = actors[i].compute_gradient.remote(model_id)
    actions.append(action_id)
```

Troubleshooting

If you are not seeing any speedup from Ray (and assuming you're using a multicore machine), the problem may be that numpy is trying to use multiple threads. When many processes are each trying to use multiple threads, the result is often no speedup. When running this example, try opening up `top` and seeing if some python processes are using more than 100% CPU. If yes, then this is likely the problem.

The example tries to set `MKL_NUM_THREADS=1` in the actor. However, that only works if the numpy on your machine is actually using MKL. If it's using OpenBLAS, then you'll need to set `OPENBLAS_NUM_THREADS=1`. In fact, you may have to do this **before** running the script (it may need to happen before numpy is imported).

```
export OPENBLAS_NUM_THREADS=1
```

CHAPTER 11

Policy Gradient Methods

This code shows how to do reinforcement learning with policy gradient methods. View the code for this example.

To run this example, you will need to install TensorFlow with GPU support (at least version `1.0.0`) and a few other dependencies.

```
pip install gym[atari]
pip install tensorflow
```

Then install the package as follows.

```
cd ray/examples/policy_gradient/
python setup.py install
```

Then you can run the example as follows.

```
python/ray/rllib/policy_gradient/example.py --environment=Pong-ram-v3
```

This will train an agent on the `Pong-ram-v3` Atari environment. You can also try passing in the `Pong-v0` environment or the `CartPole-v0` environment. If you wish to use a different environment, you will need to change a few lines in `example.py`.

Current and historical training progress can be monitored by pointing TensorBoard to the log output directory as follows.

```
tensorboard --logdir=/tmp/ray
```

Many of the TensorBoard metrics are also printed to the console, but you might find it easier to visualize and compare between runs using the TensorBoard UI.

CHAPTER 12

ResNet

This code adapts the TensorFlow ResNet example to do data parallel training across multiple GPUs using Ray. View the code for this example.

To run the example, you will need to install TensorFlow (at least version `1.0.0`). Then you can run the example as follows.

First download the CIFAR-10 or CIFAR-100 dataset.

```
# Get the CIFAR-10 dataset.
curl -o cifar-10-binary.tar.gz https://www.cs.toronto.edu/~kriz/cifar-10-binary.tar.gz
tar -xvf cifar-10-binary.tar.gz

# Get the CIFAR-100 dataset.
curl -o cifar-100-binary.tar.gz https://www.cs.toronto.edu/~kriz/cifar-100-binary.tar.
↪gz
tar -xvf cifar-100-binary.tar.gz
```

Then run the training script that matches the dataset you downloaded.

```
# Train Resnet on CIFAR-10.
python ray/examples/resnet/resnet_main.py \
    --eval_dir=/tmp/resnet-model/eval \
    --train_data_path=cifar-10-batches-bin/data_batch* \
    --eval_data_path=cifar-10-batches-bin/test_batch.bin \
    --dataset=cifar10 \
    --num_gpus=1

# Train Resnet on CIFAR-100.
python ray/examples/resnet/resnet_main.py \
    --eval_dir=/tmp/resnet-model/eval \
    --train_data_path=cifar-100-binary/train.bin \
    --eval_data_path=cifar-100-binary/test.bin \
    --dataset=cifar100 \
    --num_gpus=1
```

The script will print out the IP address that the log files are stored on. In the single-node case, you can ignore this and run tensorboard on the current machine.

```
python -m tensorflow.tensorboard --logdir=/tmp/resnet-model
```

If you are running Ray on multiple nodes, you will need to go to the node at the IP address printed, and run the command.

The core of the script is the actor definition.

```
@ray.remote(num_gpus=1)
class ResNetTrainActor(object):
    def __init__(self, data, dataset, num_gpus):
        # data is the preprocessed images and labels extracted from the dataset.
        # Thus, every actor has its own copy of the data.
        # Set the CUDA_VISIBLE_DEVICES environment variable in order to restrict
        # which GPUs TensorFlow uses. Note that this only works if it is done before
        # the call to tf.Session.
        os.environ['CUDA_VISIBLE_DEVICES'] = ','.join([str(i) for i in ray.get_gpu_
↪ids()])
        with tf.Graph().as_default():
            with tf.device('/gpu:0'):
                # We omit the code here that actually constructs the residual network
                # and initializes it. Uses the definition in the Tensorflow Resnet␣
↪Example.

    def compute_steps(self, weights):
        # This method sets the weights in the network, runs some training steps,
        # and returns the new weights. self.model.variables is a TensorFlowVariables
        # class that we pass the train operation into.
        self.model.variables.set_weights(weights)
        for i in range(self.steps):
            self.model.variables.sess.run(self.model.train_op)
        return self.model.variables.get_weights()
```

The main script first creates one actor for each GPU, or a single actor if *num_gpus* is zero.

```
train_actors = [ResNetTrainActor.remote(train_data, dataset, num_gpus) for _ in␣
↪range(num_gpus)]
```

Then the main loop passes the same weights to every model, performs updates on each model, averages the updates, and puts the new weights in the object store.

```
while True:
    all_weights = ray.get([actor.compute_steps.remote(weight_id) for actor in train_
↪actors])
    mean_weights = {k: sum([weights[k] for weights in all_weights]) / num_gpus for k␣
↪in all_weights[0]}
    weight_id = ray.put(mean_weights)
```

CHAPTER 13

Asynchronous Advantage Actor Critic (A3C)

This document walks through A3C, a state-of-the-art reinforcement learning algorithm. In this example, we adapt the OpenAI Universe Starter Agent implementation of A3C to use Ray.

View the code for this example.

To run the application, first install **ray** and then some dependencies:

```
pip install tensorflow
pip install six
pip install gym[atari]
pip install opencv-python
pip install scipy
```

You can run the code with

```
python/ray/rllib/a3c/example.py --num-workers=N
```

Reinforcement Learning

Reinforcement Learning is an area of machine learning concerned with **learning how an agent should act in an environment** so as to maximize some form of cumulative reward. Typically, an agent will observe the current state of the environment and take an action based on its observation. The action will change the state of the environment and will provide some numerical reward (or penalty) to the agent. The agent will then take in another observation and the process will repeat. **The mapping from state to action is a policy**, and in reinforcement learning, this policy is often represented with a deep neural network.

The **environment** is often a simulator (for example, a physics engine), and reinforcement learning algorithms often involve trying out many different sequences of actions within these simulators. These **rollouts** can often be done in parallel.

Policies are often initialized randomly and incrementally improved via simulation within the environment. To improve a policy, gradient-based updates may be computed based on the sequences of states and actions that have been observed. The gradient calculation is often delayed until a termination condition is reached (that is, the simulation

has finished) so that delayed rewards have been properly accounted for. However, in the Actor Critic model, we can begin the gradient calculation at any point in the simulation rollout by predicting future rewards with a Value Function approximator.

In our A3C implementation, each worker, implemented as a Ray actor, continuously simulates the environment. The driver will create a task that runs some steps of the simulator using the latest model, computes a gradient update, and returns the update to the driver. Whenever a task finishes, the driver will use the gradient update to update the model and will launch a new task with the latest model.

There are two main parts to the implementation - the driver and the worker.

Worker Code Walkthrough

We use a Ray Actor to simulate the environment.

```
import numpy as np
import ray

@ray.remote
class Runner(object):
    """Actor object to start running simulation on workers.
        Gradient computation is also executed on this object."""
    def __init__(self, env_name, actor_id):
        # starts simulation environment, policy, and thread.
        # Thread will continuously interact with the simulation environment
        self.env = env = create_env(env_name)
        self.id = actor_id
        self.policy = LSTMPolicy()
        self.runner = RunnerThread(env, self.policy, 20)
        self.start()

    def start(self):
        # starts the simulation thread
        self.runner.start_runner()

    def pull_batch_from_queue(self):
        # Implementation details removed - gets partial rollout from queue
        return rollout

    def compute_gradient(self, params):
        self.policy.set_weights(params)
        rollout = self.pull_batch_from_queue()
        batch = process_rollout(rollout, gamma=0.99, lambda_=1.0)
        gradient = self.policy.get_gradients(batch)
        info = {"id": self.id,
                "size": len(batch.a)}
        return gradient, info
```

Driver Code Walkthrough

The driver manages the coordination among workers and handles updating the global model parameters. The main training script looks like the following.

```
import numpy as np
import ray

def train(num_workers, env_name="PongDeterministic-v3"):
    # Setup a copy of the environment
    # Instantiate a copy of the policy - mainly used as a placeholder
    env = create_env(env_name, None, None)
    policy = LSTMPolicy(env.observation_space.shape, env.action_space.n, 0)
    obs = 0

    # Start simulations on actors
    agents = [Runner(env_name, i) for i in range(num_workers)]

    # Start gradient calculation tasks on each actor
    parameters = policy.get_weights()
    gradient_list = [agent.compute_gradient.remote(parameters) for agent in agents]

    while True: # Replace with your termination condition
        # wait for some gradient to be computed - unblock as soon as the earliest
↪arrives
        done_id, gradient_list = ray.wait(gradient_list)

        # get the results of the task from the object store
        gradient, info = ray.get(done_id)[0]
        obs += info["size"]

        # apply update, get the weights from the model, start a new task on the same
↪actor object
        policy.model_update(gradient)
        parameters = policy.get_weights()
        gradient_list.extend([agents[info["id"]].compute_gradient(parameters)])
    return policy
```

Benchmarks and Visualization

For the `PongDeterministic-v3` and an Amazon EC2 m4.16xlarge instance, we are able to train the agent with 16 workers in around 15 minutes. With 8 workers, we can train the agent in around 25 minutes.

You can visualize performance by running `tensorboard --logdir [directory]` in a separate screen, where `[directory]` is defaulted to `/tmp/ray/`. If you are running multiple experiments, be sure to vary the directory to which Tensorflow saves its progress (found in `a3c.py`).

CHAPTER 14

Batch L-BFGS

This document provides a walkthrough of the L-BFGS example. To run the application, first install these dependencies.

```
pip install tensorflow
pip install scipy
```

You can view the code for this example.

Then you can run the example as follows.

```
python ray/examples/lbfgs/driver.py
```

Optimization is at the heart of many machine learning algorithms. Much of machine learning involves specifying a loss function and finding the parameters that minimize the loss. If we can compute the gradient of the loss function, then we can apply a variety of gradient-based optimization algorithms. L-BFGS is one such algorithm. It is a quasi-Newton method that uses gradient information to approximate the inverse Hessian of the loss function in a computationally efficient manner.

The serial version

First we load the data in batches. Here, each element in `batches` is a tuple whose first component is a batch of `100` images and whose second component is a batch of the `100` corresponding labels. For simplicity, we use TensorFlow's built in methods for loading the data.

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
batch_size = 100
num_batches = mnist.train.num_examples // batch_size
batches = [mnist.train.next_batch(batch_size) for _ in range(num_batches)]
```

Now, suppose we have defined a function which takes a set of model parameters `theta` and a batch of data (both images and labels) and computes the loss for that choice of model parameters on that batch of data. Similarly, suppose

we've also defined a function that takes the same arguments and computes the gradient of the loss for that choice of model parameters.

```
def loss(theta, xs, ys):
    # compute the loss on a batch of data
    return loss

def grad(theta, xs, ys):
    # compute the gradient on a batch of data
    return grad

def full_loss(theta):
    # compute the loss on the full data set
    return sum([loss(theta, xs, ys) for (xs, ys) in batches])

def full_grad(theta):
    # compute the gradient on the full data set
    return sum([grad(theta, xs, ys) for (xs, ys) in batches])
```

Since we are working with a small dataset, we don't actually need to separate these methods into the part that operates on a batch and the part that operates on the full dataset, but doing so will make the distributed version clearer.

Now, if we wish to optimize the loss function using L-BFGS, we simply plug these functions, along with an initial choice of model parameters, into `scipy.optimize.fmin_l_bfgs_b`.

```
theta_init = 1e-2 * np.random.normal(size=dim)
result = scipy.optimize.fmin_l_bfgs_b(full_loss, theta_init, fprime=full_grad)
```

The distributed version

In this example, the computation of the gradient itself can be done in parallel on a number of workers or machines.

First, let's turn the data into a collection of remote objects.

```
batch_ids = [(ray.put(xs), ray.put(ys)) for (xs, ys) in batches]
```

We can load the data on the driver and distribute it this way because MNIST easily fits on a single machine. However, for larger data sets, we will need to use remote functions to distribute the loading of the data.

Now, lets turn `loss` and `grad` into methods of an actor that will contain our network.

```
class Network(object):
    def __init__():
        # Initialize network.

    def loss(theta, xs, ys):
        # compute the loss
        return loss

    def grad(theta, xs, ys):
        # compute the gradient
        return grad
```

Now, it is easy to speed up the computation of the full loss and the full gradient.

```
def full_loss(theta):
    theta_id = ray.put(theta)
```

```
    loss_ids = [actor.loss(theta_id) for actor in actors]
    return sum(ray.get(loss_ids))

def full_grad(theta):
    theta_id = ray.put(theta)
    grad_ids = [actor.grad(theta_id) for actor in actors]
    return sum(ray.get(grad_ids)).astype("float64") # This conversion is necessary␣
↪for use with fmin_l_bfgs_b.
```

Note that we turn `theta` into a remote object with the line `theta_id = ray.put(theta)` before passing it into the remote functions. If we had written

```
[actor.loss(theta_id) for actor in actors]
```

instead of

```
theta_id = ray.put(theta)
[actor.loss(theta_id) for actor in actors]
```

then each task that got sent to the scheduler (one for every element of `batch_ids`) would have had a copy of `theta` serialized inside of it. Since `theta` here consists of the parameters of a potentially large model, this is inefficient. *Large objects should be passed by object ID to remote functions and not by value*.

We use remote actors and remote objects internally in the implementation of `full_loss` and `full_grad`, but the user-facing behavior of these methods is identical to the behavior in the serial version.

We can now optimize the objective with the same function call as before.

```
theta_init = 1e-2 * np.random.normal(size=dim)
result = scipy.optimize.fmin_l_bfgs_b(full_loss, theta_init, fprime=full_grad)
```

CHAPTER 15

Evolution Strategies

This document provides a walkthrough of the evolution strategies example. To run the application, first install some dependencies.

```
pip install tensorflow
pip install gym
```

You can view the code for this example.

The script can be run as follows. Note that the configuration is tuned to work on the `Humanoid-v1` gym environment.

```
python/ray/rllib/evolution_strategies/example.py
```

At the heart of this example, we define a `Worker` class. These workers have a method `do_rollouts`, which will be used to perform simulate randomly perturbed policies in a given environment.

```
@ray.remote
class Worker(object):
    def __init__(self, config, policy_params, env_name, noise):
        self.env = # Initialize environment.
        self.policy = # Construct policy.
        # Details omitted.

    def do_rollouts(self, params):
        # Set the network weights.
        self.policy.set_trainable_flat(params)
        perturbation = # Generate a random perturbation to the policy.

        self.policy.set_trainable_flat(params + perturbation)
        # Do rollout with the perturbed policy.

        self.policy.set_trainable_flat(params - perturbation)
        # Do rollout with the perturbed policy.

        # Return the rewards.
```

In the main loop, we create a number of actors with this class.

```
workers = [Worker.remote(config, policy_params, env_name, noise_id)
           for _ in range(num_workers)]
```

We then enter an infinite loop in which we use the actors to perform rollouts and use the rewards from the rollouts to update the policy.

```
while True:
    # Get the current policy weights.
    theta = policy.get_trainable_flat()
    # Put the current policy weights in the object store.
    theta_id = ray.put(theta)
    # Use the actors to do rollouts, note that we pass in the ID of the policy
    # weights.
    rollout_ids = [worker.do_rollouts.remote(theta_id), for worker in workers]
    # Get the results of the rollouts.
    results = ray.get(rollout_ids)
    # Update the policy.
    optimizer.update(...)
```

In addition, note that we create a large object representing a shared block of random noise. We then put the block in the object store so that each `Worker` actor can use it without creating its own copy.

```
@ray.remote
def create_shared_noise():
    noise = np.random.randn(250000000)
    return noise

noise_id = create_shared_noise.remote()
```

Recall that the `noise_id` argument is passed into the actor constructor.

CHAPTER 16

Using Ray with TensorFlow

This document describes best practices for using Ray with TensorFlow.

To see more involved examples using TensorFlow, take a look at hyperparameter optimization, A3C, ResNet, Policy Gradients, and LBFGS.

If you are training a deep network in the distributed setting, you may need to ship your deep network between processes (or machines). For example, you may update your model on one machine and then use that model to compute a gradient on another machine. However, shipping the model is not always straightforward.

For example, a straightforward attempt to pickle a TensorFlow graph gives mixed results. Some examples fail, and some succeed (but produce very large strings). The results are similar with other pickling libraries as well.

Furthermore, creating a TensorFlow graph can take tens of seconds, and so serializing a graph and recreating it in another process will be inefficient. The better solution is to create the same TensorFlow graph on each worker once at the beginning and then to ship only the weights between the workers.

Suppose we have a simple network definition (this one is modified from the TensorFlow documentation).

```
import tensorflow as tf
import numpy as np

x_data = tf.placeholder(tf.float32, shape=[100])
y_data = tf.placeholder(tf.float32, shape=[100])

w = tf.Variable(tf.random_uniform([1], -1.0, 1.0))
b = tf.Variable(tf.zeros([1]))
y = w * x_data + b

loss = tf.reduce_mean(tf.square(y - y_data))
optimizer = tf.train.GradientDescentOptimizer(0.5)
grads = optimizer.compute_gradients(loss)
train = optimizer.apply_gradients(grads)

init = tf.global_variables_initializer()
sess = tf.Session()
```

To extract the weights and set the weights, you can use the following helper method.

```
import ray
variables = ray.experimental.TensorFlowVariables(loss, sess)
```

The `TensorFlowVariables` object provides methods for getting and setting the weights as well as collecting all of the variables in the model.

Now we can use these methods to extract the weights, and place them back in the network as follows.

```
# First initialize the weights.
sess.run(init)
# Get the weights
weights = variables.get_weights()  # Returns a dictionary of numpy arrays
# Set the weights
variables.set_weights(weights)
```

Note: If we were to set the weights using the `assign` method like below, each call to `assign` would add a node to the graph, and the graph would grow unmanageably large over time.

```
w.assign(np.zeros(1))  # This adds a node to the graph every time you call it.
b.assign(np.zeros(1))  # This adds a node to the graph every time you call it.
```

Complete Example

Putting this all together, we would first embed the graph in an actor. Within the actor, we would use the `get_weights` and `set_weights` methods of the `TensorFlowVariables` class. We would then use those methods to ship the weights (as a dictionary of variable names mapping to numpy arrays) between the processes without shipping the actual TensorFlow graphs, which are much more complex Python objects.

```
import tensorflow as tf
import numpy as np
import ray

ray.init()

BATCH_SIZE = 100
NUM_BATCHES = 1
NUM_ITERS = 201

class Network(object):
    def __init__(self, x, y):
        # Seed TensorFlow to make the script deterministic.
        tf.set_random_seed(0)
        # Define the inputs.
        self.x_data = tf.constant(x, dtype=tf.float32)
        self.y_data = tf.constant(y, dtype=tf.float32)
        # Define the weights and computation.
        w = tf.Variable(tf.random_uniform([1], -1.0, 1.0))
        b = tf.Variable(tf.zeros([1]))
        y = w * self.x_data + b
        # Define the loss.
        self.loss = tf.reduce_mean(tf.square(y - self.y_data))
        optimizer = tf.train.GradientDescentOptimizer(0.5)
        self.grads = optimizer.compute_gradients(self.loss)
        self.train = optimizer.apply_gradients(self.grads)
```

```
        # Define the weight initializer and session.
        init = tf.global_variables_initializer()
        self.sess = tf.Session()
        # Additional code for setting and getting the weights
        self.variables = ray.experimental.TensorFlowVariables(self.loss, self.sess)
        # Return all of the data needed to use the network.
        self.sess.run(init)

    # Define a remote function that trains the network for one step and returns the
    # new weights.
    def step(self, weights):
        # Set the weights in the network.
        self.variables.set_weights(weights)
        # Do one step of training.
        self.sess.run(self.train)
        # Return the new weights.
        return self.variables.get_weights()

    def get_weights(self):
        return self.variables.get_weights()

# Define a remote function for generating fake data.
@ray.remote(num_return_vals=2)
def generate_fake_x_y_data(num_data, seed=0):
    # Seed numpy to make the script deterministic.
    np.random.seed(seed)
    x = np.random.rand(num_data)
    y = x * 0.1 + 0.3
    return x, y

# Generate some training data.
batch_ids = [generate_fake_x_y_data.remote(BATCH_SIZE, seed=i) for i in range(NUM_
↪BATCHES)]
x_ids = [x_id for x_id, y_id in batch_ids]
y_ids = [y_id for x_id, y_id in batch_ids]
# Generate some test data.
x_test, y_test = ray.get(generate_fake_x_y_data.remote(BATCH_SIZE, seed=NUM_BATCHES))

# Create actors to store the networks.
remote_network = ray.remote(Network)
actor_list = [remote_network.remote(x_ids[i], y_ids[i]) for i in range(NUM_BATCHES)]

# Get initial weights of some actor.
weights = ray.get(actor_list[0].get_weights.remote())

# Do some steps of training.
for iteration in range(NUM_ITERS):
    # Put the weights in the object store. This is optional. We could instead pass
    # the variable weights directly into step.remote, in which case it would be
    # placed in the object store under the hood. However, in that case multiple
    # copies of the weights would be put in the object store, so this approach is
    # more efficient.
    weights_id = ray.put(weights)
    # Call the remote function multiple times in parallel.
    new_weights_ids = [actor.step.remote(weights_id) for actor in actor_list]
    # Get all of the weights.
    new_weights_list = ray.get(new_weights_ids)
    # Add up all the different weights. Each element of new_weights_list is a dict
```

```
    # of weights, and we want to add up these dicts component wise using the keys
    # of the first dict.
    weights = {variable: sum(weight_dict[variable] for weight_dict in new_weights_
↪list) / NUM_BATCHES for variable in new_weights_list[0]}
    # Print the current weights. They should converge to roughly to the values 0.1
    # and 0.3 used in generate_fake_x_y_data.
    if iteration % 20 == 0:
        print("Iteration {}: weights are {}".format(iteration, weights))
```

How to Train in Parallel using Ray

In some cases, you may want to do data-parallel training on your network. We use the network above to illustrate how to do this in Ray. The only differences are in the remote function `step` and the driver code.

In the function `step`, we run the grad operation rather than the train operation to get the gradients. Since Tensorflow pairs the gradients with the variables in a tuple, we extract the gradients to avoid needless computation.

Extracting numerical gradients

Code like the following can be used in a remote function to compute numerical gradients.

```
x_values = [1] * 100
y_values = [2] * 100
numerical_grads = sess.run([grad[0] for grad in grads], feed_dict={x_data: x_values,␣
↪y_data: y_values})
```

Using the returned gradients to train the network

By pairing the symbolic gradients with the numerical gradients in a feed_dict, we can update the network.

```
# We can feed the gradient values in using the associated symbolic gradient
# operation defined in tensorflow.
feed_dict = {grad[0]: numerical_grad for (grad, numerical_grad) in zip(grads,␣
↪numerical_grads)}
sess.run(train, feed_dict=feed_dict)
```

You can then run `variables.get_weights()` to see the updated weights of the network.

For reference, the full code is below:

```
import tensorflow as tf
import numpy as np
import ray

ray.init()

BATCH_SIZE = 100
NUM_BATCHES = 1
NUM_ITERS = 201

class Network(object):
    def __init__(self, x, y):
        # Seed TensorFlow to make the script deterministic.
```

```
        tf.set_random_seed(0)
        # Define the inputs.
        x_data = tf.constant(x, dtype=tf.float32)
        y_data = tf.constant(y, dtype=tf.float32)
        # Define the weights and computation.
        w = tf.Variable(tf.random_uniform([1], -1.0, 1.0))
        b = tf.Variable(tf.zeros([1]))
        y = w * x_data + b
        # Define the loss.
        self.loss = tf.reduce_mean(tf.square(y - y_data))
        optimizer = tf.train.GradientDescentOptimizer(0.5)
        self.grads = optimizer.compute_gradients(self.loss)
        self.train = optimizer.apply_gradients(self.grads)
        # Define the weight initializer and session.
        init = tf.global_variables_initializer()
        self.sess = tf.Session()
        # Additional code for setting and getting the weights
        self.variables = ray.experimental.TensorFlowVariables(self.loss, self.sess)
        # Return all of the data needed to use the network.
        self.sess.run(init)

    # Define a remote function that trains the network for one step and returns the
    # new weights.
    def step(self, weights):
        # Set the weights in the network.
        self.variables.set_weights(weights)
        # Do one step of training. We only need the actual gradients so we filter␣
↪over the list.
        actual_grads = self.sess.run([grad[0] for grad in self.grads])
        return actual_grads

    def get_weights(self):
        return self.variables.get_weights()

# Define a remote function for generating fake data.
@ray.remote(num_return_vals=2)
def generate_fake_x_y_data(num_data, seed=0):
    # Seed numpy to make the script deterministic.
    np.random.seed(seed)
    x = np.random.rand(num_data)
    y = x * 0.1 + 0.3
    return x, y

# Generate some training data.
batch_ids = [generate_fake_x_y_data.remote(BATCH_SIZE, seed=i) for i in range(NUM_
↪BATCHES)]
x_ids = [x_id for x_id, y_id in batch_ids]
y_ids = [y_id for x_id, y_id in batch_ids]
# Generate some test data.
x_test, y_test = ray.get(generate_fake_x_y_data.remote(BATCH_SIZE, seed=NUM_BATCHES))

# Create actors to store the networks.
remote_network = ray.remote(Network)
actor_list = [remote_network.remote(x_ids[i], y_ids[i]) for i in range(NUM_BATCHES)]
local_network = Network(x_test, y_test)

# Get initial weights of local network.
weights = local_network.get_weights()
```

```
# Do some steps of training.
for iteration in range(NUM_ITERS):
    # Put the weights in the object store. This is optional. We could instead pass
    # the variable weights directly into step.remote, in which case it would be
    # placed in the object store under the hood. However, in that case multiple
    # copies of the weights would be put in the object store, so this approach is
    # more efficient.
    weights_id = ray.put(weights)
    # Call the remote function multiple times in parallel.
    gradients_ids = [actor.step.remote(weights_id) for actor in actor_list]
    # Get all of the weights.
    gradients_list = ray.get(gradients_ids)

    # Take the mean of the different gradients. Each element of gradients_list is a␣
↪list
    # of gradients, and we want to take the mean of each one.
    mean_grads = [sum([gradients[i] for gradients in gradients_list]) / len(gradients_
↪list) for i in range(len(gradients_list[0]))]

    feed_dict = {grad[0]: mean_grad for (grad, mean_grad) in zip(local_network.grads,␣
↪mean_grads)}
    local_network.sess.run(local_network.train, feed_dict=feed_dict)
    weights = local_network.get_weights()

    # Print the current weights. They should converge to roughly to the values 0.1
    # and 0.3 used in generate_fake_x_y_data.
    if iteration % 20 == 0:
        print("Iteration {}: weights are {}".format(iteration, weights))
```

CHAPTER 17

An Overview of the Internals

In this document, we trace through in more detail what happens at the system level when certain API calls are made.

Connecting to Ray

There are two ways that a Ray script can be initiated. It can either be run in a standalone fashion or it can be connect to an existing Ray cluster.

Running Ray standalone

Ray can be used standalone by calling `ray.init()` within a script. When the call to `ray.init()` happens, all of the relevant processes are started. These include a local scheduler, a global scheduler, an object store and manager, a Redis server, and a number of worker processes.

When the script exits, these processes will be killed.

Note: This approach is limited to a single machine.

Connecting to an existing Ray cluster

To connect to an existing Ray cluster, simply pass the argument address of the Redis server as the `redis_address=` keyword argument into `ray.init`. In this case, no new processes will be started when `ray.init` is called, and similarly the processes will continue running when the script exits. In this case, all processes except workers that correspond to actors are shared between different driver processes.

Defining a remote function

A central component of this system is the **centralized control plane**. This is implemented using one or more Redis servers. Redis is an in-memory key-value store.

We use the centralized control plane in two ways. First, as persistent store of the system's control state. Second, as a message bus for communication between processes (using Redis's publish-subscribe functionality).

Now, consider a remote function definition as below.

```
@ray.remote
def f(x):
    return x + 1
```

When the remote function is defined as above, the function is immediately pickled, assigned a unique ID, and stored in a Redis server. You can view the remote functions in the centralized control plane as below.

```
TODO: Fill this in.
```

Each worker process has a separate thread running in the background that listens for the addition of remote functions to the centralized control state. When a new remote function is added, the thread fetches the pickled remote function, unpickles it, and can then execute that function.

Notes and limitations

- Because we export remote functions as soon as they are defined, that means that remote functions can't close over variables that are defined after the remote function is defined. For example, the following code gives an error.

  ```
  @ray.remote
  def f(x):
      return helper(x)

  def helper(x):
      return x + 1
  ```

  If you call `f.remote(0)`, it will give an error of the form.

  ```
  Traceback (most recent call last):
      File "<ipython-input-3-12a5beeb2306>", line 3, in f
  NameError: name 'helper' is not defined
  ```

  On the other hand, if `helper` is defined before `f`, then it will work.

Calling a remote function

When a driver or worker invokes a remote function, a number of things happen.

- First, a task object is created. The task object includes the following.
  - The ID of the function being called.
  - The IDs or values of the arguments to the function. Python primitives like integers or short strings will be pickled and included as part of the task object. Larger or more complex objects will be put into the object store with an internal call to `ray.put`, and the resulting IDs are included in the task object. Object IDs that are passed directly as arguments are also included in the task object.
  - The ID of the task. This is generated uniquely from the above content.
  - The IDs for the return values of the task. These are generated uniquely from the above content.
- The task object is then sent to the local scheduler on the same node as the driver or worker.

- The local scheduler makes a decision to either schedule the task locally or to pass the task on to a global scheduler.
  - If all of the task's object dependencies are present in the local object store and there are enough CPU and GPU resources available to execute the task, then the local scheduler will assign the task to one of its available workers.
  - If those conditions are not met, the task will be passed on to a global scheduler. This is done by adding the task to the **task table**, which is part of the centralized control state. The task table can be inspected as follows.

    ```
    TODO: Fill this in.
    ```

    A global scheduler will be notified of the update and will assign the task to a local scheduler by updating the task's state in the task table. The local scheduler will be notified and pull the task object.
- Once a task has been scheduled to a local scheduler, whether by itself or by a global scheduler, the local scheduler queues the task for execution. A task is assigned to a worker when enough resources become available and the object dependencies are available locally, in first-in, first-out order.
- When the task has been assigned to a worker, the worker executes the task and puts the task's return values into the object store. The object store will then update the **object table**, which is part of the centralized control state, to reflect the fact that it contains the newly created objects. The object table can be viewed as follows.

  ```
  TODO: Fill this in.
  ```

  When the task's return values are placed into the object store, they are first serialized into a contiguous blob of bytes using the Apache Arrow data layout, which is helpful for efficiently sharing data between processes using shared memory.

Notes and limitations

- When an object store on a particular node fills up, it will begin evicting objects in a least-recently-used manner. If an object that is needed later is evicted, then the call to `ray.get` for that object will initiate the reconstruction of the object. The local scheduler will attempt to reconstruct the object by replaying its task lineage.

TODO: Limitations on reconstruction.

Getting an object ID

Several things happen when a driver or worker calls `ray.get` on an object ID.

```
ray.get(x_id)
```

- The driver or worker goes to the object store on the same node and requests the relevant object. Each object store consists of two components, a shared-memory key-value store of immutable objects, and a manager to coordinate the transfer of objects between nodes.
  - If the object is not present in the object store, the manager checks the object table to see which other object stores, if any, have the object. It then requests the object directly from one of those object stores, via its manager. If the object doesn't exist anywhere, then the centralized control state will notify the requesting manager when the object is created. If the object doesn't exist anywhere because it has been evicted from all object stores, the worker will also request reconstruction of the object from the local scheduler. These checks repeat periodically until the object is available in the local object store, whether through reconstruction or through object transfer.

- Once the object is available in the local object store, the driver or worker will map the relevant region of memory into its own address space (to avoid copying the object), and will deserialize the bytes into a Python object. Note that any numpy arrays that are part of the object will not be copied.

CHAPTER 18

Serialization in the Object Store

This document describes what Python objects Ray can and cannot serialize into the object store. Once an object is placed in the object store, it is immutable.

There are a number of situations in which Ray will place objects in the object store.

1. The return values of a remote function.
2. The value `x` in a call to `ray.put(x)`.
3. Arguments to remote functions (except for simple arguments like ints or floats).

A Python object may have an arbitrary number of pointers with arbitrarily deep nesting. To place an object in the object store or send it between processes, it must first be converted to a contiguous string of bytes. This process is known as serialization. The process of converting the string of bytes back into a Python object is known as deserialization. Serialization and deserialization are often bottlenecks in distributed computing.

Pickle is one example of a library for serialization and deserialization in Python.

Pickle (and the variant we use, cloudpickle) is general-purpose. It can serialize a large variety of Python objects. However, for numerical workloads, pickling and unpickling can be inefficient. For example, if multiple processes want to access a Python list of numpy arrays, each process must unpickle the list and create its own new copies of the arrays. This can lead to high memory overheads, even when all processes are read-only and could easily share memory.

In Ray, we optimize for numpy arrays by using the Apache Arrow data format. When we deserialize a list of numpy arrays from the object store, we still create a Python list of numpy array objects. However, rather than copy each numpy array, each numpy array object holds a pointer to the relevant array held in shared memory. There are some advantages to this form of serialization.

- Deserialization can be very fast.
- Memory is shared between processes so worker processes can all read the same data without having to copy it.

What Objects Does Ray Handle

Ray does not currently support serialization of arbitrary Python objects. The set of Python objects that Ray can serialize using Arrow includes the following.

1. Primitive types: ints, floats, longs, bools, strings, unicode, and numpy arrays.
2. Any list, dictionary, or tuple whose elements can be serialized by Ray.

For a more general object, Ray will first attempt to serialize the object by unpacking the object as a dictionary of its fields. This behavior is not correct in all cases. If Ray cannot serialize the object as a dictionary of its fields, Ray will fall back to using pickle. However, using pickle will likely be inefficient.

Notes and limitations

- We currently handle certain patterns incorrectly, according to Python semantics. For example, a list that contains two copies of the same list will be serialized as if the two lists were distinct.

```
l1 = [0]
l2 = [l1, l1]
l3 = ray.get(ray.put(l2))

l2[0] is l2[1]  # True.
l3[0] is l3[1]  # False.
```

- For reasons similar to the above example, we also do not currently handle objects that recursively contain themselves (this may be common in graph-like data structures).

```
l = []
l.append(l)

# Try to put this list that recursively contains itself in the object store.
ray.put(l)
```

  This will throw an exception with a message like the following.

```
This object exceeds the maximum recursion depth. It may contain itself␣
↪recursively.
```

- Whenever possible, use numpy arrays for maximum performance.

Last Resort Workaround

If you find cases where Ray serialization doesn't work or does something unexpected, please let us know so we can fix it. In the meantime, you may have to resort to writing custom serialization and deserialization code (e.g., calling pickle by hand).

```
import pickle

@ray.remote
def f(complicated_object):
    # Deserialize the object manually.
    obj = pickle.loads(complicated_object)
    return "Successfully passed {} into f.".format(obj)
```

```
# Define a complicated object.
l = []
l.append(l)

# Manually serialize the object and pass it in as a string.
ray.get(f.remote(pickle.dumps(l)))  # prints 'Successfully passed [[...]] into f.'
```

Note: If you have trouble with pickle, you may have better luck with cloudpickle.

CHAPTER 19

Fault Tolerance

This document describes the handling of failures in Ray.

Machine and Process Failures

Currently, each **local scheduler** and each **plasma manager** send heartbeats to a **monitor** process. If the monitor does not receive any heartbeats from a given process for some duration of time (about ten seconds), then it will mark that process as dead. The monitor process will then clean up the associated state in the Redis servers. If a manager is marked as dead, the object table will be updated to remove all occurrences of that manager so that other managers don't try to fetch objects from the dead manager. If a local scheduler is marked as dead, all of the tasks that are marked as executing on that local scheduler in the task table will be marked as lost and all actors associated with that local scheduler will be recreated by other local schedulers.

Lost Objects

If an object is needed but is lost or was never created, then the task that created the object will be re-executed to create the object. If necessary, tasks needed to create the input arguments to the task being re-executed will also be re-executed.

Actors

When a local scheduler is marked as dead, all actors associated with that local scheduler that were still alive will be recreated by other local schedulers. By default, all of the actor methods will be re-executed in the same order that they were initially executed. If actor checkpointing is enabled, then the actor state will be loaded from the most recent checkpoint and the actor methods that occurred after the checkpoint will be re-executed. Note that actor checkpointing is currently an experimental feature.

Unhandled Failures

At the moment, Ray does not handle all failure scenarios. We are working on addressing these problems.

Process Failures

1. Ray does not recover from the failure of any of the following processes: a Redis server, the global scheduler, the monitor process.
2. If a driver fails, that driver will not be restarted and the job will not complete.

Lost Objects

1. If an object is constructed by a call to `ray.put` on the driver, is then evicted, and is later needed, Ray will not reconstruct this object.
2. If an object is constructed by an actor method, is then evicted, and is later needed, Ray will not reconstruct this object.

CHAPTER 20

Using Ray on a Cluster

The instructions in this document work well for small clusters. For larger clusters, follow the instructions for managing a cluster with parallel ssh.

Deploying Ray on a Cluster

This section assumes that you have a cluster running and that the nodes in the cluster can communicate with each other. It also assumes that Ray is installed on each machine. To install Ray, follow the instructions for installation on Ubuntu.

Starting Ray on each machine

On the head node (just choose some node to be the head node), run the following. If the `--redis-port` argument is omitted, Ray will choose a port at random.

```
ray start --head --redis-port=6379
```

The command will print out the address of the Redis server that was started (and some other address information).

Then on all of the other nodes, run the following. Make sure to replace `<redis-address>` with the value printed by the command on the head node (it should look something like `123.45.67.89:6379`).

```
ray start --redis-address=<redis-address>
```

If you wish to specify that a machine has 10 CPUs and 1 GPU, you can do this with the flags `--num-cpus=10` and `--num-gpus=1`. If these flags are not used, then Ray will detect the number of CPUs automatically and will assume there are 0 GPUs.

Now we've started all of the Ray processes on each node Ray. This includes

- Some worker processes on each machine.
- An object store on each machine.

- A local scheduler on each machine.
- Multiple Redis servers (on the head node).
- One global scheduler (on the head node).

To run some commands, start up Python on one of the nodes in the cluster, and do the following.

```
import ray
ray.init(redis_address="<redis-address>")
```

Now you can define remote functions and execute tasks. For example, to verify that the correct number of nodes have joined the cluster, you can run the following.

```
import time

@ray.remote
def f():
    time.sleep(0.01)
    return ray.services.get_node_ip_address()

# Get a list of the IP addresses of the nodes that have joined the cluster.
set(ray.get([f.remote() for _ in range(1000)]))
```

Stopping Ray

When you want to stop the Ray processes, run `ray stop` on each node.

CHAPTER 21

Using Ray on a Large Cluster

Deploying Ray on a cluster requires a bit of manual work. The instructions here illustrate how to use parallel ssh commands to simplify the process of running commands and scripts on many machines simultaneously.

Booting up a cluster on EC2

- Create an EC2 instance running Ray following instructions for installation on Ubuntu.
  - Add any packages that you may need for running your application.
  - Install the pssh package: `sudo apt-get install pssh`.
- Create an AMI with Ray installed and with whatever code and libraries you want on the cluster.
- Use the EC2 console to launch additional instances using the AMI you created.
- Configure the instance security groups so that they machines can all communicate with one another.

Deploying Ray on a Cluster

This section assumes that you have a cluster of machines running and that these nodes have network connectivity to one another. It also assumes that Ray is installed on each machine.

Additional assumptions:

- All of the following commands are run from a machine designated as the **head node**.
- The head node will run Redis and the global scheduler.
- The head node has ssh access to all other nodes.
- All nodes are accessible via ssh keys
- Ray is checked out on each node at the location `$HOME/ray`.

Note: The commands below will probably need to be customized for your specific setup.

Connect to the head node

In order to initiate ssh commands from the cluster head node we suggest enabling ssh agent forwarding. This will allow the session that you initiate with the head node to connect to other nodes in the cluster to run scripts on them. You can enable ssh forwarding by running the following command before connecting to the head node (replacing `<ssh-key>` with the path to the private key that you would use when logging in to the nodes in the cluster).

```
ssh-add <ssh-key>
```

Now log in to the head node with the following command, where `<head-node-public-ip>` is the public IP address of the head node (just choose one of the nodes to be the head node).

```
ssh -A ubuntu@<head-node-public-ip>
```

Build a list of node IP addresses

On the head node, populate a file `workers.txt` with one IP address on each line. Do not include the head node IP address in this file. These IP addresses should typically be private network IP addresses, but any IP addresses which the head node can use to ssh to worker nodes will work here. This should look something like the following.

```
172.31.27.16
172.31.29.173
172.31.24.132
172.31.29.224
```

Confirm that you can ssh to all nodes

```
for host in $(cat workers.txt); do
  ssh -o "StrictHostKeyChecking no" $host uptime
done
```

You may need to verify the host keys during this process. If so, run this step again to verify that it worked. If you see a **permission denied** error, you most likely forgot to run `ssh-add <ssh-key>` before connecting to the head node.

Starting Ray

Start Ray on the head node

On the head node, run the following:

```
ray start --head --redis-port=6379
```

Start Ray on the worker nodes

Create a file `start_worker.sh` that contains something like the following:

```
# Make sure the SSH session has the correct version of Python on its path.
# You will probably have to change the line below.
export PATH=/home/ubuntu/anaconda3/bin/:$PATH
ray start --redis-address=<head-node-ip>:6379
```

This script, when run on the worker nodes, will start up Ray. You will need to replace `<head-node-ip>` with the IP address that worker nodes will use to connect to the head node (most likely a **private IP address**). In this example we also export the path to the Python installation since our remote commands will not be executing in a login shell.

Warning: You will probably need to manually export the correct path to Python (you will need to change the first line of `start_worker.sh` to find the version of Python that Ray was built against). This is necessary because the `PATH` environment variable used by `parallel-ssh` can differ from the `PATH` environment variable that gets set when you `ssh` to the machine.

Warning: If the `parallel-ssh` command below appears to hang or otherwise fails, `head-node-ip` may need to be a private IP address instead of a public IP address (e.g., if you are using EC2). It's also possible that you forgot to run `ssh-add <ssh-key>` or that you forgot the `-A` flag when connecting to the head node.

Now use `parallel-ssh` to start up Ray on each worker node.

```
parallel-ssh -h workers.txt -P -I < start_worker.sh
```

Note that on some distributions the `parallel-ssh` command may be called `pssh`.

Verification

Now you have started all of the Ray processes on each node. These include:

- Some worker processes on each machine.
- An object store on each machine.
- A local scheduler on each machine.
- Multiple Redis servers (on the head node).
- One global scheduler (on the head node).

To confirm that the Ray cluster setup is working, start up Python on one of the nodes in the cluster and enter the following commands to connect to the Ray cluster.

```
import ray
ray.init(redis_address="<redis-address>")
```

Here `<redis-address>` should have the form `<head-node-ip>:6379`.

Now you can define remote functions and execute tasks. For example, to verify that the correct number of nodes have joined the cluster, you can run the following.

```
import time

@ray.remote
def f():
    time.sleep(0.01)
    return ray.services.get_node_ip_address()

# Get a list of the IP addresses of the nodes that have joined the cluster.
set(ray.get([f.remote() for _ in range(1000)]))
```

Stopping Ray

Stop Ray on worker nodes

Create a file `stop_worker.sh` that contains something like the following:

```
# Make sure the SSH session has the correct version of Python on its path.
# You will probably have to change the line below.
export PATH=/home/ubuntu/anaconda3/bin/:$PATH
ray stop
```

This script, when run on the worker nodes, will stop Ray. Note, you will need to replace `/home/ubuntu/anaconda3/bin/` with the correct path to your Python installation.

Now use `parallel-ssh` to stop Ray on each worker node.

```
parallel-ssh -h workers.txt -P -I < stop_worker.sh
```

Stop Ray on the head node

```
ray stop
```

Upgrading Ray

Ray remains under active development so you may at times want to upgrade the cluster to take advantage of improvements and fixes.

Create an upgrade script

On the head node, create a file called `upgrade.sh` that contains the commands necessary to upgrade Ray. It should look something like the following:

```
# Make sure the SSH session has the correct version of Python on its path.
# You will probably have to change the line below.
export PATH=/home/ubuntu/anaconda3/bin/:$PATH
# Do pushd/popd to make sure we end up in the same directory.
pushd .
# Upgrade Ray.
cd ray
git remote set-url origin https://github.com/ray-project/ray
git checkout master
git pull
cd python
python setup.py install --user
popd
```

This script executes a series of git commands to update the Ray source code, then builds and installs Ray.

Stop Ray on the cluster

Follow the instructions for *Stopping Ray*.

Run the upgrade script on the cluster

First run the upgrade script on the head node. This will upgrade the head node and help confirm that the upgrade script is working properly.

```
bash upgrade.sh
```

Next run the upgrade script on the worker nodes.

```
parallel-ssh -h workers.txt -P -t 0 -I < upgrade.sh
```

Note here that we use the `-t 0` option to set the timeout to infinite. You may also want to use the `-p` flag, which controls the degree of parallelism used by parallel ssh.

It is probably a good idea to ssh to one of the other nodes and verify that the upgrade script ran as expected.

Sync Application Files to other nodes

If you are running an application that reads input files or uses python libraries then you may find it useful to copy a directory on the head node to the worker nodes.

You can do this using the `parallel-rsync` command:

```
parallel-rsync -h workers.txt -r <workload-dir> /home/ubuntu/<workload-dir>
```

where `<workload-dir>` is the directory you want to synchronize. Note that the destination argument for this command must represent an absolute path on the worker node.

Troubleshooting

Problems with parallel-ssh

If any of the above commands fail, verify that the head node has SSH access to the other nodes by running

```
for host in $(cat workers.txt); do
  ssh $host uptime
done
```

If you get a permission denied error, then make sure you have SSH'ed to the head node with agent forwarding enabled. This is done as follows.

```
ssh-add <ssh-key>
ssh -A ubuntu@<head-node-public-ip>
```

CHAPTER 22

Using Ray and Docker on a Cluster (EXPERIMENTAL)

Packaging and deploying an application using Docker can provide certain advantages. It can make managing dependencies easier, help ensure that each cluster node receives a uniform configuration, and facilitate swapping hardware resources between applications.

Create your Docker image

First build a Ray Docker image by following the instructions for Installation on Docker. This will allow you to create the `ray-project/deploy` image that serves as a basis for using Ray on a cluster with Docker.

Docker images encapsulate the system state that will be used to run nodes in the cluster. We recommend building on top of the Ray-provided Docker images to add your application code and dependencies.

You can do this in one of two ways: by building from a customized Dockerfile or by saving an image after entering commands manually into a running container. We describe both approaches below.

Creating a customized Dockerfile

We recommend that you read the official Docker documentation for Building your own image ahead of starting this section. Your customized Dockerfile is a script of commands needed to set up your application, possibly packaged in a folder with related resources.

A simple template Dockerfile for a Ray application looks like this:

```
# Application Dockerfile template
FROM ray-project/deploy
RUN git clone <my-project-url>
RUN <my-project-installation-script>
```

This file instructs Docker to load the image tagged `ray-project/deploy`, check out the git repository at `<my-project-url>`, and then run the script `<my-project-installation-script>`.

Build the image by running something like:

```
docker build -t <my-app> .
```

Replace <app-tag> with a tag of your choice.

Creating a Docker image manually

Launch the ray-project/deploy image interactively

```
docker run -t -i ray-project/deploy
```

Next, run whatever commands are needed to install your application. When you are finished type exit to stop the container.

Run

```
docker ps -a
```

to identify the id of the container you just exited.

Next, commit the container

```
docker commit -t <app-tag> <container-id>
```

Replace <app-tag> with a name for your container and replace <container-id> id with the hash id of the container used in configuration.

Publishing your Docker image to a repository

When using Amazon EC2 it can be practical to publish images using the Repositories feature of Elastic Container Service. Follow the steps below and see documentation for creating a repository for additional context.

First ensure that the AWS command-line interface is installed.

```
sudo apt-get install -y awscli
```

Next create a repository in Amazon's Elastic Container Registry. This results in a shared resource for storing Docker images that will be accessible from all nodes.

```
aws ecr create-repository --repository-name <repository-name> --region=<region>
```

Replace <repository-name> with a string describing the application. Replace <region> with the AWS region string, e.g., us-west-2. This should produce output like the following:

```
{
    "repository": {
        "repositoryUri": "123456789012.dkr.ecr.us-west-2.amazonaws.com/my-app",
        "createdAt": 1487227244.0,
        "repositoryArn": "arn:aws:ecr:us-west-2:123456789012:repository/my-app",
        "registryId": "123456789012",
        "repositoryName": "my-app"
    }
}
```

Take note of the `repositoryUri` string, in this example `123456789012.dkr.ecr.us-west-2.amazonaws.com/my-app`.

Tag the Docker image with the repository URI.

```
docker tag <app-tag> <repository-uri>
```

Replace the `<app-tag>` with the container name used previously and replace `<repository-uri>` with URI returned by the command used to create the repository.

Log into the repository:

```
eval $(aws ecr get-login --region <region>)
```

Replace `<region>` with your selected AWS region.

Push the image to the repository:

```
docker push <repository-uri>
```

Replace `<repository-uri>` with the URI of your repository. Now other hosts will be able to access your application Docker image.

Starting a cluster

We assume a cluster configuration like that described in instructions for using Ray on a large cluster. In particular, we assume that there is a head node that has ssh access to all of the worker nodes, and that there is a file `workers.txt` listing the IP addresses of all worker nodes.

Install the Docker image on all nodes

Create a script called `setup-docker.sh` on the head node.

```
# setup-docker.sh
sudo apt-get install -y docker.io
sudo service docker start
sudo usermod -a -G docker ubuntu
exec sudo su -l ubuntu
eval $(aws ecr get-login --region <region>)
docker pull <repository-uri>
```

Replace `<repository-uri>` with the URI of the repository created in the previous section. Replace `<region>` with the AWS region in which you created that repository. This script will install Docker, authenticate the session with the container registry, and download the container image from that registry.

Run `setup-docker.sh` on the head node (if you used the head node to build the Docker image then you can skip this step):

```
bash setup-docker.sh
```

Run `setup-docker.sh` on the worker nodes:

```
parallel-ssh -h workers.txt -P -t 0 -I < setup-docker.sh
```

Launch Ray cluster using Docker

To start Ray on the head node run the following command:

```
eval $(aws ecr get-login --region <region>)
docker run \
    -d --shm-size=<shm-size> --net=host \
    <repository-uri> \
    ray start --head \
        --object-manager-port=8076 \
        --redis-port=6379 \
        --num-workers=<num-workers>
```

Replace `<repository-uri>` with the URI of the repository. Replace `<region>` with the region of the repository. Replace `<num-workers>` with the number of workers, e.g., typically a number similar to the number of cores in the system. Replace `<shm-size>` with the the amount of shared memory to make available within the Docker container, e.g., `8G`.

To start Ray on the worker nodes create a script `start-worker-docker.sh` with content like the following:

```
eval $(aws ecr get-login --region <region>)
docker run -d --shm-size=<shm-size> --net=host \
    <repository-uri> \
    ray start \
        --object-manager-port=8076 \
        --redis-address=<redis-address> \
        --num-workers=<num-workers>
```

Replace `<redis-address>` with the string `<head-node-private-ip>:6379` where `<head-node-private-ip>` is the private network IP address of the head node.

Execute the script on the worker nodes:

```
parallel-ssh -h workers.txt -P -t 0 -I < setup-worker-docker.sh
```

Running jobs on a cluster

On the head node, identify the id of the container that you launched as the Ray head.

```
docker ps
```

the container id appears in the first column of the output.

Now launch an interactive shell within the container:

```
docker exec -t -i <container-id> bash
```

Replace `<container-id>` with the container id found in the previous step.

Next, launch your application program. The Python program should contain an initialization command that takes the Redis address as a parameter:

```
ray.init(redis_address="<redis-address>")
```

Shutting down a cluster

Kill all running Docker images on the worker nodes:

```
parallel-ssh -h workers.txt -P 'docker kill $(docker ps -q)'
```

Kill all running Docker images on the head node:

```
docker kill $(docker ps -q)
```

CHAPTER 23

Troubleshooting

This document discusses some common problems that people run into when using Ray as well as some known problems. If you encounter other problems, please let us know.

No Speedup

You just ran an application using Ray, but it wasn't as fast as you expected it to be. Or worse, perhaps it was slower than the serial version of the application! The most common reasons are the following.

- **Number of cores:** How many cores is Ray using? When you start Ray, it will determine the number of CPUs on each machine with `psutil.cpu_count()`. Ray usually will not schedule more tasks in parallel than the number of CPUs. So if the number of CPUs is 4, the most you should expect is a 4x speedup.
- **Physical versus logical CPUs:** Do the machines you're running on have fewer **physical** cores than **logical** cores? You can check the number of logical cores with `psutil.cpu_count()` and the number of physical cores with `psutil.cpu_count(logical=False)`. This is common on a lot of machines and especially on EC2. For many workloads (especially numerical workloads), you often cannot expect a greater speedup than the number of physical CPUs.
- **Small tasks:** Are your tasks very small? Ray introduces some overhead for each task (the amount of overhead depends on the arguments that are passed in). You will be unlikely to see speedups if your tasks take less than ten milliseconds. For many workloads, you can easily increase the sizes of your tasks by batching them together.
- **Variable durations:** Do your tasks have variable duration? If you run 10 tasks with variable duration in parallel, you shouldn't expect an N-fold speedup (because you'll end up waiting for the slowest task). In this case, consider using `ray.wait` to begin processing tasks that finish first.
- **Multi-threaded libraries:** Are all of your tasks attempting to use all of the cores on the machine? If so, they are likely to experience contention and prevent your application from achieving a speedup. You can diagnose this by opening `top` while your application is running. If one process is using most of the CPUs, and the others are using a small amount, this may be the problem. This is very common with some versions of `numpy`, and in that case can usually be setting an environment variable like `MKL_NUM_THREADS` (or the equivalent depending on your installation) to `1`.

If you are still experiencing a slowdown, but none of the above problems apply, we'd really like to know! Please create a GitHub issue and consider submitting a minimal code example that demonstrates the problem.

Crashes

If Ray crashed, you may wonder what happened. Currently, this can occur for some of the following reasons.

- **Stressful workloads:** Workloads that create many many tasks in a short amount of time can sometimes interfere with the heartbeat mechanism that we use to check that processes are still alive. On the head node in the cluster, you can check the files `/tmp/raylogs/monitor-******.out` and `/tmp/raylogs/monitor-******.err`. They will indicate which processes Ray has marked as dead (due to a lack of heartbeats). However, it is currently possible for a process to get marked as dead without actually having died.
- **Starting many actors:** Workloads that start a large number of actors all at once may exhibit problems when the processes (or libraries that they use) contend for resources. Similarly, a script that starts many actors over the lifetime of the application will eventually cause the system to run out of file descriptors. This is addressable, but currently we do not garbage collect actor processes until the script finishes.
- **Running out of file descriptors:** As a workaround, you may be able to increase the maximum number of file descriptors with a command like `ulimit -n 65536`. If that fails, double check that the hard limit is sufficiently large by running `ulimit -Hn`. If it is too small, you can increase the hard limit as follows (these instructions work on EC2).
  - Increase the hard ulimit for open file descriptors system-wide by running the following.

    ```
    sudo bash -c "echo $USER hard nofile 65536 >> /etc/security/limits.conf"
    ```

  - Logout and log back in.

Hanging

If a workload is hanging and not progressing, the problem may be one of the following.

- **Reconstructing an object created with put:** When an object that is needed has been evicted or lost, Ray will attempt to rerun the task that created the object. However, there are some cases that currently are not handled. For example, if the object was created by a call to `ray.put` on the driver process, then the argument that was passed into `ray.put` is no longer available and so the call to `ray.put` cannot be rerun (without rerunning the driver).
- **Reconstructing an object created by actor task:** Ray currently does not reconstruct objects created by actor methods.

Serialization Problems

Ray's serialization is currently imperfect. If you encounter an object that Ray does not serialize/deserialize correctly, please let us know. For example, you may want to bring it up on this thread.

- Objects with multiple references to the same object.
- Subtypes of lists, dictionaries, or tuples.

Outdated Function Definitions

Due to subtleties of Python, if you redefine a remote function, you may not always get the expected behavior. In this case, it may be that Ray is not running the newest version of the function.

Suppose you define a remote function `f` and then redefine it. Ray should use the newest version.

```
@ray.remote
def f():
    return 1

@ray.remote
def f():
    return 2

ray.get(f.remote())  # This should be 2.
```

However, the following are cases where modifying the remote function will not update Ray to the new version (at least without stopping and restarting Ray).

- **The function is imported from an external file:** In this case, `f` is defined in some external file `file.py`. If you `import file`, change the definition of `f` in `file.py`, then re-`import file`, the function `f` will not be updated.

  This is because the second import gets ignored as a no-op, so `f` is still defined by the first import.

  A solution to this problem is to use `reload(file)` instead of a second `import file`. Reloading causes the new definition of `f` to be re-executed, and exports it to the other machines. Note that in Python 3, you need to do `from importlib import reload`.

- **The function relies on a helper function from an external file:** In this case, `f` can be defined within your Ray application, but relies on a helper function `h` defined in some external file `file.py`. If the definition of `h` gets changed in `file.py`, redefining `f` will not update Ray to use the new version of `h`.

  This is because when `f` first gets defined, its definition is shipped to all of the workers, and is unpickled. During unpickling, `file.py` gets imported in the workers. Then when `f` gets redefined, its definition is again shipped and unpickled in all of the workers. But since `file.py` has been imported in the workers already, it is treated as a second import and is ignored as a no-op.

  Unfortunately, reloading on the driver does not update `h`, as the reload needs to happen on the worker.

  A solution to this problem is to redefine `f` to reload `file.py` before it calls `h`. For example, if inside `file.py` you have

  ```
  def h():
      return 1
  ```

  And you define remote function `f` as

  ```
  @ray.remote
  def f():
      return file.h()
  ```

  You can redefine `f` as follows.

  ```
  @ray.remote
  def f():
      reload(file)
      return file.h()
  ```

This forces the reload to happen on the workers as needed. Note that in Python 3, you need to do `from importlib import reload`.

Index