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](https://github.com/ray-project/ray)
Installing Ray

- pip install ray

- verify installation

```python
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` parallelly
- What if we want to make compute a remote function too?

```python
# Suppose we have three functions defined as follows

def get_arg1(x):
    return x**x

def get_arg2(x):
    return x**x**x

def compute(x, y):
    return x + y

a = get_arg1(10)
b = get_arg2(42)
res = compute(a, b)
```
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()`

```python
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

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>futures = f.remote(args)</code></td>
<td>Execute function ( f ) remotely. ( f\text{.remote}() ) can take objects or futures as inputs and returns one or more futures. This is non-blocking.</td>
</tr>
<tr>
<td><code>objects = ray.get(futures)</code></td>
<td>Return the values associated with one or more futures. This is blocking.</td>
</tr>
<tr>
<td><code>ready_futures = ray.wait(futures, k, timeout)</code></td>
<td>Return the futures whose corresponding tasks have completed as soon as either ( k ) have completed or the timeout expires.</td>
</tr>
<tr>
<td><code>actor = Class.remote(args)</code></td>
<td>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.</td>
</tr>
</tbody>
</table>

Table 1: Ray API

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.™