UTSA
Tech Solutions

An Introduction to Python Parallelization

Zhiwei Wang, Ph.D.
Research Computing Support Group
University of Texas at San Antonio

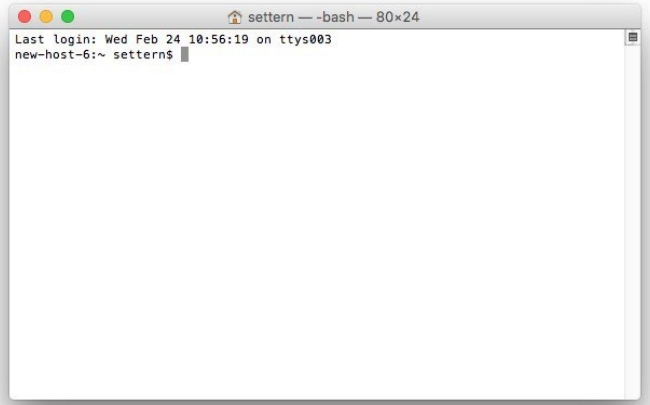
Today's Agenda:

- Introduction to Multitasking
- Understanding the Limitations of Multithreading in Python
- Exploring True Parallelism with Python Multiprocessing
- Scaling Up with MPI for Python
- Leveraging Built-in Parallelism in High-Level Data Libraries (Cupy, NumPy, TensorFlow, PyTorch)
- Common job configuration mistakes that prevent proper scaling

How to access Arc

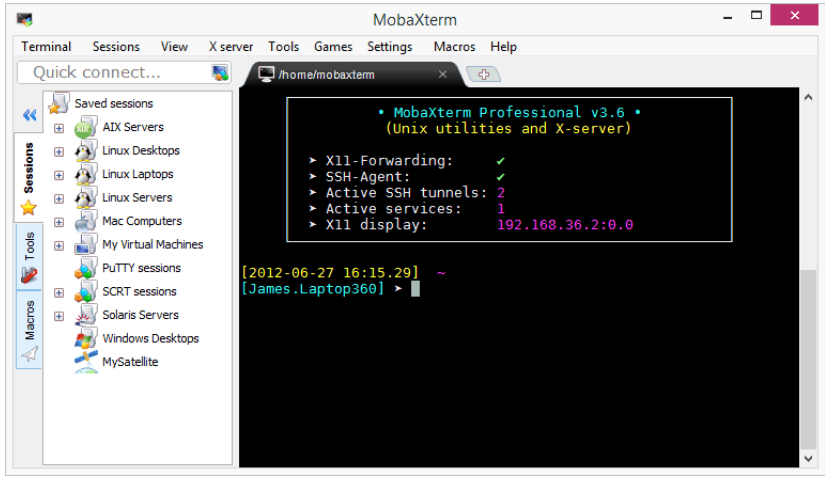
From Linux and Mac

ssh abc123@arc.utsa.edu
ssh –X abc123@arc.utsa.edu



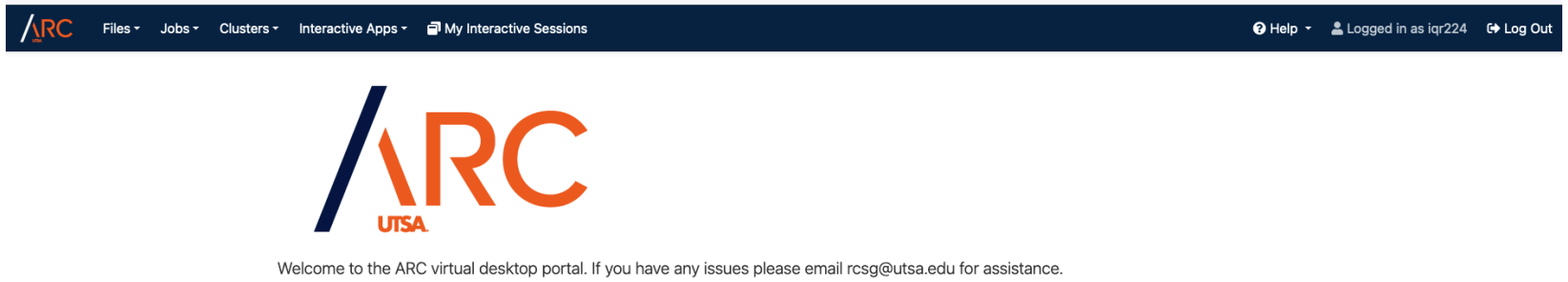
From Windows

Install a SSH client program, such as MobaXterm.



How to access Arc

From Web Portal : portal.arc.utsa.edu



Python Parallelization Techniques

- **Multi-tasking**

Concurrent execution of multiple tasks on the same node becomes essential when parallelizing an application is not feasible. In this scenario, each instance of the task processes a specific portion of the data.

- **Multi-threading**

Multithreading employs threads to achieve parallelization. A thread represents a sequence of instructions within a program capable of running independently of other code. However, in Python, due to the Global Interpreter Lock (GIL), multiple threads will never execute simultaneously.

- **Multi-processing**

Multi-processing enables a Python program to utilize multiple cores and achieves true parallelism on a single node.

- **MPI**

MPI for Python is an implementation of the Message Passing Interface for Python programming language. It enables a Python program to run across multiple nodes in a cluster environment.

- **High level libraries**

Some operations of higher level libraries such as numpy if configured properly.

- **GPU acceleration**

Some operations of higher level libraries such as Cupy, Tensorflow, and PyTorch can automatically utilize GPU if configured properly.

Python Virtual Enviroment

module load anaconda3

conda create -n "myenv"

conda activate myenv

conda install cupy numpy

module load gcc openmpi

pip install mpi4py

conda deactivate

Sample Code

- All sample codes for this training are located at /work/training/parallel-python
- Use the following command to make a copy of the sample code in your directory.

$ cp -r /work/trainings/parallel-python/ .

Multi-Tasking

- Concurrent execution of multiple tasks on the same node involves each instance of the task processing a portion of the data.
- Accelerates the overall process, conserves valuable system resources, and optimizes single node utilization.
- Enables us to execute more tasks, circumventing the ten-job restriction on Arc.

multi-task.py

```
import sys
print(f'\nProcessing data
protion:{sys.argv[1]}')
#.....
#processing the selected portion of the data
```

multi-task.job

```
#!/bin/bash
#SBATCH -J multi-task
#SBATCH -p compute1
#SBATCH -t 00:02:00
#SBATCH -N 1
#SBATCH -n 1
#SBATCH -c 40

module load anaconda3
python multi-task.py 1 &
python multi-task.py 2 &
python multi-task.py 3 &
python multi-task.py 4 &
python multi-task.py 5
```

Calculate Integral in Python

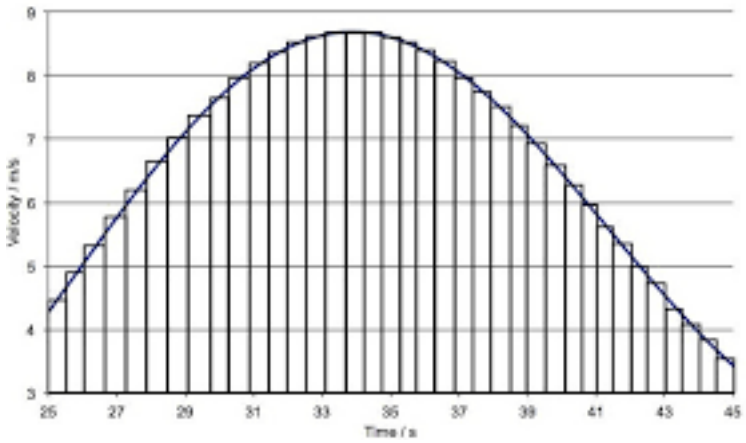
```
def f(x):
    return x * x

def compute_integral(a, b, n):
    h = (b - a) / n
    x = a
    integral = 0
    for i in range(n):
        integral = integral + ((f(x) + f(x + h)) / 2.0) * h
        x = x + h

    return integral


integral = compute_integral(0, 100, 100000000)

print(f"integral is {integral}")
```



Multithreading in Python

```
import time
import multiprocessing
import concurrent.futures
start = time.perf_counter()
def f(x):
    return x*x
def compute_integral(a, b, n):
    h = (b-a)/n
    x = a
    integral = 0
    for i in range(n):
        integral = integral + ((f(x) + f(x +h))/2.0)*h
        x = x + h
    return integral
processes = []
a = 0
b = 100
n = 100000000
h = (b-a)/n
N = 8
with concurrent.futures.ThreadPoolExecutor() as executor:
for i in range(N):
        n1 = int(n/N)
a1 = a + i*n1*h
        b1 = a + (i+1)*n1*h
        p = executor.submit(compute_integral, a1, b1, n1)
        processes.append(p)

    final = 0
    for p in processes:
        final = final + p.result()
    print(f'final integral is {final}\n')


finish = time.perf_counter()

run_time = finish - start

print(f'finish in {round(run_time, 3)} second')
```

Multi-processing in Python

```
with concurrent.futures.ProcessPoolExecutor() as executor:
    for i in range(N):
        n1 = int(n/N)
        a1 = a + i*n1*h
        b1 = a + (i+1)*n1*h
        p = executor.submit(compute_integral, a1, b1, n1)
        processes.append(p)
    final = 0
    for p in processes:
        final = final + p.result()
    print(f'final integral is {final}\n')
finish = time.perf_counter()
run_time = finish - start
print(f'finish in {round(run_time, 3)} second’)
```

Run Multi-threading/Multi-processing code on Arc

- Interactively execute a Python program

$srun -p compute1 -n 1 -t 02:00:00 --cpus-per-task=40--pty bash

$module load anaconda3

$python integral-mprocessing.py

- Submit a batch job (on a login node)

$sbatch mprocessing.job

mprocessing.job

```
#!/bin/bash
#SBATCH -J multi-task
#SBATCH -p compute1
#SBATCH -t 00:02:00
#SBATCH -N 1
#SBATCH -n 1
#SBATCH -c 40

module load anaconda3
python integral-mprocessing.py
```

MPI for Python

```
a = 0
b = 100
n = 100000000
h = (b-a)/n
if rank == 0:
    partial = 0
    integral = 0
    for i in range(1, size):
        partial = comm.recv()
        integral = integral + partial
        print(f"master received value {partial} from process
 {i}")
    finish = time.perf_counter()
    print(f'the integral is {integral}')
    print(f'finish in {round(finish - start, 3)} second')
```

```
else:
    print(f"process {rank} starts")
    n1 = (int)(n/(size -1))
    a1 = a + (rank -1)*n1*h
    b1 = a + rank*n1*h
    partial = compute_integral(a1, b1, n1)
    comm.send(partial, dest=0)
```

Submit Python MPI Batch Jobs on Arc

- To submit a batch job on a login node:

$sbatch mpi.job

- To check the status of a submitted job:

$squeue -u abc123

- Delete a job from the queue

$ scancel jobID

Must be done on a login node

MPI.job

```
#!/bin/bash
#SBATCH --job-name=integral_mpi
#SBATCH --output=out.ext
#SBATCH --mail-type=ALL
#SBATCH --mail-user=zhiwei.wang@utsa.edu
#SBATCH --ntasks=80
#SBATCH --nodes=2

# Load one of these
module load openmpi
module load anaconda3
conda activate parallel
mpirun -n $SLURM_NTASKS python3 integral-mpi.py
```

About –c and –n

- -n 80 or –ntasks=80 specifies the number of computing slots your MPI program needs to reserve. For non-MPI jobs, always use 1 . (otherwise, multiple runs will be performed in case of using "srun app")

- -c 40 or --cpus-per-task=40 specifies the number of cores each of your program instance need to reserve. It is typically for non-MPI programs with multi-threading or multi-processing parallelization.

- In case of hybrid MPI program, you can specify more than 1 for –c or --cpus-per-task. The total cores reserved for your program will the multiplication of the two numbers.

MPI.job

```
#!/bin/bash
#SBATCH --job-name=integral_mpi
#SBATCH --output=out.ext
#SBATCH --mail-type=ALL
#SBATCH --mail-user=zhiwei.wang@utsa.edu
#SBATCH --ntasks=80
#SBATCH --nodes=2
#SBATCH --cpus-per-tasks=1

# Load one of these
module load shared openmpi
module load anaconda3
mpirun -n $SLURM_NTASKS python3 integral-mpi.py
```

Numpy

- Many NumPy operations do run in parallel if linked to optimized numerical libraries such as Intel MKL, OpenBlas, or Apple Accelerate, which handle parallel execution under the hood.
- To check if NumPy is running in parallel, first run the test program. Then, set the environment variables below and run the test again. If the second run is significantly slower, your NumPy is parallelized.

export OPENBLAS_NUM_THREADS=1
export MKL_NUM_THREADS=1

- Note that elementwise ops and loops in Python are not parallelized

```
import numpy as np
import time
d = 10000
# Two 200000 x 200000 need 320G RAM, leading to "out of
memeory"
a = np.random.rand(d, d)
b = np.random.rand(d, d)

def matrix_multiply(A, B):
    if len(A[0]) != len(B):
        raise ValueError("Incompatible dimensions for matrix
multiplication")
    result = [[0 for _ in range(len(B[0]))] for _ in range(len(A))]
    for i in range(len(A)):             # Rows of A
        for j in range(len(B[0])):      # Columns of B
            for k in range(len(B)):     # Columns of A / Rows of B
                result[i][j] += A[i][k] * B[k][j]
    return result
start = time.time()
c = a @ b
print("Time of numpy:", time.time() - start)
start = time.time()
c = matrix_multiply(a, b)
print("Time of manual looping:", time.time() - start)
```

Cupy For GPU Computing

CuPy is a GPU-accelerated array library for Python that has a NumPy-compatible API. It allows you to perform high-performance numerical computations on NVIDIA GPUs using CUDA.

```
import numpy as np
import cupy as cp
import time
d = 10000

a = np.random.rand(d, d)
b = np.random.rand(d, d)

start = time.time()
# Perform dot product on CPU with numpy (matrix multiplication)
c = a @ b
print("Time of numpy:", time.time() - start)

a_gpu = cp.asarray(a)
b_gpu = cp.asarray(b)
# Perform dot product on GPU (matrix multiplication)
start = time.time()
c_gpu = a_gpu @ b_gpu    # or cp.dot(a, b)
print("Time of cupy:", time.time() - start)

# Transfer result back to CPU if needed
c = cp.asnumpy(c_gpu)
```

Common Misconfiguration

- Requesting 40 CPUs (cores) but using 1 thread in your code
- Requesting GPU but running CPU-only code
- Requesting multiple nodes for non-MPI code
  - -n or –node= should be 1 for all non-MPI jobs

Why Scaling Is Hard

Amdahl's Law

Speedup limited by serial fraction:

$$Speedup = \frac{1}{S + \frac{P}{N}}$$

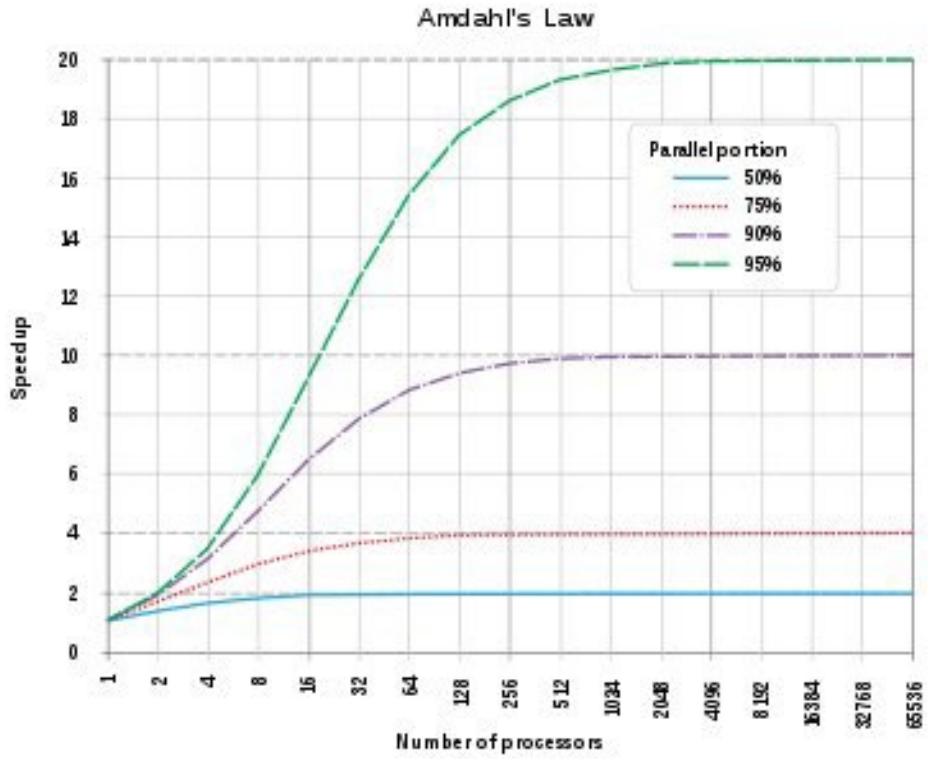
Where:

S = serial portion

P = parallel portion

N = number of processors

If 10% is serial → max speedup = 10× (no matter how many cores)



Why Scaling Is Hard

Real-World Bottlenecks

- Communication latency
- Synchronization barriers
- Memory bandwidth
- File system contention (important for Arc users)

Law of Diminishing Returns

Add more cores

- Communication overhead increases
- Memory contention increases
- IO bottlenecks dominate