AuE 8930: Computing and Simulation for Autonomy

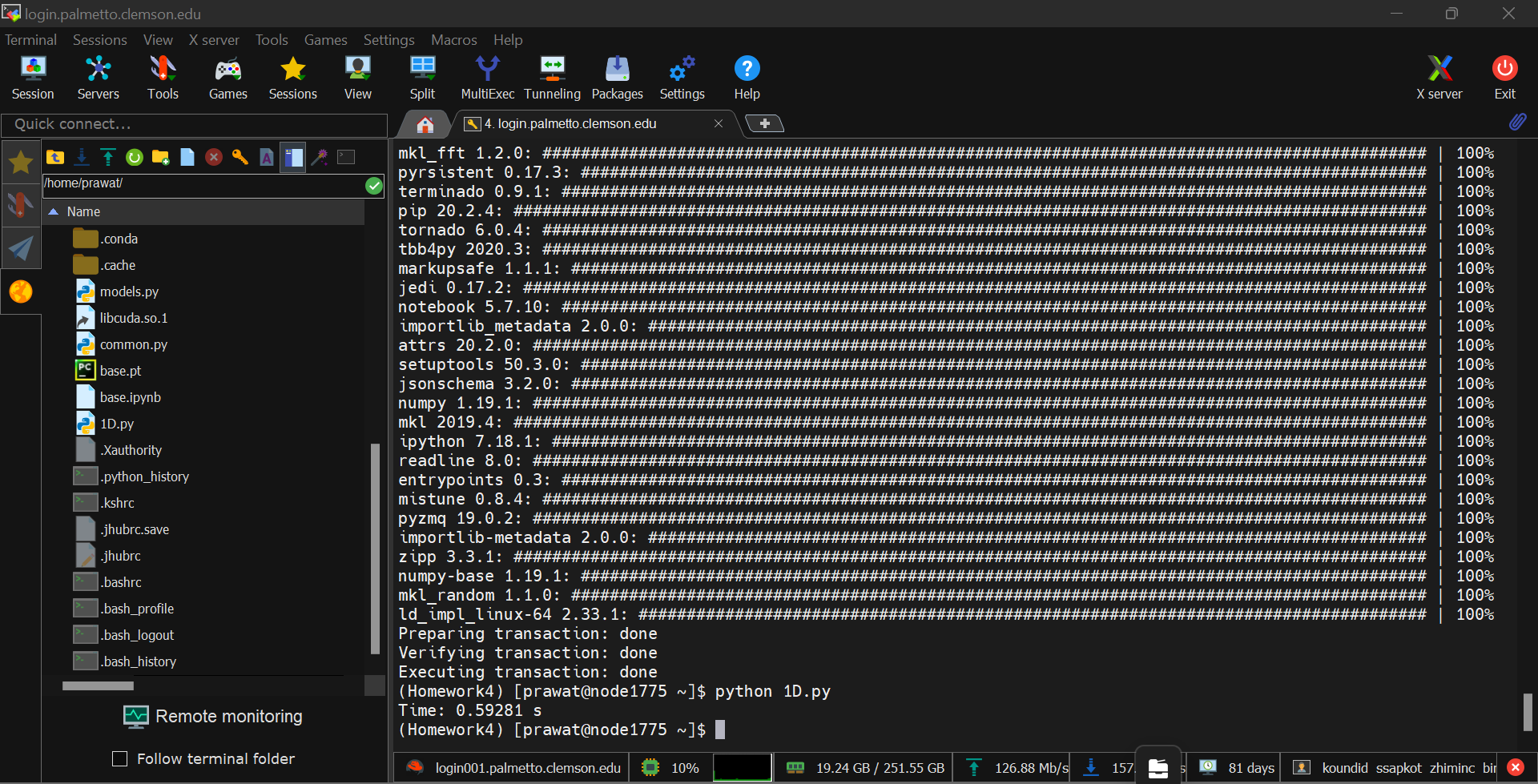
**Priyanshu Rawat**

**GitHub Link:** <https://github.com/priyanshurawat1509/HPC_Homework_4>

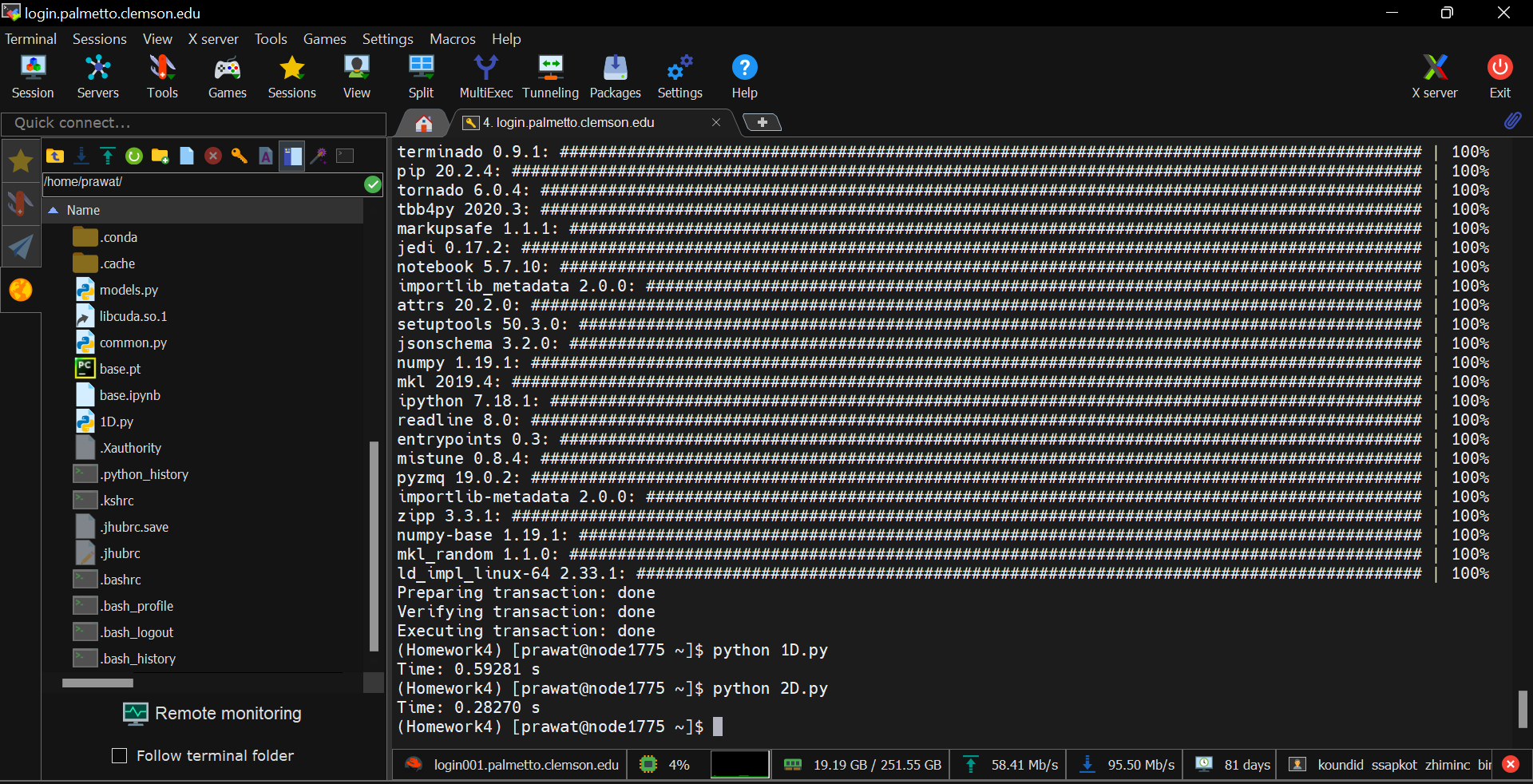
**Question 1)**

**(1) Install Pycuda on Palmetto Cluster and run 1D.py and 2D.py, show screenshots. (5 points)**

1D.py



2D.py



**(2) Try different block size, grid size and matrix dimensions. Analyze the running time. If error**

**occurs with some block/grid size, analyze the reason. (5 points)**

1D.py

|  |  |  |  |
| --- | --- | --- | --- |
| **Block Size** | **Grid Size** | **Matrix Dimensions** | **Time** |
| 128 | 1600\*1600/128 | 1600\*1600 | 0.58552 s |
| 512 | 1600\*1600/512 | 1600\*1600 | 0.58646 s |
| 128 | 600\*600/128 | 600\*600 | 1.44461 s |
| 512 | 3000\*3000/512 | 3000\*3000 | 1.44549 s |

2D.py

|  |  |  |  |
| --- | --- | --- | --- |
| **Block Size** | **Grid Size** | **Matrix Dimensions** | **Time** |
| 32 | 3000\*3000/32 | 3000\*3000 | 0.28245 s |
| 64 | 3000\*3000/64 | 3000\*3000 | 1.89878 s |
| 32 | 800\*800/32 | 800\*800 | 1.89175 s |
| 64 | 800\*800/64 | 800\*800 | 0.03623 s |

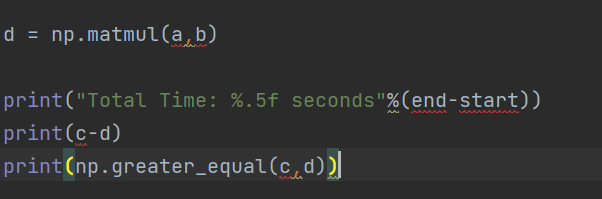
From the result, we can infer that the block size is limited by the selection of hardware. In other words, the total number of running threads is determined by choice of the GPU. For instance, if larger block size is defined, the GPU will reach its maximum capacity and will not be able to process the threads.

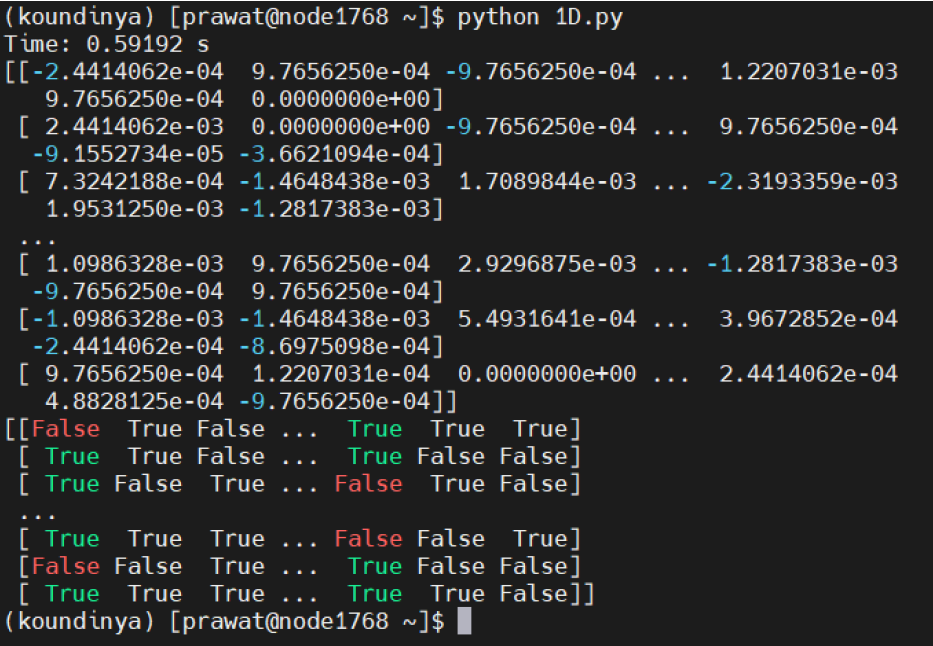
**(3) Implement a CPU based matrix multiplication using programming language you prefer,**

**compare it with the above GPU sample implementation. Check if the product is the same as GPU**

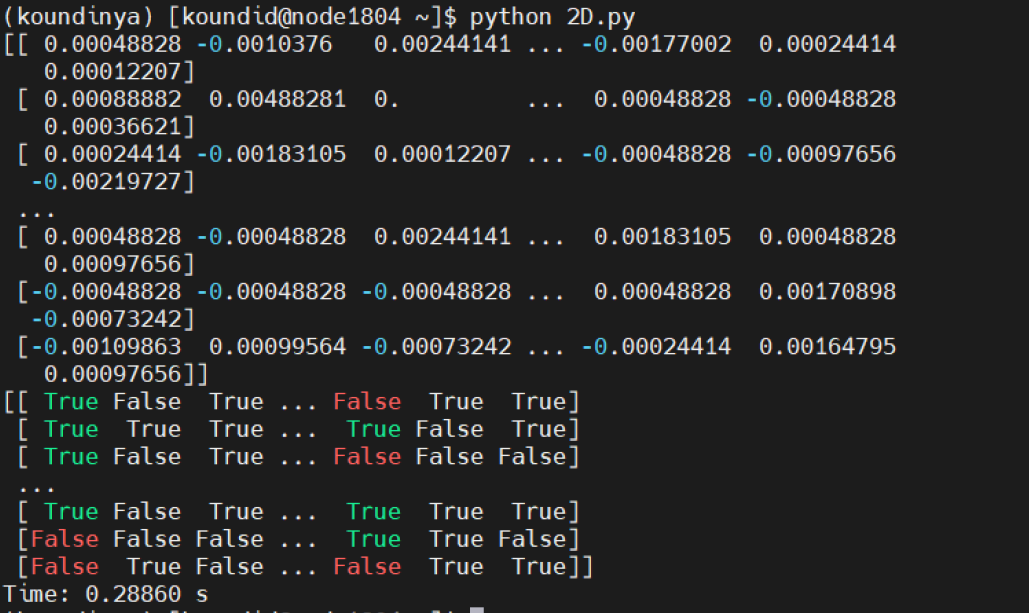
**implementation, if not, compute and analyze the difference. (15 points)**

To perform the matrix multiplication using CPU, I have used NumPy library and made the following changes.





The above results are for the 1D.py code. The first result in the matrix shows the error between both calculations. While the second result shows which element is greater in value than the other corresponding element.



This difference is mainly because of the approximation in values by CPU. Whereas, the GPU outputs are much more precise, since the GPU does not round off values like the CPU. This result shows that such errors can multiply manifold the CPU is subjected to large calculations (as seen in this case).

**(4) Analyze the difference between 1D.py and 2D.py, why 2D.py is quicker than 1D.py? (5 points)**

Thread distribution is the primary difference between 1D.py and 2D.py. It is 1-dimensional in the 1D.py code and 2-dimensional in the 2D.py code. This the reason why 2D.py can do a greater number of parallel tasks compared to 1D.py and is thus quicker than latter.

**(5) Survey and describe some methods to further speedup the 2D.py (5 points)**

To further speedup 2D.py, we can use libraries like tensor flow. The matrix multiplication of tensor flow can improve the speed. On top of that, defining global variables instead of local variables can affect the speed positively. Last but not the least, threading can be optimized further to improve the efficiency of the code.

**Question 2**

**(1) Refer to the sample code in Q3 folder, complete it by importing libraries and implementing your main function. Run and analyze the behavior of the code. (15 points) You can refer to the readme.md.**

Initial Image -



Final Image **-**



**(2) Write a Python code using CPU to mimic the behavior of the above GPU code, compare and analyze the result and speed. You can use third party library such as the OpenCV. (15 points)**

CPU Implementation (using OpenCV library)-





****

GPU Implementation-

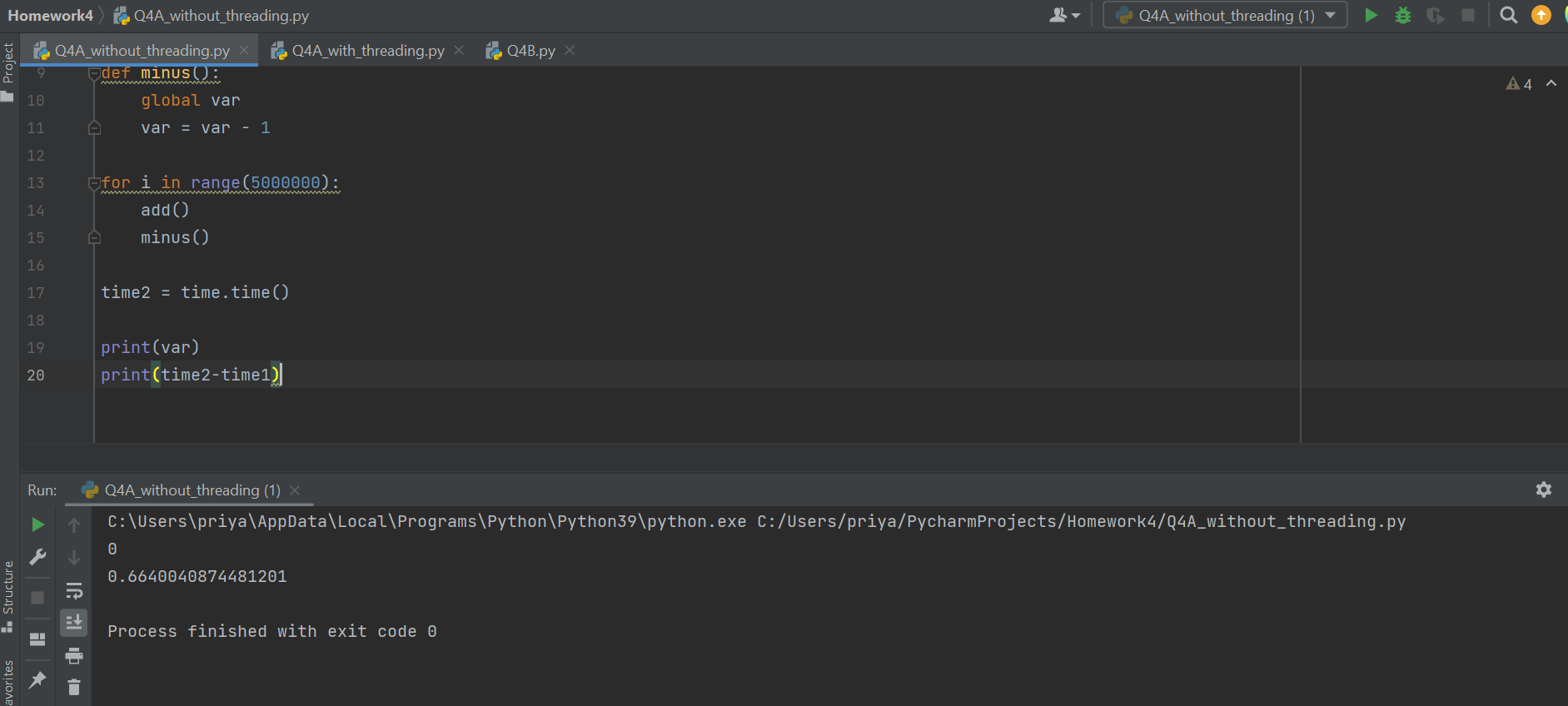


****

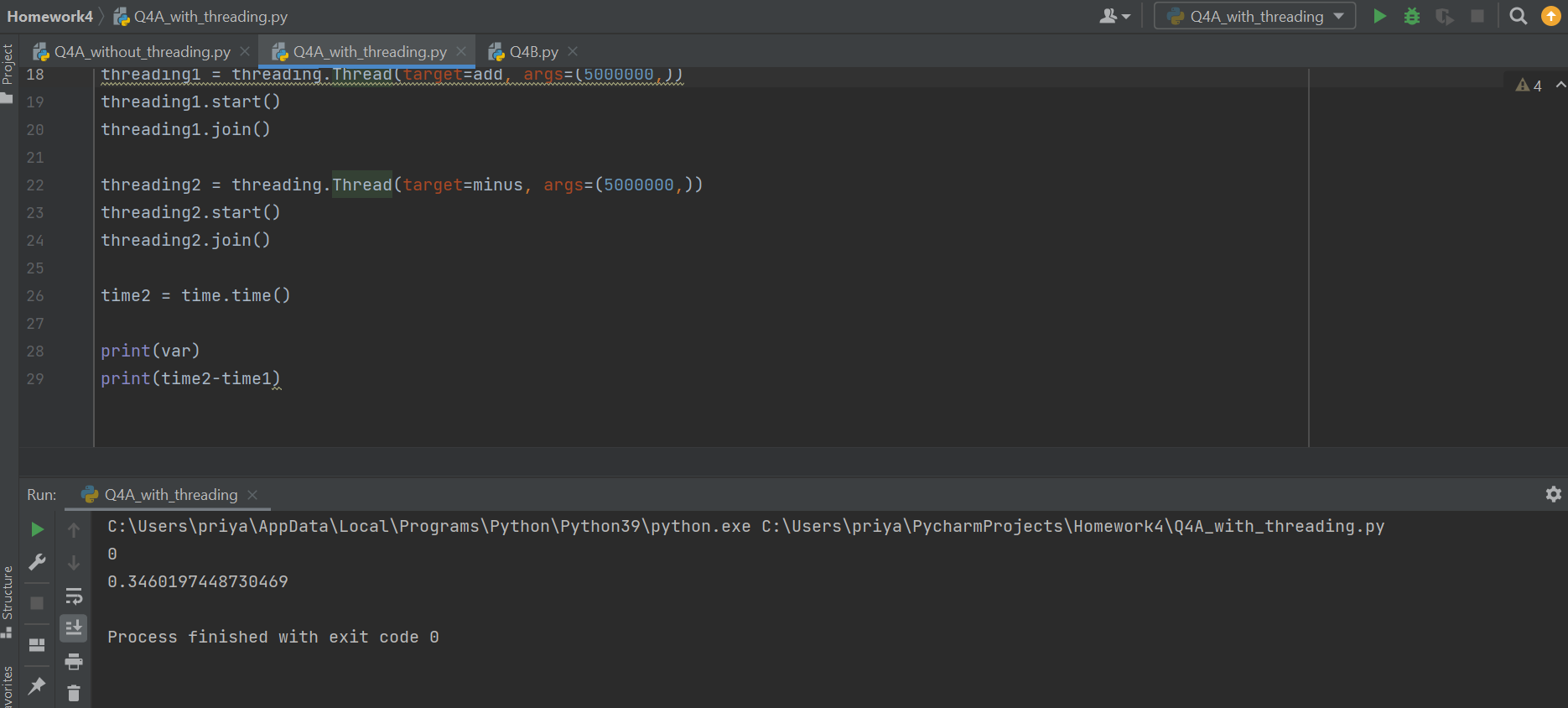
**Question** **3.1)**

**(1) Use “import threading” library to speedup Q4A.py as quick as you can. (10 points) (e.g., you may use threading.Thread() … to create your new thread to do the computing)**

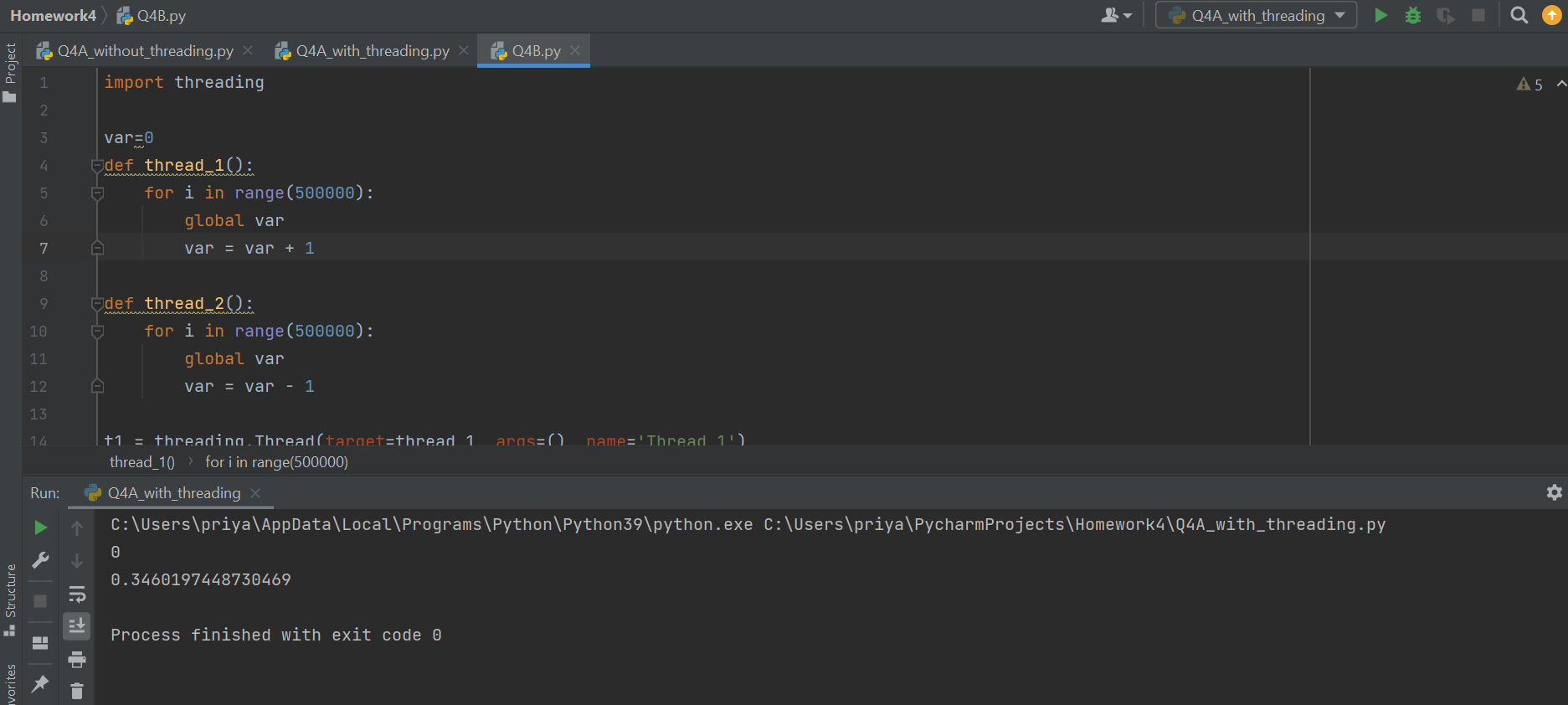
Without Threading



With Threading –

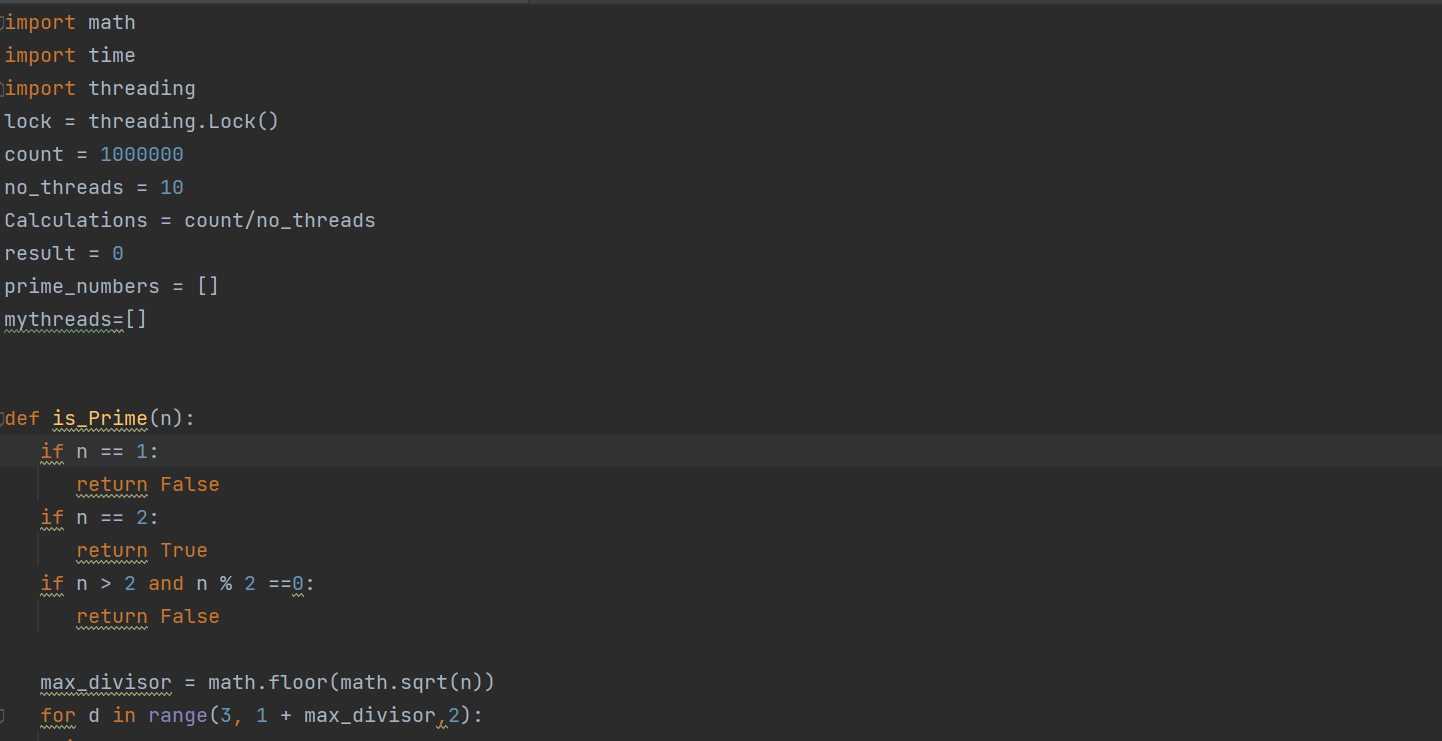


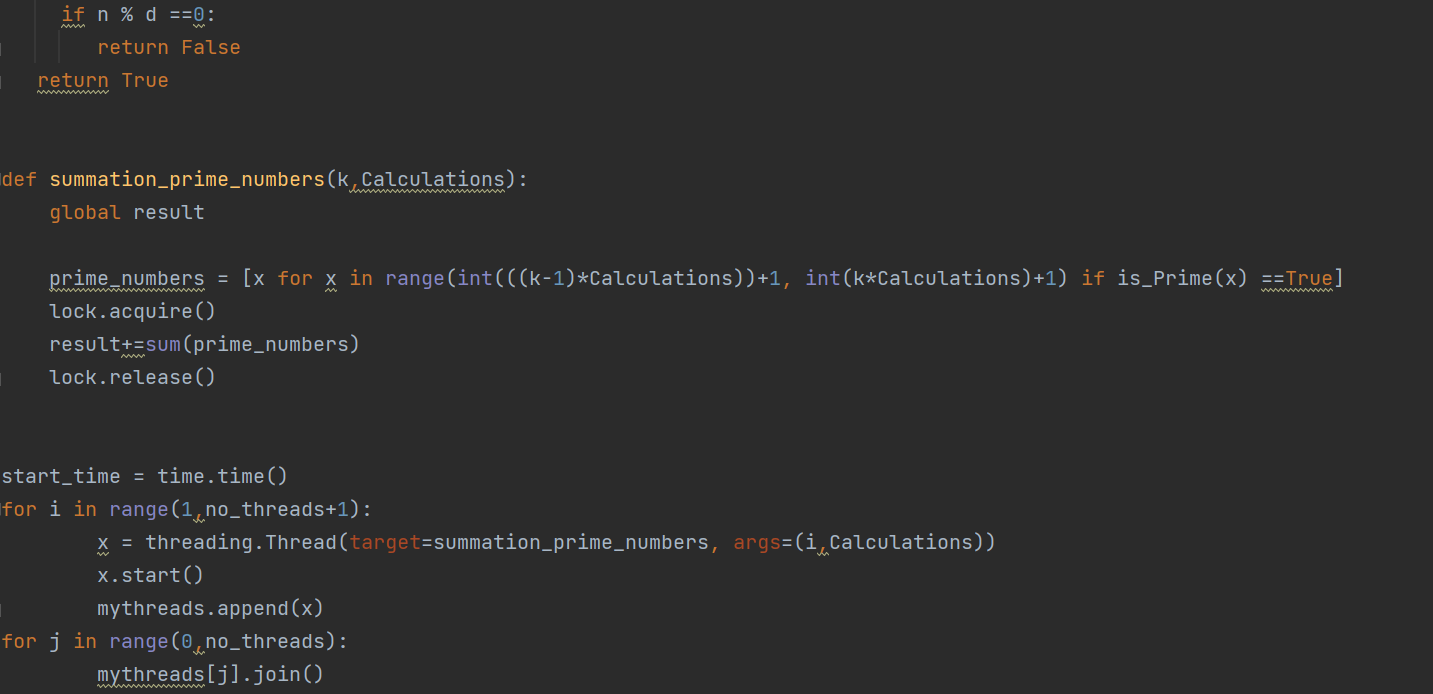
**(2) Run Q4B.py several times, analyze what’s wrong with it, try to fix it (you may use Lock). (10 points)**

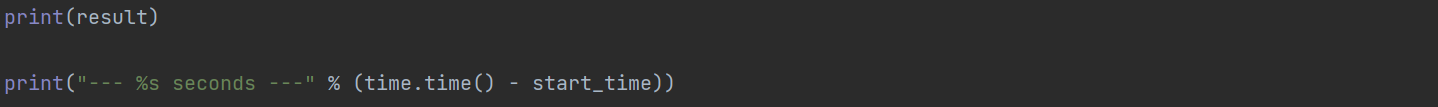


The output of the code should be ‘0’, but this was not the case. This discrepancy in the output occurred as two threads were running simultaneously leading to a situation called ‘racing between threads.’ This problem can be solved by using threading.join() command to join the two threads so both can run synchronously. After making the necessary correction, the result of the code came as expected.

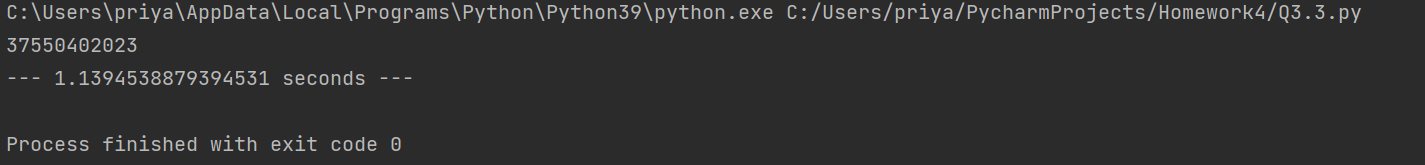
**Question 3.3)**





****

Result -

****