**ASSIGNMENT 2**

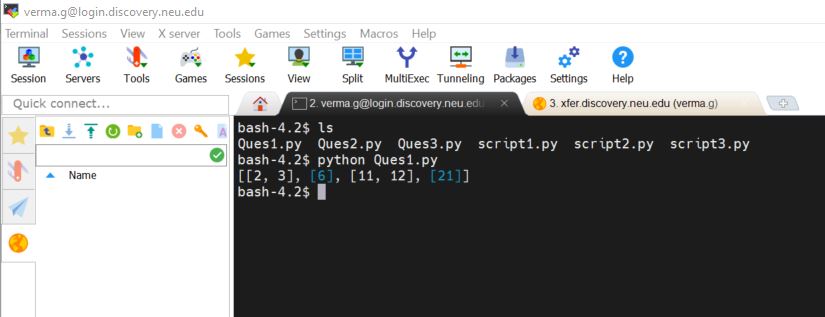
**Part 1: 15 points**

1. Use Pool.apply() to get the row wise common items in list\_a and list\_b; and print the result.    (5 pts)

list\_a = [[1, 2, 3], [5, 6, 7, 8], [10, 11, 12], [20, 21]]

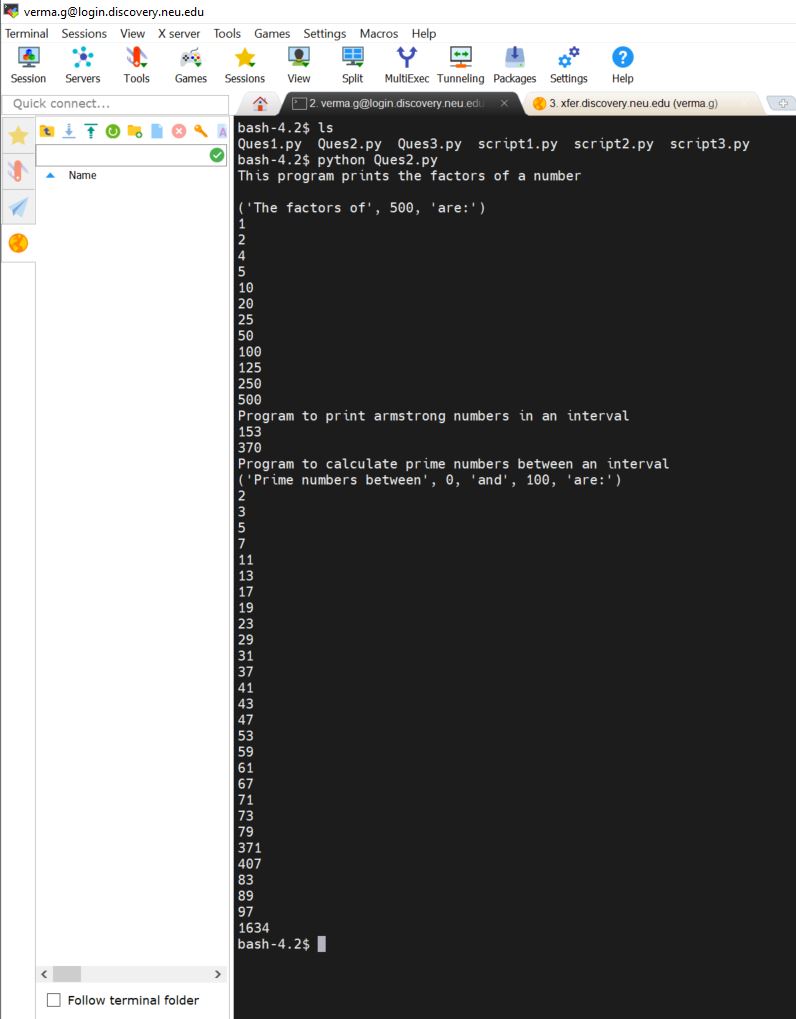
list\_b = [[2, 3, 4, 5], [6, 9, 10], [11, 12, 13, 14], [21, 24, 25]]

Answer:



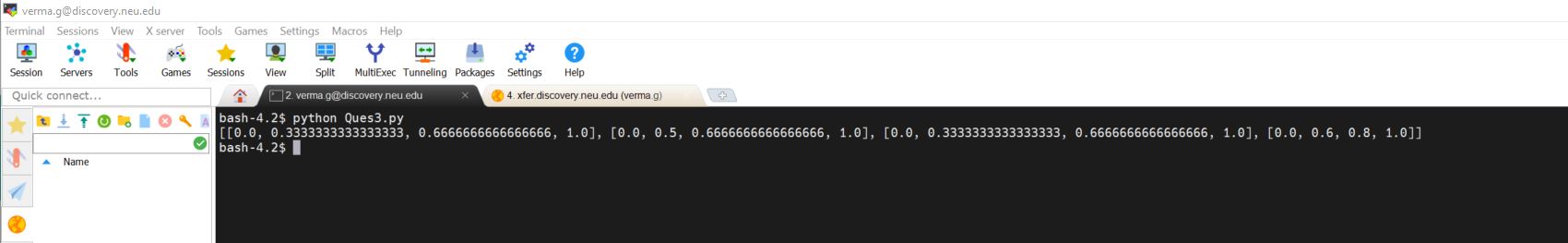
1. Use Pool.map() to run the following python scripts in parallel; and print the result.                    (5 pts)

Script1.py script1.py script3.py



1. Normalize each row of 2d array (list) list\_c to vary between 0 and 1. Parallelize the function with any subfunction of Pool; and print the result.     (5 pts)

list\_c = [[2, 3, 4, 5], [6, 9, 10, 12], [11, 12, 13, 14], [21, 24, 25, 26]]

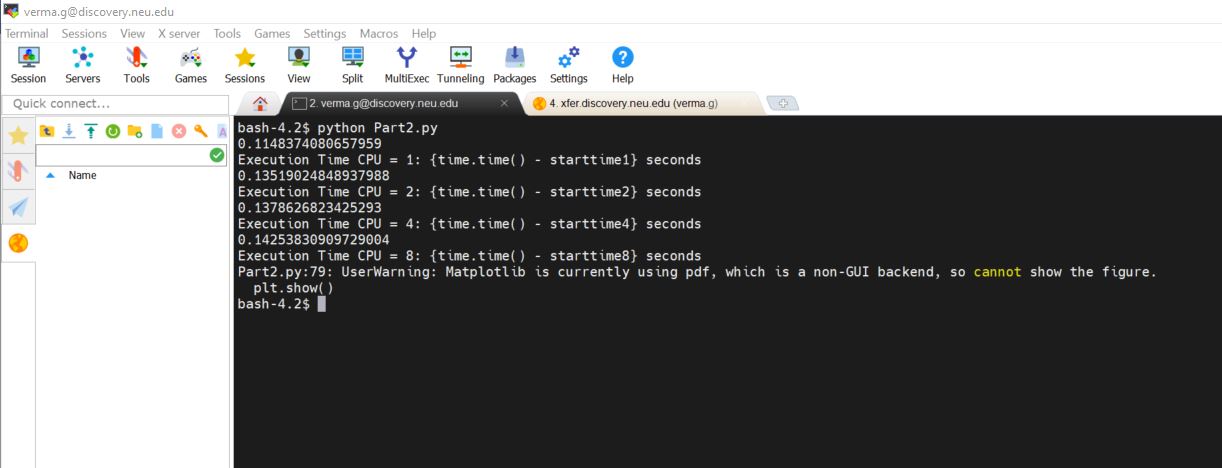


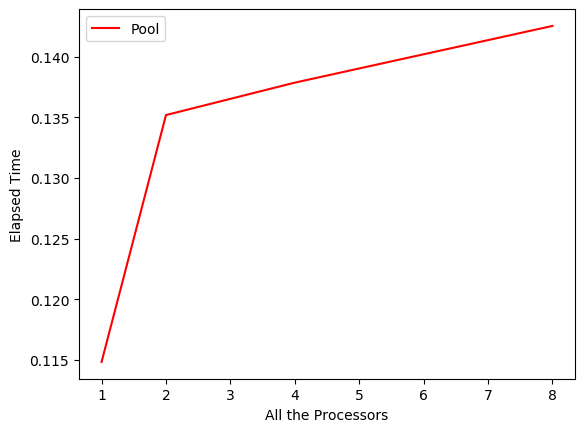
**Part 2: 25 points**

Please complete the following tasks:

* Create a dataframe. The required values are arbitrary numerical numbers, the shape of the values is 20000\*100 (5pts)
* Define a function to find the maximum and minimum values of each column, sum the square of these two numbers, then find square root.    (5pts)
* Parallelize the function with any method of multiprocessing.Pool.           (5pts)
* Set a timer to calculate the elapsed time for the parallelized code when CPU=1, 2, 4, and 8.  (4pts)
* Using matplotlib to plot the trend curve of speedup as number of CPU (1,2,4,8) and save the figure as an image file (4pts)
* Create a word document with following
* Create a table and fill in the elapsed times you obtained when running the program using different CPU numbers (2pts)
* Insert the plot image in this Word document and analyze your results: the speedup, the overhead, optimal results.etc

|  |  |
| --- | --- |
| CPU Numbers | Elapsed Time (seconds) (approx.) |
| 1 | 0.11 |
| 2 | 0.135 |
| 4 | 0.137 |
| 8 | 0.142 |





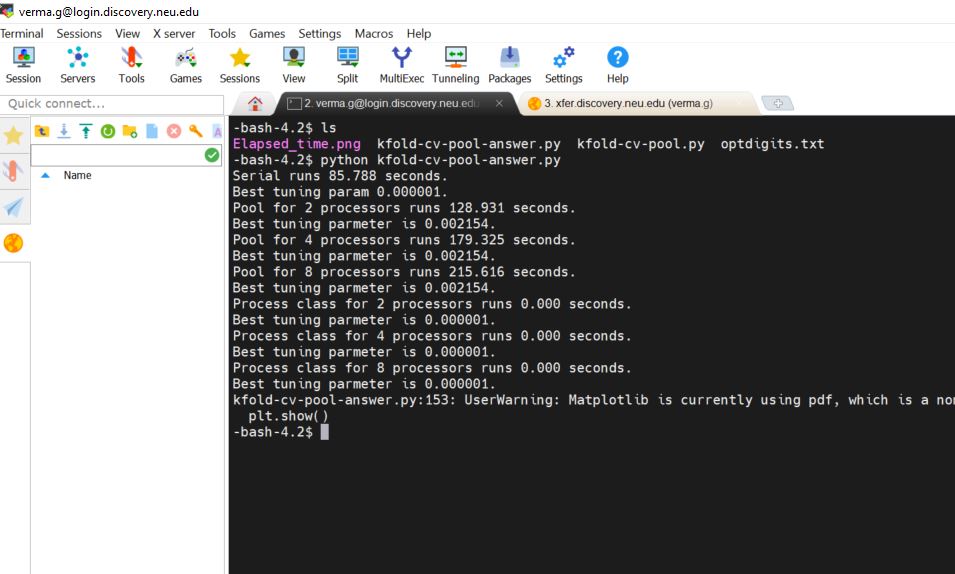
**Analysis:**

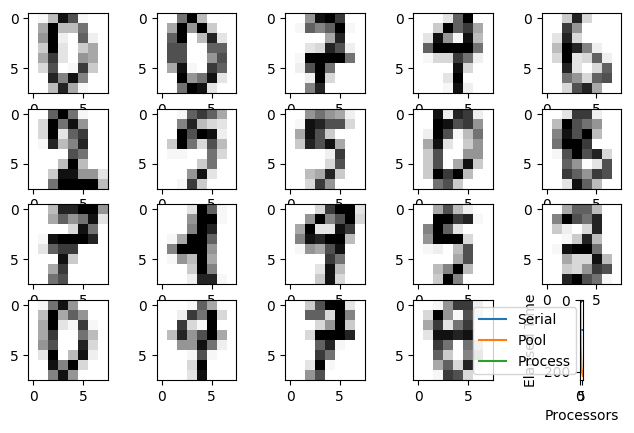
* As number of cores increases, speedup of parallel applications also increases.
* More number of cores will lead to high speed up of parallel applications, highly sequential applications will be favored by a powerful core.
* In case of strong scaling, the number of processors is increased while the problem size remains constant. This also results in a reduced workload per processor.
* A system will adapt to the actual level of parallelism though, if possible, changing the core configuration will incur some overhead and will challenge application developers.
* Considering the re-configuration overhead, the speedup of automatic core configuration will be superior to either symmetric or asymmetric core design.

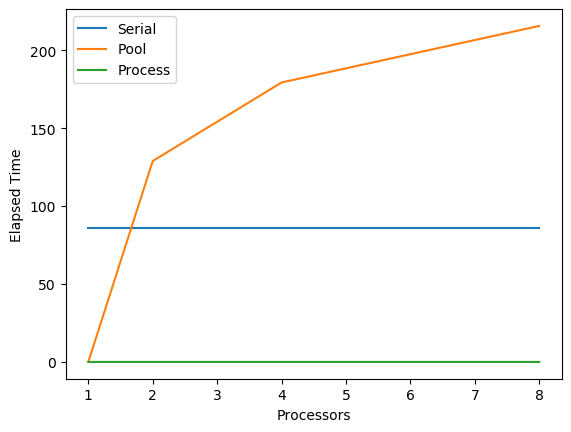
**Part 3: 15 points**

* Based on the parallelized code file “kfold-cv-pool.py”, set a timer and build a for-loop to calculate the elapsed time for the parallelized code when CPU=1, 2, 4, and 8.                          (4 pts)
* Using matplotlib to plot the trend curve of speedup as the number of CPU (1,2,4,8). (2pts)
* Create a word document. In this, you should do:
* Create a table and fill in the elapsed time (seconds) you obtained when running the parallelized program using different CPU numbers (2pts)
* Insert the plot image in this Word document and analyze your results: the speedup, the overhead and optimal results.etc (2pts)

|  |  |
| --- | --- |
| CPU Numbers | Elapsed Time (seconds) (approx.) |
| 1 | 85.78 |
| 2 | 128.931 |
| 4 | 179.325 |
| 8 | 215.616 |







**Analysis:**

* More cores will lead to high speed up of highly parallel applications, a powerful core will favor highly sequential applications.
* A system will adapt to the actual level of parallelism though, if feasible, changing the core configuration will incur some overhead and will challenge application developers.
* Even considering the re-configuration overhead, the speedup of automatic core configuration will be superior to either symmetric or asymmetric core design.
* Even in the case where number of processors is increased even though the problem size remains constant sometimes reduces workload per processor in case of strong scaling.

**Part 4: 20 points**

* In numpy array, draw random samples from a normal (Gaussian) distribution, the mean of the distribution is 10, the standard deviation of the distribution is 0.1, and the output shape is 20000x20000. Then take the "mean" of x along axis=0 with a step of 100.                 (10 pts)
* In dask array, do the same thing as above. (Tips: you can set the size of “chunks” to 1000x1000)                      (10 pts)

Note: finish the part 4 in one Jupyter file.

**JUPYTER NOTEBOOK ATTACHED**

**Part 5: 25 points**

Please install nycflight dataframe package in your Anaconda environment (you can do it on “local” or “discovery”):

              pip install nycflights13  
              from nycflights13 import flights

* Remove the samples with “NaN” in the feature “dep\_delay” in this dataframe.                  (5 pts)
* Start a Dask Client within JupterLab extension, and set “n\_workers=4” for this client.                                    (5 pts)
* Using dask dataframe to compute the mean and standard deviation for departure delay “dep\_delay” of all flights.            (5 pts)
* You are required to take a screenshot for running the step 3 with the 3 observed processes “Dask Process”, “Dask Graph”, “Dask Task Stream” in a same JupyterLab window, which I showed in the class. (Tip: since the dataset is small, the execution would be very fast. You may need to take a very fast screen capture.)                    (10 pts)

