Multithreading and speed in finding factorials

# OS Project Loay AlBarmawi 20170111

## Introduction:

This purpose of this report is to show the time difference between multithreaded and single threaded programs that find factorials of integers, the language used for the 2 versions of the program was python on a windows PC.

## The Problem:

Calculations that require a large number of operations are very demanding in terms of computational power, in this case, finding the factorial of a number N would require N multiplication operations and the results of the operations are astronomical.

## The Solution:

Using multiple threads instead of one to perform the calculations simultaneously where we divide the work into smaller pieces and calculate them then multiply the results with each other.

import threading, time  
from queue import Queue

Libraries used for the project

def fact(arr, q):  
 print("working")  
 result = 1  
 for i in arr:  
 result \*= i  
 q.put(result)

The factorial function used

The function gets an array of numbers and finds the multiplication of all of them then returns it into a queue, I added print("working") to make sure that the threads were all working.

if \_\_name\_\_ == "\_\_main\_\_":  
 q = Queue()  
 num = int(input("enter number"))  
 t1 = time.time()  
 numbers = range(1, num)  
 parts = [numbers[i::2] for i in range(2)]  
 th = threading.Thread(target=fact, args=(parts[0], q))  
 th.start()  
 th1 = threading.Thread(target=fact, args=(parts[1], q))  
 th1.start()  
 th.join()  
 th1.join()  
  
 result = 1  
 for c in range(0, 2):  
 result \*= q.get()  
 print(result)  
 print("time: " + str((time.time() - t1)))

The main code

I started with defining a queue then asking the user to insert a number, the time starts after the user inserts the number to find the factorial of, then it fills the array numbers with the numbers from 1 to the number specified, then it cuts it into 2 arrays and stores it into 1 array called parts.

Then it starts 2 threads, each one uses half of the numbers and gets their multiplication result, after that the program multiplies the results with each other, giving us the final result and the time it took to calculate it.

The single threaded version was the same code but without the division of the numbers and with only one thread created instead of two.

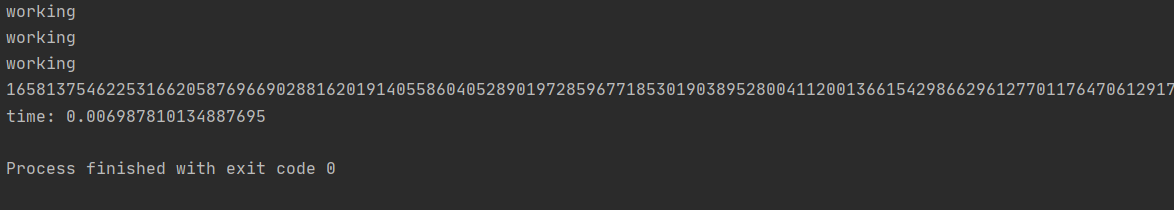
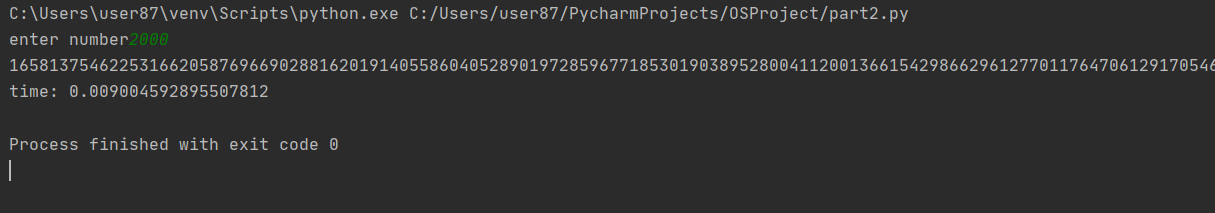
## Results:

Before we get into the results, the multiplication results will not be shown completely because they are astronomical values.

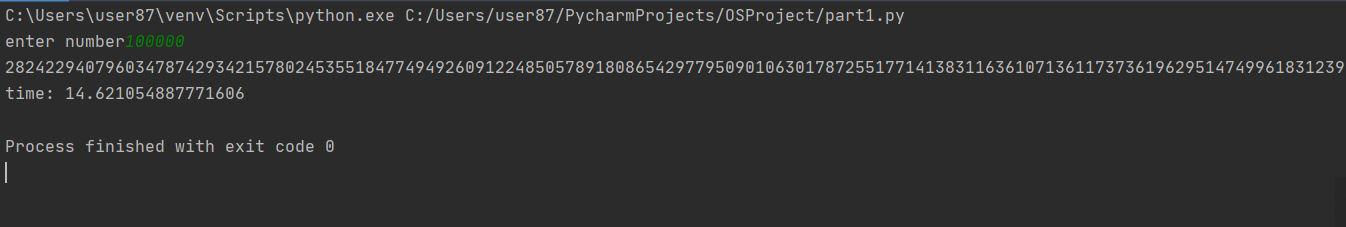
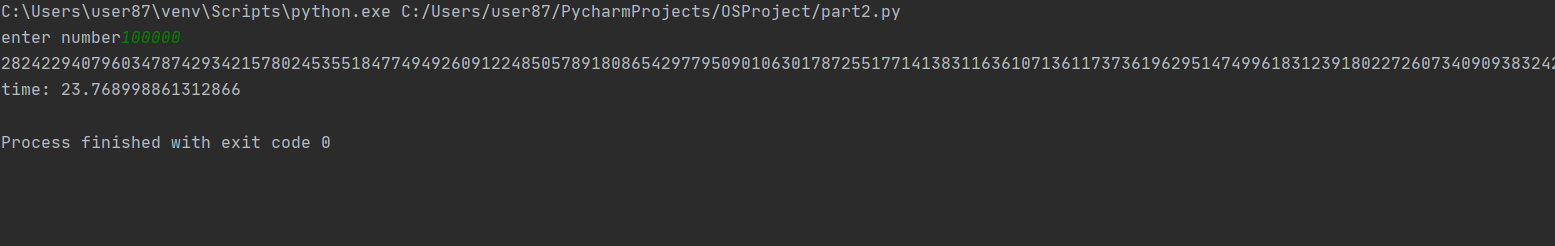
In terms of small numbers, the single threaded version performs closely to the multithreaded version as specified in the screenshots below; the difference was so small it could be ignored.

The number entered here was 2000

Single Threaded Multithreaded

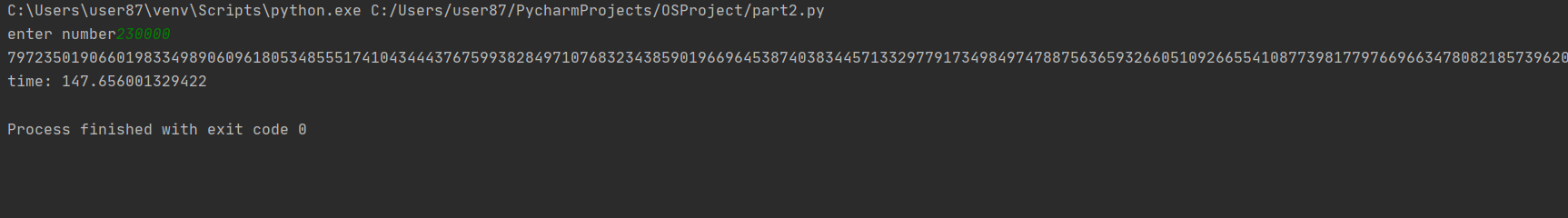
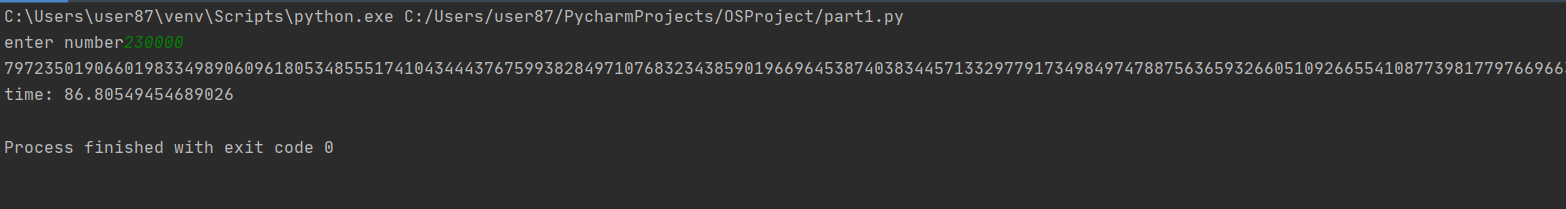


When I increased the number to 100000 the difference started being more prominent, where the multithreaded version took much less time than the single threaded one.



Single Threaded Multithreaded

Again, I increased the number to 230000 and the difference got even more apparent, the time difference was over a minute between the two.

Single Threaded Multithreaded

## Conclusion:

The results were noticeable the more I increased the number, this proves that it did divide the work successfully.

This experiment shows that multithreading is very useful in terms of time for calculations that require a lot of processing power and could be divided.

What could be changed to make it more efficient is dividing the list of numbers in a way where they would be distributed as odds and evens to distribute the weight of the calculation more equally on the threads, in other words, if we want to find 100000!, list1=[1,3,5,7,……….,99999] and list1=[2,4,6,8,……….,100000].