Multithreading and Multiprocessing

Recall the phrase "many hands make light work". This is as true in programming as anywhere else.

What if you could engineer your Python program to do four things at once? What would normally take an hour could (almost) take one fourth the time.*

This is the idea behind parallel processing, or the ability to set up and run multiple tasks concurrently.

* *We say almost, because you do have to take time setting up four processors, and it may take time to pass information between them.*

Threading vs. Processing

A good illustration of threading vs. processing would be to download an image file and turn it into a thumbnail.

The first part, communicating with an outside source to download a file, involves a thread. Once the file is obtained, the work of converting it involves a process. Essentially, two factors determine how long this will take; the input/output speed of the network communication, or I/O, and the available processor, or CPU.

I/O-intensive processes improved with multithreading:

- webscraping
- reading and writing to files
- sharing data between programs
- network communications

CPU-intensive processes improved with multiprocessing:

- computations
- text formatting
- image rescaling
- data analysis

Multithreading Example: Webscraping

Historically, the programming knowledge required to set up multithreading was beyond the scope of this course, as it involved a good understanding of Python's Global Interpreter Lock (the GIL prevents multiple threads from running the same Python code at once). Also, you had to set up special classes that behave like Producers to divvy up the work, Consumers (aka "workers") to perform the work, and a Queue to hold tasks and provide communcations. And that was just the beginning.

Fortunately, we've already learned one of the most valuable tools we'll need – the `map()` function. When we apply it using two standard libraries, *multiprocessing* and *multiprocessing.dummy*, setting up parallel processes and threads becomes fairly straightforward.

Here's a classic multithreading example provided by IBM (http://www.ibm.com/developerworks/aix/library/au-threadingpython/) and adapted by Chris Kiehl (http://chriskiehl.com/article/parallelism-in-one-line/) where you divide the task of retrieving web pages across multiple threads:

```
import time
import threading
import Queue
import urllib2

class Consumer(threading.Thread):
  def __init__(self, queue):
    threading.Thread.__init__(self)
    self._queue = queue

  def run(self):
    while True:
      content = self._queue.get()
      if isinstance(content, str) and content == 'quit':
        break
      response = urllib2.urlopen(content)
    print 'Thanks!'


def Producer():
  urls = [
    'http://www.python.org', 'http://www.yahoo.com'
    'http://www.scala.org', 'http://www.google.com'
    # etc..
  ]
  queue = Queue.Queue()
  worker_threads = build_worker_pool(queue, 4)
  start_time = time.time()

  # Add the urls to process
  for url in urls:
    queue.put(url)
  # Add the poison pill
  for worker in worker_threads:
    queue.put('quit')
  for worker in worker_threads:
    worker.join()

  print 'Done! Time taken: {}'.format(time.time() - start_time)

def build_worker_pool(queue, size):
  workers = []
  for _ in range(size):
    worker = Consumer(queue)
    worker.start()
    workers.append(worker)
  return workers

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

Using the multithreading library provided by the *multiprocessing.dummy* module and `map()` all of this becomes:

```
import urllib2
from multiprocessing.dummy import Pool as ThreadPool

pool = ThreadPool(4) # choose a number of workers

urls = [
'http://www.python.org', 'http://www.yahoo.com'
'http://www.scala.org', 'http://www.google.com'
# etc..
]

results = pool.map(urllib2.urlopen, urls)
pool.close()
pool.join()
```

In the above code, the *multiprocessing.dummy* module provides the parallel threads, and `map(urllib2.urlopen, urls)` assigns the labor!

Multiprocessing Example: Monte Carlo

Let's code out an example to see how the parts fit together. We can time our results using the *timeit* module to measure any performance gains. Our task is to apply the Monte Carlo Method to estimate the value of Pi.

Monte Carle Method and Estimating Pi

If you draw a circle of radius 1 (a unit circle) and enclose it in a square, the areas of the two shapes are given as

Area Formulas	
circle	πr^2
square	$4 r^2$

Therefore, the ratio of the volume of the circle to the volume of the square is

$$\frac{\pi}{4}$$

The Monte Carlo Method plots a series of random points inside the square. By comparing the number that fall within the circle to those that fall outside, with a large enough sample we should have a good approximation of Pi. You can see a good demonstration of this here (https://academo.org/demos/estimating-pi-monte-carlo/) (Hit the **Animate** button on the page).

For a given number of points *n*, we have

$$\pi = \frac{4 \cdot points\ inside\ circle}{total\ points\ n}$$

To set up our multiprocessing program, we first derive a function for finding Pi that we can pass to `map()` :

In [1]:

```python
from random import random  # perform this import outside the function

def find_pi(n):
    """
    Function to estimate the value of Pi
    """
    inside=0

    for i in range(0,n):
        x=random()
        y=random()
        if (x*x+y*y)**(0.5)<=1:  # if i falls inside the circle
            inside+=1

    pi=4*inside/n
    return pi
```

Let's test `find_pi` on 5,000 points:

In [2]:

```python
find_pi(5000)
```

Out[2]:

```
3.1064
```

This ran very quickly, but the results are not very accurate!

Next we'll write a script that sets up a pool of workers, and lets us time the results against varying sized pools. We'll set up two arguments to represent *processes* and *total_iterations*. Inside the script, we'll break *total_iterations* down into the number of iterations passed to each process, by making a processes-sized list. For example:

```
total_iterations = 1000
processes = 5
iterations = [total_iterations//processes]*processes
iterations
# Output: [200, 200, 200, 200, 200]
```

This list will be passed to our `map()` function along with `find_pi()`

In [3]:

```
%%writefile test.py
from random import random
from multiprocessing import Pool
import timeit

def find_pi(n):
    """
    Function to estimate the value of Pi
    """
    inside=0

    for i in range(0,n):
        x=random()
        y=random()
        if (x*x+y*y)**(0.5)<=1:  # if i falls inside the circle
            inside+=1

    pi=4*inside/n
    return pi

if __name__ == '__main__':
    N = 10**5  # total iterations
    P = 5      # number of processes

    p = Pool(P)
    print(timeit.timeit(lambda: print(f'{sum(p.map(find_pi, [N//P]*P))/P:0.7
f}'), number=10))
    p.close()
    p.join()
    print(f'{N} total iterations with {P} processes')
```

```
Writing test.py
```

In [4]:

```
! python test.py
```

```
3.1466800
3.1364400
3.1470400
3.1370400
3.1256400
3.1398400
3.1395200
3.1363600
3.1437200
3.1334400
0.2370227286270967
100000 total iterations with 5 processes
```

Great! The above test took under a second on our computer.

Now that we know our script works, let's increase the number of iterations, and compare two different pools. Sit back, this may take awhile!

In [5]:

```
%%writefile test.py
from random import random
from multiprocessing import Pool
import timeit

def find_pi(n):
    """
    Function to estimate the value of Pi
    """
    inside=0

    for i in range(0,n):
        x=random()
        y=random()
        if (x*x+y*y)**(0.5)<=1:  # if i falls inside the circle
            inside+=1

    pi=4*inside/n
    return pi

if __name__ == '__main__':
    N = 10**7  # total iterations

    P = 1      # number of processes
    p = Pool(P)
    print(timeit.timeit(lambda: print(f'{sum(p.map(find_pi, [N//P]*P))/P:0.7
f}'), number=10))
    p.close()
    p.join()
    print(f'{N} total iterations with {P} processes')

    P = 5      # number of processes
    p = Pool(P)
    print(timeit.timeit(lambda: print(f'{sum(p.map(find_pi, [N//P]*P))/P:0.7
f}'), number=10))
    p.close()
    p.join()
    print(f'{N} total iterations with {P} processes')
```

```
Overwriting test.py
```

In [6]:

```
! python test.py
```

```
3.1420964
3.1417412
3.1411108
3.1408184
3.1414204
3.1417656
3.1408324
3.1418828
3.1420492
3.1412804
36.03526345242264
10000000 total iterations with 1 processes
3.1424524
3.1418376
3.1415292
3.1410344
3.1422376
3.1418736
3.1420540
3.1411452
3.1421652
3.1410672
17.300921846344366
10000000 total iterations with 5 processes
```

Hopefully you saw that with 5 processes our script ran faster!

More is Better ...to a point.

The gain in speed as you add more parallel processes tends to flatten out at some point. In any collection of tasks, there are going to be one or two that take longer than average, and no amount of added processing can speed them up. This is best described in Amdahl's Law (https://en.wikipedia.org/wiki/Amdahl%27s_law).

Advanced Script

In the example below, we'll add a context manager to shrink these three lines

```
p = Pool(P)
...
p.close()
p.join()
```

to one line:

```
with Pool(P) as p:
```

And we'll accept command line arguments using the *sys* module.

In [7]:

```
%%writefile test2.py
from random import random
from multiprocessing import Pool
import timeit
import sys

N = int(sys.argv[1])  # these arguments are passed in from the command line
P = int(sys.argv[2])

def find_pi(n):
    """
    Function to estimate the value of Pi
    """
    inside=0

    for i in range(0,n):
        x=random()
        y=random()
        if (x*x+y*y)**(0.5)<=1:  # if i falls inside the circle
            inside+=1

    pi=4*inside/n
    return pi

if __name__ == '__main__':

    with Pool(P) as p:
        print(timeit.timeit(lambda: print(f'{sum(p.map(find_pi, [N//P]*P))/P:0.5
f}'), number=10))
    print(f'{N} total iterations with {P} processes')
```

```
Writing test2.py
```

In [8]:

```
! python test2.py 10000000 500
```

```
3.14121
3.14145
3.14178
3.14194
3.14109
3.14201
3.14243
3.14150
3.14203
3.14116
16.871822701405073
10000000 total iterations with 500 processes
```

Great! Now you should have a good understanding of multithreading and multiprocessing!