Parallel/Distributed Programming with Python

Abdus Salam Azad

Parallel Programming

- Running parts of a program “simultaneously” in multiple processes
- Reduces the overall processing time

```python
import time

def compute(idx):
    """a dummy function abstracting
    some heavy computation"""
    time.sleep(1)  #simulating a time delay
    return idx*idx


tasks=4
res = []
for i in range(tasks):
    x = compute(i)
    res.append(x)

print(res)
```

The sequential code would take ~4 seconds to complete

what if we could execute compute() parallely?

How to do Parallel Processing in Python??

- The built-in “`multiprocessing`” module
- A widely adopted open-source library “`ray`”
- other open-source libraries are available too e.g “dask” ...

Ray

- Developed by Berkeley researchers from RiseLab
  - disclaimer: the speaker is also a member of RiseLab

- https://github.com/ray-project/ray

Installing Ray

- `pip install ray`
- verify installation

```
import ray

ray.init()
```

Let’s try to parallelize our previous example [demo]

```python
import time

def compute(idx):
    """a dummy function abstracting
    some heavy computation"""
    time.sleep(1)  #simulating a time delay
    return idx*idx


tasks=4
res = []
for i in range(4):
    res.append(compute(i))

print(res)
```

Can we answer the following questions?

- Why is the serial execution time exactly not 4 times the parallel execution time?

Can we answer the following questions?

- Why is the serial execution time exactly not 4 times the parallel execution time?
  - invoking `compute.remote()` or `ray.get()` has some overhead
  - Each program has some serial part which we do not parallelize
    - we only parallelize some parts of the program, the rest remains sequential

Measuring Performance

- How to quantify parallelization performance?

$$\text{speedup} = \frac{\text{Time to execute Sequential Program}}{\text{Time to execute Parallel Program}}$$

Can we answer the following questions?

- What if we change `time.sleep(1)` to `time.sleep(5)`
    - How is speedup impacted ?

Can we answer the following questions?

- What if we change `task = 4` to `task = 15 ??`

Parallel Computation Model: Task and Future

- Each remote function is called a “Task” e,g., compute
- The initiation of task, i.e., compute.remote() is a non-blocking call
  - returns to the main program immediately
- compute.remote() returns a *future*
  - also called a *promise*
- *Future*/*Promise* objects holds a promise to you:
  - keep working on other parts of the program and when you will need me I will be there
  - just call me with ray.get([future])

Tasks

- Takes in input, does some computation and returns the computed result

- “side-effect” free:
  - doesn’t change any program state outside of the task
  - “stateless”

Parallel Data Processing with Task Dependencies

- we can execute *get_arg1* and *get_arg2* parallely
- What if we want to make `compute` a remote function too?

```
1   #Suppose we have three
2   #functions defined as follows
3
4   def get_arg1(x):
5       return x*x
6
7   def get_arg2(x):
8       return x*x*x
9
10  def compute(x, y):
11      return x + y
12
13  a = get_arg1(10)
14  b = get_arg2(42)
15  res = compute(a, b)
16
17
18
```

Example: Merge Sort (Demo)

Misc.

- ray.init(num_cpus=<int>)
  - specifies how many parallel processes/workers to use
  - should be less than the number of cores your machine has

- you can also call a remote function sequentially with ._function()

```
ray.remote
def compute():
     …

compute._function() # calls sequentially/locally
                      # similar to a typical function
                   # e.g, like calling compute() if it
              wasn’t
                     #declared as a remote function
```

Misc.

- @ray.remote
  - it is a decorator
  - “By definition, a decorator is a function that takes another function and extends the behavior of the latter function without explicitly modifying it.”
  - further details: https://realpython.com/primer-on-python-decorators/

Ray API

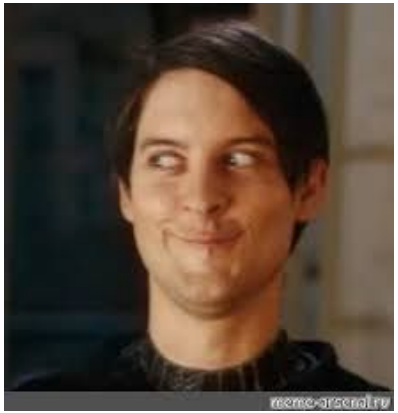
Name	Description
$futures = \textbf{f.remote}(args)$	Execute function f remotely. **f.remote**() can take objects or futures as inputs and returns one or more futures. This is non-blocking.
$objects = \textbf{ray.get}(futures)$	Return the values associated with one or more futures. This is blocking.
$ready\_futures = \textbf{ray.wait}(futures, k, timeout)$	Return the futures whose corresponding tasks have completed as soon as either k have completed or the timeout expires.
$actor = \textbf{Class.remote}(args)$ $futures = actor.\textbf{method.remote}(args)$	Instantiate class *Class* as a remote actor, and return a handle to it. Call a method on the remote actor and return one or more futures. Both are non-blocking.

Table 1: Ray API

[1]https://arxiv.org/pdf/1712.05889.pdf

Parallel Computation Model: Actors

- Tasks / Remote Functions are stateless
  - takes input, computes a output and returns it
  - Doesn’t change any program states
  - “side-effect” free
- Sometimes you need to maintain a “state”
- We can use actors
  - Actors retain state
  - methods of an actor are executed sequentially
  - Hence, to ensure parallelism, Multiple actors have to be designed




MAY THE FORCE
BE WITH YOU.™