# Distributed Parallel System
# Assignment 2 - Ray Framework

Gahan Saraiya (18MCEC10)

April, 2019

## I Introduction

Parallel and distributed computing are a staple of modern applications. We need to leverage multiple cores or multiple machines to speed up applications or to run them at a large scale. The infrastructure for crawling the web and responding to search queries are not single-threaded programs running on someone's laptop but rather collections of services that communicate and interact with one another.

The cloud promises unlimited scalability in all directions (memory, compute, storage, etc). Realizing this promise requires new tools for programming the cloud and building distributed applications.

To task advantage of this unlimited scalability this experiment aims towards building application with Ray Framework build in python which can scale from `laptop to large cluster`.

### I Significance of Ray

Ray is capable to handle requirements of modern days application listed below:

- Running the same code on more than one machine.
- Building micro-services and actors that have state and can communicate.
- Gracefully handling machine failures.
- Efficiently handling large objects and numerical data.

Traditional Programming rely on 2 core concepts:

- functions
- classes

Using these building blocks, programming languages allow us to build countless applications.

However, when we migrate our applications to the distributed setting, the concepts typically change.

On one end of the spectrum, we have tools like OpenMPI, `Python multiprocessing`, and ZeroMQ, which provide low-level primitives for sending and receiving messages. These tools are very powerful, but they provide a different abstraction and so single-threaded applications must be rewritten from scratch to use them.

On the other end of the spectrum, we have domain-specific tools like TensorFlow for model training, Spark for data processing and SQL, and Flink for stream processing. These tools provide higher-level abstractions like neural networks, datasets, and streams. However, because they differ from the abstractions used for serial programming, applications again must be rewritten from scratch to leverage them.

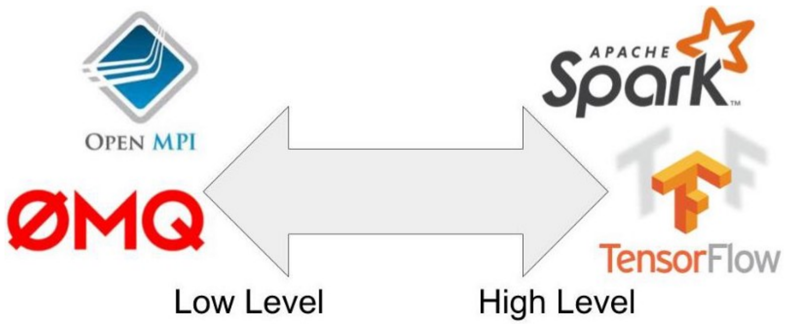

Figure 1: Tools for distributed computing on an axis from low-level primitives to high-level abstractions.

Ray occupies a unique middle ground. Instead of introducing new concepts. Ray takes the existing concepts of functions and classes and translates them to the distributed setting as tasks and actors. This API choice allows serial applications to be parallelized without major modifications.

## II Introduction to Ray

Ray is a 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 raylet per node assigns tasks to workers on the same node.

- 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 raylet and get objects from the object store, but it is different in that the raylet 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.

## I Initializing Ray

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

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

The `ray.init()` command starts all of the relevant Ray processes.

On a cluster, this is the only line that needs to change (we need to pass in the cluster address). These processes include the following:

- A number of worker processes for executing Python functions in parallel (roughly one worker per CPU core).
- A scheduler process for assigning "tasks" to workers (and to other machines). A task is the unit of work scheduled by Ray and corresponds to one function invocation or method invocation.
- A shared-memory object store for sharing objects efficiently between workers (without creating copies).
- An in-memory database for storing metadata needed to rerun tasks in the event of machine failures.

Ray workers are separate processes as opposed to threads because support for multi-threading in Python is very limited due to the global interpreter lock.

## II 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 below:

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

A remote function looks like this.

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

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.

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

```python
import time

def foo():
    time.sleep(1)

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

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

# The following takes one second (assuming the system has at least ten
↪  CPUs).
ray.get([bar.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 raylet, 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))
```

## III Parallelism with Tasks

To turn a Python function `foo` into a “remote function” (a function that can be executed remotely and asynchronously), we declare the function with the `@ray.remote` decorator. Then function invocations via `f.remote()` will immediately return futures (a future is a reference to the eventual output), and the actual function execution will take place in the background (we refer to this execution as a task).

```python
#!/home/jarvis/.virtualenvs/ray/bin/python3
# -*- coding: utf-8 -*-
"""
Author: Gahan Saraiya
GiT: https://github.com/gahan9
StackOverflow: https://stackoverflow.com/users/story/7664524

Example of Task Parallellism with Ray
"""
import ray
import time


@ray.remote
def foo(x):
    time.sleep(1)
    return x

# Start 4 tasks in parallel.
result_ids = []
for i in range(4):
    result_ids.append(foo.remote(i))

if __name__ == "__main__":
    ray.init()
    # Wait for the tasks to complete and retrieve the results.
    # With at least 4 cores, this will take 1 second.
    results = ray.get(result_ids)  # [0, 1, 2, 3]
```

Because the call to foo.remote(i) returns immediately, four copies of foo can be executed in parallel simply by running that line four times.

## I Task Dependencies

Tasks can also depend on other tasks. Below, the multiply_matrices task uses the outputs of the two create_matrix tasks, so it will not begin executing until after the first two tasks have executed. The outputs of the first two tasks will automatically be passed as arguments into the third task and the futures will be replaced with their corresponding values). In this manner, tasks can be composed together with arbitrary DAG dependencies.

## II Aggregating Values Efficiently

Task dependencies can be used in much more sophisticated ways. For example, suppose we wish to aggregate 8 values together. This example uses integer addition, but in many applications, aggregating large vectors across multiple machines can be a bottleneck. In this case, changing a single line of code can change the aggregation's running time from linear to logarithmic in the number of values being aggregated.

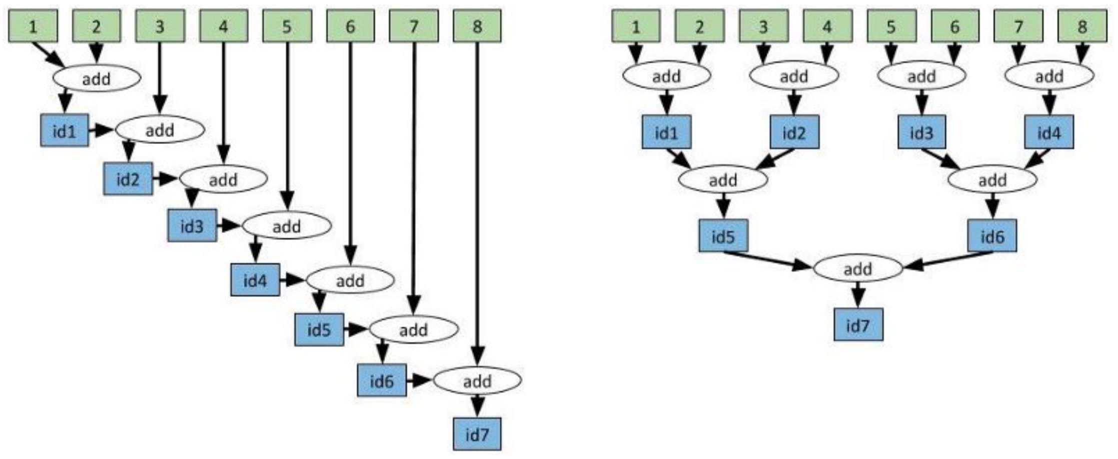

Figure 2: The dependency graph on the left has depth 7. The dependency graph on the right has depth 3. The computations yield the same result, but the one on the right is much faster.

As described above, to feed the output of one task as an input into a subsequent task, simply pass the future returned by the first task as an argument into the second task. This task dependency will automatically be taken into account by Ray's scheduler. The second task will not execute until the first task has finished, and the output of the first task will automatically be shipped to the machine on which the second task is executing.

```
#!/home/jarvis/.virtualenvs/ray/bin/python3
```

```python
# -*- coding: utf-8 -*-
"""
Author: Gahan Saraiya
GiT: https://github.com/gahan9
StackOverflow: https://stackoverflow.com/users/story/7664524

Explicit Aggregation of addition function
"""

import ray
import time
import cProfile

@ray.remote
def add(x, y):
    time.sleep(1)
    return x + y

def approach1():
    """
    Aggregate the values slowly. This approach takes O(n) where n is the
    number of values being aggregated. In this case, 7 seconds.
    """
    id1 = add.remote(1, 2)
    id2 = add.remote(id1, 3)
    id3 = add.remote(id2, 4)
    id4 = add.remote(id3, 5)
    id5 = add.remote(id4, 6)
    id6 = add.remote(id5, 7)
    id7 = add.remote(id6, 8)
    result = ray.get(id7)
    return result

def tree_approach():
    """
    Aggregate the values in a tree-structured pattern. This approach
    takes O(log(n)). In this case, 3 seconds.
    """
    id1 = add.remote(1, 2)
    id2 = add.remote(3, 4)
    id3 = add.remote(5, 6)
    id4 = add.remote(7, 8)
    id5 = add.remote(id1, id2)
    id6 = add.remote(id3, id4)
    id7 = add.remote(id5, id6)
    result = ray.get(id7)

if __name__ == "__main__":
    # Start Ray.
    ray.init()
    cProfile.run("approach1()")
```

The above code is very explicit, but note that both approaches can be implemented in a more concise fashion using while loops.

```
# -*- coding: utf-8 -*-
"""
Author: Gahan Saraiya
GiT: https://github.com/gahan9
StackOverflow: https://stackoverflow.com/users/story/7664524

Speeding up aggregation
"""
import ray
import time
import cProfile

@ray.remote
def add(x, y):
    time.sleep(1)
    return x + y

```

```python
if __name__ == "__main__":
    # Slow approach.
    values = [1, 2, 3, 4, 5, 6, 7, 8]
    while len(values) > 1:
        values = [add.remote(values[0], values[1])] + values[2:]
    result = ray.get(values[0])


    # Fast approach.
    values = [1, 2, 3, 4, 5, 6, 7, 8]
    while len(values) > 1:
        values = values[2:] + [add.remote(values[0], values[1])]
    result = ray.get(values[0])
```

## IV From Classes to Actors

It's challenging to write interesting applications without using classes, and this is as true in the distributed setting as it is on a single core.

Ray allows you to take a Python class and declare it with the @ray.remote decorator. Whenever the class is instantiated, Ray creates a new "actor", which is a process that runs somewhere in the cluster and holds a copy of the object. Method invocations on that actor turn into tasks that run on the actor process and can access and mutate the state of the actor. In this manner, actors allow mutable state to be shared between multiple tasks in a way that remote functions do not.

Individual actors execute methods serially (each individual method is atomic) so there are no race conditions. Parallelism can be achieved by creating multiple actors.

```python
#!/home/jarvis/.virtualenvs/ray/bin/python3
# -*- coding: utf-8 -*-
"""
Author: Gahan Saraiya
GiT: https://github.com/gahan9
StackOverflow: https://stackoverflow.com/users/story/7664524

using Actor classes
"""

import ray
import time

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

    def inc(self):
        self.x += 1

    def get_value(self):
        return self.x

```

```python
if __name__ == "__main__":
    ray.init()
    # Create an actor process.
    c = Counter.remote()

    # Check the actor's counter value.
    print(ray.get(c.get_value.remote()))  # 0

    # Increment the counter twice and check the value again.
    c.inc.remote()
    c.inc.remote()
    print(ray.get(c.get_value.remote())) # 2
```

The above example is the simplest possible usage of actors. The line `Counter.remote()` creates a new actor process, which has a copy of the Counter object. The calls to `c.get_value.remote()` and c.inc.remote() execute tasks on the remote actor process and mutate the state of the actor.

## I Actor Handles

In the above example, we only invoked methods on the actor from the main Python script. One of the most powerful aspects of actors is that we can pass around handles to an actor, which allows other actors or other tasks to all invoke methods on the same actor.

The following example creates an actor that stores messages. Several worker tasks repeatedly push messages to the actor, and the main Python script reads the messages periodically.

```python
#!/home/jarvis/.virtualenvs/ray/bin/python3
# -*- coding: utf-8 -*-
"""
Author: Gahan Saraiya
GiT: https://github.com/gahan9
StackOverflow: https://stackoverflow.com/users/story/7664524

Code for invoking methods on an actor from multiple concurrent tasks.
"""

```

```python
__doc__ = """Code for invoking methods on an actor from multiple
↪  concurrent tasks.
"""

__author__ = "Gahan Saraiya"

import ray
import time


@ray.remote
class MessageActor(object):
    def __init__(self):
        self.messages = []

    def add_message(self, message):
        self.messages.append(message)

    def get_and_clear_messages(self):
        messages = self.messages
        self.messages = []
        return messages


# Define a remote function which loops around and pushes
# messages to the actor.
@ray.remote
def worker(message_actor, j):
    for i in range(100):
        time.sleep(1)
        message_actor.add_message.remote(
            "Message {} from worker {}.".format(i, j))

if __name__ == "__main__":
    ray.init()
    # Create a message actor.
    message_actor = MessageActor.remote()

    # Start 3 tasks that push messages to the actor.
    [worker.remote(message_actor, j) for j in range(3)]

    # Periodically get the messages and print them.
    for _ in range(100):
        new_messages = ray.get(message_actor.get_and_clear_messages.remote())
        print("New messages:", new_messages)
        time.sleep(1)

# This script prints something like the following:
# New messages: []
# New messages: ['Message 0 from worker 1.', 'Message 0 from worker 0.']
# New messages: ['Message 0 from worker 2.', 'Message 1 from worker 1.', 'Message 1 from worker 0.', 'Message 1 from worker 2.']
# New messages: ['Message 2 from worker 1.', 'Message 2 from worker 0.', 'Message 2 from worker 2.']
# New messages: ['Message 3 from worker 2.', 'Message 3 from worker 1.', 'Message 3 from worker 0.']
# New messages: ['Message 4 from worker 2.', 'Message 4 from worker 0.', 'Message 4 from worker 1.']
# New messages: ['Message 5 from worker 2.', 'Message 5 from worker 0.', 'Message 5 from worker 1.']
```

Actors are extremely powerful. They allow you to take a Python class and instantiate it as a microservice which can be queried from other actors and tasks and even other applications.

Tasks and actors are the core abstractions provided by Ray. These two concepts are very general and can be used to implement sophisticated applications including Ray's builtin libraries for reinforcement learning, hyperparameter tuning, speeding up Pandas, and much more.

## V Tuning and profiling

For the profiling I have implemented a simple code of $O(n^2)$ as shown below:

```
sum([i*j*1
    for i in range(n)
    for j in range(n)
    for k in range(1)
])
```

As you can see the code will compute multiplication of every permutation of $n$ (integer number provided for computation) and then calculating sum of all these numbers.

### I Sequential Code

```
#!/home/jarvis/.virtualenvs/ray/bin/python3
# -*- coding: utf-8 -*-
"""
Author: Gahan Saraiya
GiT: https://github.com/gahan9
StackOverflow: https://stackoverflow.com/users/story/7664524

Experiment of loop of n^2 complexity without Ray
"""

import time


def stress_function(num):
    return sum([i * j * 1
                for i in range(num)
                for j in range(num)
                for k in range(1)
                ])


```

```python
if __name__ == "__main__":
    TEST_LIS = [100, 200, 300, 500, 700, 1000]
    for t in TEST_LIS:
        start = time.time()
        result = [stress_function(t) for _ in range(t)]
        print("Time Elapsed for Input Size {}: {:.4f}".format(t,
         ↪  time.time() - start))
```

### I.1 Output

```
Time Elapsed for Input Size 100: 0.3270
Time Elapsed for Input Size 200: 2.6102
Time Elapsed for Input Size 300: 9.6823
Time Elapsed for Input Size 500: 46.1485
Time Elapsed for Input Size 700: 117.5654
Time Elapsed for Input Size 1000: 387.8440
```

## II Parallel Code

Below is the implementation of parallel code with single node and 4 cores:

```python
#!/home/jarvis/.virtualenvs/ray/bin/python3
# -*- coding: utf-8 -*-
"""
Author: Gahan Saraiya
GiT: https://github.com/gahan9
StackOverflow: https://stackoverflow.com/users/story/7664524

Experiment of loop of n^2 complexity with Ray
"""
import time
import ray

```

```python

@ray.remote
def stress_function(num):
    # Example of loop O(n^2)
    return sum([i * j * 1
                for i in range(num)
                for j in range(num)
                for k in range(1)
                ])


if __name__ == "__main__":
    # Initialize ray
    ray.init()
    # ray.init(redis_address="0.0.0.0:6667")
    TEST_LIS = [100, 200, 300, 500, 700, 1000]
    for t in TEST_LIS:
        start = time.time()
        # Remote function is invoked by .remote keyword
        result = ray.get([stress_function.remote(t) for _ in range(t)])
        print("Time Elapsed for Input Size {}: {:.4f}".format(t,
         ↪  time.time() - start))
```

### II.1 Output

```
Time Elapsed for Input Size 100: 2.0011
Time Elapsed for Input Size 200: 2.8617
Time Elapsed for Input Size 300: 5.8519
Time Elapsed for Input Size 500: 28.4275
Time Elapsed for Input Size 700: 77.6339
Time Elapsed for Input Size 1000: 227.4916
```

## VI Summarizing Performance

As you can see in the code implemented in section V.I.I is sequential and the amount of time (in seconds) is increased by the number of iterations.

| **Number of Data ($n$)** | **Sequential code [V.I.I] execution (in seconds)** | **Parallel code [V.II.II] execution (in seconds) (with *1 node and 4 cores*)** |
|---|---|---|
| 100 | 0.3270 | 2.0011 |
| 200 | 2.6102 | 2.8617 |
| 300 | 9.6823 | 5.8519 |
| 500 | 46.1485 | 28.4275 |
| 700 | 117.5654 | 77.6339 |
| 1000 | 387.8440 | 227.4916 |

Table 1: result of observation (Note that due to limitation of resources the code can not be executed with more than one nodes)

As observed in result (refer Table 1) for small data chunk size the overhead observed with Ray implementation, however as data size increases, gradually speedup observed is increased and the parallel version is able to complete the same task can be completed lot quicker.

Various other Tuning implementation with Ray are available at `https://github.com/gahan9/ray/tree/master/python/ray/tune/examples`.