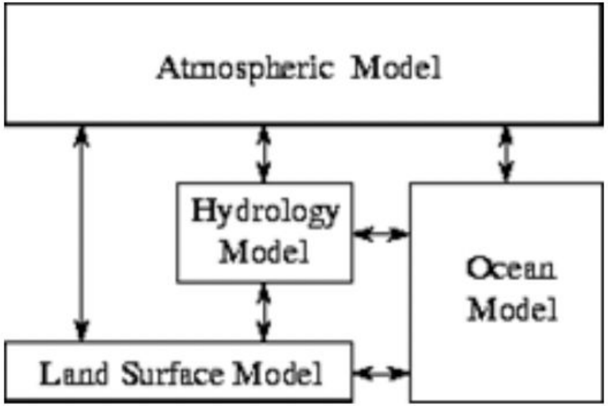
PA2 Discussion Session

DSC 204a, Spring 2025

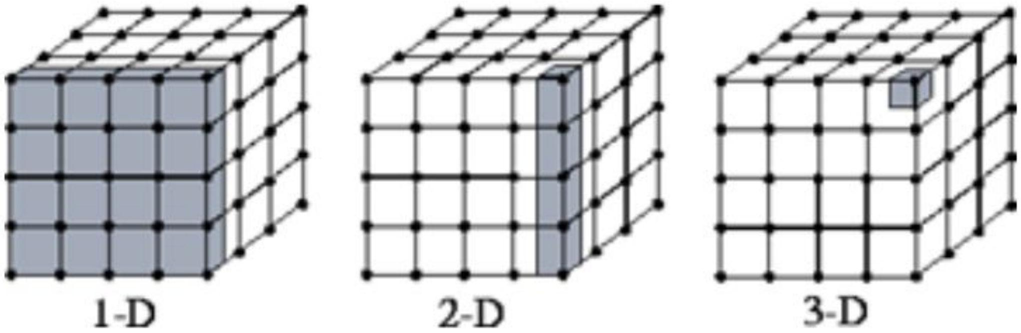
Task Parallelism

- Different tasks or processes run simultaneously on multiple processing units.
- Different operations are performed concurrently on different or the same data.
- Typically in real scenario divide your program into discrete tasks that can be executed independently.
- Efficient resource utilization
- **Challenge:** managing task dependencies, handling shared resources, and balancing workloads effectively.



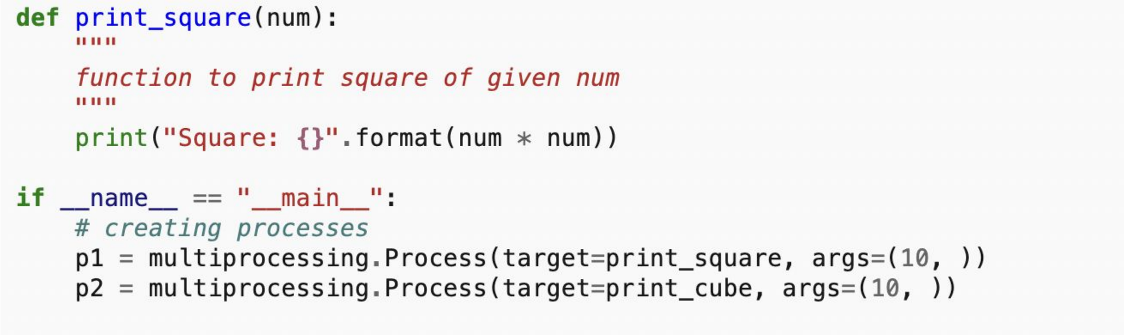
Data Parallelism

- Same operation on different portions of the data.
- divide and conquer
- Simpler because we're repeating the same operation across different data portions.
- Challenges - Synchronization, Data Partition management



Multiprocessing in Python

- Works when:
  a. A computer with more than one central processor.
  b. Or a single computing component with two or more independent actual processing units (called “cores”).
- Step 1 - create a process
  - **target**: the function to be executed by process
  - **args**: the arguments to be passed to the target function



```
def print_square(num):
    """
    function to print square of given num
    """
    print("Square: {}".format(num * num))

if __name__ == "__main__":
    # creating processes
    p1 = multiprocessing.Process(target=print_square, args=(10, ))
    p2 = multiprocessing.Process(target=print_cube, args=(10, ))
```

2. Start method of Process class

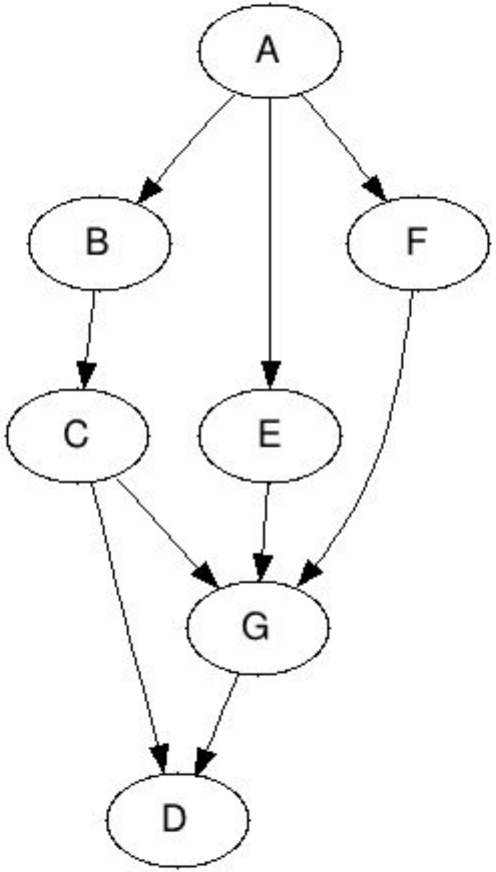
```
p1.start()
p2.start() # concurrently
```

3. Wait for process to finish

```
p1.join() # blocking operation
```

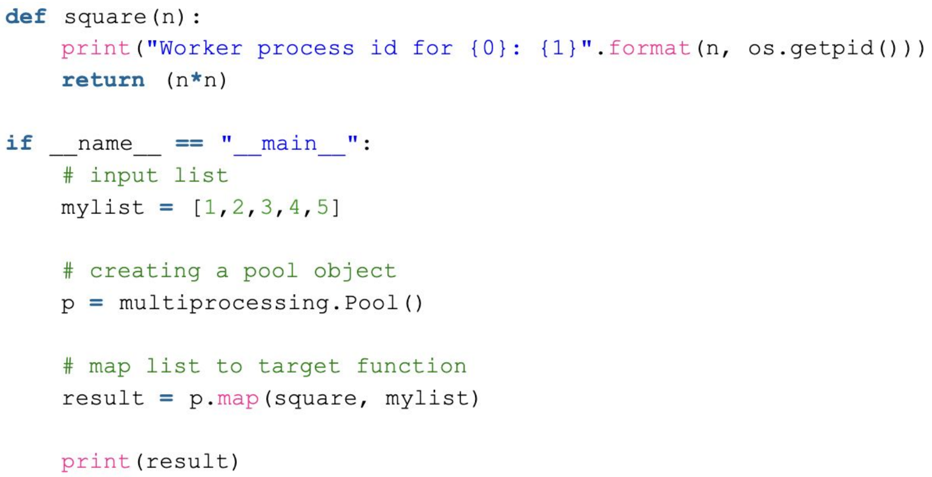
Note:

1. Any updates done are local to the process. Need to create a shared memory for communicating data -> More on this in notebook



Data Parallelism using multiprocessing lib

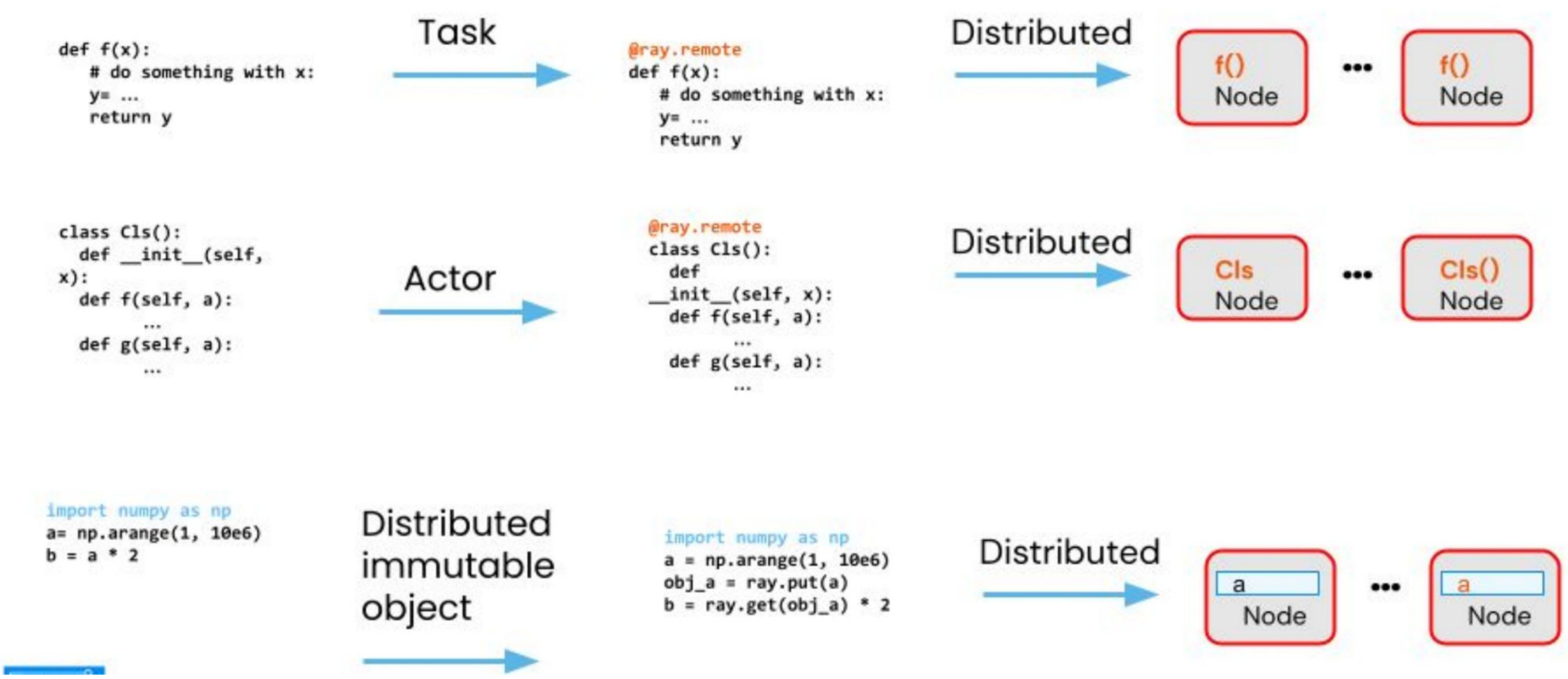
1. Using Pool



```
def square(n):
    print("Worker process id for {0}: {1}".format(n, os.getpid()))
    return (n*n)

if __name__ == "__main__":
    # input list
    mylist = [1,2,3,4,5]

    # creating a pool object
    p = multiprocessing.Pool()

    # map list to target function
    result = p.map(square, mylist)

    print(result)
```

2. Manually dividing the chunks and assigning to a subprocess

Revisiting Ray Core



```
def f(x):
   # do something with x:
   y= ...
   return y
```

```
@ray.remote
def f(x):
   # do something with x:
   y= ...
   return y
```

```
class Cls():
  def __init__(self,
x):
  def f(self, a):
        ...
  def g(self, a):
        ...
```

```
@ray.remote
class Cls():
  def
__init__(self, x):
  def f(self, a):
        ...
  def g(self, a):
        ...
```

```
import numpy as np
a= np.arange(1, 10e6)
b = a * 2
```

```
import numpy as np
a = np.arange(1, 10e6)
obj_a = ray.put(a)
b = ray.get(obj_a) * 2
```

Managing Multi processing

1. How do you handle concurrent updates?

   -> Ray handles the synchronization automatically. No Manual Locks

2. How can we do this in a multi-node setting?

   -> No code change. Same code works in Multi node setting