Parallel Processing in Python

Parallel processing can increase the number of tasks done by our program which reduces the overall processing time. These help to handle large scale problems.

Introduction to parallel processing

For parallelism, it is important to divide the problem into sub-units that do not depend on other sub-units (or less dependent). A problem where the sub-units are totally independent of other sub-units is called embarrassingly parallel.

For example, An element-wise operation on an array. In this case, the operation needs to aware of the particular element it is handling at the moment.

In another scenario, a problem which is divided into sub-units have to share some data to perform operations. These results in the performance issue because of the communication cost.

There are two main ways to handle parallel programs:

Shared Memory

In shared memory, the sub-units can communicate with each other through the same memory space. The advantage is that you don't need to handle the communication explicitly because this approach is sufficient to read or write from the shared memory. But the problem arises when multiple process access and change the same memory location at the same time. This conflict can be avoided using synchronization techniques.

Distributed memory

In distributed memory, each process is totally separated and has its own memory space. In this scenario, communication is handled explicitly between the processes. Since the communication happens through a network interface, it is costlier compared to shared memory.

Threads are one of the ways to achieve parallelism with shared memory. These are the independent sub-tasks that originate from a process and share memory.
Due to **Global Interpreter Lock** (GIL), threads can't be used to increase performance in Python. GIL is a mechanism in which Python interpreter design allow only one Python instruction to run at a time. GIL limitation can be completely avoided by using processes instead of thread.
Using processes have few disadvantages such as less efficient inter-process communication than shared memory, but it is more flexible and explicit.

Multithreading

```
from threading import Thread,current_thread
print(current_thread().getName())
def mt():
    print("Child Thread")
    for i in range(11,15):
            print(i*2)
def disp():
   for i in range(5):
         print(i*2)
child=Thread(target=mt)
child.start()
disp()
print("Executing thread name :",current_thread().getName())
```

Output:

```
MainThread
Child Thread
22
24
26
28
0
2
4
6
8
```

Program:

```
from threading import Thread,current_thread
class mythread(Thread):
    def run(self):
        for x in range(7):
            print("Hi from child")
a = mythread()
a.start()
a.join()
print("Bye from",current_thread().getName())
```

Output:

```
Hi from child
Hi from child
Hi from child
Hi from child
Hi from child
Hi from child
Hi from child
Bye from MainThread
```

Multiprocessing for parallel processing

Using the standard multiprocessing module, we can efficiently parallelize simple tasks by creating child processes. This module provides an easy-to-use interface and contains a set of utilities to handle task submission and synchronization.

Process and Pool Class

Process

By subclassing multiprocessing.process, we can create a process that runs independently. By extending the __init__ method we can initialize resource and by implementing Process.run() method we can write the code for the subprocess.
To spawn the process, we need to initialize our Process object and invoke Process.start() method. Here Process.start() will create a new process and will invoke the Process.run() method.
The code after p.start() will be executed immediately before the task completion of process p. To wait for the task completion, we can use Process.join().
Program:

```
import multiprocessing
import time
class Process(multiprocessing.Process):
    def __init__(self, id):
        super(Process, self).__init__()
        self.id = id

    def run(self):
        time.sleep(1)
        print("I'm the process with id: {}".format(self.id))

if __name__ == '__main__':
    p = Process(0)
    p.start()
    p.join()
    p = Process(1)
    p.start()
    p.join()
```

Output:

```
I'm the process with id: 0
I'm the process with id: 1
```

Pool class

Pool class can be used for parallel execution of a function for different input data. The multiprocessing.Pool() class spawns a set of processes called workers and can submit tasks using the methods apply/apply_async and map/map_async. For parallel mapping, you should first initialize a multiprocessing.Pool() object. The first argument is the number of workers; if not given, that number will be equal to the number of cores in the system.
In this example, we will see how to pass a function which computes the square of a number. Using Pool.map() we can map the function to the list and passing the function and the list of inputs as arguments, as follows:

```
import multiprocessing
import time
```

```
def square(x):
        return x * x

if __name__ == '__main__':
        pool = multiprocessing.Pool()
        pool = multiprocessing.Pool(processes=4)
        inputs = [0,1,2,3,4]
        outputs = pool.map(square, inputs)
        print("Input: {}".format(inputs))
        print("Output: {}".format(outputs))
```

Output

```
        Input: [0, 1, 2, 3, 4]
        Output: [0, 1, 4, 9, 16]
```

When we use the normal map method, the execution of the program is stopped until all the workers completed the task. Using map_async(), the AsyncResult object is returned immediately without stopping the main program and the task is done in the background. The result can be retrieved by using the AsyncResult.get() method at any time as shown below:

```
import multiprocessing
import time
def square(x):
        return x * x
if __name__ == '__main__':
        pool = multiprocessing.Pool()
        inputs = [0,1,2,3,4]
        outputs_async = pool.map_async(square, inputs)
        outputs = outputs_async.get()
        print("Output: {}".format(outputs))
```

Output:

```
        Output: [0, 1, 4, 9, 16]
```

Pool.apply_async assigns a task consisting of a single function to one of the workers. It takes the function and its arguments and returns an AsyncResult object.

```
import multiprocessing
import time
def square(x):
        return x * x
if __name__ == '__main__':
        pool = multiprocessing.Pool()
        result_async = [pool.apply_async(square, args = (i, )) for i in
                                        range(10)]
        results = [r.get() for r in result_async]
        print("Output: {}".format(results))
```

Output

```
        Output: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Concurrent Programming Paradigm

Threads vs. Processes

A process is a basic operating system abstraction. It is a program that is in execution—in other words, code that is running. Multiple processes are always running in a computer, and they are executing in parallel.

A process can have multiple threads. They execute the same code belonging to the parent process. Ideally, they run in parallel, but not necessarily. The reason why processes aren't enough is because applications need to be responsive and listen for user actions while updating the display and saving a file.

PROCESSES	THREADS
Processes don't share memory	Threads share memory
Spawning/switching processes is expensive	Spawning/switching threads is less expensive
Processes require more resources	Threads require fewer resources (are sometimes called lightweight processes)
No memory synchronization needed	We need to use synchronisation mechanisms to be sure we're correctly handling the data

Concurrent Programming

Concurrency implies scheduling independent code to be executed in a cooperative manner. Take advantage of the fact that a piece of code is waiting on I/O operations, and during that time run a different but independent part of the code.

In Python, we can achieve lightweight concurrent behaviour via greenlets. From a parallelization perspective, using threads or greenlets is equivalent because neither of them runs in parallel. Greenlets are even less expensive to create than threads. Because of that, greenlets are heavily used for performing a huge number of simple I/O tasks, like the ones usually found in networking and web servers.

Now that we know the difference between threads and processes, parallel and concurrent, we can illustrate how different tasks are performed on the two paradigms. Here's what we're going to do: we will run, multiple times, a task outside the GIL and one inside it. We're running them serially, using threads and using processes. Let's define the tasks:

Program

```
import os
import time
import threading
import multiprocessing
 NUM_WORKERS = 4
 def only_sleep():
    """ Do nothing, wait for a timer to expire """
    print("PID: %s, Process Name: %s, Thread Name: %s" % (
        os.getpid(),
        multiprocessing.current_process().name,
        threading.current_thread().name)
    )
    time.sleep(1)



def crunch_numbers():
    """ Do some computations """
    print("PID: %s, Process Name: %s, Thread Name: %s" % (
        os.getpid(),
        multiprocessing.current_process().name,
        threading.current_thread().name)
    )
    x = 0
    while x < 10000000:
        x += 1
## Run tasks serially
start_time = time.time()
for _ in range(NUM_WORKERS):
    only_sleep()
end_time = time.time()
 print("Serial time=", end_time - start_time)
 # Run tasks using threads
start_time = time.time()
threads = [threading.Thread(target=only_sleep) for _ in range(NUM_WORKERS)]
[thread.start() for thread in threads]
```

```
[thread.join() for thread in threads]
end_time = time.time()
print("Threads time=", end_time - start_time)
# Run tasks using processes
start_time = time.time()
processes = [multiprocessing.Process(target=only_sleep()) for _in range(NUM_WORKERS)]
[process.start() for process in processes]
[process.join() for process in processes]
end_time = time.time()
print("Parallel time=", end_time - start_time)
```

Output

```
PID: 95726, Process Name: MainProcess, Thread Name: MainThread
PID: 95726, Process Name: MainProcess, Thread Name: MainThread
PID: 95726, Process Name: MainProcess, Thread Name: MainThread
PID: 95726, Process Name: MainProcess, Thread Name: MainThread
Serial time= 4.018089056015015
PID: 95726, Process Name: MainProcess, Thread Name: Thread-1
PID: 95726, Process Name: MainProcess, Thread Name: Thread-2
PID: 95726, Process Name: MainProcess, Thread Name: Thread-3
PID: 95726, Process Name: MainProcess, Thread Name: Thread-4
Threads time= 1.0047411918640137
PID: 95728, Process Name: Process-1, Thread Name: MainThread
PID: 95729, Process Name: Process-2, Thread Name: MainThread
PID: 95730, Process Name: Process-3, Thread Name: MainThread
PID: 95731, Process Name: Process-4, Thread Name: MainThread
Parallel time= 1.014023780822754
```

We've created two tasks. Both of them are long-running, but only crunch_numbers actively perform computations. Let's run only_sleep serially, multithreaded and using multiple processes and compare the results:

Here are some observations:

In the case of the serial approach, we're running the tasks one after the other. All four runs are executed by the same thread of the same process.

Using processes we cut the execution time down to a quarter of the original time, simply because the tasks are executed in parallel. Notice how each task is performed in a different process and on the MainThread of that process.

Using threads we take advantage of the fact that the tasks can be executed concurrently. The execution time is also cut down to a quarter, even though nothing is running in parallel. Here's how that goes: we spawn the first thread and it starts waiting for the timer to expire. We pause its execution, letting it wait for the timer to expire, and in this time, we spawn the second thread. We repeat this for all the threads. At one moment the timer of the first thread expires so we switch execution to it and we terminate it. The algorithm is repeated for the second and for all the other threads. At the end, the result is as if things were run in parallel. The four different threads branch out from and live inside the same process: MainProcess.

The threaded approach is quicker than the truly parallel one. That's due to the overhead of spawning processes. As we noted previously, spawning and switching processes is an expensive operation.

Let's do the same routine but this time running the crunch_numbers task:

Program

```
start_time = time.time()
for _ in range(NUM_WORKERS):
    crunch_numbers()
end_time = time.time()
print("Serial time=", end_time - start_time)
start_time = time.time()
threads = [threading.Thread(target=crunch_numbers) for _ in range(NUM_WORKERS)]
[thread.start() for thread in threads]
[thread.join() for thread in threads]
end_time = time.time()
print("Threads time=", end_time - start_time)
start_time = time.time()
processes = [multiprocessing.Process(target=crunch_numbers) for _ in range(NUM_WORKERS)]
[process.start() for process in processes]
```

```
[process.join() for process in processes]
end_time = time.time()
print("Parallel time=", end_time - start_time)
```

Output

```
PID: 96285, Process Name: MainProcess, Thread Name: MainThread
PID: 96285, Process Name: MainProcess, Thread Name: MainThread
PID: 96285, Process Name: MainProcess, Thread Name: MainThread
PID: 96285, Process Name: MainProcess, Thread Name: MainThread
Serial time= 2.705625057220459
PID: 96285, Process Name: MainProcess, Thread Name: Thread-1
PID: 96285, Process Name: MainProcess, Thread Name: Thread-2
PID: 96285, Process Name: MainProcess, Thread Name: Thread-3
PID: 96285, Process Name: MainProcess, Thread Name: Thread-4
Threads time= 2.6961309909820557
PID: 96289, Process Name: Process-1, Thread Name: MainThread
PID: 96290, Process Name: Process-2, Thread Name: MainThread
PID: 96291, Process Name: Process-3, Thread Name: MainThread
PID: 96292, Process Name: Process-4, Thread Name: MainThread
Parallel time= 0.8014059066772461
```

The main difference here is in the result of the multithreaded approach. This time it performs very similarly to the serial approach, and here's why: since it performs computations and Python doesn't perform real parallelism, the threads are basically running one after the other, yielding execution to one another until they all finish.

The Python Parallel/Concurrent Programming Ecosystem

Python has rich APIs for doing parallel/concurrent programming.

threading: The standard way of working with threads in Python. It is a higher-level API wrapper over the functionality exposed by the _thread module, which is a low-level interface over the operating system's thread implementation.
concurrent. futures: A module part of the standard library that provides an even higher-level abstraction layer over threads. The threads are modelled as asynchronous tasks.
multiprocessing: Similar to the threading module, offering a very similar interface but using processes instead of threads.
gevent and greenlets: Greenlets, also called micro-threads, are units of execution that can be scheduled collaboratively and can perform tasks concurrently without much overhead.
celery: A high-level distributed task queue. The tasks are queued and executed concurrently using various paradigms like multiprocessing or gevent.

Concurrent.futures

The concurrent.futures module provides a high-level interface for asynchronously executing callables.

The asynchronous execution can be performed with threads, using ThreadPoolExecutor, or separate processes, using ProcessPoolExecutor. Both implement the same interface, which is defined by the abstract Executor class.

Executor Objects

class concurrent.futures.Executor

An abstract class that provides methods to execute calls asynchronously. It should not be used directly, but through its concrete subclasses.

submit(fn, /, *args, **kwargs)

Schedules the callable, fn, to be executed as fn(*args, **kwargs) and returns a Future object representing the execution of the callable.

```
with ThreadPoolExecutor(max_workers=1) as executor:
    future = executor.submit(pow, 323, 1235)
```

```
print(future.result())
```

map(func, *iterables, timeout=None, chunksize=1)

Similar to map(func, *iterables) except:

- the iterables are collected immediately rather than lazily;
- func is executed asynchronously and several calls to func may be made concurrently.

shutdown(wait=True, *, cancel_futures=False)

Signal the executor that it should free any resources that it is using when the currently pending futures are done executing. Calls to Executor.submit() and Executor.map() made after shutdown will raise RuntimeError.

If wait is True then this method will not return until all the pending futures are done executing and the resources associated with the executor have been freed. If wait is False then this method will return immediately and the resources associated with the executor will be freed when all pending futures are done executing. Regardless of the value of wait, the entire Python program will not exit until all pending futures are done executing.

If cancel_futures is True, this method will cancel all pending futures that the executor has not started running. Any futures that are completed or running won't be cancelled, regardless of the value of cancel_futures.

If both cancel_futures and wait are True, all futures that the executor has started running will be completed prior to this method returning. The remaining futures are cancelled.

Example

```
import shutil
with ThreadPoolExecutor(max_workers=4) as e:
    e.submit(shutil.copy, 'src1.txt', 'dest1.txt')
    e.submit(shutil.copy, 'src2.txt', 'dest2.txt')
    e.submit(shutil.copy, 'src3.txt', 'dest3.txt')
    e.submit(shutil.copy, 'src4.txt', 'dest4.txt')
```

ThreadPoolExecutor

ThreadPoolExecutor is an Executor subclass that uses a pool of threads to execute calls asynchronously.

Example

```
import time
```

```
def wait_on_b():
   time.sleep(5)
   print(b.result())  # b will never complete because it is waiting on a.
   return 5
def wait_on_a():
   time.sleep(5)
   print(a.result())  # a will never complete because it is waiting on b.
   return 6
executor = ThreadPoolExecutor(max_workers=2)
a = executor.submit(wait_on_b)
b = executor.submit(wait_on_a)
def  wait_on_future():
   f = executor.submit(pow, 5, 2)
   # This will never complete because there is only one worker thread and
   # it is executing this function.
   print(f.result())
executor = ThreadPoolExecutor(max_workers=1)
executor.submit(wait_on_future)
class concurrent.futures.ThreadPoolExecutor(max_workers=None,
thread_name_prefix='', initializer=None, initargs=())
```

ProcessPoolExecutor

The ProcessPoolExecutor class is an Executor subclass that uses a pool of processes to execute calls asynchronously. ProcessPoolExecutor uses the multiprocessing module, which allows it to side-step the Global Interpreter Lock but also means that only picklable objects can be executed and returned.

The __main__ module must be importable by worker subprocesses. This means that ProcessPoolExecutor will not work in the interactive interpreter.

Calling Executor or Future methods from a callable submitted to a ProcessPoolExecutor will result in deadlock.

```
class concurrent.futures.ProcessPoolExecutor(max_workers=None,
mp_context=None, initializer=None, initargs=())
```

Example

```
import concurrent.futures
import math

PRIMES = [
    112272535095293,
    112582705942171,
    112272535095293,
    115280095190773,
    115797848077099,
    1099726899285419]
def is_prime(n):
    if n < 2:
        return False
    if n == 2:
        return True
    if n % 2 == 0:
        return False

    sqrt_n = int(math.floor(math.sqrt(n)))
    for i in range(3, sqrt_n + 1, 2):
        if n % i == 0:
            return False
    return True

def main():
    with concurrent.futures.ProcessPoolExecutor() as executor:
        for number, prime in zip(PRIMES, executor.map(is_prime, PRIMES)):
            print('%d is prime: %s' % (number, prime))

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

FUNCTIONAL PROGRAMMING PARADIGM

Introduction

- Functional programming is a programming paradigm in which we try to bind everything in pure mathematical functions style.
- It is a declarative type of programming style.
- Its main focus is on "what to solve" in contrast to an imperative style where the main focus is "how to solve".
- It uses expressions instead of statements.
- An expression is evaluated to produce a value whereas a statement is executed to assign variables.

Functional Programming is based on Lambda Calculus:

- Lambda calculus can be called as the smallest programming language in the world.
- It gives the definition of what is computable.
- Anything that can be computed by lambda calculus is computable.
- It is equivalent to Turing machine in its ability to compute.

Programming Languages that support functional programming: Haskell, JavaScript, Python, Scala, Erlang, Lisp, ML, Clojure, OCaml, Common Lisp, Racket.

Concepts of functional programming:

- Pure functions
- Recursion
- Referential transparency
- Functions are First-Class and can be Higher-Order
- Variables are Immutable

Pure functions:

1) They always produce the same output for same arguments irrespective of anything else.
2) They have no side-effects i.e. they do not modify any arguments or local/global variables or input/output streams.
3) Immutable: The pure function's only result is the value it returns. They are deterministic.

Programs done using functional programming are easy to debug because pure functions have no side effects or hidden I/O. Pure functions also make it easier to write parallel/concurrent applications.

Example:

```
sum(x, y)          // sum is function taking x and y as arguments
   return x + y    // sum is returning sum of x and y without changing them
```

Recursion: There are no "for" or "while" loop in functional languages. Iteration in functional languages is implemented through recursion. Recursive functions repeatedly call themselves, until it reaches the base case.

Example:

```
fib(n)
   if (n <= 1)
      return 1;
```

```
        else
           return fib(n - 1) + fib(n - 2);
```

Referential transparency: In functional programs variables once defined do not change their value throughout the program. Functional programs do not have assignment statements. If we have to store some value, we define new variables instead. This eliminates any chances of side effects because any variable can be replaced with its actual value at any point of execution. State of any variable is constant at any instant.

Example:

```
x = x + 1 // this changes the value assigned to the variable x.
          // So the expression is not referentially transparent.
```

Functions are **First-Class** and can be **Higher-Order**: First-class functions are treated as first-class variable. The **first class** variables can be passed to functions as parameter, can be returned from functions or stored in data structures. **Higher order functions** are the functions that take other functions as arguments and they can also return functions.

Variables are Immutable: In functional programming, we can't modify a variable after it's been initialized. We can create new variables – but we can't modify existing variables, and this really helps to maintain state throughout the runtime of a program. Once we create a variable and set its value, we can have full confidence knowing that the value of that variable will never change.

Advantages and Disadvantages of Functional programming

Advantages:

1) Pure functions are easier to understand because they don't change any states and depend only on the input given to them.
2) The ability of functional programming languages to treat functions as values and pass them to functions as parameters make the code more readable and easily understandable.
3) Testing and debugging is easier. Since pure functions take only arguments and produce output, they don't produce any changes don't take input or produce some hidden output. They use immutable values, so it becomes easier to check some problems in programs written uses pure functions.
4) It is used to implement concurrency/parallelism because pure functions don't change variables or any other data outside of it.
5) It adopts lazy evaluation which avoids repeated evaluation because the value is evaluated and stored only when it is needed.

Disadvantages:

1) Sometimes writing pure functions can reduce the readability of code.
2) Writing programs in recursive style instead of using loops can be bit intimidating.
3) Writing pure functions are easy but combining them with the rest of the application and I/O operations is a difficult task.
4) Immutable values and recursion can lead to decrease in performance.

1). The map() function:

The map() function is a higher-order function. As previously stated, this function accepts another function and a sequence of 'iterables' as parameters and provides output after applying the function to each iterable in the sequence. It has the following syntax:

SYNTAX: map(function, iterables)

The function is used to define an expression which is then applied to the 'iterables'. User-defined functions and lambda functions can both be sent to the map function.
User-defined functions can be sent to the map() method. The user or programmer is the only one who can change the parameters of these functions.

EXAMPLE

```
def function(a):
   return a*a
x = map(function, (1,2,3,4))  #x is the map object
print(x)
print(set(x))
```

OUTPUT

```
{16, 1, 4, 9}
```

x is a map object, as you can see. The map function is displayed next, which takes "function()" as a parameter and then applies "a * a" to all 'iterables'. As a result, all iterables' values are multiplied by themselves before being returned.

Lambda within map() functions:

Functions with no name are known as lambda functions. These functions are frequently used as input to other functions. Let's try to integrate Lambda functions into the map() function.

EXAMPLE

```
tup= (5, 7, 22, 97, 54, 62, 77, 23, 73, 61)
newtuple = tuple(map(lambda x: x+3 , tup))
print(newtuple)
```

OUTPUT

```
(8, 10, 25, 100, 57, 65, 80, 26, 76, 64)
```

The filter() function:
The filter() function is used to generate an output list of values that return true when the function is called. It has the following syntax:

SYNTAX: filter (function, iterables)

This function like map(), can take user-defined functions and lambda functions as parameters.

EXAMPLE

```
def func(x):
   if x>=3:
      return x
y = filter(func, (1,2,3,4))
print(y)
print(list(y))
```

OUTPUT

```
[3, 4]
```

As you can see, y is the filter object, and the list is a collection of true values for the condition (x>=3).

Lambda within filter() functions:

The condition to be checked is defined by the lambda function that is provided as an argument.

EXAMPLE

```
y = filter(lambda x: (x>=3), (1,2,3,4))
print(list(y))
```

OUTPUT

```
[3, 4]
```

The reduce() function:

The reduce() function applies a provided function to 'iterables' and returns a single value, as the name implies.

SYNTAX: reduce(function, iterables)

The function specifies which expression should be applied to the 'iterables' in this case. The function tools module must be used to import this function.

EXAMPLE

```
from functools import reduce
reduce(lambda a,b: a+b,[23,21,45,98])
```

OUTPUT

```
187
```

The reduce function in the preceding example adds each iterable in the list one by one and returns a single result.

Partial Functions in functools Module of Python

Functools is a standard library module of Python. With its help, we can extend the utility of the functions or callable objects without rewriting them. The purpose of functools is to make functional programming more agile. It makes working with high-order functions effortless.

Currently, functools contain these 11 functions.

cmp_to_key()
cached_property()
lru_catch()
partial()
partialmethod()
reduce()
singledispatch()
singledispatchmethod()
total_ordering()
update_wrapper()
wraps()

Introduction to partial function

functools partial is a function that takes a function as an argument and some inputs which are pre-filled to return a new function that is prepopulated.

Syntax of partial function
The partial function takes a function and a preassigned value of a parameter of that function as the arguments.

```
partial(func: (*args: Any , **kwargs: Any) , *args: Any, **kwargs: Any)
```

However,, we can write in the following way

```
partial(sum, 5)
```

In the above example, the sum is a function that takes two arguments and returns the added product of the two arguments. Simultaneously, we have fixed one of the arguments as 5. Hence, we have developed a new function that can take only one argument, and we will return a product that is the sum of 5 and that argument.

Parameters
The partial method takes these parameters as arguments to return a new function.

Syntax:
func(*args:Any, **kwargs: Any)   e.g. sum(5,9) -> 14

Return Value of the partial function
As notified before, Partial is a function that returns a new function to increase the readability and maintenance of the codebase.

Example:

```
from functools import partial
```

```
def sum(a,b):
   return a+b
add_10 = partial(sum,10)
print(add_10(5))
```

Output:

```
15
```