Python Multiprocessing on the FASRC Clusters

Learning Objectives

- Serial Processing
- Why use Multiprocessing in Python?
- Optimizing cluster usage variables for multiprocessing
- Basic process-based parallelism
- Controlling utilization with pooling
- Accelerating your code with numpy
- Other helpful tools & training

Serial Processing - What is it & When to use?

- Default mode of Python
- Tasks executed one after the other, in a strict sequence
- Easy to implement & understand

- Lots of very short operations &/or trivial computations
- Anything where setup and teardown of processes would slow down execution
- Ex: Working with I/O on small files where the overhead of spawning threads or processes is higher
- Code has dependencies between tasks

- But inefficient for computationally intensive tasks; dealing with large datasets

Accelerating Your Python - Parallel Processing

- Multiple tasks executed simultaneously, utilizing multiple cores
- Achieve faster execution times by dividing workload
- Require synchronization & communication between processes or precaution between threads using global interpreter lock (GIL)
- Process-based
  - Separate processes, each with their own memory & Python interpreter => avoids GIL
  - Harder to share objects between processes
- Thread-based
  - Threads share same memory space
  - Could write to the same memory at the same time => needs GIL
- https://medium.com/@bfortuner/python-multithreading-vs-multiprocessing-73072ce5600b
- https://www.python-engineer.com/courses/advancedpython/15-thread-vs-process/

MultiProcessing

- Multiprocessing - **Process-based** parallelism
  - Ability of a system to run multiple processors at one time
- Allows several processes to run in parallel
- Multiprocessing module allocates tasks to different processors and makes better use of a multi-core machine
- No shared memory means better isolation between tasks, reducing the risk of data corruption
- Amplifies program efficiency & resource utilization
- multiprocessing — Process-based parallelism — Python 3.12.4 documentation
- https://medium.com/@surve.aasim/python-process-based-parallelization-3f91645ac4cb

Multithreading - Thread-based Parallelism

Pros:

- Threads are lightweight execution units within a process
- Share memory making communication between threads efficient
- Good for IO bound tasks

Cons:

- Must manage to avoid race conditions, synchronization issues
- Python's Global Interpreter Lock (GIL) limits the effectiveness of threads in CPU bound tasks by preventing the execution of python bytecode simultaneously

Multiprocessing vs Multithreading

Multiprocessing is parallelism/**doing** multiple things at the same time.	Multithreading is concurrency/**dealing** with multiple things at the same time.
Multiprocessing is for increasing speed	Multithreading is for hiding latency
Multiprocessing is best for computations	Multithreading is best for IO

- One thread running at any given time in a python process
- For CPU heavy tasks, use multiprocessing, `n_process=n_cores`, and never more

🔗 From Multithreading vs. Multiprocessing in Python (very informative)

Software Based Multiprocessing vs Python Coding

- Software with multiprocessing options:
  - May be limited in configuration variables and thus performance
  - Is probably just threading
- See significant performance gains writing your own Python multiprocessing code
- Tailor parallel execution to your needs
  - Process data efficiently
  - Control process communication
  - Handle errors and logging
- Community support around Python multiprocessing in guides, manuals, and books
- Submit your slurm job and walk away

Training Material

https://docs.rc.fas.harvard.edu/kb/training-materials/

```
# Login to Cannon
ssh <username>@login.rc.fas.harvard.edu

# Check current location & change if desired for this training: pwd
cd <desired-location>

# Clone FASRC User Codes repository:
https://github.com/fasrc/User_Codes/tree/master
SSH - git clone git@github.com:fasrc/User_Codes.git
HTTPS - git clone https://github.com/fasrc/User_Codes.git

# Create a training folder & go to that folder:
mkdir python-training; cd python-training

# Copy Python folders from the User Codes directory:
cp -r ../User_Codes/Languages/Python .
cp -r ../User_Codes/Parallel_Computing/Python/Python-Multiprocessing-Tutorial .
```

Python Package Installation - Interactive

- Go to a compute node on the test partition:

```
salloc -p test --nodes=1 --cpus-per-task=2 --mem=12GB --time=01:00:00
```

- Create a vanilla mamba/conda environment (for multiprocessing exercise):

```
module load python
mamba create --prefix=/n/holylabs/LABS/<desired-folder>/multiproc_env
python=3.11 -y
```

- Alternatively, if default *$HOME* is desired, then do following instead:

```
module load python
conda create --name multiproc_env python=3.11 -y
```

- See [Python Package Installation]

Python Package Installation

- Activate conda/mamba environment:

```
mamba activate /n/holylabs/LABS/<desired-folder>/multiproc_env
```

- Or if $HOME used, then: `mamba activate multiproc_env`
- Install relevant python packages (Mamba recommended):

```
mamba install numpy pandas matplotlib -y
pip install jupyterlab swifter
```

- Always pip install inside a conda environment to avoid package conflicts
- https://docs.rc.fas.harvard.edu/kb/python-package-installation/#Pip_Installs
- Deactivate the environment: `mamba deactivate`

Python Package Installation - sbatch

https://github.com/fasrc/User_Codes/tree/master/Languages/Python/Example2

```
# Go to Multiprocessing Tutorial
cd Python-Multiprocessing-Tutorial
# Submit job
sbatch run_multiproc.sbatch
```

multiprocbuild_env.sh: bash script for creating the multiproc_env mamba environment

```
#!/bin/bash
#SBATCH -J multi_proc           # job name
#SBATCH -o multi_proc.out       # standard output file
#SBATCH -e multi_proc.err       # standard error file
#SBATCH --cpus-per-task=1     # number of cores
#SBATCH --partition=test      # partition
#SBATCH --time=0-01:00        # time in D-HH:MM
#SBATCH --mem=10G             # memory in GB

# Load required modules
module load python

# Build the environment
sh multiprocbuild_env.sh

# Activate the environment
mamba activate multiproc_env

# Install pip packages
pip install jupyterlab swifter
```

Multiprocessing - Process-based Parallelism - Basic

- [Multiprocessing in Python - MachineLearningMastery.com]()
- Two functions declared to execute print statements after sleeping for 2 & 3 seconds, resp.
- 3 processes created using *multiprocessing.Process* inside *main()*
- The *Process()* utilizes *target* argument to run target process
- Processes are run using *start()*
- Use *join()* to run & exit a processes before the main program process

```
import multiprocessing
import time
def worker():
    name = multiprocessing.current_process().name
    print(name, 'Starting')
    time.sleep(2)
    print(name, 'Exiting')
def my_service():
    name = multiprocessing.current_process().name
    print(name, 'Starting')
    time.sleep(3)
    print(name, 'Exiting')
if __name__ == '__main__':
    service = multiprocessing.Process(name='my_service', target=my_service)
    worker_1 = multiprocessing.Process(name='worker 1', target=worker)
    worker_2 = multiprocessing.Process(target=worker)
    worker_1.start()
    worker_2.start()
    service.start()
```

Multiprocessing in Python

- On the cluster, difference between number of CPUs allocated to the job vs total number of CPUs available on the node
- Go to a compute node on the test partition requesting 10 cores:

```
salloc -p test --nodes=1 --cpus-per-task=10 --mem=12GB --time=01:00:00
```

- See total number of cores available on the node:

```
scontrol show node <nodename>
```

- Execute **cpu-count.py** to see which command gives you the number of cores allocated to your job:

```
cd Python-Multiprocessing-Tutorial
python cpu-count.py
```

- See How to find out the number of CPUs using python - Stack Overflow

Multiprocessing - Pooling

- Run 1000 processes together - may not be possible
- Create a process pool to limit number of processes that can be run at a time
- Function declared to return the cube
- The *multiprocessing.Process* doesn't work with p.start() & p.join(), would need an output queue as well. But faster than *Pool()*
- The *multiprocessing.Pool* module easier to use, returns ordered result using *pool.map()*, & causes less overhead
- See [Python multiprocessing: How to know to use Pool or Process? - Stack Overflow](#)

```
import multiprocessing
import time
import os

def cube(x):
    return x**3

if __name__ == '__main__':

  # The Process class
  processes = [multiprocessing.Process(target=cube, args=(x,)) for x in range(1,len(os.sched_getaffinity(0)))]
  [p.start() for p in processes]
  result_process = [p.join() for p in processes]

  # The Pool class
  pool = multiprocessing.Pool(processes=len(os.sched_getaffinity(0)))
  result_pool = pool.map(cube, range(1,len(os.sched_getaffinity(0)))]
```

Multiprocessing + Numpy with JupyterLab notebook

- Using Multiprocessing along with Numpy to accelerate python program
- Go to OOD (Cannon or FASSE) & launch JupyterLab notebook on *test* with
  - *52 CPUs*
  - *gcc/12.2.0-fasrc01* loaded as a module
  - *multiproc_env* loaded as a kernel
  - In *python-training/Python-Multiprocessing-Tutorial*
- Problem Statement:
  - A sample data file has 4 columns and 1000 entries. Columns correspond to the time a job was submitted, when it started, when it ended, and number of CPUs allocated.
  - Calculate the total number of CPUs in use by currently running jobs for every submitted job

Multiprocessing + Numpy

- Convert numerical columns to Numpy arrays.
- Declare a function to calculate CPUs utilized: *calculate_cpus_utilized()*
- Multiple methods utilized for the calculation:
  - Use the function over each submitted-job entry
    - Pandas *apply()*
    - swifter.apply()
  - Using Numpy arrays & for-loop
  - Using Multiprocessing with a pool of processes = #CPUs requested for OOD job
- Run the notebook to see which method gives the fastest result
- Fastest: Combination of Numpy and Multiprocessing

Accelerate Python - Other Tools

- Numba
  - https://numba.pydata.org/
- Swifter
  - Speed up your Pandas Processing with Swifter | by Cornellius Yudha Wijaya | Towards Data Science
  - GitHub - jmcarpenter2/swifter: A package which efficiently applies any function to a pandas dataframe or series in the fastest available manner
- Dask
  - https://www.dask.org/

FASRC documentation

- FASRC docs: https://docs.rc.fas.harvard.edu/
- FASRC Python docs:
  - https://docs.rc.fas.harvard.edu/kb/python/
  - https://docs.rc.fas.harvard.edu/kb/python-package-installation/
- GitHub User_codes: https://github.com/fasrc/User_Codes/
- Getting help
  - Office hours: https://www.rc.fas.harvard.edu/training/office-hours/
  - Ticket
    - Portal: http://portal.rc.fas.harvard.edu/rcrt/submit_ticket (requires login)
    - Email: rchelp@rc.fas.harvard.edu

Upcoming Trainings

Training calendar: https://www.rc.fas.harvard.edu/upcoming-training/

Informatics: SNPArcher tutorial

Training is focused on the snakemake workflow for variant calling in non-model organisms

Details: https://informatics.fas.harvard.edu/events/

Survey

Please, fill out our course survey. Your feedback is essential for us to improve our trainings!!

http://tinyurl.com/FASRCsurvey

Thank you :)

FAS Research Computing