Parallel Processing in Python

Ade Fewings, Aaron Owen
Bangor University
Updated: 2022-02-17

eResearch

- Supercomputing Wales
  - Available to researchers at Bangor
- Research Software Engineers
- Collate expert knowledge into an open and shared centralised repository
  - Yammer
  - Github
  - Workshops
  - Projects
  - Acknowledgements

Training Workshops

- Introduction to the Linux Shell
- Version Control Using Git
- Programming Principles and Practice using Python
- Advanced Python
- **Parallel Processing in Python**
- Machine Learning with Python*

See and discuss on the Yammer group.

Suggestions for new training welcome.

Environment

Python at Bangor

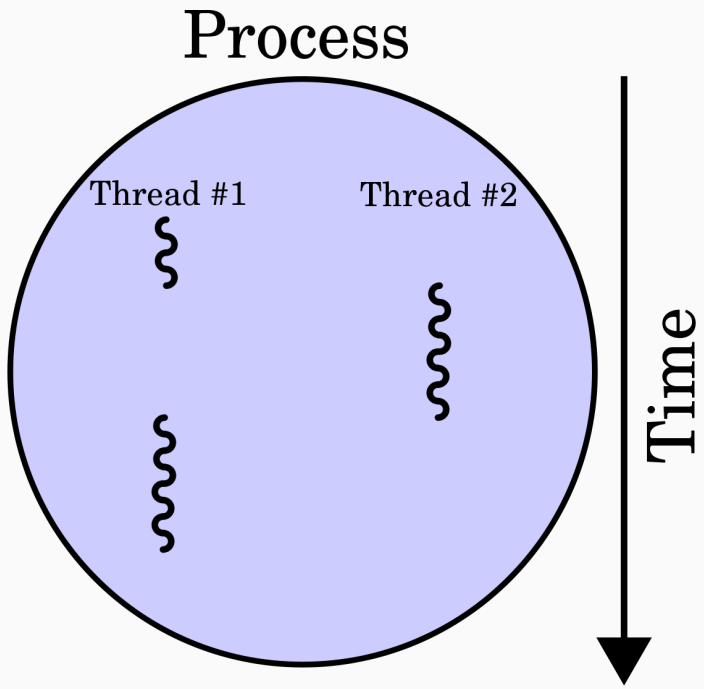
- Visit https://jupyter.bangor.ac.uk/jupyter/hub/login
- Login using your Bangor University credentials.
- Click the `Python 3.8` notebook.

Overview

- What are processes and threads?
- What is parallel processing?
- Multiprocessing vs Multithreading.
- CPU vs Core.
- Determine available resources.
- Python's GIL problem.
- Multiprocessing examples.
- Using the Supercomputing Wales.
- Desktop to cluster.
- GNU Parallel.

What are processes and threads?

- A **process** is an instance of a running program.
- Processes create **threads** (sub-processes) to handle sub-tasks.
- **Threads** live and operate inside a **process**.



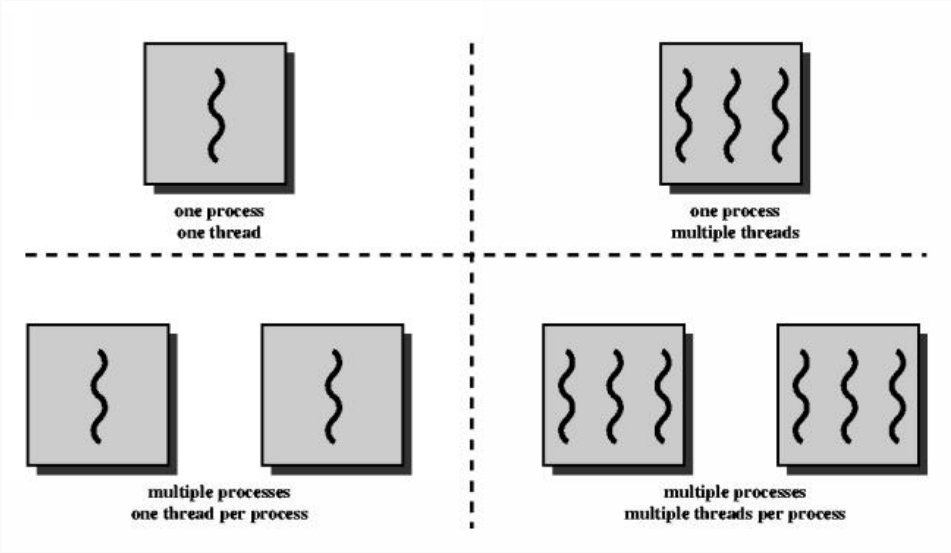
Note: Running on a single processor.

Application example

- When you open a text editor, you create a **process**.
- When you start typing, the process creates (spawns) **threads**.
  - One to read keystrokes
  - One to display text
  - One to autosave your file
  - One to highlight spelling mistakes
- By creating multiple threads, the text editor takes advantage of idle CPU time (waiting for keystrokes or files to load) and makes you more productive.
- **Diagram**

Multithreading vs Multiprocessing

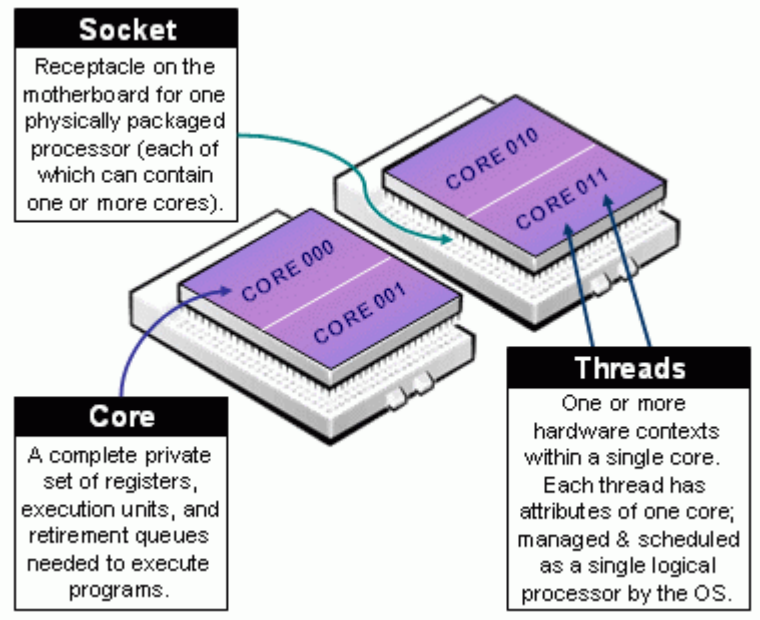
- **Multithreading** - Single processes, multiple threads
- **Multiprocessing** - Multiple processes
- **Multithreading** and **Multiprocessing** - Mulitple processes and multiple threads



- We will just focus on **multiprocessing** (multiple processes, one thread per process)

CPU vs Core

- The CPU, or processor, manages the fundamental computation work of the computer (processes).
- **CPUs** have one or more **cores**, allowing the CPU to execute code simultaneously.

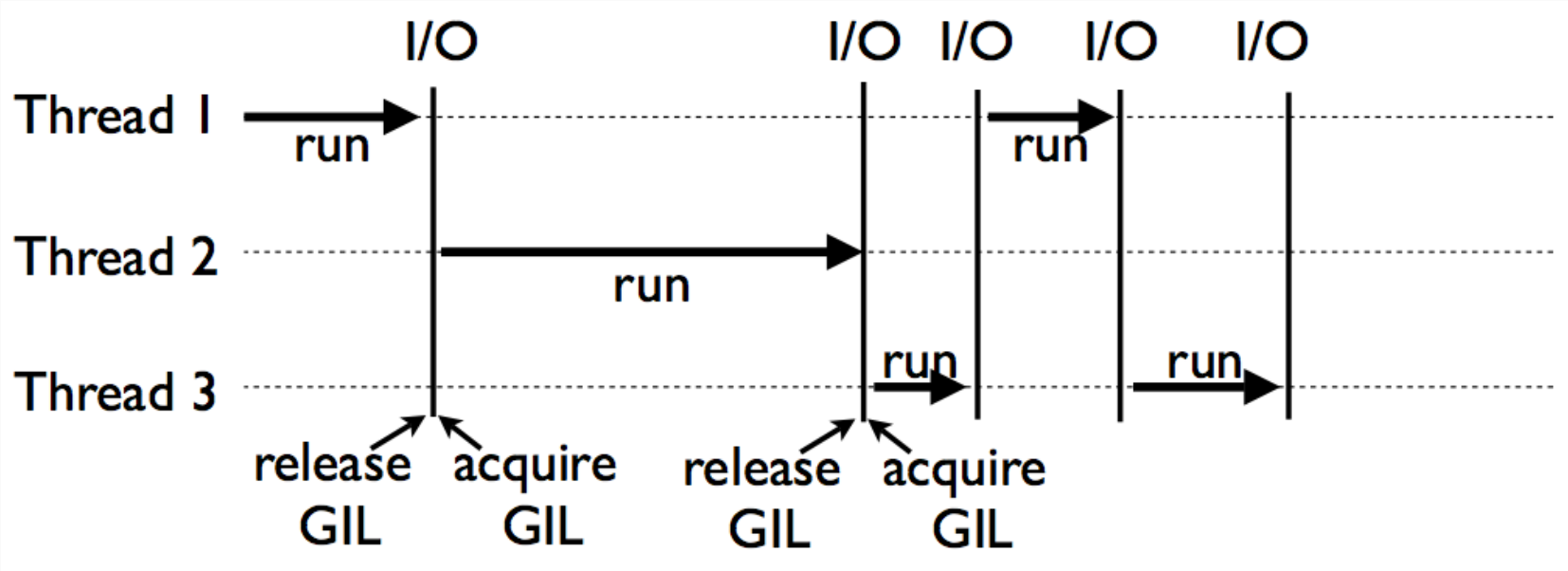


- SCW
  - 40 core node - 2 CPU with 20 cores each
  - 64 core node - 2 CPU with 32 cores each

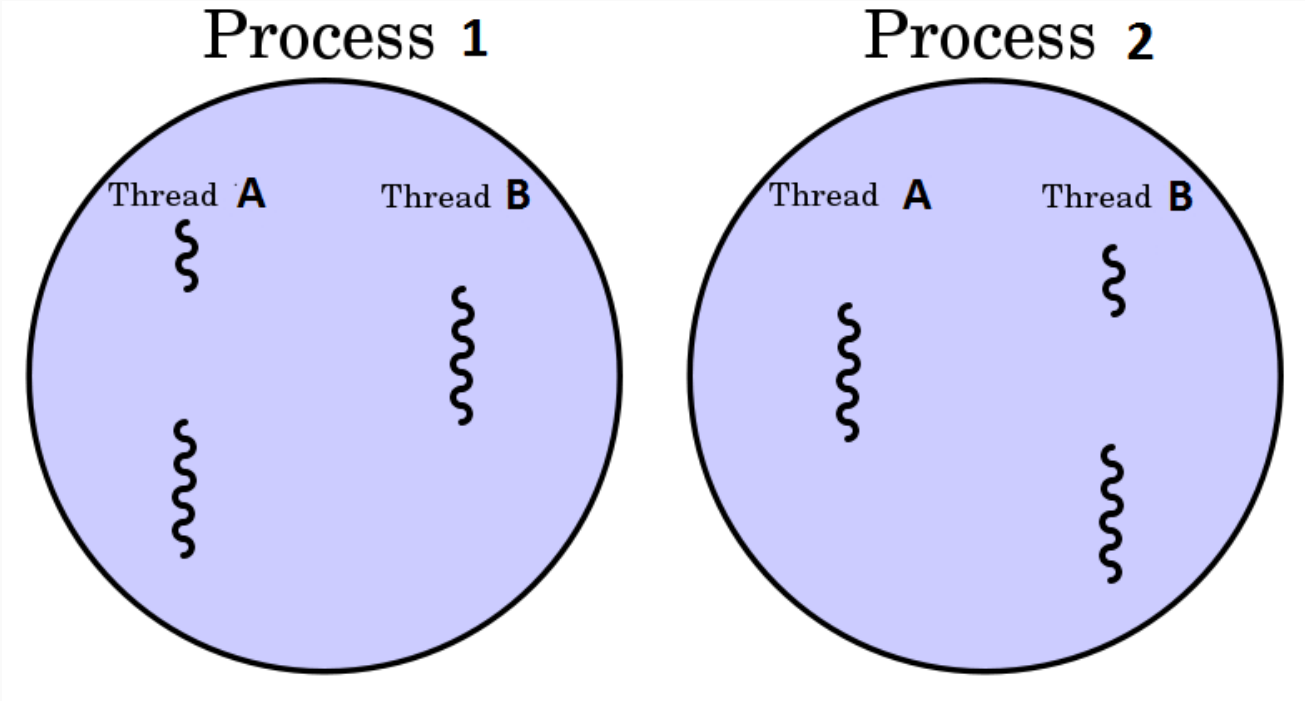
Python's GIL problem

- Python has something called the GIL (Global Interpreter Lock).
- It prevents two threads from executing simultaneously in the same process.
- Libraries like Numpy bypass this limitation by running external code in C which can manually release the GIL to speed up computations.
- The most popular solution is to use **multiprocessing**.
- Python provides a library to simplify multiprocessing.
- Python processes get their own Python interpreter and memory space so the GIL won't be a problem.

Python's GIL problem



Multiple processes



When to use threads and processes?

- **Processes** speed up Python operations that are CPU intensive because they benefit from multiple cores and avoid the GIL problem.
- **Threads** are best for I/O tasks or tasks involving external systems because threads can work more efficiently together (lower overhead).
- **Threads** provide no benefit in Python for CPU intensive tasks because of the GIL problem.

When to use threads and processes?

- If your code has a lot of I/O or Network usage
  - Multithreading is your best bet because of its low overhead
- If you have a graphical user interface application
  - Multithreading so your UI thread doesn't get locked up
- If your code is CPU heavy (bound)
  - You should use multiprocessing (if your machine has multiple cores)

What is parallel processing?

- Parallel processing is a mode of operation where a task (program) is executed simultaneously in multiple processes in the same computer.
- It is meant to reduce the overall processing time.
- Python provides a built in module to run independent parallel processes.
- It enables users to use multiple processors on a machine.
- The management of worker processes can be simplified with the *Pool* object.

Process example

```
from multiprocessing import Process

def func(name):
    print(f'Hello {name}')

def main():
    p = Process(target=func, args=('Aaron', ))
    p.start()

if __name__ == '__main__':
    main()
    print('Why is this line printing first?')
```

Process example

```
from multiprocessing import Process

def func(name):
    print(f'Hello {name}')

def main():
    p = Process(target=func, args=('Aaron', ))
    p.start()
    p.join()

if __name__ == '__main__':
    main()
    print('Why is this printing last?')
```

Process id

```
from multiprocessing import Process
import os

def func():
    print(f"\tfunc's process id {os.getpid()}")
    print(f"\t\tfunc's parent process id = {os.getppid()}")

def main():
    p1 = Process(target=func)
    p1.start()
    p1.join()

    p2 = Process(target=func)
    p2.start()
    p2.join()

if __name__ == '__main__':
    print(f"Main's process id is {os.getpid()}")
    main()
```

Determine available resources

- Firstly, we need to determine what resources are available to our program.
- The maximum number of processes you can run at a time is limited by the number of processors in your computer.
- We can use python to determine the number of processors.
- Enables us to avoid assumptions about the number of processors.

```
import multiprocessing as mp

print(f'Number of processors: {mp.cpu_count()}')

> Number of processes: 4
```

- On the SCW cluster, we have access to a few more than 4 processors.

Multiprocessing

- Let's treat the available cores as a pool of resources - *workers* (diagram).
- Generate a list of items to be worked on.
- Each item in the list will be processed by *worker* core in the pool of resources.
- The multiprocessing pool can be used for parallel execution of a function across multiple input values.
- Distributing the input data across processes (data parallelism).
- Example: Distributing work to each individual in the workshop.

Multiprocessing Pool

- Map `func` to list items and print value

```
import multiprocessing as mp

def func(x):
    print(x * x)

def main():
    num_cores = mp.cpu_count()
    pool = mp.Pool(num_cores)
    pool.map(func, [4, 2, 3])

if __name__ == '__main__':
    main()
    print('done')
```

Multiprocessing Pool

- Map `func` to list items and return values

```
import multiprocessing as mp

def func(x):
    return (x * x)

def main():
    num_cores = mp.cpu_count()
    pool = mp.Pool(num_cores)
    result = pool.map(func, [4, 2, 3])
    print(result)

if __name__ == '__main__':
    main()
    print('done')
```

Multiprocessing Pool

```
import multiprocessing as mp

work_items = ["Aaron", "Beth", "George", "Mia"]


def worker_func(work_item):
    print(f'Process {work_item}')


def main():
    num_cores = mp.cpu_count()
    work_pool = mp.Pool(num_cores)
    work_pool.map(worker_func, work_items)


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

Multiprocessing Pool with process id

- Print process id

```
import multiprocessing as mp

# This could be a list of any Python datatype (lists, functions, dicts)
work_items = ["Aaron", "Beth", "George", "Mia"]


def worker_func(work_item):
    process_id = mp.current_process()
    print(f'Process {work_item} on process {process_id}')


def main():
    num_cores = mp.cpu_count()
    work_pool = mp.Pool(num_cores)
    work_pool.map(worker_func, work_items)  # map_async


if __name__ == '__main__':
    main()
    print('done')
```

Launch multiple python programs

- We can use **Pool.map()** to run multiple python scripts in parallel.

```
import os
import multiprocessing as mp

# List of scripts or args to program?
scripts = ['script_1.py', 'script_2.py', 'script_3.py']


def run_python_script(script):
    os.system(f'python {script}')  # args?


def main():
    num_cores = mp.cpu_count()
    pool = mp.Pool(processes=num_cores)
    pool.map(run_python_script, scripts)


if __name__ == '__main__':
    main()
    print('done')
```

Launch multiple programs

```
def run_script(script):
    app = None

    if script.endswith('.py'):
      app = 'python3'

    if script.endswith('.R'):
      app = 'Rscript'

    if app:
      os.system(f'{app} {script}')  # args?
    else:
      print('Unable to determine app')
```

Challenge

- Extend the code to pass through the command line args

map vs imap

- A more optimised method is `imap`.
- This method does not duplicate the memory space of the original Python process to different workers.
  - The outcome of using `imap` is identical to `map`, but reduces memory usage.
- One thing to note is that `imap` and `map` can only pass one parameter to the function to be parallelised.
- We can pass more than one argument using `starmap`.

starmap

```python
import multiprocessing as mp

def worker_func(a, b, c, d):
    process_id = mp.current_process()
    print(f'Process {a} {b} {c} {d} on process {process_id}')

def main():
    num_cores = mp.cpu_count()
    work_pool = mp.Pool(num_cores)
    work_pool.starmap(
        worker_func,
        [
            (1, 2, 3, 4),  # process 1
            (5, 6, 7, 8),  # process 2
        ])
    # starmap_async

if __name__ == '__main__':
    main()
    print('done')
```

What is Synchronous and Asynchronous

- In parallel processing, there are two types of execution:
  - **Synchronous** and **Asynchronous**.
- **Synchronous** execution is where the processes are completed in the same order in which they were started.
- **Asynchronous** execution is where the processes may not be completed in the same order in which they were started. However, it may process the items quicker.

Locks

- If required, we can manually aquire and release a lock to control the order in which processes run.

```
from multiprocessing import Process, Lock

def func(lock, num):
    lock.acquire()
    try:
        print(f'The sentence is not mixed up.')
    finally:
        lock.release()

def main():
    lock = Lock()

    for num in range(10):
        Process(target=func, args=(lock, num)).start()

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

- Without using the lock output from the different processes is liable to be mixed up.

Pool and Process

- There are two main classes in the multiprocessing module to implement parallel execution: The **Pool** Class and the **Process** Class.
- The general way to parallelise any operation is to take a particular function that should run multiple times and make it run in parallel on different processors.
- Feel free to ask Aaron for help when adding parallel features to code.

SuperComputing Wales

- Login
- Modules
- Python Shell
- Queue
- Home / Scratch
- Requesting resources
- GNU parallel
    - Store a file of commands - Automatic management of resource pool

Multiprocessing slurm script

```
#!/usr/bin/env bash

# Run main.py on 4 cores
# Usage: sbatch run.slurm

#SBATCH --job-name=map

#SBATCH --output=logs/map.%J.out
#SBATCH --error=logs/map.%J.err

#SBATCH --partition=htc
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=4

#SBATCH --account=scw1124
#SBATCH --time=00-00:05

module purge
module load python/3.7.0

python3 main.py
```

Multiprocessing python script

```python
import os
import multiprocessing as mp

def worker_func(data):
    process_id = mp.current_process()
    print(f'worker_func is running on process {process_id}, data={data}')

def main():
    # How many cores were assigned to the slurm job allocation?
    num_cores = int(os.getenv('SLURM_CPUS_PER_TASK'))
    print(f'Found {num_cores} cores')

    pool = mp.Pool(num_cores)
    worker_data = range(10)
    pool.map(worker_func, worker_data)

if __name__ == "__main__":
    main()
```

GNU Parallel slurm script

```
#!/usr/bin/env bash

#SBATCH --job-name=parallel

#SBATCH --output=%J.out
#SBATCH --error=%J.err

#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=4

#SBATCH --account=scw1124

#SBATCH --time=00-03:00

module purge

module load python/3.9.2
module load parallel

echo "Start!"
time parallel < commands.txt
echo "Finished!"
```

commands.txt

```
python3 -c "print('Hello from line 1')"
python3 -c "print('Hello from line 2')"
python3 -c "print('Hello from line 3')"
python3 -c "print('Hello from line 4')"
```