

# Parallel Programming with Python


Trung Nguyen, Ph.D.

Research Computing Center

Aug 10, 2023



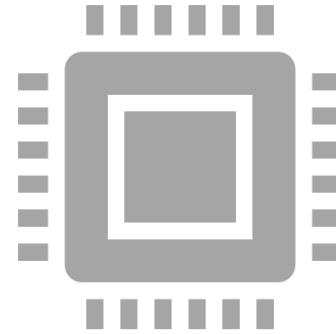
You will  
know

- multithreading and multiprocessing models
  - commonly used strategies for parallelizing a serial Python code
  - popular Python packages for parallelizing your code
- 

# Related workshops



Parallel programming with OpenMP and MPI  
(Debbie Samaddar, RCC)



Python with GPUs  
(with Kris Keipert, NVIDIA)

# RCC Midway Clusters

time for you to log in to Midway3 ...

1. Log in to the login node via SSH, or via ThinLinc
  - `ssh [your-cnetid]@midway3.rcc.uchicago.edu`
2. Clone the github repo for the examples
  - `git clone https://github.com/rcc-uchicago/parallel-python.git`
3. Request an interactive job
  - `sinteractive -N 1 --ntasks-per-node=8 --account=rcc-guest`
4. Load the modules and activate the environment
  - `module load python/anaconda-2021.05 openmpi/4.1.2+gcc-7.4.0`
  - `ulimit -l unlimited`
  - `source activate parallel`

# Why parallelize your code?



Having repeated tasks on different datasets or input params



Having access to enough computing resources



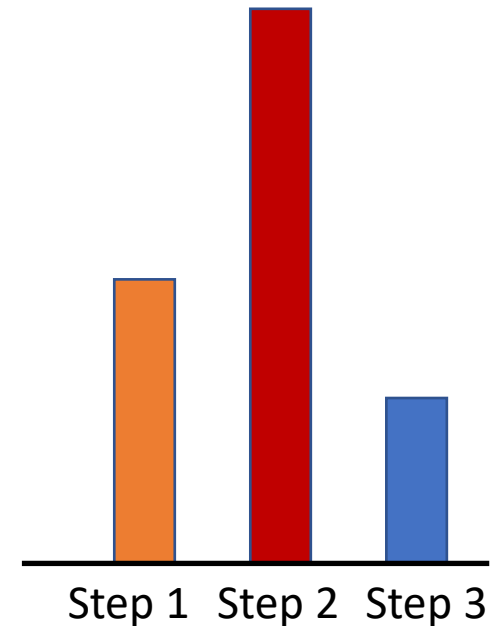
Having limited time until deadlines



Curious?

# Understand your code: How can you parallelize it?

- Identify the bottlenecks in the flow chart of your program – using timers and profilers
  - How often data I/O with hard drive is performed?
  - Data layout: Are data structures arranged to memory access friendly patterns?
  - Computation: Any heavy `for` loops? any external modules/packages calls in the nested inner loop?
- Can the bottleneck(s) be parallelized?
  - or, can the workload be distributed among processing units, aka “workers”?



# Profiling tools for (serial) Python codes

Commonly used Python modules:

- **time**
- **pyinstrument**
- **cProfile**

Note: Fine-grained profiling may distort actual performance.

# Profiling a python code segment with time (Exercise 1)

`python profiling.py`

```
import time
```

```
start_time = time.time()
```

```
# your code segment
```

```
elapsed_time = time.time() - start_time
```

NOTE: `time.perf_count()` is recommended for high-resolution timings (for short events) as it relies on the CPU clock vs. `time.time()` using OS `time()` function. For long events, the diff between the two becomes negligible.



# Profiling a python code with pyinstrument

python profiling.py

```
from pyinstrument import Profiler
```

```
profiler = Profiler()
```

```
profiler.start()
```

```
# your code segment
```

```
profiler.stop()
```

```
profiler.print()
```

# Amdahl's Law

- Theoretical speedup is limited by the contribution of the non-parallelized parts

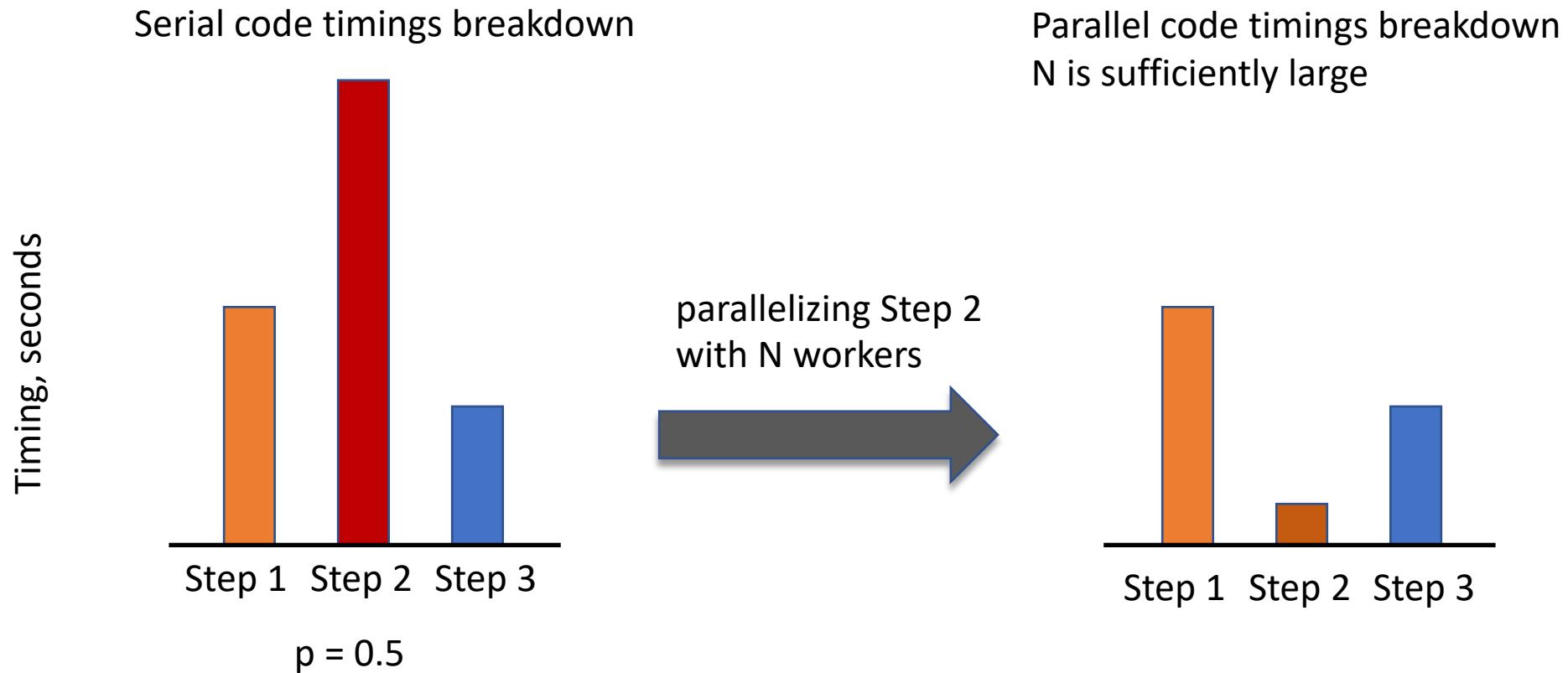
$$S(N) = \frac{1}{(1 - p) + \frac{p}{N}} \qquad \lim_{N \rightarrow \infty} S(N) = \frac{1}{1 - p}$$

S = theoretical speedup with N processing units

p = time percentage of the parallelized parts

N = theoretical speedup gained for the task with N processing units

# Amdahl's Law



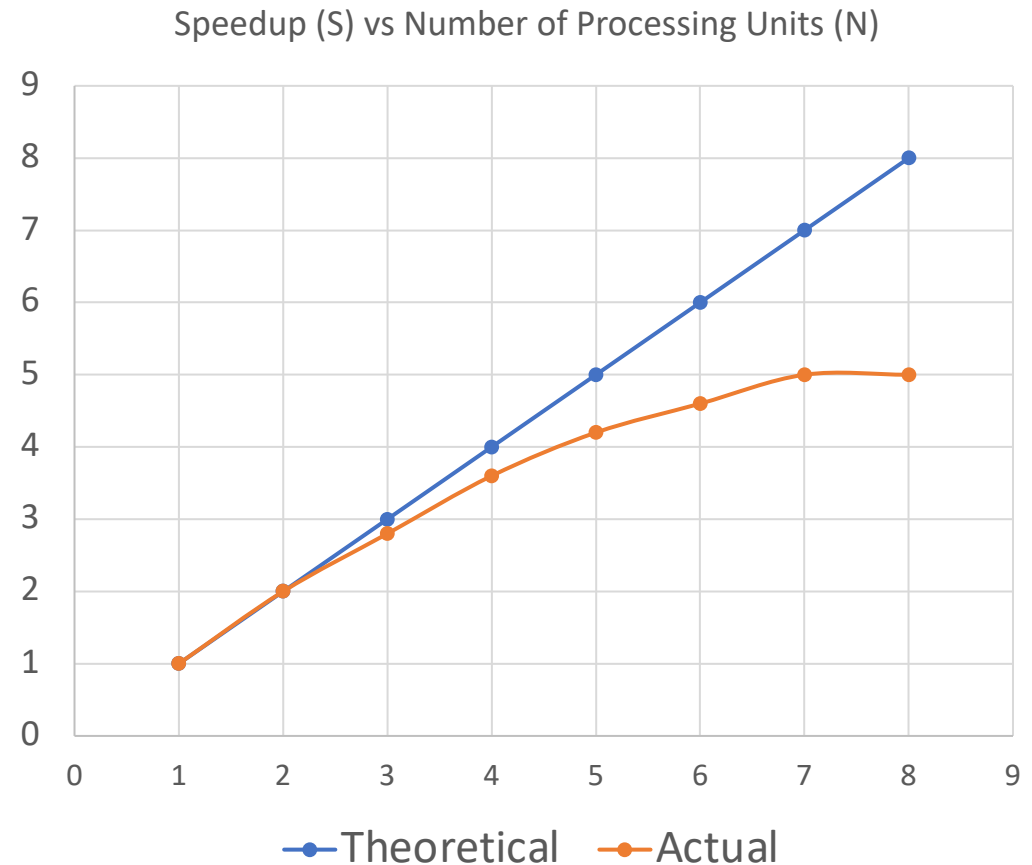
$$\lim_{N \rightarrow \infty} S(N) = \frac{1}{1 - p} = \frac{1}{1 - 0.5} = ?$$

# Parallelization strategies: Ways to distribute workload among workers

- Process level:
  - embarrassingly parallel (multiprocessing)
  - map/reduce (multiprocessing)
- Data level:
  - data decomposition (multiprocessing, or multithreading)
- Instruction level:
  - just-in-time compilation for multi-core CPU or GPU targets (multithreading)

# Parallelization performance: Strong scaling

- Parallelization introduces overhead:
  - communication/sync between workers
- Strong scaling analysis show time to solution, or speedup, as a function of number of workers (threads, or processes)
  - linear scaling:  $S_{\text{linear}} = t_1/t_N = N$
  - computation vs. communication break-even point
    - $P^*$  where speedup stops increasing with  $P$



# Know the hardware where your code is running

to decide which parallelization strategies would be optimal

- Single-node configuration
  - Multi-core CPUs: how many physical CPU cores? hardware threading off/on? ([lscpu](#), [/etc/cpuinfo](#)) supporting vectorization (avx512)? L1 cache size?
  - Memory
  - GPUs attached? Types? Memory size and bandwidth?
  - Storage
- Multiple-node configuration
  - Interconnect bandwidth: Infiniband?

# Midway3 Compute Nodes

to decide which parallelization strategies would be optimal

- Show the partition information:  
    `scontrol show partition broadwl`  
    `sinfo -p broadwl`
- Show the node information: `scontrol show node midway3-xxxx`
  - How many physical CPU cores?
  - Memory
  - GPUs attached?

# Processes and threads

A process is a program managed by the operating system (OS)

- operating on separate memory spaces
- performing a series of executions, or
- consisting of an infinite loop waiting for OS events (GUI programs)
- able to spawn/fork child processes

A thread is a sub-process created and managed by a process

- sharing the memory space with peer threads in the same process
- able to spawn/fork child threads

By default, Python is a program (an interpreter) with a single thread

```
python your_script.py
```



# Multithreading and Multiprocessing Programming Models

- **Multithreading:**

- the main thread spawns/forks multiple threads
- each thread access to the data in the shared memory pool
- each thread executes the instructions, may sync with other threads
- threads are terminated (joined) when done

- **Multiprocessing:**

- create child processes, or launch peer processes (mpirun or srun)
- each process allocates data in its memory space
- each process executes the instructions, may send/receive among the processes
- processes are closed (finalized) when done

# Multiprocessing Programming Models

- **Spawning vs Forking**

- each **forked** Python process inherits all the variables and states, and modules of the parent Python process, progressing independently from the forking point.
- each **spawned** process is a fresh Python process, the modules are reimported, new copies of the variables are created.

Action	fork	spawn
Create new PID for processes	yes	yes
Module-level variables and functions present	yes	yes
Each child process calls <code>plot_function</code> on multiple pool args	yes	yes
Child processes independently track variable state	yes	yes
Import module at start of each child process	no	yes
Variables have same id as in parent process	yes	no
Child process gets variables defined in <code>name == main</code> block	yes	no
Parent process variables are updated from child process state	no	no
Threads from parent process run in child processes	no	no
Threads from parent process modify child variables	no	no

# Process-based parallelization with multiprocessing Module (Exercise 2)

test-multiprocessing.py

```
from multiprocessing import Process

def func(input_args):
    (input_args)

p1 = Process(target=func, args=(args1,))
p2 = Process(target=func, args=(args2,))
p1.start()
p2.start()
p1.join()
p2.join()
```

`fork` is the default mode of creating a new process with the `multiprocessing` module.

# Communicate between processes with Queue (Exercise 3 )

python map-reduce-pi.py

- Each proc puts the result into a queue, and get the result

```
from multiprocessing import Process, Queue
```

```
def func(my_queue, input_args):  
    my_queue.put(result)
```

```
q = Queue()  
p = Process(target=func, args=(q, args,))  
p.start()  
res = q.get()  
p.join()
```

# Map and reduce

```
python map-reduce-pi.py
```

- Mapping functions to process-owned data
- Reducing (tallying) the results from the queues

# Map and reduce with multiprocessing Pool

python map-reduce-pi.py

```
from multiprocessing import Pool

def func(input_args):
    return result

with Pool(Ncores) as pool:
    results = pool.map(func, [args[i] for i in range(Ncores)])
np.sum(results)/Ncores
```

# Map and reduce with multiprocessing Pool

python map-reduce-pi.py

```
from multiprocessing import Pool

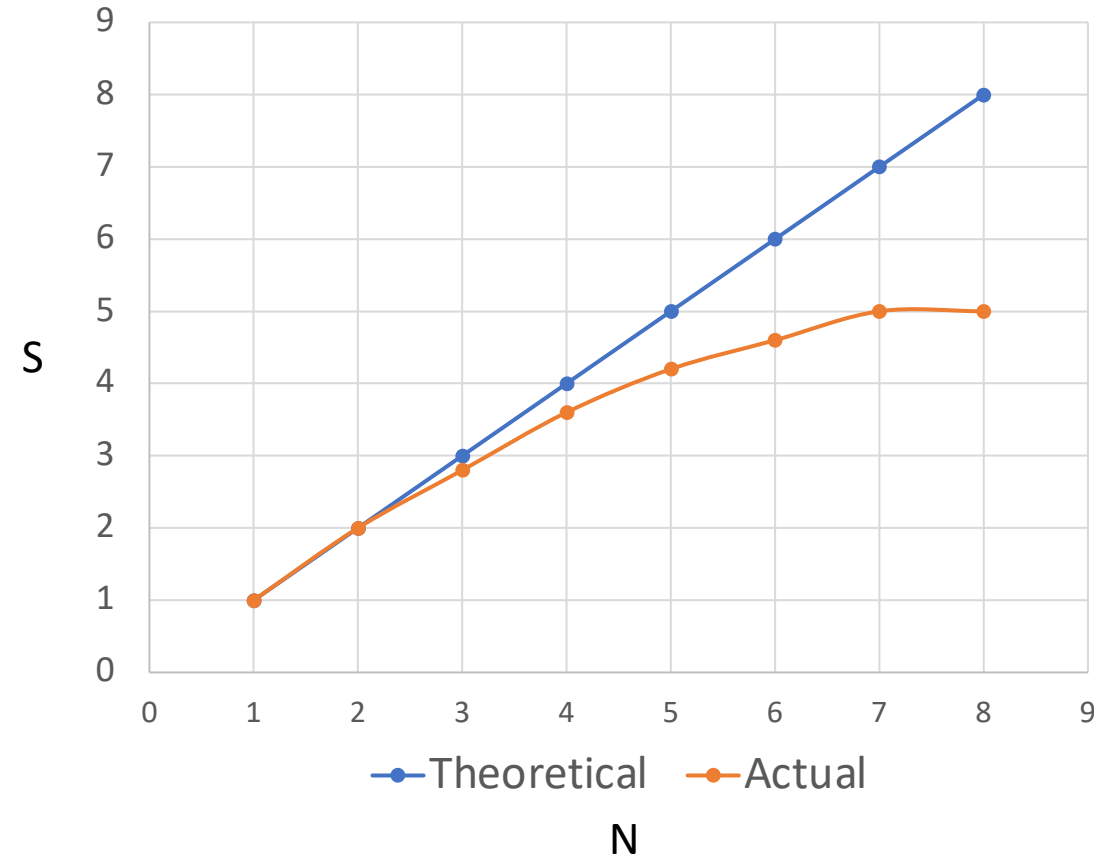
def func(input_args):
    return result

with Pool(Ncores) as pool:
    results = pool.map(func, [args[i] for i in range(Ncores)])
np.sum(results)/Ncores
```

# Parallelization efficiency: Strong scaling (Exercise 4)

`python performance.py`

Speedup vs Number of Processing Units



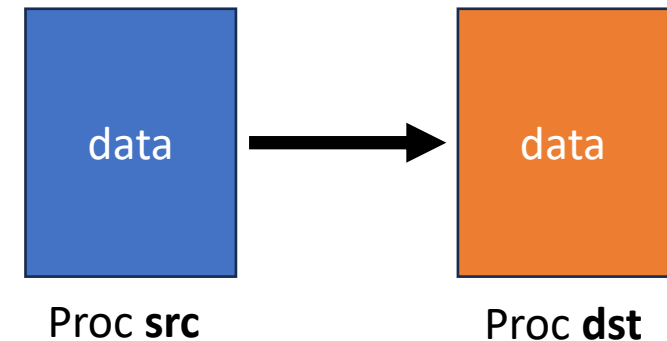


# Communication between processes: Message Passing Interface (MPI) Basics

- Message Passing Interface (MPI) is a specification (programming model) for exchange data between processes
  - communications: point-to-point, collective (one-to-all, all-to-all), one-sided
  - C/C++/Fortran bindings
  - Google search/ChatGPT: MPI tutorials, examples
- Different implementations (vendors): OpenMPI, Intel MPI and MPICH
  - [module avail openmpi](#)
  - [module avail intelmpi](#)
- Compile your C/C++/Fortran codes with wrappers like mpicc, mpicxx, and mpifort

# Communication between processes: Message Passing Interface (MPI) Basics

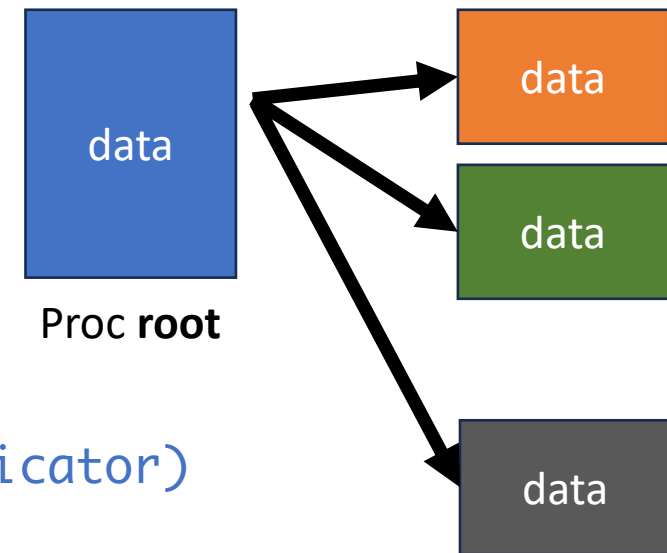
- Point-to-point communications: Send/Receive data between 2 procs
  - Non-blocking vs Blocking operations



```
if (rank == src)
    MPI_Send(&data, count, MPI_DOUBLE, dst, tag, communicator)
else if (rank == dst)
    MPI_Recv(&data, count, MPI_DOUBLE, src, tag, communicator, &status)
```

# Communication between processes: Message Passing Interface (MPI) Basics

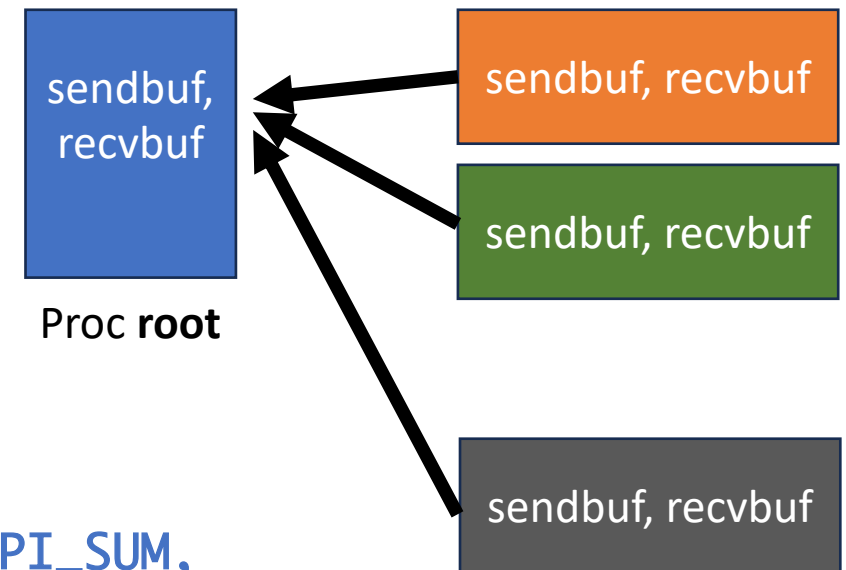
- Collective communications: Broadcast from a proc to other procs



`MPI_Bcast(&data, count, MPI_DOUBLE, root, communicator)`

# Communication between processes: Message Passing Interface (MPI) Basics

- Collective communications: Reduce (tally) from other procs to a proc



```
MPI_Reduce(&sendbuf, &recvbuf, count, MPI_DOUBLE, MPI_SUM,  
          root, communicator)
```

# Process-level parallelization: mpi4py Module

- work as a wrapper for an underlying Message Passing Interface (MPI) library (OpenMPI, MPICH or Intel MPI)
- mpirun launches multiple Python instances

```
from mpi4py import MPI
```

```
comm = MPI.COMM_WORLD
```

```
nprocs = comm.Get_size()
```

```
my_rank = comm.Get_rank()
```

```
func(input_args, my_rank, nprocs)
```

# Communication between processes: MPI binding (Exercise 5)

```
mpirun -np 4 python mpi-comm-ops.py
```

```
from mpi4py import MPI
```

```
comm = MPI.COMM_WORLD
```

```
nprocs = comm.Get_size()
```

```
my_rank = comm.Get_rank()
```

```
func(input_args, my_rank, nprocs)
```

Midway3: ulimit -l unlimited (if getting errors with UCX workers init)

# Communication between processes: MPI binding (Exercise 5)

```
mpirun -np 4 python mpi-comm-ops.py
```

```
# non-blocking send, and receive
if rank == 0:
    data = {'a': 7, 'b': 3.14}
    comm.isend(data, dest=1, tag=11)
elif rank == 1:
    req = comm.irecv(source=0, tag=11)
    data = req.wait()
    print(data)
```

- Point-to-point communication
  - blocking: MPI\_Send/MPI\_Recv
  - non-blocking: MPI\_Isend/MPI\_Irecv
  - MPI\_Wait

Midway3: ulimit -l unlimited (if getting errors with UCX workers init)

# Communication between processes: MPI binding (Exercise 5)

```
mpirun -np 4 python mpi-comm-ops.py
```

```
# Bcast
# allocate array properly on all procs
if rank == 0:
    data = np.arange(100, dtype='i')
else:
    data = np.empty(100, dtype='i')

comm.Bcast(data, root=0)
for i in range(100):
    assert data[i] == i
```

- Collective communication
  - MPI\_Gather/MPI\_Reduce
  - MPI\_Bcast/MPI\_Scatter



# DIY: Calculating $\pi$ with mpi4py

10 minutes

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
nprocs = comm.Get_size()
my_rank = comm.Get_rank()

local_result = calculate_pi()

comm.Allreduce(local_result, result, op=MPI.SUM)
```

# Map/reduce with mpi4py MPIPoolExecutor

analogous to `multiprocessing Pool`

```
from mpi4py.futures import MPIPoolExecutor

def func(args):
    # do smth

with MPIPoolExecutor() as executor:
    input_args_list = (input_args[i] for i in range(Nprocs))
    result = executor.map(func, input_args_list)
```

Your homework assignment!

# Data-level parallelization: multithreading

- Multithreading is generally prohibited by the Global Interpreter Lock (GIL) used by Python
  - to avoid write conflicts
  - to prevent leaked memory (object mem allocation and release)
- Module `multithreading` is still useful for launching I/O bound tasks concurrently

# Comparing I/O bound vs CPU-bound tasks: multithreading (Exercise 6)

```
python test-multithreading.py
```

```
from threading import Thread
```

```
def func(input_args):  
    (input_args)
```

```
t1 = Thread(target=func, args=(args1,))
```

```
t2 = Thread(target=func, args=(args2,))
```

```
t1.start()
```

```
t2.start()
```

```
t1.join()
```

```
t2.join()
```

# Common issues and Debugging Python codes

- Deadlocks during process communications:
  - Send/Recv
  - Bcast
- Write conflicts to shared variables
- Debugging tools
  - **print** on selected ranks
  - **pdb**

```
python -m pdb myscript.py
```

# Practicing with some bugs...

- Deadlocks during process communications ([mpi-comm-ops.py](#))
  - Send/Recv: unmatched src/dst ranks
  - Bcast: called from some proc(s)
- Write conflicts to shared variables
- Debugging tools
  - **print** on selected ranks
  - [pdb](#)

```
python -m pdb myscript.py
```

# What about Python with GPU?

- Offload computation to the GPU: Single-Instruction Multiple-Threads model
- Drop-in options: **numba**, **cupy**

---

# Summary

---

- Profiling and Amdahl's law
- Multiprocessing and multithreading models
- Popular Python modules





# Questions?

[help@rcc.uchicago.edu](mailto:help@rcc.uchicago.edu)  
[ndtrung@uchicago.edu](mailto:ndtrung@uchicago.edu)