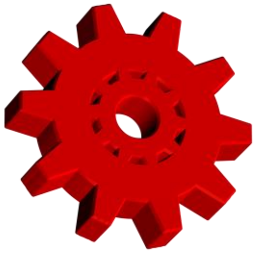
DISTRIBUTED SYSTEM DESIGN

Lab 4

Multiprocessing and MPI

What is a Process?

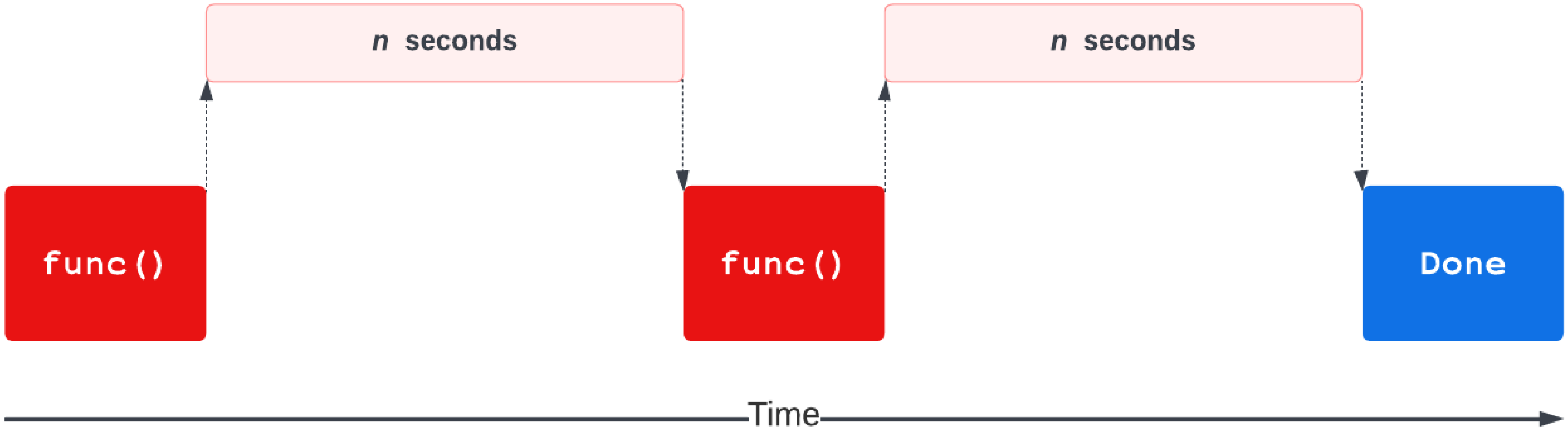
- An **independent** process-of-control
- Process are **Share nothing**
- Processes are **Big**
- Processes run on **multiple cores**
- When to use? **CPU-bound** applications



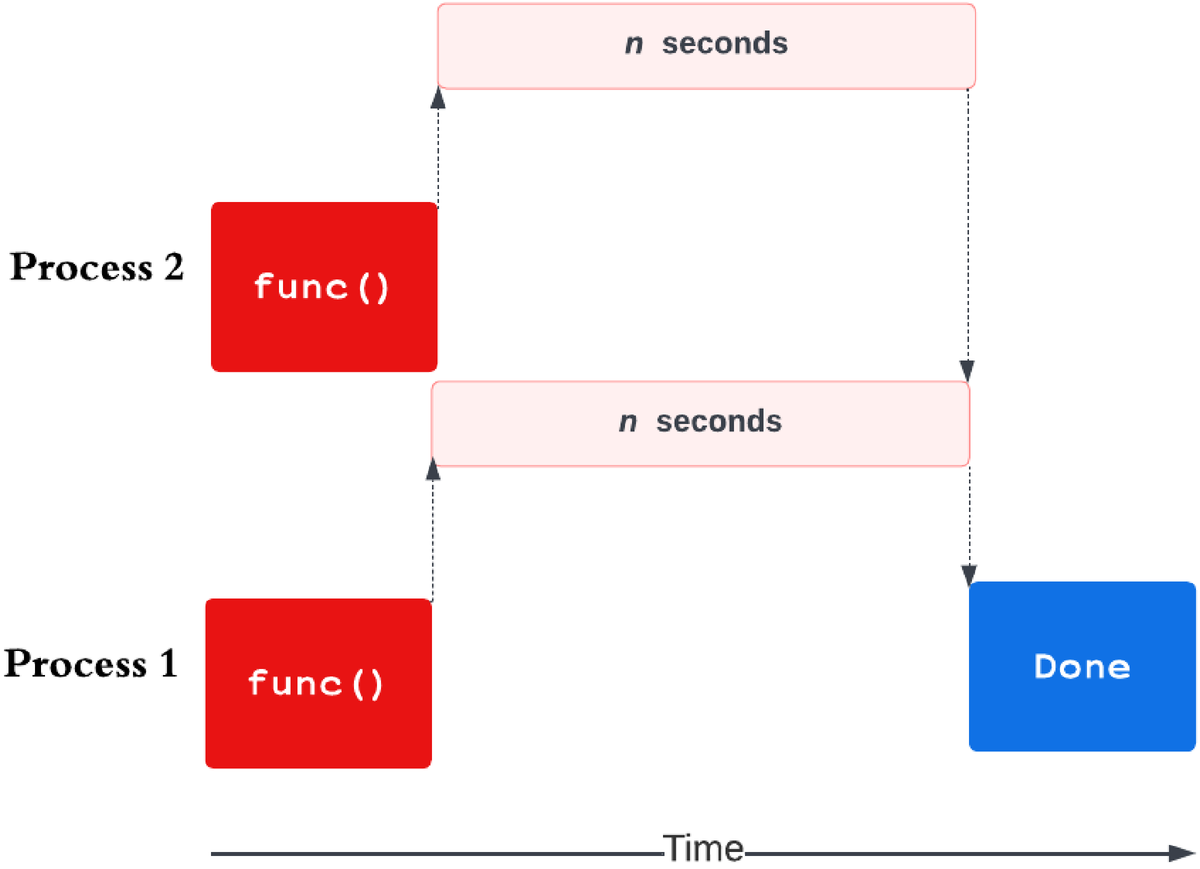
Threads v/s Processes

	Threads	Processes
1.	System calls are not involved	System calls are involved
2.	Context switching is faster	Context switching is slower
3.	Blocking a thread will block entire process	Blocking a process will not block another process
4.	Threads share same copy of code and data	Different processes have different copies of code and data
5.	Interdependent	Independent
6.	I/O bound	CPU bound

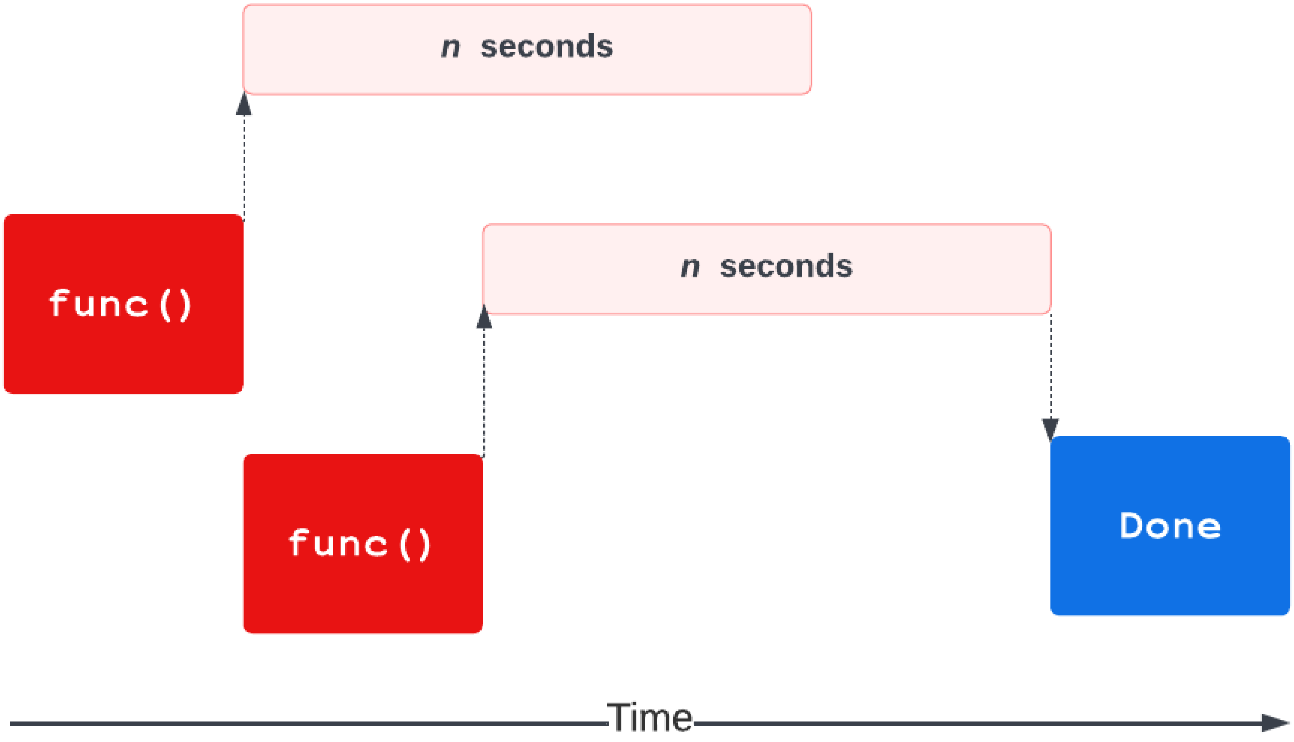
Visualizing execution: serial



Visualizing execution: processes



Visualizing execution: threads



Hello world in Multiprocessing

```
import os
import multiprocessing
from time import time, sleep

def compute():
    print('computing...')
    sleep(1)

def compute_multi_processing():
    print('using multi-processing\nCPU-core(s) available: ‘, os.cpu_count())
    start = time()
    p1 = multiprocessing.Process(target=compute)
    p2 = multiprocessing.Process(target=compute)
    p1.start()
    p2.start()
    p1.join()
    p2.join()
    finished = time()
    print(f'time taken (multi-processing): {round(finished - start, 2)}
second(s)’)

if __name__ == '__main__':
    compute_multi_processing()
```

Handling multiple processes

- Pool class in multiprocessing can handle an enormous number of processes.
- It **allows you to run multiple jobs per process** (due to its ability to queue the jobs)
- The memory is allocated only to the executing processes, unlike the Process class, which allocates memory to all the processes.

```
import time
from multiprocessing import Pool

def sum_square(number):
    s = 0
    for i in range(number):
        s += i * i
    return s

def compute_multiprocessing(numbers):
    print('using multiprocessing')
    start = time.time()
    p = Pool()
    result = p.map(sum_square, numbers)
    p.close()
    p.join()
    finish = time.time()
    print(f'time taken (multiprocessing execution): {round(finish - start, 2)} second(s)')

if __name__ == '__main__':
    n = range(30000)
    compute_multiprocessing(numbers=n)
```

MPI

What is MPI?

- **M**essage **P**assing **I**nterface

- Not a new programming language. MPI is collection of functions and macros, or a library that can be used in programs written in C, C++, Fortran and Python (via mpi4py)

- Language independent communications protocol

- Portable and platform independent

- Various implementations exists (Open MPI, vendor versions)

What is MPI? (cont'd…)

- MPI uses a **statistic**ally allocated group of processes (number is set at the beginning of the program unlike threads)

- Each process is assigned a unique **rank**, from 0 to p-1, where p is the number of processes

- Since processes do not share memory, explicit function calls must be made.

- Process which has data calls a Send function and process which has to receive data calls a Receive function.

Few definitions in MPI

- **COMM**: communication “world” defined by MPI
- **RANK**: an ID number given to each internal process
- **SIZE**: total number of processes allocated
- **BROADCAST**: one-to-many communication
- **SCATTER**: One-to-many data distribution (in chunks)
- **GATHER**: Many-to-one data distribution

Few definitions in MPI (cont'd) …

- **SEND**:
- **RECV**:

point-to-point communicators
“blocking” commands

```
comm.send(obj, dest, tag=0)
comm.recv(source=MPI.ANY_SOURCE,tag=MPI.ANY_TAG,
         status=None)
```

- tag is used as a filter
- dest is the rank of communicator
- source can be rank or a wild card
- status is used to retrieve information about received message

MPI in Python

- **mpi4py** (MPI for Python) provides bindings for MPI in Python

- Object oriented, user-friendly. Automatically determines many required arguments to MPI calls (which are explicitly given when using other languages)

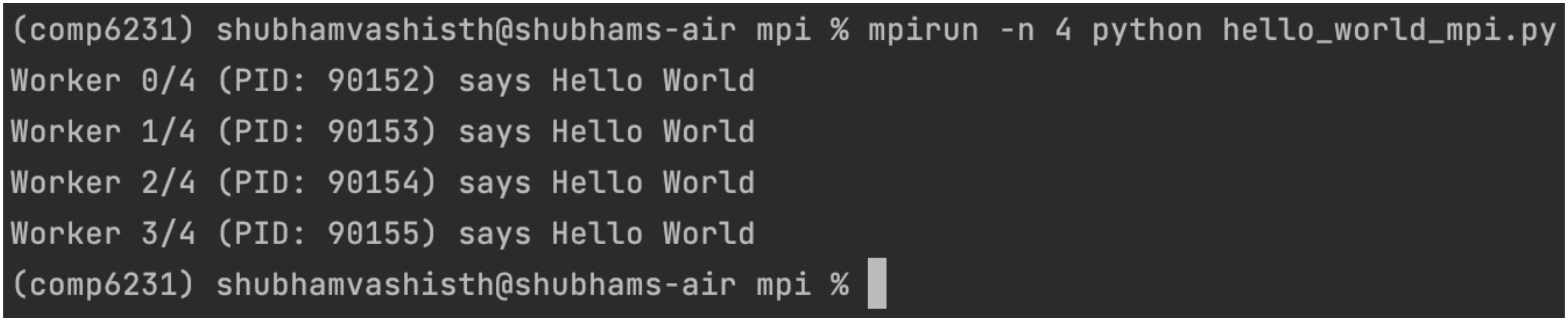
- Docs: https://mpi4py.readthedocs.io/en/stable/

- Installation:

```
conda create -n comp6231 python=3.8 mpi4py numpy pandas jupyter tqdm

conda activate comp6231

conda install -c conda-forge mpi4py openmpi
```

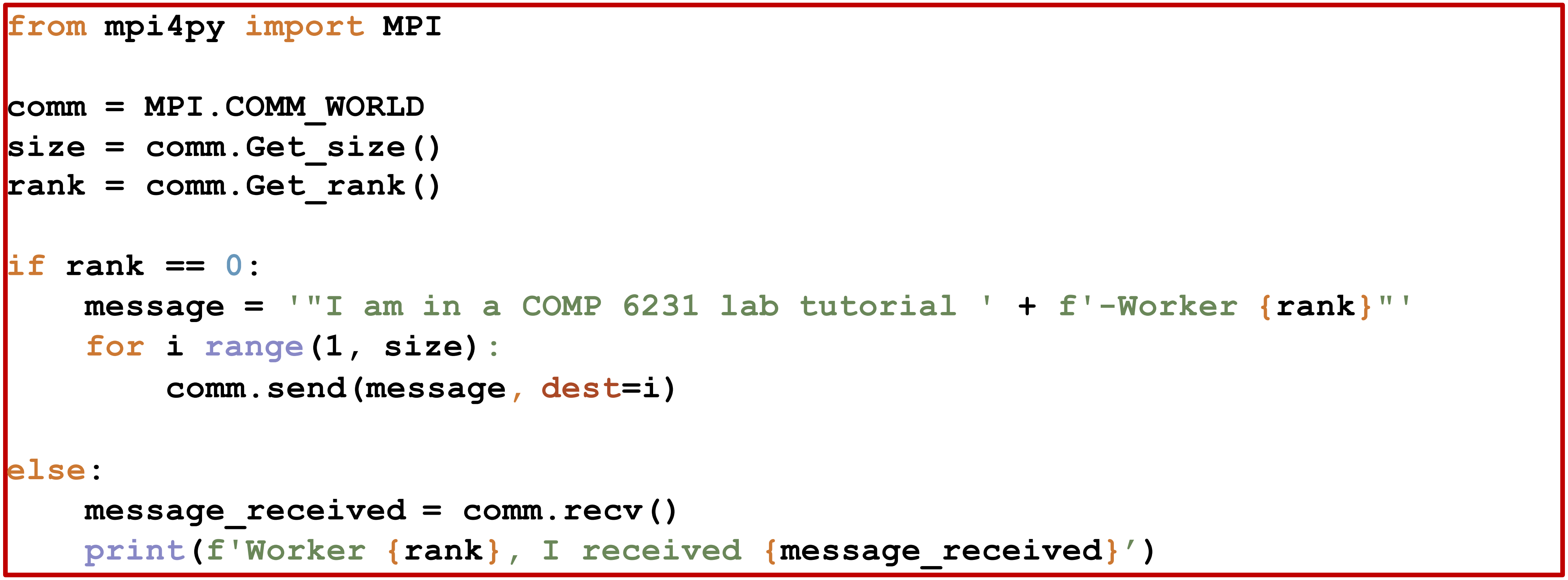
Hello world in MPI

```python
from mpi4py import MPI
import os

comm = MPI.COMM_WORLD   # instantiate communication world
size = comm.Get_size()   # get size of communication world
rank = comm.Get_rank()   # get rank of particular process
PID = os.getpid()

print(f'Worker {rank}/{size} (PID: {PID}) says Hello World')
```

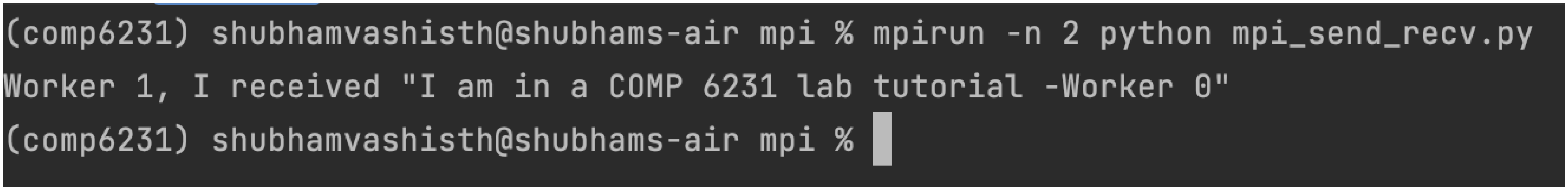


```
(comp6231) shubhamvashisth@shubhams-air mpi % mpirun -n 4 python hello_world_mpi.py
Worker 0/4 (PID: 90152) says Hello World
Worker 1/4 (PID: 90153) says Hello World
Worker 2/4 (PID: 90154) says Hello World
Worker 3/4 (PID: 90155) says Hello World
(comp6231) shubhamvashisth@shubhams-air mpi % 
```

send() recv() in MPI



```python
from mpi4py import MPI

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

if rank == 0:
    message = '"I am in a COMP 6231 lab tutorial ' + f'-Worker {rank}"'
    for i range(1, size):
        comm.send(message, dest=i)

else:
    message_received = comm.recv()
    print(f'Worker {rank}, I received {message_received}’)
```



```
(comp6231) shubhamvashisth@shubhams-air mpi % mpirun -n 2 python mpi_send_recv.py
Worker 1, I received "I am in a COMP 6231 lab tutorial -Worker 0"
(comp6231) shubhamvashisth@shubhams-air mpi %
```

Exercises

1. Run the following:
    I. pandas.ipynb
    II. pandas_multiprocessing,py
    III. hello_world_mpi.py
    IV. mpi_send_recv.py
    V. pandas_mpi.py
2. Run Pandas with multiprocessing
3. Modify *pandas_multiprocessing.py (from exercise)* example to calculate missing years *(NAN)* **serially**
4. Modify *pandas_multiprocessing.py (from exercise)* example to calculate missing years *(NAN)* using **multiprocessing**
5. Modify *pandas_mpi.py (from exercise)* example to calculate missing years *(NAN)* using **MPI**

you should get **410974** missing values for exercise 3, 4 and 5.