Scalable and Distributed Computing

Assignment 2 - Parallel Programming in Python

Rafaela Becerra Robalino, Marta Cortés Ocaña, Anna Subirós de Arriaga 29/02/2020

Contents

1 Introduction **2**

2 Theoretical Framework **2**
2.1 Parallelization in Python 2

3 Methodology **2**
3.1 Serial Programming 2
3.2 Parallel Programming 3
3.2.1 Multiprocessing 3
3.2.2 Threading 4

4 Results **4**

5 Conclusions **4**

6 Bibliography **5**

7 Appendix **5**

1 Introduction

The aim of the present work is the address of a serial problem to a parallelization process using Python.

The dataset intended to use is constructed with 345,180 sequences of proteins and contains two variables 'id' and 'Sequence'. The purpose is to create a serial program that matches a pattern introduced using the keyboard against all the proteins in the file, returning a histogram of the number of occurrences and the maximal number of it. The need for parallelization will be derived from the time cost of the computational complexity analysis of searching the pattern occurrence in each sequence, in consequence, a parallel program using multiprocessing and threads will be contrasted to measure performance.

2 Theoretical Framework

2.1 Parallelization in Python

In `python`, parallelization can be done using the `multiprocessing` module, which is used to run independent parallel processes by using subprocesses. While a process can be thought almost like a completely different program defined as a collection of resources, consequently, this could be interpreted as if each process runs in its own Python interpreter (Anderson, J).

The maximum number of processes will be restricted by the number of processors in the computer and can be used to control multiple processors on both Windows and Unix. Moreover, there are two main objects to implement a parallelization: Pool Class and the Process class. The Pool class is the most convenient to use and serves most in common practical applications (Prabjakaran, 2018), then is the one that will be used.

The multiple process spawning can be done using the Pool mechanism where the spawned process accepts and returns a single integer argument (Bogaerts, p. 34,52). This pool can be done in three ways: `Pool.apply`, `Pool.map`, and `Pool.starmap`, existing the option of an asynchronous variation, which does not involve locking and changes the order of results, for each one. `Pool.apply` creates processes to execute functions in parallel and handles multiple arguments, conversely, `Pool.map`, will allow only one iterable as the argument. Furthermore, the `Pool.starmap` will handle only one iterable as an argument, but this can also be iterable, then it is similar to pool.map but allows multiple arguments.

In contrast with Multiprocessing, Threading will run on a single processor and therefore only runs one process at a time. In threading, the operating system knows about each thread and can interrupt it at any time to start running a different thread (Anderson, J).

3 Methodology

3.1 Serial Programming

Initially, we need to create the serial program that finds the match of our pattern into the protein sequence.

In order to be able to work with data frames we have to import the `pandas` library first.

After that, we use the function `input` to ask the user for the pattern which has to be inserted in the console. To keep the program robust, there is a check done with a regular expression that confirms that the pattern inserted by the user only consists in letters from a-z independently on the upper or lower case. However, since the protein sequences are recorded in uppercase letters we have to transform the pattern with the function `.upper`.

Then, we create the function called `count_pattern` which has the arguments 'sequence' and 'pattern' and applies the function `count` which extracts the number of coincidences of our pattern in the protein

sequence by using the function `idxmax`.

Finally, we implement a *'for loop'* that executes the function `count_pattern` across all the lines (sequences) in the dataset, calculates the total number of coincidences of the pattern into each sequence, and stores that value in a new column.

Also, we save the time employed in the computation to compare it afterwards with the ones of the next sections. The same pattern is going to be used to make the time calculation easier. The time used in the computation of the serial program has been 481.8799 seconds.

For a more visual approach, we present the coincidences (without taking into account 0 occurrences) in a histogram plot using `matplotlib` library (Figure 1). The program also prints the sequence that includes the maximum number of the pattern.

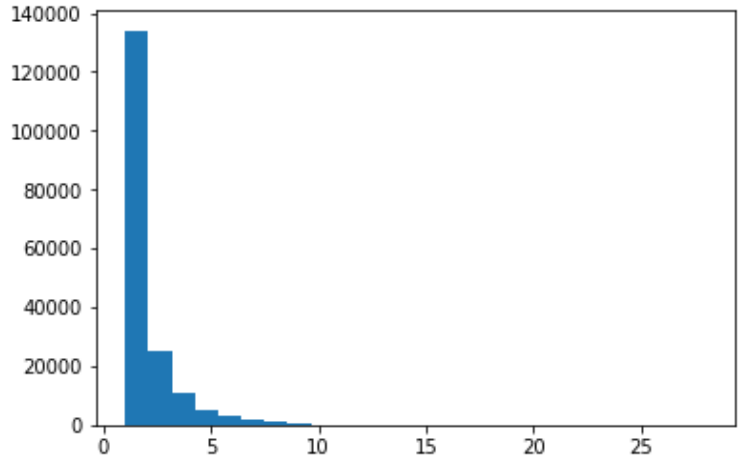

Figure 1: Number of coincidences of the pattern 'AR' into the sequences using serial program

Additionally, we have also implemented another function `count_pattern` for the serial program which uses a dictionary scheme to compare its computational cost with the one used above. The main idea of this new program is to convert the data frame in a dictionary and use keys and values to iterate instead of the rows of a data frame. In a 300 iterations run, the meantime of the program is 0.3547 seconds, as seen, this makes the speed increase drastically.

In order to measure times, as the program lasts less than a second, there is a possibility of generating many iterations to get the mean time of the program. This is done by inserting the main core of the program in a function that runs the number of times the user asks and returns the time of each iteration and the mean of the whole process. The histogram and the sequence are printed but only using the last result that will be the same as all of the previous ones and without randomness in the function.

3.2 Parallel Programming

3.2.1 Multiprocessing

Multiprocessing is a module of parallelization in Python which uses independent subprocesses to run a completed process. In our case, the limit number of subprocesses is 4 which is the number of processors of our computer.

In order to implement multiprocessing we need to import the `multiproccesing` library. Then, we repeat the same procedure of the serial program but we change the *'for loop'* for a *'pool'* process.

Initially, we have to open the pool with the function `mp.Pool` indicating the number of *'CPUs'*. The function chosen for doing the pool is the `apply` which works similar than the *loop*, applying the created

function `count_pattern` in each sequence of the protein dataset but splitting the effort between the four processors.

Finally, we close the *pool*, we save the time and input the histogram as done in the previous section and we obtain the same results both in the histogram and the sequence that has the maximum number of times the pattern asked. In this case, the time employed in the execution of the program has been 211.1536 seconds. It can be seen that the time has decreased by more than 50%. Instead of the `apply` technique, starmap is going to be used as there is more than one argument in the function.

Following with the previous section idea, we have changed the panda approach to the dictionary one and we have tried to emulate the serial program. The mean time for the multiprocessing using the `Pool.starmap` is 0.53626 seconds. This is around 0.20 seconds more than in the serial case. This means that, in this case, the parallelization is not worth as it takes more time to create the pools than the increase of time that supposes the parallelization.

3.2.2 Threading

On the contrary, Threading parallelization only runs one process using one single processors at a time but it can interrupt and changes for one thread to another anytime.

In this case, it is important to implement a function that does not need to work with a established order. In that way, we use the second type of our serial program as a reference because this is made with a dictionary and pandas does not work correctly with this type of parallelization.

We first import the `threading` library and then we repeat the procedure of the serial program applying the function `count_pattern` with the dictionary approach.

After that, we define the number of threads which is 3 and we run the function into each thread through a *'for loop'*. In order to end the process, we join the operation of all threads with the function `th.join`

Finally, we calculate the time employed in the execution using 300 iterations mean (0.8991 seconds) and we implement the histogram which, again, generates the same result than before.

4 Results

In Table 1, the conglomerate of the results for each approach is presented.

Approach	Serial	Parallel	
		Multiproccesing	Threading
Pandas dataframe	481.8799	211.1536	—-
Dictionaries	0.3547	0.53626	0.8991

Table 1: Summary of time results (in seconds) regarding the approaches, operation and method

It can be seen that multiproccesing parallelization get to decrease the computational time more than 50% with regard to the serial program in Pandas and it has also better performance than threading parallelization in the version of the dictionary program. Nevertheless, as it has been said before, parallelization is not worthy in the dictionary version.

5 Conclusions

Parallelization can be done to improve the performance time of serial programs that involve a high computational cost. When the functions presented are simple, paralellization may not be useful because it could increase the computational time.

The use of dataframes in Python derivates to an increase in the computational time when the target is a simple function. In this case, the use of dictionaries is recommended considering that the dataset contemplates a simple structure that can be transform to a collection of key-value pairs and has been demonstrated that the time of performance associated to dictionaries is better than Pandas data frames.

If there is a need of structuring the problem through Pandas data frames, with a multiproccesing parallelization a better time can be achieved.

6 Bibliography

[1] Anderson, J. Speed Up Your Python Program With Concurrency. Retrieved from: https://realpython.com/python-concurrency/

[2] Bogaerts, S., and Stough, J. (2015). "Chapter 3 - Parallelism in Python for Novices." Topics in Parallel and Distributed Computing. 2015. 25-58. Web.

[3] Prabhakaran, S (2018). Parallel Processing in Python – A Practical Guide with Examples. Retrivied from: https://www.machinelearningplus.com/python/parallel-processing-python/

7 Appendix

In the attach files the codes for the programs can be found:

- Using panda dataframes: only a pattern will be asked to the user
  - `serial.py`: Serial program
  - `multi.py`: Paralellization using multiprocessing program
- Using dictionary approach: a pattern will be asked to the user and the number of iterations to see the time results:
  - `serial_time.py`: Serial program.
  - `multi_time.py`: Paralellization using multiprocessing program
  - `thread_time.py`: Paralellization using threading program