Parallel Programming: Practice 4

April 15 – 19, 2024

Answers must be delivered in a single document (pdf format). This document must include the full name of all group members. If the exercise is an implementation exercise, the source code can be delivered in separate .py files within a compressed archive.

Exercise 1

Python's thread-based parallelism is limited by the GIL. To carry out true parallel execution it is necessary to use processes. In the following exercises we are going to parallelize with processes the program from the previous practices that downloads all the files referenced by an HTML page. We will start from this sequential version:

```
import you
import urllib.request
import urllib.parse
import re
import shutil
import time
import multiprocessing

total_file = 0
bytes_total = 0

def download_file(address):
    global total_file, total_bytes
    id = multiprocessing.current_process().name total_file +=
    1
    print(id, total_file, address) s =
    address.split('/')
    file = 'download/' + s[-1] try:

        i = urllib.request.urlopen(address) with open(file,
        'wb') as f:
```

```
                        shutil.copyfileobj(i, f) bytes_total
                        += f.tell() except Exception as
                        err:
                print(err)

def download_html(root_url, urls):
        r = urllib.request.urlopen(root_url) html =
        r.read().decode('utf-8', 'ignore')
        for m in re.finditer(r'src\w*=\w*"([-_./0-9a-zA-Z]*)"', html, re.I):

                address                    =               urllib.parse.urljoin(root_url,
m.group(1))
                urls.append(address)

def main():
        root_url = 'http://www.uv.es' if not
        os.path.exists('download'):
                os.mkdir('download')

        urls = []
        t1 = time.time()
        download_html(root_url, urls) for
        address in urls:
                download_file(address) t2 =
                time.time()
        total_time = t2 - t1

        print('---------------------') print('Time', total_time) print('Files', total_file)
        print('MBytes', total_bytes / (1024.0 ** 2)) print('Bandwidth (MBit/s)',
        (bytes_total * 8 / (1024.0



* * 2)) / total_time)

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

The bookstoremultiprocessingprovides for processes an interface similar to that ofthreading.For example, to start a process the routine is usedprocesswhich is similar toThread

```
p = multiprocessing.Process(target=..., args=...)
```

The variablepcontains an object of typeprocesswhich contains the methodsstart() andjoin()with the same functionality as inThread. An important detail is that the part corresponding to the main process must be inside a conditional such as:

```
if   ___yam ____  == 'main        ___':
```

to prevent child processes from executing that code. As with threads, you have to save the child processes in a list and start them with start().Also the main process must wait for all its children with
a call tojoin().

- Run the program and check that it works by looking at the contents of the directorydownload.Note the execution time.

- Make a copy of the previous program but making each call todownload_file be executed in a different process with the function multiprocessing.Process().Compare the times of the sequential version with the parallel one.

Exercise 2

A problem with the previous exercise is that it creates as many processes as there are downloaded files. This is even less efficient than the threaded version since the processes are significantly more expensive to create. As with threads, the solution is to create a fixed number of processes and have the producer place the data in a queue from which the consumers take it. Typically the number of processes is equal to the number of system processors that can be obtained with the function os.cpu_count().
Classmultiprocessing.Queueimplements synchronized queue for processes. The queue must be shared between all processes, for this it is created in the main process and passed as a parameter to the child processes.

One difference important between queue.Queue and multiprocessing.Queueis that the last one does not have the methods task_doneandjoin.Instead we need a mechanism to notify consumers that the job is finished. The simplest solution is to insert (after the producer has finished) a special value into the queue (for example, the empty string). In this way, consumers must obtain the value from the queue (with get()) and check whether or not it is the empty string. If it is the empty string, since there are no files left to download they must finish the process and if not continue downloading the file.

- Modify the above program to create a fixed number of processes and the downloads will be distributed among the processes using the class multiprocessing.Queue.Remember that the main process must insert as many empty strings as there are processes for them to terminate.

- Compare the times of this version with the previous ones.

Exercise 3

The execution model with processes is also known as *shared memory parallelism* . Each run has its own copy
of memory, that's why the variablestotal_fileandbytes_totalof the main process remain at zero. Python provides the classes multiprocessing.Valueand multiprocessing.Arrayto share data between processes. These classes include the locks necessary to avoid access conflicts. Note: to do arithmetic operations with these variables it is necessary to use the attributevalue.

- Modify the previous program to be able to share variables between processes. To do this, remove the global variablestotal_fileand bytes_total and change them to parametersdownload_file. Now these variables must be built in the main process with:

```
file_total = multiprocessing.Value('i', 0) bytes_total =
multiprocessing.Value('i', 0)
```

Exercise 4

To implement process parallelism in Python there is the Pool class, a mechanism that is higher level than Thread. ClassPoolrepresents a set of processes to which work (function calls) can be submitted in different ways. The distribution of work and communication between processes is automatic. The class is created with the constructor (if not specified
the number of processes usedos.cpu_count()):

```
pool = multiprocessing.Pool(processes=4)
```

One of the simplest functions isapply_asyncwhich launches a function call with its corresponding parameters. Work is assigned to one of the processes created byPooland runs asynchronously. When you have finished sending jobs toPoolyou have to call the method closeand then to the method jointo wait for all processes to finish.

- Do a parallel version of the sequential program in exercise 1 usingPool.The call to submit work is similar to the one used to create a process:

  ```
  pool.apply_async(download_file, (address, ))
  ```

- Compare the times of the sequential version with the parallel one.

Exercise 5

A class advantagePoolis that functions executed by worker processes can return results to the main process. For this, the methodapply_asyncreturns an object of the classApplyResult from which it can be extracted withgetthe results that the function sent with return.Obviously, these results are not available until the feature ends.

- Save the objectsApplyResultthat returnsapply_asyncin a list and calculates the total sums in the main program.

Exercise 6

Another interface to run in parallel with the Pool class is the operation map.Like the sequential version,mapIt takes a function and an iterable as arguments and executes the function with each of the elements as a parameter. The return values are returned as a list.

The operationmapit waits for the processes to finish, so there is no longer a need to calljoin.But it is necessary to callclose or terminateto destroy working processes. This can be automated usingwithwith which the call tomapit looks like this:

```
with multiprocessing.Pool() as pool:
        result = pool.map(download_file, urls)
```

- Modify the program from the previous exercise to usemapratherapply_async