Parallel Programming: Practice 1

12-16 February 2024

The answers must be submitted in a single document (pdf format). This document must include the full name of all members of the group. If the exercise is an implementation exercise, the source code can be delivered in separate .py files inside a compressed archive (exercises 5 and 6).

Exercise 1

The product of matrices is a basic operation in linear algebra with many practical applications. The following implementation uses Python lists to store the matrices:

```
import random
import time

def multiply(n, a, b, c):
    for i in range(n):
        for j in range(n):
            d = 0
            for k in range(n):
                d = d + a[j][k] * b[k][i]
            c[i][j] = d

n = 500

random.seed(0)
a = [[random.random() for i in range(n)] for j in range(n)]
b = [[random.random() for i in range(n)] for j in range(n)]
c = [[0 for i in range(n)] for j in range(n)]

time = []
for r in range(3):
    t1 = time.perf_counter()

    multiply(n, a, b, c) t2

    = time.perf_counter()
    print("Time [", r, "]", t2 - t1)
    time.append(t2 - t1)

print("Size", n, "Minimum time", min(time))
```

To measure execution time reliably, it is necessary to repeat several runs to check that no other processes are interfering. Generally there is not much variation and the median (or minimum) time is chosen.

- Run the program varying the size of the matrices (n = 1000, 2000, 5000...) and make a graph with the times on the Y-axis and the size n on the X-axis.
- Does the graph match the theoretical cost $O(n^3)$?

Exercise 2

The product cost of arrays makes it an excellent candidate to take advantage of parallel execution. The following code spreads the computation of each element of the result over np processes.

```
import random
import time
import multiprocessing

def multiply(i,j): d
    = 0
    for k in range(n):
        d = d + a[j][k] * b[k][i]
    return d

def list_to_array(l,n):
    return [l[i:i+n] for i in range(0, len(l), n)]

if name == ' main ':

    n = 500
    np = 2

    random.seed(0)
    a = [[random.random() for i in range(n)] for j in range(n)]]
    b = [[random.random() for i in range(n)] for j in range(n)]]
    c = [[0 for i in range(n)] for j in range(n)]

    time = []


    with multiprocessing.Pool(np) as p:
```

```
            for r in range(3):
                 t1 = time.perf_counter()

                 l = []
                 for i in range(n): for
                     j in range(n):
                         l.append([i,j])
                 s = p.starmap(multiply, l) c
                 = matrix_list(s, n)

                 t2 = time.perf_counter()
                 print("Time [", r, "]", t2 - t1)
                 time.append(t2 - t1)

     print("Processes", np, "Size", n, "Minimum time", min(time))
```

- Execute the program by varying the number of processors (np = 1, np=2, np = 4, np = 6 and np = 8).
- Make a graph with time on the Y-axis and the number of processes on the X-axis. How does time behave as the number of processes increases?

Exercise 3

Using Python lists to store numeric arrays is not the best option. In an array all elements are of the same type, but lists can store different types. This is an overhead of several bytes for each element, and with a large array this can be a lot of memory. The `numpy` library provides array objects without this problem where elements are stored consecutively. And it also allows a much more natural syntax `a[i,j]` to be used instead of `a[i][j]`. The following program is a version of exercise 1 using `numpy`.

```
import random
import time
import numpy

def multiply(n, a, b, c): for
    i in range(n):
        for j in range(n):
            d = 0
```

```
            for k in range(n):
                d = d + a[j,k] * b[k,i]
            c[i,j] = d

n = 500

random.seed(0)
a = numpy.array([[random.random() for i in
range(n)] for j in range(n)])
b = numpy.array([[random.random() for i in
range(n)] for j in range(n)])
c = numpy.zeros((n,n))

time = []
for r in range(3):
    t1 = time.perf_counter() multiply(n,

    a, b, c)

    t2 = time.perf_counter() print("Time
    [", r, "]", t2 - t1) time.append(t2
    - t1)

print("Size", n, "Minimum time", min(time))
```

- How long does it take? Is it faster than the version in exercise 1? Make a graph and compare it with exercise 1.

Exercise 4

Replaces the call to the multiply function with the matrix multiply operation implemented in numpy:

```
    numpy.matmul(a, b, out = c)
```

- Execute the modified program with the above matrix sizes.
  How long does it take now?
- Make a graph of the times with numpy. How does it compare with the times of the previous exercises? Justify the results.

Exercise 5

Given the following Python code in which we have 2 threads executing the buyer function at the same time:

```
import threading

orange_counter=0 repetitions=10

def buyer():
    global orange_counter
    for i in range(repetitions):
        orange_counter+=1




if name == ' main ':
    jorge=threading.Thread(target=buyer)
    ana=threading.Thread(target=buyer)

    jorge.start()
    ana.start()

    jorge.join()
    ana.join()
    print("We have:    ",    orange_counter,        '
oranges')
```

- First run this code and then change the number of repetitions to 1000000. Run the code several times, what do you observe, why do you think this is happening? Modify the code to get the correct result every time.

Exercise 6

Given the following problem: a group of students are sitting around a round table in the library. On the table there is a collection of books available for studying different subjects. Each student needs two specific books to study for their next exam, but can only take the books that are to their left and to their right.

right. The challenge is to ensure that every student can get the books they need to study. We have this Python code that tries to solve it:

```
import threading
import time
import random

libroA=threading.Lock()
libroB=threading.Lock()
libroC=threading.Lock()


def student(name, firstbook, secondbook, reviews):
    while repasos>0:
        primerlibro.acquire()
        print(" The student: ", name, " has the
first book")
        secondbook.acquire()
        print(" The student: ", name, " has the
second book")

        if repasos>0:
            reps-=1 time.sleep(random.randint(1,
            3)) print(" Student: ", name, " has
reviewed ")

        secondbook.release()
        firstbook.release()



if name == ' main ': jorge=threading.Thread(target=student,
args=("Jorge", bookA, bookB,5))
    ana=threading.Thread(target=student,
args=("Ana", bookB, bookC,5))
    maria=threading.Thread(target=student,
args=(("Maria", bookC, bookA,5)))

    jorge.start()
    ana.start()
    maria.start()

    jorge.join()
    ana.join()
    maria.join()
    print("We have finished studying")
```

- What happens when you run it? Repeat the execution several times. Explain what is happening and try to solve it.