

# Parallel Programming with Python


Trung Nguyen, Ph.D.

Research Computing Center

May 23, 2023



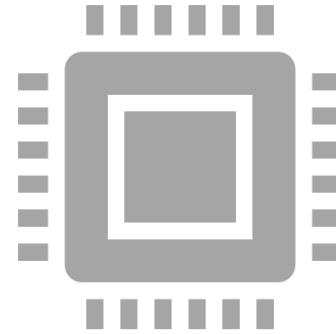
You will  
know

- 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 ...

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

# Why parallelize your code?



Having repeated tasks on different datasets or input params



Having access to enough computing resources



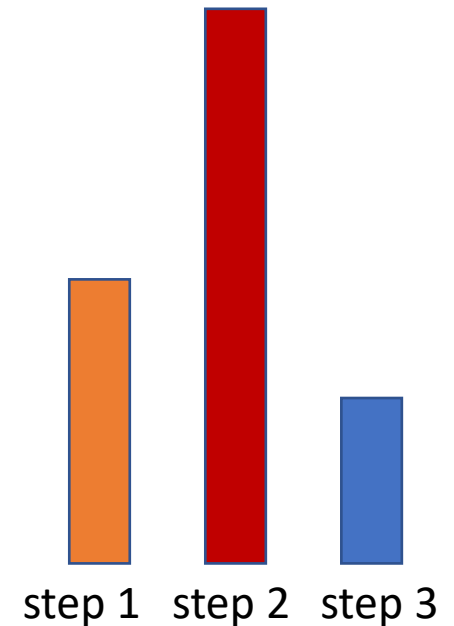
Having limited time until deadlines



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”?



# Amdahl's Law

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

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

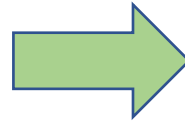
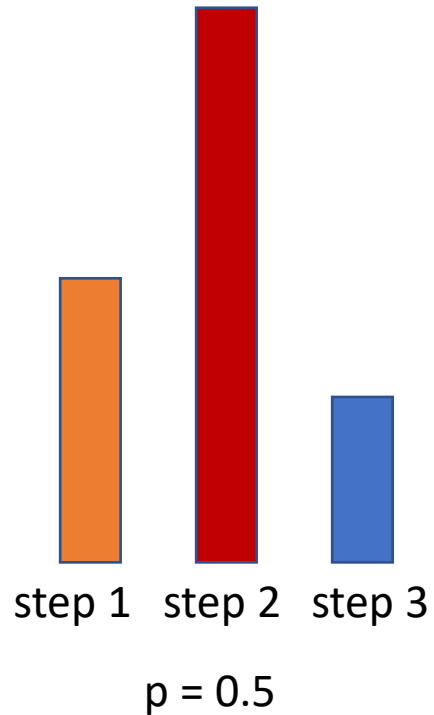
S = theoretical speedup with N processing units

p = time percentage of the parallelized parts

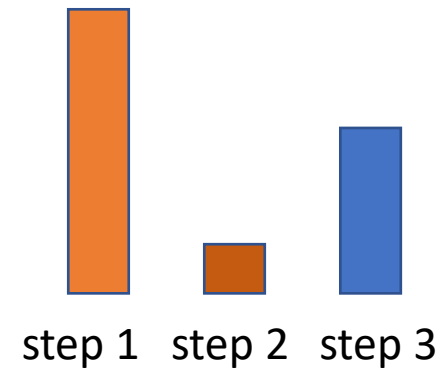
N = theoretical speedup gained for the task with N processing units

# Amdahl's Law

Serial code timings breakdown



Parallel code timings breakdown  
N is sufficiently large (e.g., at breakeven)



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

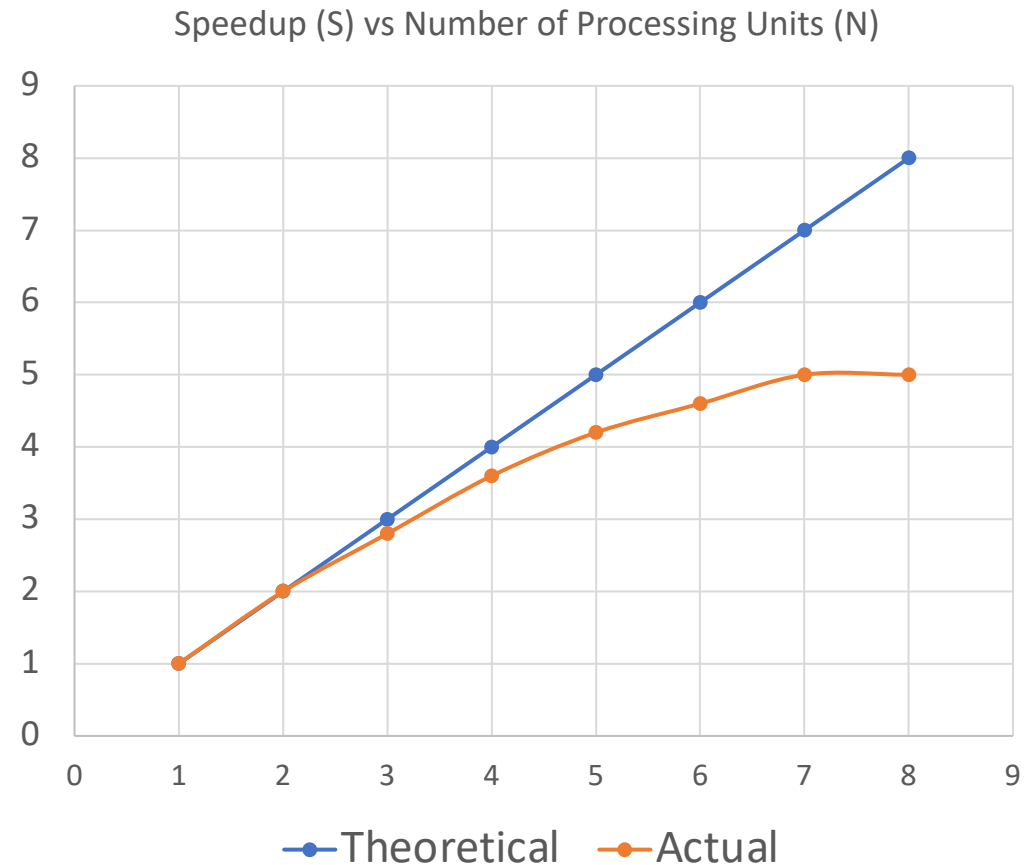


# Parallelization strategies: Ways to distribute workload among workers

- 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  $P$



# 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

# 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? 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?

# Profiling tools for (serial) Python codes

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

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
```



# Profiling a python code with pyinstrument

python profiling.py

```
from pyinstrument import Profiler
```

```
profiler = Profiler()
```

```
profiler.start()
```

```
# your code segment
```

```
profiler.stop()
```

```
profiler.print()
```

# 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()
```

# 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 (tally) 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

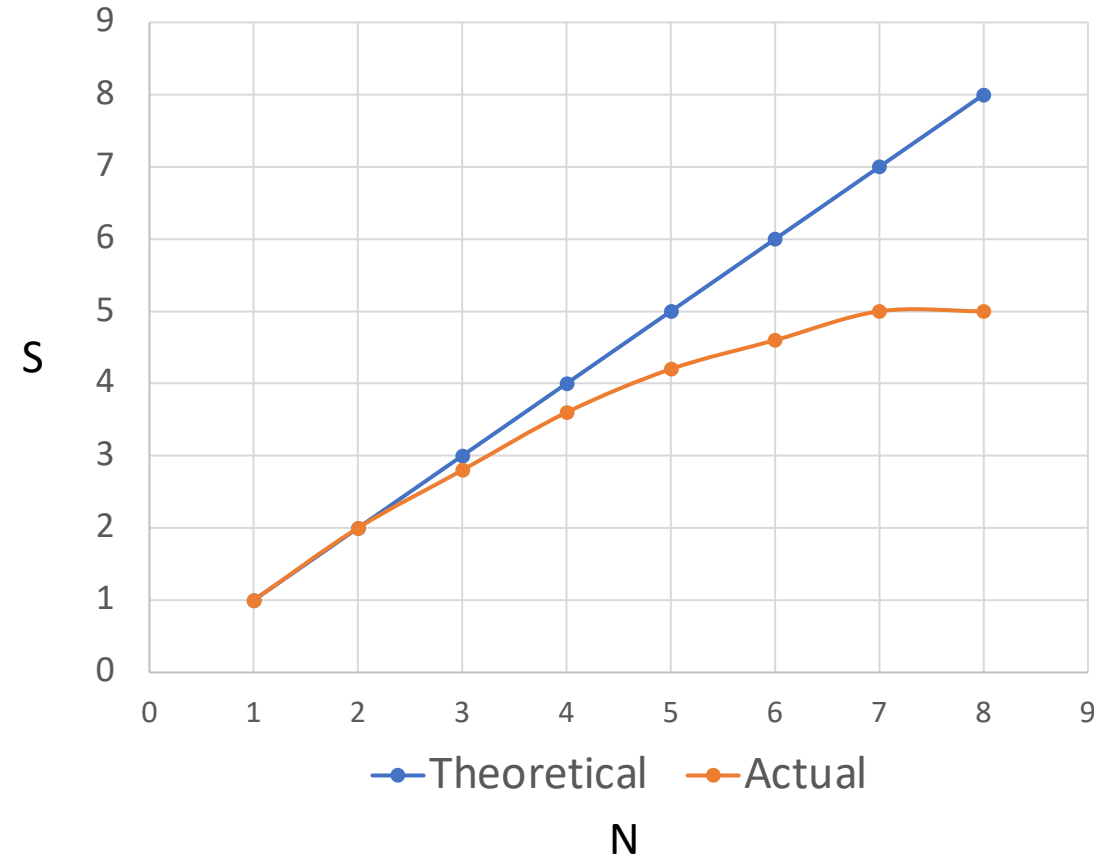
def func(input_args):
    return result

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



# Process-level parallelization: mpi4py Module

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

```
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)
```

Parallel programming with MPI workshop:

- Process initialization/finalization
- Point-to-point communication
  - MPI\_Send/MPI\_Recv
  - MPI\_Wait
- Collective communication
  - MPI\_Gather/MPI\_Reduce
  - MPI\_Bcast/MPI\_Scatter

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

# DIY: Calculate $\pi$ 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(result_local, result, op=MPI.SUM)
```

# Map/reduce with mpi4py MPIPoolExecutor

analogous to `multiprocessing Pool`

```
from mpi4py.futures import MPIPoolExecutor
```

```
def func(input_args):  
    # do smth
```

```
with MPIPoolExecutor() as executor:  
    iterable =  
    result = executor.map(func, input_args)
```

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 useful for I/O bound tasks

# 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)