Jose Maria Martínez Marín - 100443343

Francesco Mora – 100439601

Ignacio Soria Ramírez - 100443660

TECHNOLOGICAL FUNDAMENTALS OF BIG DATA: PROTEINS

Universidad Carlos III de Madrid

1. Introduction

The dataset is a .csv file called “proteins” contains 74.099 protein chains defined by a string, in which every letter corresponds to a part of the protein chain.
The table is composed by a Structure ID and the Sequence, and the size of the dataset is 62.7Mb.

structureId	sequence
1	MVLSEGEWQLVLHVWAKVEADVAGHGQDILIRLFKSHPETLEKFDRVKH …
2	MNIFEMLRIDEGLRLKIYKDTEGYYTIGIGHLLTKSPSLNAAAKSELDKAIGR …
3	MVLSEGEWQLVLHVWAKVEADVAGHGQDILIRLFKSHPETLEKFDRFKHL …
…	…

The idea of the program is to let a user put a string of characters as an input, and to check the following things:

- Whether the pattern in input is found inside the datafile in some row.
- If there are occurrences, register the ID of the protein and the number of times the pattern occurs.

As a conclusion, a histogram of the occurrences is showed, and the list of the proteins with the highest number of occurrences is printed.

Different approaches will be tried in the multiprocessing programming:

- Pool.map() function
- Pool.starmap() function
- Threads

With each of them, the execution time will be calculated slicing the program and the data in different chunks, from 2 to 16.

Finally, the best solution for the machine which this assignment was made with will be chosen.

2. Serial program

The conceptual schema of the program is as follows:

1. Read the dataset and save it as a DataFrame with two columns. Make every string uppercase.
2. Read the input string and make it uppercase.
3. Define the function that looks for occurrences. This function associates to every ID the number of times that the pattern appears in the string.
4. Initialize the time, run the function, and stop the time.
5. Create a DataFrame with the ID of the proteins with at least one occurrence.
6. Plot a histogram with the ID on the x-axis and the number of occurrences in the y-axis.
7. Print the IDs of the proteins with the highest number of occurrences.
8. Print the execution time of the part of the code in which the function under study is running.

Trying with the random pattern “NAAKS”, the output values are:

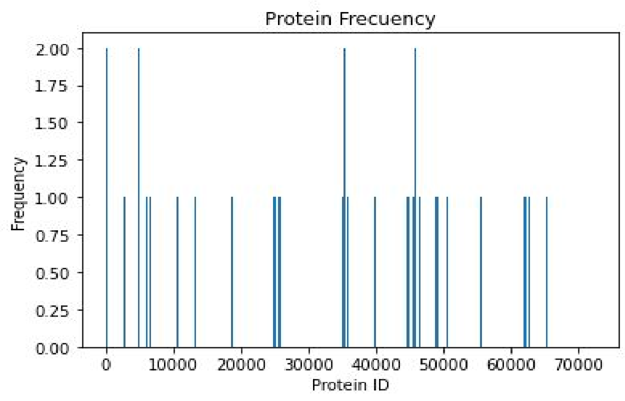
```
MAX VALUE OF OCCURRENCES IS:

 >>> 2.0


PRESENT IN THE FOLLOWING 17 INDICES:

 >>> [    76  4854  4855  4856  4857  4858  4859 25721 35357 35358 45722 45806
 45807 62248 62262 62263 62264]
```

The histogram of the occurrences is shown in the picture below.
We can clearly see that in this case most of the protein IDs with occurrences have either one or two of them.



3. Multiprocessing program

The idea with the multiprocessing approach is to divide both the dataset and the function into several independent chunks. In this way, it is possible to run a big query in parallel, taking advantage of the available cores in the CPU, and thus speeding up the program. The library in charge of the multiprocessing activity is called *multiprocessing*.

The schema of the code is:

1. Define the function that looks for occurrences. This function associates to every ID the number of times that the pattern appears in the string.
2. Read the dataset and save it as a DataFrame with two columns. Make every string uppercase.
3. Read the input string and make it uppercase.
4. Divide the dataset into equal size chunks, so that every process is in charge of a small set of data.
5. Initialize the time
6. Open the pool, run the function with a pool function, and close the pool.
7. Stop the time
8. Join the results achieved by the different processes to create a unique matrix of results.
9. Create a DataFrame with the ID of the proteins with at least one occurrence.
10. Plot a histogram with the ID on the x-axis and the number of occurrences in the y-axis.
11. Print the IDs of the proteins with the highest number of occurrences.
12. Print the execution time of the part of the code in which the function under study is running.

The main options to run a multiprocessing code in Python is to use pool.apply(), pool.map(), or pool.starmap(), depending on the type of data and on the approach to the code.
In this study pool.apply(), pool.map() and pool.starmap() were used in order to compare their performance and pool.map() and pool.starmap() where found to provide the best results. It was decided to optimise and compare these two in order to obtain the fastest possible multiprocessing version. Both implementations with pool.map() and pool.starmap() are provided.

The achieved results are exactly the same as the serial program.

4. Threads program

Unlike the processes, threads share the memory because they run on the same memory space. On the one hand working with threads can be a big advantage for the performances of the program, on the other hand there can be GIL limitations and it is even more crucial to avoid global variables.

The schema of the program with threads is as follows:

1. Define the function that looks for occurrences. This function associates to every ID the number of times that the pattern appears in the string.
2. Read the dataset and save it as a DataFrame with two columns. Make every string uppercase.
3. Create an array of protein chains starting from the DataFrame.
4. Choose the number *n* of threads.
5. Split the array of data into *n* chunks, creating a list of arrays.
6. Read the input string and make it uppercase.
7. Initialize the time.
8. Run the threads and ensure that all of them have finished.
9. Stop the time
10. Create a DataFrame with the ID of the proteins with at least one occurrence.
11. Plot a histogram with the ID on the x-axis and the number of occurrences in the y-axis.
12. Print the IDs of the proteins with the highest number of occurrences.
13. Print the execution time of the part of the code in which the function under study is running.

The achieved results are exactly the same as the serial program.

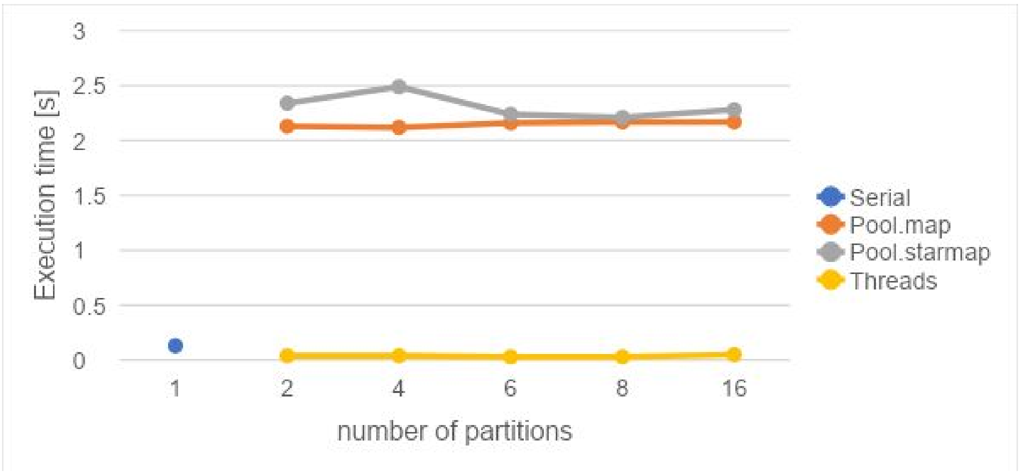
5. Results and conclusions

Here below the achieved results in terms of execution time are shown.
The conditions were maintained the same between each measurement. The machine on which the trials were performed has a 4 cores processor and Windows OS.

Execution time [s] with the original dataset

	Number of chunks					
	1	**2**	**4**	**6**	**8**	**16**
Serial	0.13	-	-	-	-	-
Pool.map	-	2.13	2.12	2.16	2.17	2.18
Pool.starmap	-	2.34	2.49	2.24	2.21	2.28
Threads	-	0.04	0.04	0.03	0.03	0.05

These results are plotted in a graph:



The following conclusions are stated:

- Programs with multiprocessing probably are finding some problems because a lower execution time was expected. Anyway it was not possible to find the root cause of it.
- Pool.map() and pool.starmap() were found to achieve better performance than pool.apply()
- The program with the threads is much faster than the serial one. The speed-up can be calculated as:

$$S = \frac{t_{ser}}{t_{par}} = 4.33$$

- The implementation with threads was capable of significantly improving the execution time, probably caused by the shared memory which allows faster access to the data.

- There is not a huge influence of the number of partitions on the execution time, considering a range between 2 and 16 partitions. However, the minimum execution time is chosen for each program.
- It was found that there is no difference in performance between using dataframes and lists as may be expected, possibly due to pandas functions acting on whole columns being optimised

A trial with a bigger dataset was done, to see how it affects the performance of each program.
In the table below the same programs are running with a 10 times bigger dataset. The best number of partitions was chosen for each program.

Execution time [s] with a 10x bigger dataset

	Number of chunks					
	1	**2**	**4**	**6**	**8**	**16**
Serial	1.28	-	-	-	-	-
Pool.map	-	-	3.89	-	-	-
Pool.starmap	-	-	-	-	4.81	-
Threads	-	-	-	0.40	-	-

With this bigger dataset, the new speedup is:

$$S = \frac{t_{ser}}{t_{par}} = 3.2$$