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.