Report 8 - Parallel Python

May 31, 2020

1 Parallel Python

In this report we will look at performing parallel computation using the Python language. Having covered an introduction to Python programming in Report 7, we extend the concepts introduced to involve efficient parallel computation. In this report we mainly cover *functional* programming, and thus in Part 1 we introduce the concept and use of functional programming.

1.1 Part 1

1.1.1 Functional Programming

Functional programming is simply a *style* of programming, in which we treat functions as objects. That is, we may choose to store functions as variables, pass them as inputs into other functions, or store them in data structures (e.g. we may have a vector or matrix of functions). Adopting this framework, we will focus on writing functions which can be ran in parallel over multiple cores; writing functions which interact and share dependencies is the difficulty in developing a "team" of operations that we distribute over nodes in a cluster or the cores of a processor.

As one might expect, functional programming is not the only way to achieve parallelism in Python. Instead we may choose to implement shared-memory parallelism, or message passing.

1.1.2 Functions as Objects

In this section we will introduce some examples of functional programming in practice. We simply define a function and then treat it as any other object — we store it as a variable, define other variables as transformations of evaluations of our function/object, and provide the function as an input to another function.

Let's define a simple function. Our function will multiply two scalar inputs.

[3]:
```
def mult(a, b):
    """Simple function which returns the multiplication of the inputs"""
    return a * b
```

Below we test that the function works as expected.

[9]:
```
# multiply 1 and 2
mult(1, 2)
```

[9]:
```
2
```

We can easily store evaluations of functions as variables.

[14]:
```
test = mult(1, 2)
print(test)
```

```
2
```

The way this works is rather simple. When we define a variable such as `a = 2`, we are storing `2` in memory and then setting the object `a` to point to this piece of memory. Similarly, if we then set `b = a` we are creating a new variable `b` which points to whatever `a` was pointing to. Thus, in writing `test = mult(1,2)` we are evaluating the function `mult(1,2)`, and `mult` points to the output of the function. Then we are setting the new variable `test` to point to what `mult` is pointing to.

We may define a function which is a transformation of this function. We define another function which returns two times the multiple of the two scalar inputs.

[6]:
```
def mult_II(x, y):
    return 2 * mult(x, y)
```

[9]:
```
mult_II(2, 2)
```

[9]:
```
8
```

We now define a function which takes another function as input. Specifically, this function will return two times the evaluation of the given function.

[10]:
```
def two_times(a, b, function):
    return 2 * function(a, b)
```

We can test the function using the `mult` function defined previously.

[11]:
```
two_times(2, 2, mult)
```

[11]:
```
8
```

We can also define functions on-the-fly using lambdas! Below we use the `two_times` function where the function provided as an input is a lambda function which adds the two inputs. Hence, given inputs x and y the function should return $2(x+y)$.

[12]:
```
two_times(2, 2, lambda x, y: x + y)
```

[12]:
```
8
```

1.1.3 Properties of a Function

As seen in Report 7, object such as integers and strings have properties attached to them which we can access. If we set the variable `a` to be an integer (e.g. `a = 2`) we can access its properties via `a.__[TAB]`. Functions also possess properties which are accessed in the same way. Below we look at some examples of the properties belonging to functions.

[15]:
```
print(mult.__name__)
print(mult.__doc__)
```

```
mult
Simple function which returns the multiplication of the inputs
```

1.1.4 Functions as Arguments

We can also supply functions as arguments to other functions. This approach is commonly used in functional programming. Below is a simple example.

[14]:
```
def call_function(func, arg1, arg2):
    """
    Simple function which calls the first argument `func` with
    arguments `arg1` and `arg2`.
    """
    return func(arg1, arg2)
```

[16]:
```
test = call_function(mult, 1, 2)
print(test)
```

```
2
```

The above function is self-explanatory — it simply calls the function provided as the first argument with the second and third arguments. This may not seem useful in this sense, but it is common to write a function which calls another function within it. For example, suppose we are writing a function to implement Gaussian process regression (GPR). In implementing GPR we must specify the covariance function via a *kernel function*. Our choice of kernel function may impact our results and thus should consider multiple different options. To avoid redefining our GPR function to use a different kernel function for every experiment, we can simply define *one* GPR function which takes a kernel function as input.

Nevertheless, let's proceed by making our `call_function` function a little more useful.

[17]:
```
def call_function(func, arg1, arg2):
    """
    Simple function which calls the first argument `func` with
    arguments `arg1` and `arg2`. Also provides some information on
    the function being called.
    """
    print(f"Calling function {func.__name__} with arguments {arg1} and {arg2}")
    output = func(arg1, arg2)
    print(f"The output is {output}")
```

[18]:
```
test = call_function(mult, 1, 2)
```

```
Calling function mult with arguments 1 and 2
The output is 2
```

1.1.5 Mapping Functions

We often want to apply a single function to an entire set of data. Below we look at using our `mult` function to multiply two vectors element-wise. Of course we could simple reduce this into a single number by summing our outputs (thus computing the inner product).

[23]:
```
a = [2, 4, 6, 8, 10]
b = [1, 3, 5, 7, 9]
```

```
# simply store the output
output = []
for i,j in zip(a, b):
    output.append(mult(i, j))
print(output)
```

```
[2, 12, 30, 56, 90]
```

[29]:
```
# reduce the output i.e. compute the inner product
output = 0
for i, j in zip(a, b):
    output += mult(i, j)
print(output)
```

```
190
```

The above functions looped over pairs of numbers in the vectors `a` and `b` and applied the function `mult` to these pairs of numbers. The first function simply appended the output of `mult` to a list whereas the second function reduced the output to a single number by adding the output of `mult`.

Mapping is very common in practice and thus Python offers an optimized implementation via `map`.

Let's write a simple function which compute the Gaussian/RBF kernel function for a set of points. The Gaussian kernel function is given by

$$K(x, y) = \exp\left(-\frac{\|x - y\|^2}{2\gamma^2}\right),$$

where $\gamma > 0$ is a hyperparameter called the bandwidth or length-scale.

[49]:
```
import math

def gaussian_kernel(x, y, gamma=1):
    """
    Evaluates the gaussian kernel function for scalar inputs x and y,
    and bandwidth parameter gamma.
    """
    return math.exp(-(x-y)**2/(2*gamma**2))
```

We can easily check that the above function works. If $x = y$ then we expect the kernel fucntion to be equal to 1. As the difference between x and y increases, the kernel function's evaluation will also decrease.

[44]:
```
# should be 1
gaussian_kernel(1, 1, 1)
```

[44]:
```
1.0
```

[45]:
```
# should be less than 1
gaussian_kernel(1, 0.5, 1)
```

[45]:
```
0.8824969025845955
```

[46]:
```
# should be less than the previous evaluation
gaussian_kernel(1, 0.1, 1)
```

[46]:
```
0.6669768108584744
```

Let's use Python's built-in function `map` to perform the above test once again.

[62]:
```
a = [1,1,1]
b = [1.0, 0.5, 0.1]

output = map(gaussian_kernel, a, b)
print(output)
```

```
<map object at 0x7fdcb0262400>
```

As seen above we cannot simply print the output as `map` simply points to an object which has not yet been evaluated; this saves unnecessary computation and avoids storing the evaluated list in memory. The obejct returned is known as an iterator. In order to access the results we must turn the object into a list.

[64]:
```
list(output)
```

[64]:
```
[1.0, 0.8824969025845955, 0.6669768108584744]
```

Note that the use of `map` is no longer necessary in Python 3 due to the existence of generator functions.

When working with kernel methods we must compute the Gram matrix K with elements $(K)_{ij} = K(x_i, x_j)$, $i, j = 1, \ldots, n$ for a given dataset $(x_i)_{i=1}^n$. In order to do this we can make use of Python's broadcasting. Note that we will need to use Numpy arrays and Numpy's exponential function in order to make use of broadcasting.

[86]:
```
import numpy as np

x = np.array([-2, -1, 0, 1, 2])

def gaussian_kernel(x, y, gamma=1):
    """
    Evaluates the gaussian kernel function for scalar inputs x and y,
    and bandwidth parameter gamma.
    """
    return np.exp(-(x-y)**2/(2*gamma**2))

m = gaussian_kernel(x[:,None], x)
print(m)
```

```
[[1.00000000e+00 6.06530660e-01 1.35335283e-01 1.11089965e-02
  3.35462628e-04]
 [6.06530660e-01 1.00000000e+00 6.06530660e-01 1.35335283e-01
  1.11089965e-02]
 [1.35335283e-01 6.06530660e-01 1.00000000e+00 6.06530660e-01
```

```
 1.35335283e-01]
[1.11089965e-02 1.35335283e-01 6.06530660e-01 1.00000000e+00
 6.06530660e-01]
[3.35462628e-04 1.11089965e-02 1.35335283e-01 6.06530660e-01
 1.00000000e+00]]
```

We see that we have generated the Gram matrix! In the above we have used some reshaping operations for Numpy arrays. By writing `x[:,None]` we can transform an array into a vertical array. When applying an operation in Python to two arrays which are not of the same dimension, Python will do its best to transform the smaller array to match the dimension of the larger array. This is called broadcasting. In this case, Python will apply the function `gaussian_kernel` with every possible combination of pairs. Below are some examples.

[96]:
```
a = np.array([1, 2, 3, 4])
b = 2
print(a + b)
```

```
[3 4 5 6]
```

In the above we see that the scalar `b` was broascast to a vector `[2, 2, 2, 2]` matching the dimensions of `a`.

[99]:
```
a = np.array([[1, 2,], [3, 4]])
b = np.array([1, 2])
print(a + b)
```

```
[[2 4]
 [4 6]]
```

In the above example we see that the 1×2 array `b` was broadcast to be a 2×2 matrix with rows equal to `b`.

1.1.6 Lambda Functions

So far we have defined functions and then provided these functions as arguments to other functions. However we can save time by using lambda functions. These are anonymous functions which we write on-the-go; we do not need to explicitly define them and assign them to a variable.

Below we will use a lambda to create an anonymous function which has the same behaviour as `mult` defined above.

[100]:
```
a = [1,2,3]
b = [2,2,2]
print(list(map(lambda x, y: x*y, a, b)))
```

```
[2, 4, 6]
```

Lambdas may be difficult to read initially. Their general syntax is as follows:

```
lambda arguments: expression
```

Which can also be written as

```
def name(arguments):
    return expression
```

Note however that this version assigns a function to `name` whereas lambdas do not assign the function to a variable.

2 Multicore (local) Parallel Programming

Now that we understand the basics of functional programming, we will use what we have learned to parallelize a Python script. There are many libraries that offer parallelism in Python, and in this report we will look at `multiprocessing`.

[104]:
```
import multiprocessing
```

You can easily access the documentation of the module by typing

```
help(multiprocessing)
```

A useful function to know in the `multiprocessing` module is `cpu_count`. This returns the number of cores your local machine has.

[106]:
```
print(multiprocessing.cpu_count())
```

```
8
```

We see that the machine which compiled this report has 8 cores, however this is also including cores which are simulated by Intel's multithreading implementation. In reality this machine has 4 physical cores upon which it can perform 4 operations simultaneously.

Multiprocessing allows you to parallelise code by running multiple copies of your script on different processor cores. One of the copies is known as the master copy and controls all other worker copies. Due to this, in order to use multiprocessing we must write a script and run it using the python interpreter in the terminal. As parallelism in this case works by running multiple copies of your script, your script must follow a certain layout to ensure that all workers have access to the same libraries and functions. A typical layout is to import modules at the beginning of the scripts, followed by all function and class definitions. Then we ensure that only the master copy of the script runs the code, and assigns operations to other workers.

2.1 Pool

A key feature of `multiprocessing` is `multiprocessing.Pool`, which provides a pool of workers that can parallelize a `map`. Below is a script which we will save as `pool.py`. Note that below we make use of IPython's built-in magic commands.

```
# %load pool.py
from functools import reduce
from multiprocessing import Pool, cpu_count

def square(x):
    """Function to return the square of the argument"""
    return x * x
```

```
if __name__ == "__main__":
    # print the number of cores
    print(f"Number of cores available equals {cpu_count()}")

    # create a pool of workers
    with Pool() as pool:
        # create an array of 5000 integers, from 1 to 5000
        r = range(1, 5001)

        result = pool.map(square, r)

    total = reduce(lambda x, y: x + y, result)

    print(f"The sum of the square of the first 5000 integers is {total}")
```

[121]:
```
%run pool.py
```

```
Number of cores available equals 8
The sum of the square of the first 5000 integers is 41679167500
```

The above code works in the following way:

- The line `with Pool() as pool:` creates a pool of workers which each have access to the script. The number of workers in pool is equal to the number returned by `multithreading.cpu_count()`, but can be changed manually by setting the `processes` variable within `Pool()` e.g. `with Pool(processes = 2) as pool:`.
- The line `result = pool.map(square, r)` performs the parallelism. The `map` operation is divided up among the workers in the pool. That is, if you have two workers then each worker will conduct half of the work. If you have four then each will conduct a quarter of the work.

We can write a similar script which explicitly tells us what worker is performing each operation, using `multithreading.current_process()`. The script below is saved as `pool_v2.py`

```
# %load pool_v2.py
from functools import reduce
from multiprocessing import Pool, current_process

def square(x):
    """Function to return the square of the argument"""
    print(f"Worker {current_process().pid} calculating square of {x}")
    return x * x

if __name__ == "__main__":
    nprocs = 8
```

```
    # print the number of cores
    print(f"Number of workers equals {nprocs}")

    # create a pool of workers
    with Pool(processes=nprocs) as pool:
        # create an array of 5000 integers, from 1 to 5000
        r = range(1, 21)

        result = pool.map(square, r)

    total = reduce(lambda x, y: x + y, result)

    print(f"The sum of the square of the first 20 integers is {total}")
```

[152]:
```
%run pool_v2.py
```

```
Number of workers equals 8
Worker 17593 calculating square of 5
Worker 17589 calculating square of 1
Worker 17592 calculating square of 4
Worker 17590 calculating square of 2
Worker 17594 calculating square of 6
Worker 17595 calculating square of 7
Worker 17593 calculating square of 8
Worker 17591 calculating square of 3
Worker 17595 calculating square of 13
Worker 17590 calculating square of 14
Worker 17596 calculating square of 10
Worker 17593 calculating square of 15
Worker 17594 calculating square of 11
Worker 17592 calculating square of 12
Worker 17594 calculating square of 17
Worker 17592 calculating square of 19
Worker 17596 calculating square of 18
Worker 17595 calculating square of 16
Worker 17593 calculating square of 20
Worker 17589 calculating square of 9
The sum of the square of the first 20 integers is 2870
```

We can also use multiple `pools` in a single script, but you must ensure that all required definitions are placed *before* the line `with Pool() as pool:`.

There are a number of issues to overcome when using the functions provided by `multithreading` as they do not always function the same way as Python's built-in functions. For example, the `map` function can only work with functions which take one argument. We can overcome this problem by using `zip` to combine two lists into one list of tuples and using the `pool.starmap` function.

Another problem you may encounter is that we cannot use lambda functions. There are many ways that you may overcome this. You can simply define all of your functions explicitly, or you

can investigate other libraries (forks of `multiprocessing`) which can convert lambda functions to binary operations.

2.2 Asynchronous Functions and Futures

The `pool.map` function tells each worker to run a certain function, but we may want different workers to perform different operations. We can achieve this by using `pool.apply` which *applies* functions to different workers. Below we will demostrate this.

The Python script below is saved as `poolapply.py`.

"'python # %load poolapply.py import time from multiprocessing import Pool, current_process

def delayed_add(x, y, nsecs): """ This function waits nsecs and then adds x and y. """

```
print(f"This is worker {current_process().pid} going to sleep for {nsecs} second(s)")

time.sleep(nsecs)

print(f"This is worker {current_process().pid} waking up and adding {x} and {y}")

return x + y
```

if **name** == "**main**":

```
print(f"This is the Master process {current_process().pid}")

with Pool() as pool:
    r1 = pool.apply(delayed_add, [2, 3, 2])
    r2 = pool.apply(delayed_add, [1, -1, 1])

print(f"The results are {r1} and {r2}.")
```

[16]:
```
%run poolapply.py
```

```
This is the Master process 10074
This is worker 10611 going to sleep for 2 second(s)
This is worker 10611 waking up and adding 2 and 3
This is worker 10612 going to sleep for 1 second(s)
This is worker 10612 waking up and adding 1 and -1
The results are 5 and 0.
```

Above is the result of running the script. We see that in this cast the parallelism is achieved by the line

```
r1 = pool.apply(delayed_add, [2, 3, 2])
```

which tells the master process to assign one worker to run the function `delayed_add` with arguments `[2, 3, 2]`. In this case you will notice that the master process does not designate other

tasks until the worker has finished its task. We can overcome this with the use of asynchronous functions.

We replace `pool.apply` with `pool.apply_async` and run the resulting script saved as `poolapplyasync.py`. You will notice that there are a few other things which we need to be careful of. Firstly, we must tell the master process to wait until our computations have finished before it prints the outputs. This is achieve by `r1.wait()` and `r2.wait()`. Also, in order to access the results we must use `r1.get()` and `r2.get()`.

```
# %load poolapplyasync.py
import time
from multiprocessing import Pool, current_process

def delayed_add(x, y, nsecs):
    """
    This function waits nsecs and then adds x and y.
    """

    print(f"This is worker {current_process().pid} going to sleep for {nsecs} second(s)")

    time.sleep(nsecs)

    print(f"This is worker {current_process().pid} waking up and adding {x} and {y}")

    return x + y

if __name__ == "__main__":

    print(f"This is the Master process {current_process().pid}")

    with Pool() as pool:
        r1 = pool.apply_async(delayed_add, [2, 3, 2])
        r2 = pool.apply_async(delayed_add, [1, -1, 1])
        r1.wait()
        r2.wait()
    print(f"The results are {r1.get()} and {r2.get()}.")
```

[17]:
```
%run poolapplyasync.py
```

```
This is the Master process 10074
This is worker 10725 going to sleep for 1 second(s)
This is worker 10724 going to sleep for 2 second(s)
This is worker 10725 waking up and adding 1 and -1
This is worker 10724 waking up and adding 2 and 3
The results are 5 and 0.
```

2.3 Asynchronous Mapping

Just as `multiprocessing` provides asyncronous functions, it also provides asyncronous maps. Recall that asynchronous simply refers to the fast that the master process is not blocked whilst

the pool of workers are carrying out computations. A simple example can be found below.

```
# %load asyncmap.py
import time
from functools import reduce
from multiprocessing import Pool, current_process

def delayed_add(x, y):
    """
    This function sleeps for 1 second and then adds x and y
    """

    print(f"This is worker {current_process().pid} performing operation delayed_add({x}, {y})")

    time.sleep(1)

    return x + y

def delayed_subtract(x, y):
    """
    This function sleeps for 1 second and then subtracts x and y
    """

    time.sleep(1)

    print(f"This is worker {current_process().pid} performing operation delayed_subtract({x}, 

    return x - y

if __name__ == "__main__":

    a = range(1, 6)
    b = range(6, 11)

    with Pool() as pool:
        r1 = pool.starmap_async(delayed_add, zip(a, b))
        r2 = pool.starmap_async(delayed_subtract, zip(a, b))
        r1.wait()
        r2.wait()

    total_sum = reduce(lambda x, y: x + y, r1.get())
    total_diff = reduce(lambda x, y: x + y, r2.get())

    print(f"a = {list(a)} \nb = {list(b)}")
    print(f"The total sum of a and b is {total_sum}")
    print(f"The sum of element-wise differences between a and b is {total_diff}")
```

[163]:
```
%run asyncmap.py
```

```
This is worker 18915 performing operation delayed_add(2, 7)
This is worker 18908 performing operation delayed_add(1, 6)
This is worker 18911 performing operation delayed_add(4, 9)
This is worker 18912 performing operation delayed_add(5, 10)
This is worker 18909 performing operation delayed_add(3, 8)
This is worker 18913 performing operation delayed_subtract(1, 6)
This is worker 18910 performing operation delayed_subtract(2, 7)
This is worker 18914 performing operation delayed_subtract(3, 8)
This is worker 18915 performing operation delayed_subtract(4, 9)
This is worker 18908 performing operation delayed_subtract(5, 10)
a = [1, 2, 3, 4, 5]
b = [6, 7, 8, 9, 10]
The total sum of a and b is 55
The sum of element-wise differences between a and b is -25
```

Dividing simple operations across processor cores can be rather inefficient if the overhead of assigning the operation is greater than that of the actual computation. This is the case with the above operations — the cost of assigning a single addition operation to a worker is far greater than the computational cost of assigning a task. To overcome this problem we can assign a *chunk* of operations to a worker.

We can do this by specifying the `chunksize` in the `pool` function. You can find an example below.

```python
# %load asyncmap-chunks.py
import time
from functools import reduce
from multiprocessing import Pool, current_process

def delayed_add(x, y):
    """
    This function sleeps for 1 second and then adds x and y
    """

    print(f"This is worker {current_process().pid} performing operation delayed_add({x}, {y})"

    time.sleep(1)

    return x + y

def delayed_subtract(x, y):
    """
    This function sleeps for 1 second and then subtracts x and y
    """

    time.sleep(1)

    print(f"This is worker {current_process().pid} performing operation delayed_subtract({x}, -
```

```
    return x - y

if __name__ == "__main__":

    a = range(1, 6)
    b = range(6, 11)

    with Pool() as pool:
        r1 = pool.starmap_async(delayed_add, zip(a, b), chunksize=3)
        r2 = pool.starmap_async(delayed_subtract, zip(a, b), chunksize=3)
        r1.wait()
        r2.wait()

    total_sum = reduce(lambda x, y: x + y, r1.get())
    total_diff = reduce(lambda x, y: x + y, r2.get())

    print(f"a = {list(a)} \nb = {list(b)}")
    print(f"The total sum of a and b is {total_sum}")
    print(f"The sum of element-wise differences between a and b is {total_diff}")
```

[169]:
```
%run asyncmap-chunks.py
```

```
This is worker 19149 performing operation delayed_add(4, 9)
This is worker 19148 performing operation delayed_add(1, 6)
This is worker 19148 performing operation delayed_add(2, 7)
This is worker 19150 performing operation delayed_subtract(1, 6)
This is worker 19151 performing operation delayed_subtract(4, 9)
This is worker 19149 performing operation delayed_add(5, 10)
This is worker 19148 performing operation delayed_add(3, 8)
This is worker 19151 performing operation delayed_subtract(5, 10)
This is worker 19150 performing operation delayed_subtract(2, 7)
This is worker 19150 performing operation delayed_subtract(3, 8)
a = [1, 2, 3, 4, 5]
b = [6, 7, 8, 9, 10]
The total sum of a and b is 55
The sum of element-wise differences between a and b is -25
```