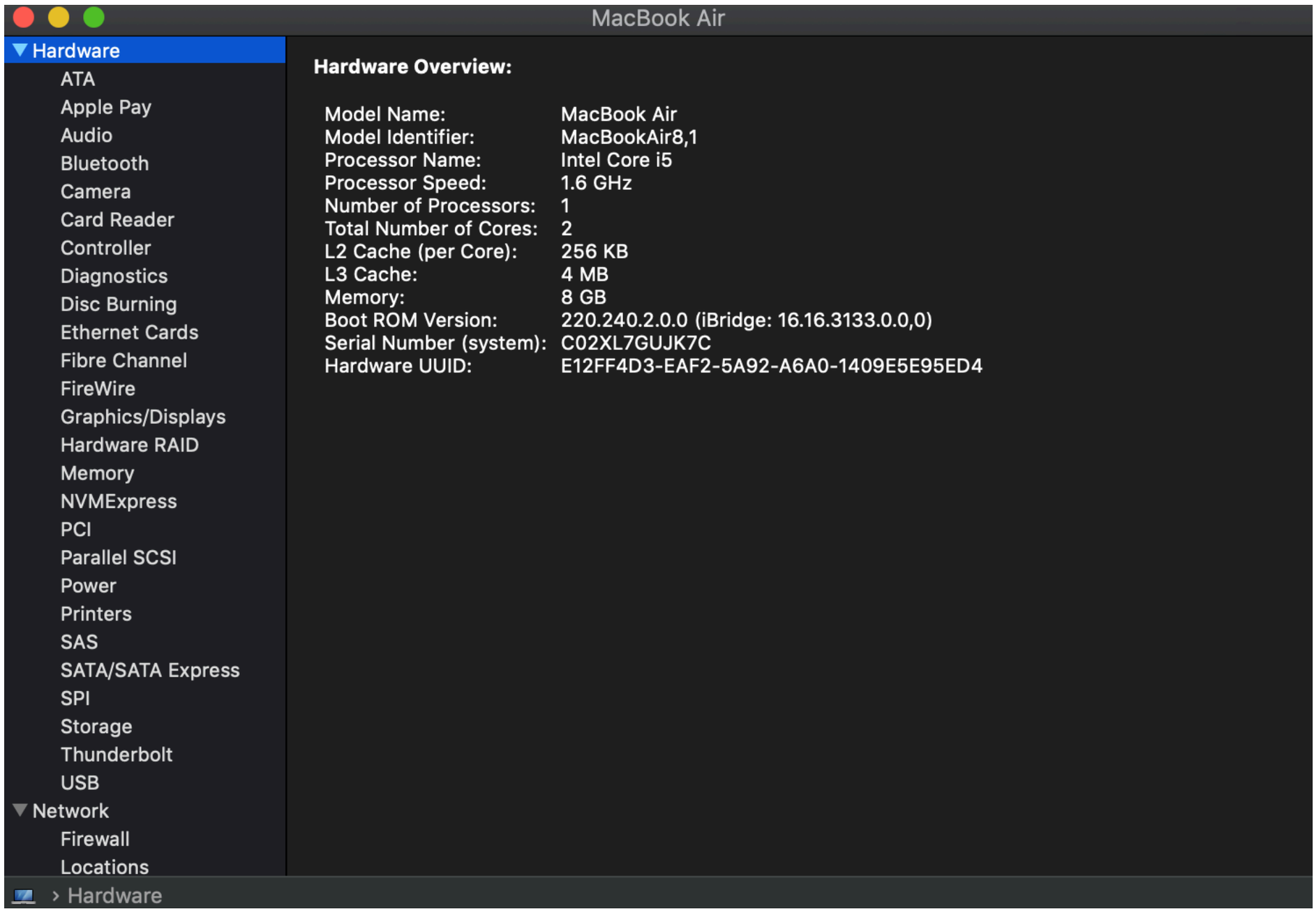
Parallel Computing

- Parallel computing
- Embarrassingly parallel
- Hadoop computing framework
- Python multiprocessing

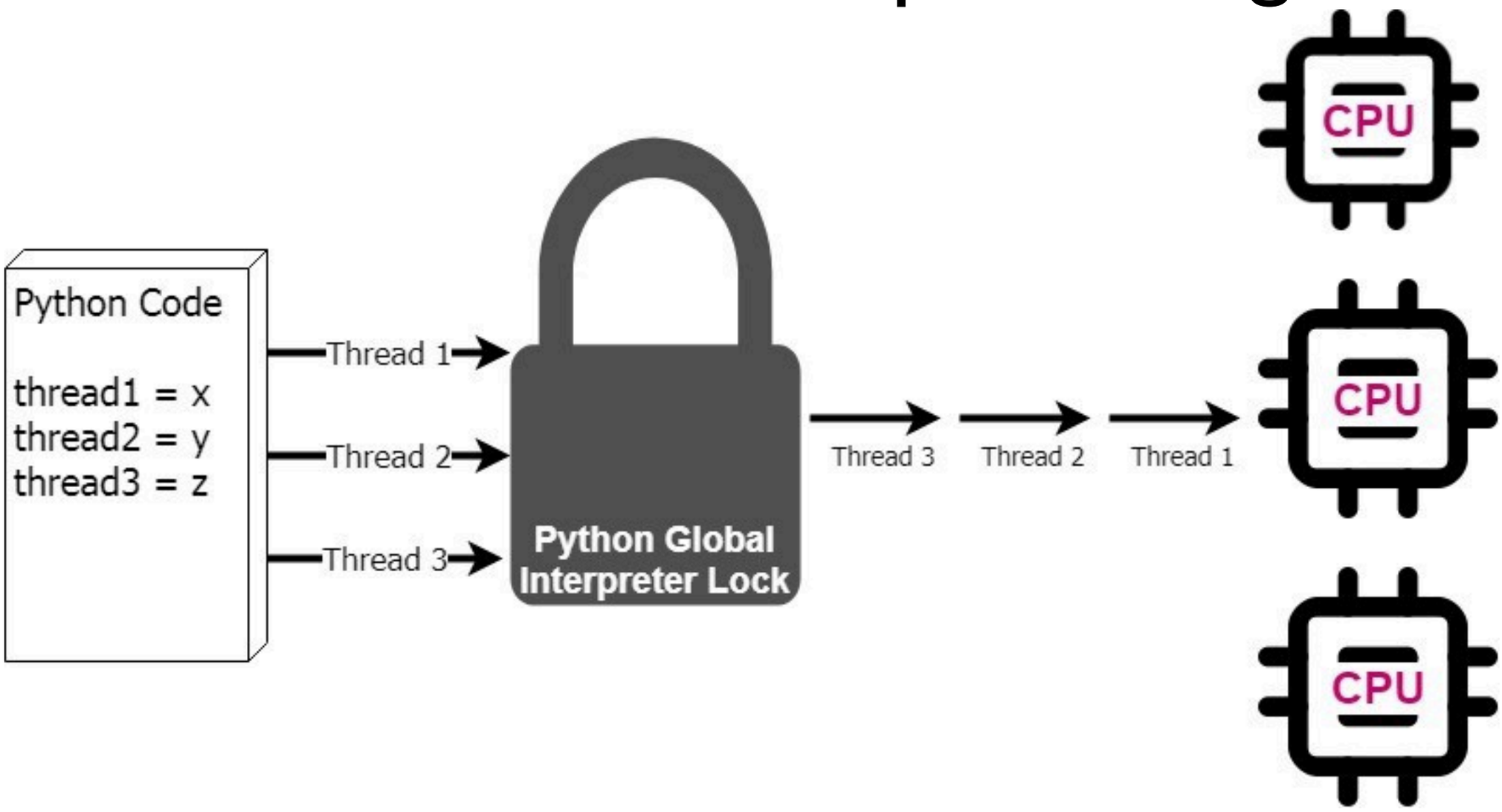
Python Multiprocessing

Checking how many CPUs




```
import multiprocessing as mp
nprocs = mp.cpu_count()
print(f"Number of CPU cores: {nprocs}")
```

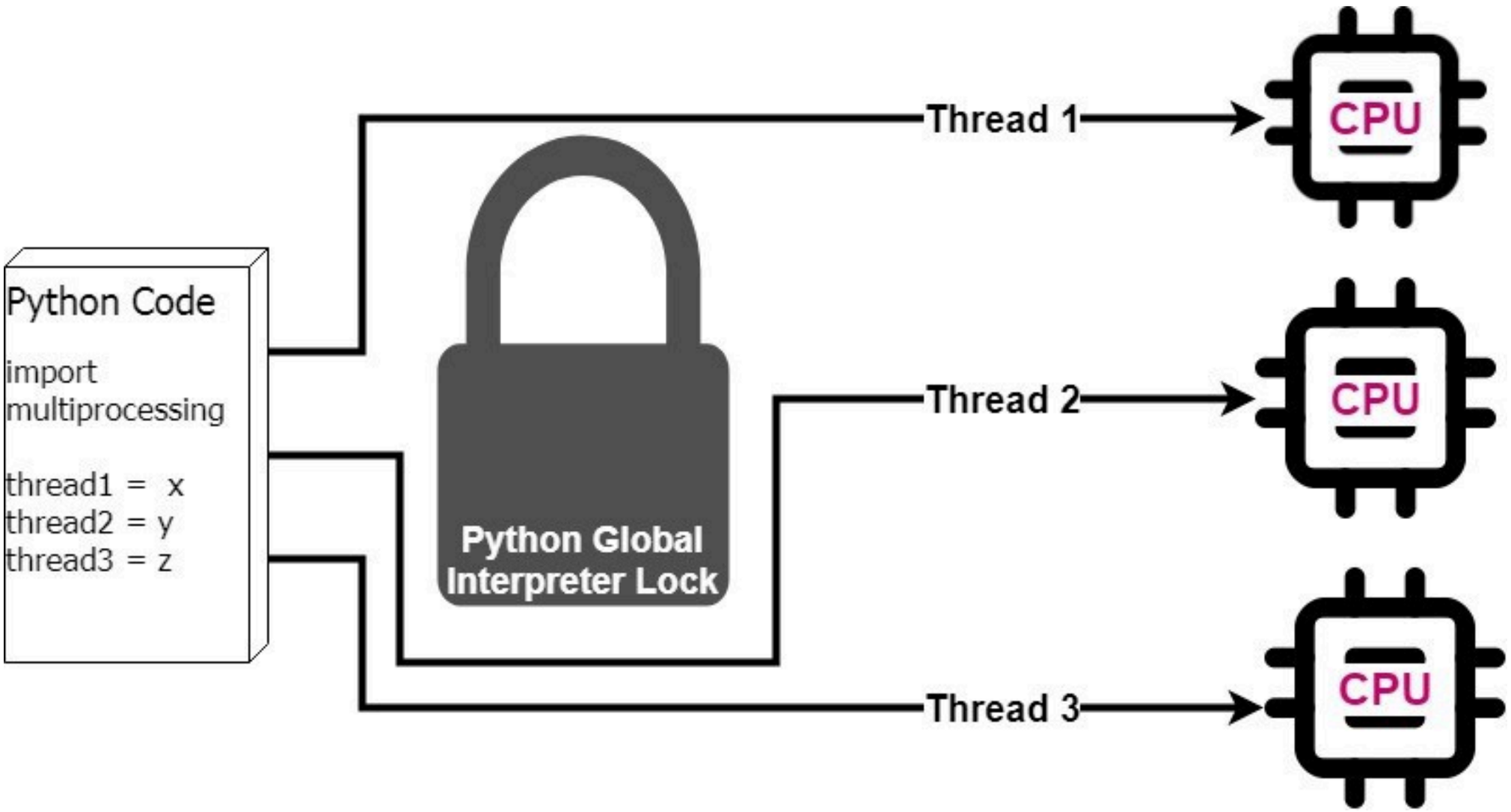
Threads vs. Multiprocessing



https://medium.com/@urban_institute/using-multiprocessing-to-make-python-code-faster-23ea5ef996ba

https://www.linuxjournal.com/content/multiprocessing-python

Multiprocessing in Python



Multiprocessing in Python

threading	→ multiprocessing
thread	→ Process
threads	→ processes
thread	→ process

Pool

A class representing a pool of worker processes

e.g. `pool = Pool(processes=4)`

- `pool` has 4 worker processes

Tasks can be offloaded to these workers

e.g. `result = pool.map(f, data)`

- `data` is an array or list
- function `f` will be applied to each member of `data` in turn

Simple Pool Example

```
from multiprocessing import Pool
from math import sqrt
newPool = Pool(processes=2) # initialize the Pool.
res = newPool.map(sqrt,[4,5,6,7])
res
Out[22]:
[2.0, 2.23606797749979, 2.449489742783178, 2.6457513110645907]
```

pool_process

1

```python
import time
import math
from multiprocessing import Pool

# A function for timing a job that uses a pool of processes.
#  f is a function that takes a single argument
#  data is an array of arguments on which f will be mapped
#  pool_size is the number of processes in the pool.
def pool_process(f, data, pool_size):
    tp1 = time.time()
    pool = Pool(processes=pool_size) # initialize the Pool.
    result = pool.map(f, data)       # map f to the data using the Pool
    pool.close() # No more processes
    pool.join()  # Wait for the pool processing to complete.
    print("Results", result)
    print("Overall Time:", int(time.time()-tp1))
```

2

```python
def my_func(x):
    return math.sqrt(x)
```

```python
dataRange = range(20)
```

3

```python
pool_process(my_func, dataRange, 2)
```

```
# This verbose version shows which process in the pool is running each task.
def my_func_verbose(x):
    s = math.sqrt(x)
    print("Task", multiprocessing.current_process(), x, s)
    return s
```

```
dataRange = range(10)
pool_process(my_func_verbose, dataRange, 1)
```

```
Task <ForkProcess(ForkPoolWorker-7, started daemon)> 0 0.0
Task <ForkProcess(ForkPoolWorker-7, started daemon)> 1 1.0
Task <ForkProcess(ForkPoolWorker-7, started daemon)> 2 1.4142135623730951
Task <ForkProcess(ForkPoolWorker-7, started daemon)> 3 1.7320508075688772
Task <ForkProcess(ForkPoolWorker-7, started daemon)> 4 2.0
Task <ForkProcess(ForkPoolWorker-7, started daemon)> 5 2.2360679774997 9
Task <ForkProcess(ForkPoolWorker-7, started daemon)> 6 2.449489742783178
Task <ForkProcess(ForkPoolWorker-7, started daemon)> 7 2.6457513110645907
Task <ForkProcess(ForkPoolWorker-7, started daemon)> 8 2.8284271247461903
Task <ForkProcess(ForkPoolWorker-7, started daemon)> 9 3.0
Results [0.0, 1.0, 1.4142135623730951, 1.7320508075688772, 2.0, 2.2360679
7749979, 2.449489742783178, 2.6457513110645907, 2.8284271247461903, 3.0]
Overall Time: 0
```

1 process, same as running in series in a loop

```
dataRange = range(10)
pool_process(my_func_verbose, dataRange, 2)
```

```
Task <ForkProcess(ForkPoolWorker-9, started daemon)> 2 1.4142135623730951
Task <ForkProcess(ForkPoolWorker-8, started daemon)> 0 0.0
Task <ForkProcess(ForkPoolWorker-8, started daemon)> 1 1.0
Task <ForkProcess(ForkPoolWorker-9, started daemon)> 3 1.7320508075688772
Task <ForkProcess(ForkPoolWorker-8, started daemon)> 4 2.0
Task <ForkProcess(ForkPoolWorker-9, started daemon)> 6 2.449489742783178
Task <ForkProcess(ForkPoolWorker-9, started daemon)> 7 2.6457513110645907
Task <ForkProcess(ForkPoolWorker-8, started daemon)> 5 2.23606797749979
Task <ForkProcess(ForkPoolWorker-8, started daemon)> 8 2.8284271247461903
Task <ForkProcess(ForkPoolWorker-8, started daemon)> 9 3.0
Results [0.0, 1.0, 1.4142135623730951, 1.7320508075688772, 2.0, 2.2360679
7749979, 2.449489742783178, 2.6457513110645907, 2.8284271247461903, 3.0]
Overall Time: 0
```

2 processes

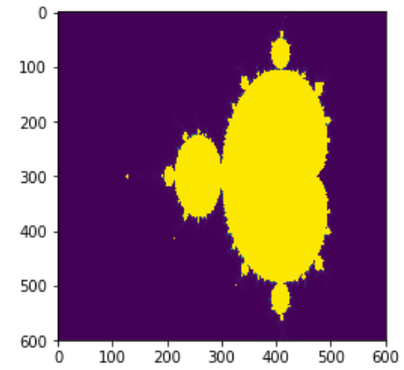
- Allows both cores to be used in parallel
- Asynchronous behaviour

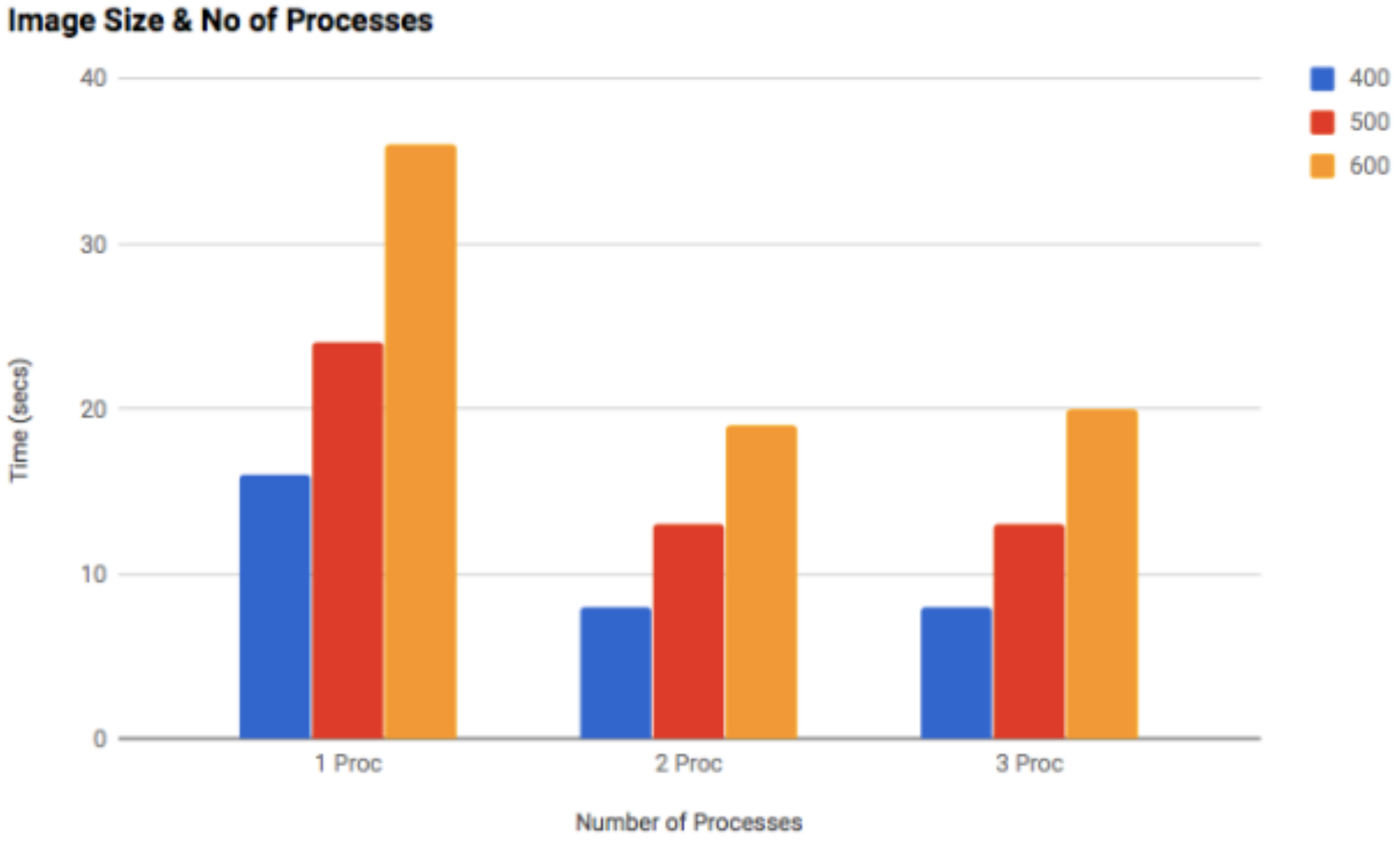
```
dataRange = range(10)
pool_process(my_func_verbose, dataRange, 10)
```

```
Task <ForkProcess(ForkPoolWorker-12, started daemon)> 2 1.4142135623730951
Task <ForkProcess(ForkPoolWorker-10, started daemon)> 0 0.0
Task <ForkProcess(ForkPoolWorker-15, started daemon)> 5 2.23606797749979
Task <ForkProcess(ForkPoolWorker-16, started daemon)> 6 2.449489742783178
Task <ForkProcess(ForkPoolWorker-14, started daemon)> 4 2.0
Task <ForkProcess(ForkPoolWorker-11, started daemon)> 1 1.0
Task <ForkProcess(ForkPoolWorker-18, started daemon)> 8 2.828427124746l903
Task <ForkProcess(ForkPoolWorker-13, started daemon)> 3 1.732050807568877Z
Task <ForkProcess(ForkPoolWorker-17, started daemon)> 7 2.645751311064590?
Task <ForkProcess(ForkPoolWorker-19, started daemon)> 9 3.0
Results [0.0, 1.0, 1.4142135623730951, 1.732050807568877Z, 2.0, 2.23606797749979, 2.449489742
783178, 2.645751311064590?, 2.828427124746l903, 3.0]
Overall Time: 0
```

10 processes but 2 cores

Testing with `Mandelbrot Images`

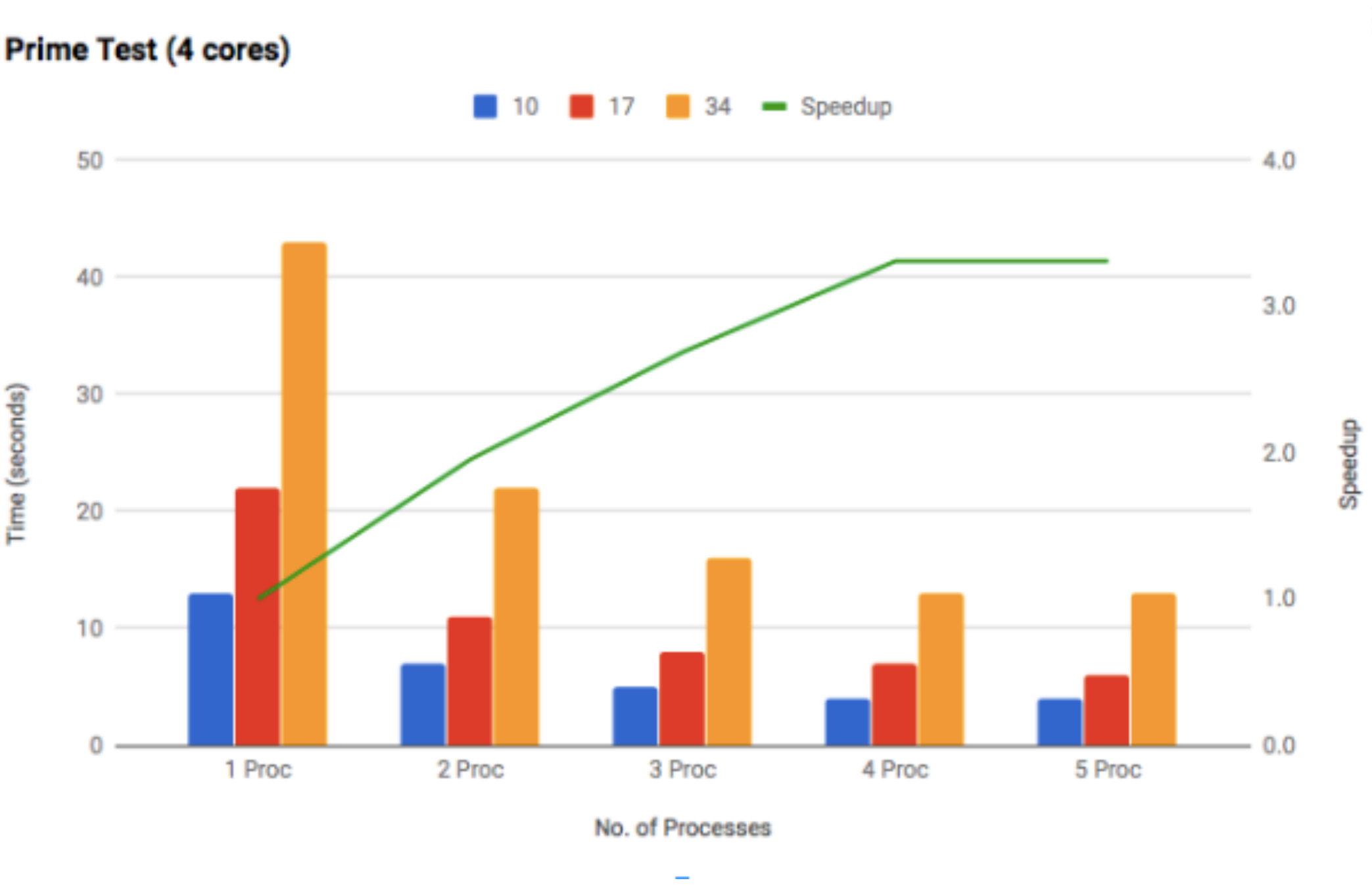




https://timothyawiseman.wordpress.com/2012/12/21/a-really-simple-multiprocessing-python-example/

https://en.wikipedia.org/wiki/Mandelbrot_set

4 Core CPU


Prime Test (4 cores)
10
17
34
Speedup
50
40
30
20
10
0
4.0
3.0
2.0
1.0
0.0
Time (seconds)
Speedup
1 Proc
2 Proc
3 Proc
4 Proc
5 Proc
No. of Processes

Assignment 1

Multiprocessing

Objective

The objective of this exercise is to evaluate speedup derived from using multiple CPU cores through the multiprocessing facility in Python.

What is provided?

A Python Notebook (MultiprocessingCore) is available on the Moodle page that provides a basic multiprocessing framework. The Notebook contains code to set up a multiprocessing Pool and use that Pool to process tasks. If multiple cores are available the Pool can use those cores to speed up processing. The function in the sample code is a simple square-root function so it is not possible to see the impact of using multiple cores.

A naive function for checking primes is also provided. If this is used to check large numbers (8 digit) it takes time and the speedup from using multiple cores will be evident.

There are plenty of primes to be found here: https://primes.utm.edu/lists/small/millions/

Further reading

https://www.linuxjournal.com/content/multiprocessing-python

https://www.quantstart.com/articles/Parallelising-Python-with-Threading-and-Multiprocessing

https://www.youtube.com/watch?v=Lu5LrKh1Zno&list=PLeo1K3hjS3uub3PRhdoCTY8BxMKSW7RjN&index=2

https://medium.com/@urban_institute/using-multiprocessing-to-make-python-code-faster-23ea5ef996ba