UNIVERSIDAD CARLOS III

TECHNOLOGICAL FUNDAMENTALS IN THE BIG DATA WORLD

Lab 1 - Protein Matching in Python

Students:

100483840, Pablo ALONSO,

100492040, Nerea IZCUE,

100484916, Damián MALENO.

Professor:

Jesús CARRETERO Pérez

October 18, 2022

Contents

Introduction and Approaching

In the following report, we are going to present different approaches and implementations to solve a specific problem that consists in creating a program that counts the matches of a pattern introduced by keyboard against all the proteins sequences in a dataset using Python.

We are going to implement three different programs and see which approach works the best for this case and why. The first program consists in a **serial** processing of data, which is a type of processing in which one task is completed at a time and all the tasks are executed by the processor in sequence. The other two implementations consist in a **parallel** processing program. However, there are clear differences between them. The first one uses **multiprocessing**, which is a type of processing in which multiple tasks are completed at a time by different logical-processors and the second one uses **multithreading** processing, which is referred to having multiple processes working at the same time on a single CPU.

The first step before computing the programs consisted in generating a protein dataset, executing the given file named “proteins-generator.py”. We only needed to specify as an argument the number of rows that we wanted to generate. In order to make the execution faster, we tried our programs with 50,000 rows, and when we were sure that our program worked, we increased the number of rows to 500,000. Therefore, the file “proteins.csv" was created and included a data set with a list of protein sequences, including the “id" and “Sequence" per protein. The “Sequence" is the description of the protein components expressed as a chain of chars, where we have to look for matches with the pattern.

Serial program

The first program, as previously mentioned, consists in the implementation of a **serial** program. Before even running any of our programs, we can expect the serial to be the least efficient and the slowest of our programs. This is because when we implement serial processing of data, the central processing unit (CPU) carries out just one machine-level operation at a time. In contrast to what happens with a parallel processing, which features more than one CPU to performing the task.

However, the main advantage versus a parallel processing is that it’s less costly at a programming but also a computational level because it doesn’t need to spend time initializing multiple processes or threads. However, this implies that the workload of the serial processor is higher on a sole logical-processor.

Focusing on the serial program, we have defined two different functions. The first one, called *get-Sequence()*, asks about the pattern that we want to search in our sequences. However, we do not let the user give us an invalid pattern, so we only accept patterns that contain the letters A, B, C or D.

```
def getSequence():
    check = False
    while check == False:
        sequence = input("Input the pattern to search matching the regex ^[A-D]+
    $ :\n").upper()
        if re.match(r"^[A-D]+$",sequence):
            check = True
    return(sequence)
```

The second one, called *countMatches()*, is the serial function that reads the CSV file and finds the given pattern in the protein sequence. Using a dictionary, we keep the key, which is the "id" and the value, which is the occurrences of the pattern in the "Sequence".

```
def countMatches(Pattern, File):
    occurrences = {}
    with open(File) as file:
        reader = csv.reader(file)
        next(reader)
        for row in reader:
            occurrences[int(row[0])] = row[1].count(Pattern)
    return occurrences
```

Therefore, we run our program and the five "id" and the sequences with the most matches are displayed. Moreover, a barplot of occurrences using the protein "id" as X and the number of occurrences as Y, presents the 10 proteins with more matches.

```
if __name__ == '__main__':
    sequence = getSequence()
    t_stamp = time.time()
    hits = countMatches(Pattern= str(sequence), File= file)
    print("Elapsed execution time: ",time.time() - t_stamp," s")
    hits = {k: v for k,v in sorted(hits.items(),key=itemgetter(1),reverse=True)}
    print({k: hits[k] for k in list(hits)[:5]})
    hits_10 = dict(itertools.islice(hits.items(),10))
    plt.bar([str(i) for i in hits_10.keys()], hits_10.values(), color='g')
    plt.show()
```

The main part of the code of the serial processing consists in calling the *get_sequence()* function to save the pattern given by the user and then, pass the pattern to the function *countMatches()* in order to find the number of times that the pattern appears in the different sequences. The last part consists in creating a barplot to plot the "id" against the number of time the pattern is repeated in that sequence.

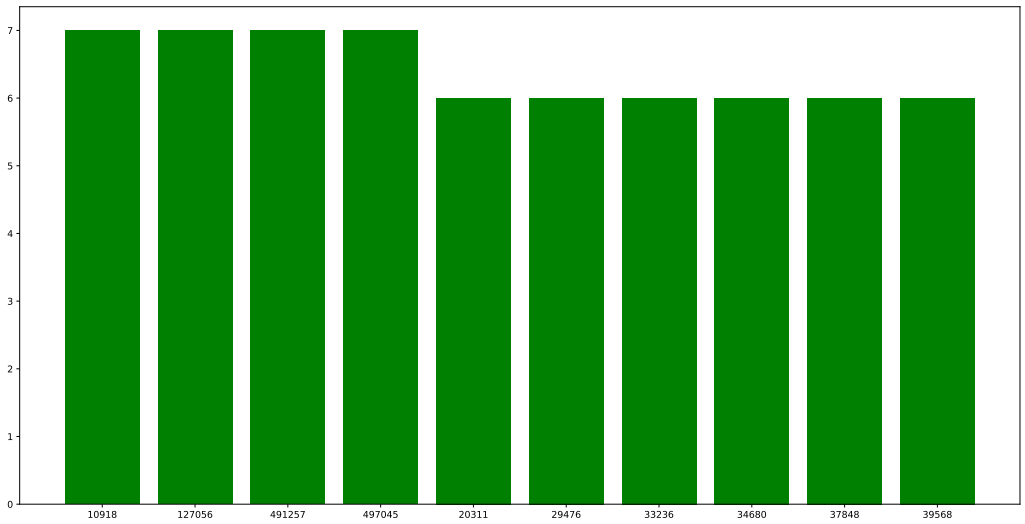

Figure 1: Barplot obtained after searching in the file with 500,000 sequences the pattern: “ABCD”.

Multiprocessing

In this second part of the lab, we are going to work with two different strategies of parallelism. The first one is the **multiprocessing**, which consists in running independent parallel processes by using sub-processes. Multiprocessing replicates processes, by replicating code, data, and files, which incurs in a computational overhead. Therefore, those processes can be run in completely separate memory locations.

Multiprocessing is very useful when the execution of a task or program is highly dependent on the CPU, because we will benefit from the fact that we are working with multiple processors. The only limitation of this method is that the maximum number of processes is limited by the number of logical-processors of the computer.

Focusing on the code we have implemented, let’s explain with more detail what we have done:

```
def getLength(file):
    return int(sp.getoutput("tail "+file+" -n 1 | tr -dc '0-9'"))
```

This function gives the number of lines our file has, which is basically the id of the last line plus one. Since we are only asking for numerical values, we specify that on the last part of the function *('0-9')*.

We have defined the *get_sequence()* function in the same way it was defined in the serial program.

```
def chunking(Lines):
    iterdata = {}
    cpus = mp.cpu_count()
    chunk_size = int(Lines/mp.cpu_count())
    for i in range (cpus): iterdata[i*chunk_size+1] = (i+1)*chunk_size+1
    return iterdata
```

The third one is the *chunking()* function. Knowing the number of cores of the computer, we will determine the rows of the file that we will give to each process. We are trying to create a process for each core because doing so, we are parallelizing as much as we can. This is why we want to set the chunk size as an integer that results of the division of the total length of our protein file by the number of cores.

```
def countMatches(Start, Stop, Pattern, File):
    occurrences = {}
    with open(File) as file:
        reader = csv.reader(islice(file,Start,Stop))
        for row in reader:
            occurrences[int(row[0])] = row[1].count(Pattern)
    return occurrences
```

The *countMatches()* function has the same behavior as it had in the serial program. The function reads the CSV file but now, the start and stop argument are defined by the chunk size created in the *chunking()* function. Therefore, we have parallelized the read of the file, dividing it in as many parts as cores we have. Hence, each processor reads simultaneously from its starting row to its ending row, looking for the matches of the given pattern in the protein sequences. Using a dictionary, we store the id as the key, and the occurrences of the pattern in the sequences as the value.

```
if __name__ == '__main__':
    sequence = getSequence()
    iterdata = chunking(Lines= getLength(file))
    iterdata = [(start,stop,sequence,file) for start, stop in iterdata.items()]
    t_stamp = time.time()
    pool = mp.Pool(mp.cpu_count())
    results = pool.starmap_async(countMatches, iterdata)
    hits = results.get()
    pool.close()
    print("Elapsed execution time: ",time.time() - t_stamp," s")
    hits = dict(ChainMap(*hits))
    hits = {k: v for k,v in sorted(hits.items(),key=itemgetter(1),reverse=True)}
    print({k: hits[k] for k in list(hits)[:5]})
    hits_10 = dict(islice(hits.items(),10))
    plt.bar([ str(i) for i in hits_10.keys()], hits_10.values(), color='g')
    plt.show()
```

The last part is the main code, where we use all the functions we have defined. First, we ask the sequence we are looking for in our protein file, calling *getSequence()*. Secondly, we use the *chunking()* function, giving the length of the file we are going to inspect looking for the given pattern as an argument.

At this stage, *iterdata* is a dictionary but in the next line convert it into a list by adding start and stop, which are the *iterdata* items that indicate the starting and ending of the chunk that each processor works over. Then we apply the *starmap_async* function given the function *countMatches()* and the argument *iterdata*.

Finally, we plot a barplot showing the sequences that have the given pattern more times repeated.

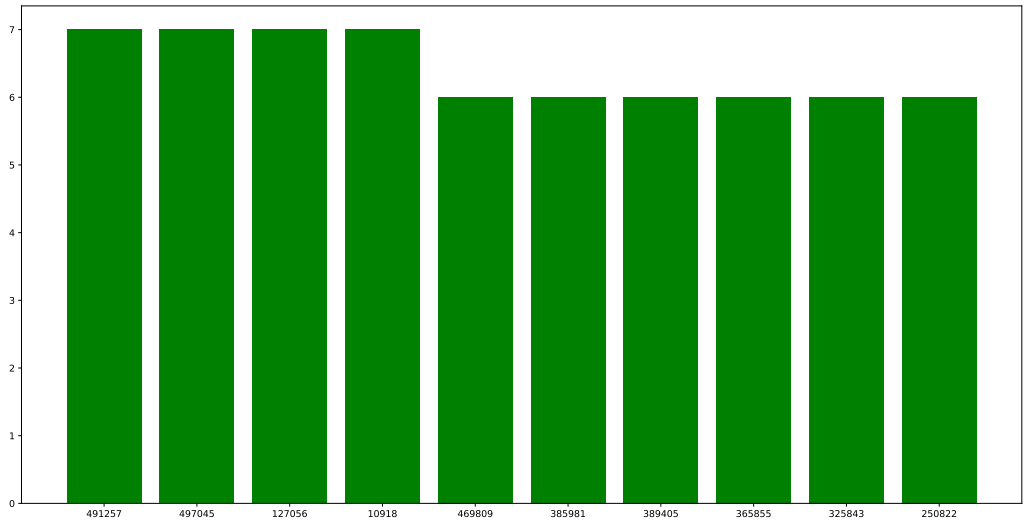

Figure 2: Barplot obtained after giving the pattern “ABCD”.

Threading

In this section, we are implementing another parallel program, which is the **threading** alternative, that performs the execution of multiple threads concurrently, where each thread runs a process.

As opposite as what happened with multiprocessing, threads share their address space, sharing the same code, data, and files but run on a different register and stack. This saves system memory, increases computing speed and improves application performance.

Multithreading is useful for programs with a large number of I/O operations, such as reading different files from a database, since each thread can run the IO-bound process concurrently. However, if we use multithreading for CPU-bound processes, the performance might slow down because of the competing resources that ensure only one thread can execute at a time, and an overhead is incurred when we are dealing with multiple threads.

Let’s break down our program and see how we have implemented the threading solution to solve our protein matching problem.

The functions *getLength()* and *get_sequence()* are the same two functions previously introduced.

```
def chunking(Lines, Num_threads):
    portions = []
    chunk_size = int(Lines/Num_threads)
    for i in range (Num_threads):
        with open(File) as file:
            reader=csv.reader(islice(file,(i*chunk_size+1),(i+1)*chunk_size+1))
            portion = []
            for row in reader:
                portion.append(row)
            portions.append(portion)
    return(portions)
```

In this case, the *chunking()* function is very similar to the previous one defined in the multiprocessing program. However, in the threading we perform the reading of the file in a serial way - the reason will be explained later. Finally, we save each fragment of the CSV file in a list prepared for the pattern detection of the threads that will be parallelized in the main code.

```
def countThread(Pattern, Rows):
    occurrences = {}
    for row in Rows:
        occurrences[int(row[0])] = row[1].count(Pattern)
    mutex.acquire()
    results.append(occurrences)
    mutex.release()
```

We have defined a counting function called *countThread()*, which is the one that we are going to parallelize in the main code with threads. In this case, we give the pattern and the rows as arguments and with that information, the function counts the number of occurrences of the given pattern. It is important to mention that adding a *mutex lock* avoids any type of concurrence with the global variable *results*.

```
if __name__ == '__main__':
    sequence = getSequence()
    num_threads = 4
    t_stamp = time.time()
    iterdata = chunking(Lines= getLength(File), Num_threads= num_threads)
    results = []
    mutex = Lock()
    iterdata = [(sequence, rows) for rows in iterdata]
    ths = []
    for i, thread in enumerate(iterdata):
```

```
        th = threading.Thread(name='th%s' %(i+1), target= countThread, args=
    iterdata[i])
        th.start()
        ths.append(th)
    for thread in ths:
        th.join()
    print("Elapsed execution time: ",time.time() - t_stamp," s")
    hits = dict(ChainMap(*results))
    hits = {k: v for k,v in sorted(hits.items(),key=itemgetter(1),reverse=True)}
    print({k: hits[k] for k in list(hits)[:5]})
    hits_10 = dict(islice(hits.items(),10))
    #plt.bar([ str(i) for i in hits_10.keys()], hits_10.values(), color='g')
    #plt.show()
```

The main part of the code is similar to the other programs. We call the *getSequence()* function to ask the pattern and set the number of threads at 4, which is the optimal number found for 500,000 lines in our case - it might change depending on the system where the execution is performed. We define an iterdata adapted to the lines of the file and the number of threads, which later embodies the pattern to match. The *countThread* function call the basic function *threading.Thread()* in which we give as an argument iterdata. Finally, in the same way as the other programs, we plot the barplot with the sequences that has the pattern more times repeated.[1]

Previously, we said that the optimal number of threads was 4, due to the fact that different runs were performed with a different number of threads, and we chose the optimal number of threads depending on the shortest execution time. We provide the table that shows that information below:

Nº PROTEINS	**50k**	**500k**	**5Million**
2 Threads	0.1511	1.5656	14.488
4 Threads	0.1759	1.3083	13.9932
5 Threads	0.1749	1.5385	13.8675
6 Threads	0.1648	1.6185	13.4445
8 Threads	0.1375	1.6543	16.4327

[1] Every barplot has to look the same. This is because they must return the same values since we are running the same protein file, but the order of apparition can change.

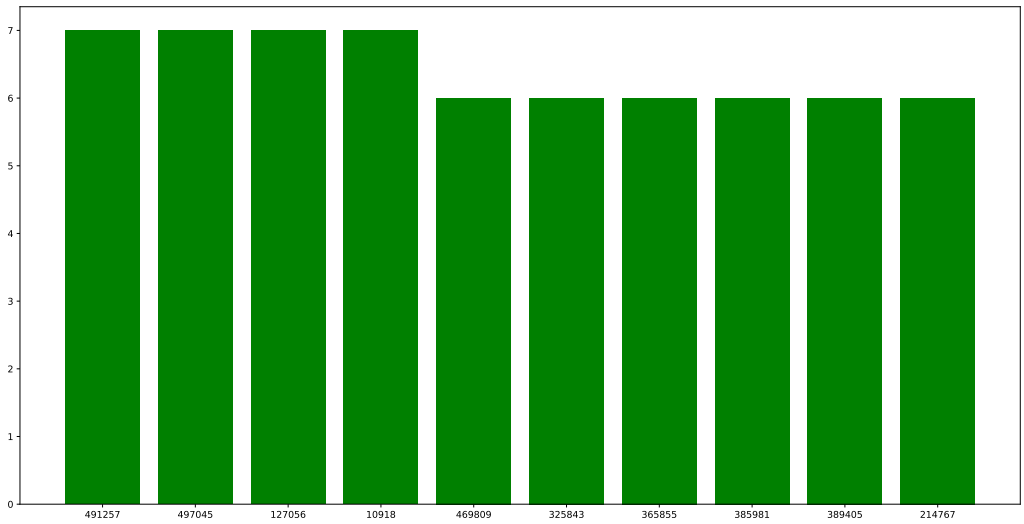

Figure 3: Barplot obtained after giving the pattern “ABCD”

Considerations, Result Comparison and Conclusions

Before comparing the results, we should highlight some considerations about the most important choices we had to make. Related to the **multiprocessing** parallelization, we have used the *Pool* library, and specifically we decided to implement it using the *starmap* function. We considered using the *apply* function, but it was a more general function and its performance was slower compared to the performance of other functions. We also dismiss the *map* function because it was not applicable in this case, since we are working with an iterable of iterables and this function doesn’t allow that type of argument. Therefore, considering the type of argument we had, we decided that the appropriate function was the *starmap*.

The other important choice about multiprocessing was selecting if we were using the synchronous or asynchronous *starmap*. On the one hand, if we are concerned by the order of the elements, we must use a synchronous method. However, if we are not, which was our case since we don’t care abut which protein is analyzed first, the asynchronous was the optimal way to proceed to get a better performance. In conclusion, we decided to apply the *starmap_async*.

Related to **threading** parallelization, the main concern was selecting the portion of the code to parallelize to obtain the optimal execution time. In the multiprocessing program, we parallelized the reading of the CSV file and the count of occurrences in each protein, so we thought that would be an optimal approach in the threading parallelization. However, after its implementation, we realized it was not the best option, since the threads share the same environment and reading the same file at the same time caused the execution to be interrupted momentarily. This is because when applying multiprocessing, the space is duplicated and each process works over its own environment, so this was not a problem. Nevertheless, this doesn’t happen when using threading and for doing so, threads need to wait until the previous thread

finishes reading the file, blocking the rest of the threads until it is done. Hence, this wastes a lot of time and resources. Therefore, we decided to only parallelize the count of occurrences and not the reading.

In the following table, we show the different execution times between both implementations.

Nº PROTEINS	**50k**	**500k**	**5Million**
First implementation	0.1266	1.891	17.897
Second implementation	0.1375	1.3083	13.4445

After explaining the reasoning behind our implementations, we are going to present a table with the different execution times between the different programs and number of protein sequences in the file, interpreting the results.

Nº PROTEINS	**50k**	**500k**	**5Million**
SERIAL	0.1023	0.9788	9.6764
MULTIPROCESSING	0.0524	0.3593	3.3345
OPTIMAL THREADING	0.1375	1.3083	13.4445

Looking at the results[2], we can conclude that our predictions were in part wrong. In this case, the **serial** program works better than the **threading** program, but worse than the **multiprocessing** program. This is due to the fact that using threads is not the best idea when we need to access and update the same resource (set a variable, write to a file). Therefore, the more common problem when applying threads is blocking for resources, since they can access every address in the task.

In our first threading implementation, the program was not very efficient due to the lost in performance related to the read of the file, since each thread has to wait until the previous thread ends reading the file, resulting in a bottleneck. In our current threading implementation, this problem was solved by moving the file read to the sequential part of the code, which still being underperforming if compared to the multiprocessing approach but faster than our first threading approach. However, this improvement makes our program not scalable to files ten or more times larger, since the memory limit of Python variables would be exceeded (although it works perfectly with the file with which it is going to be evaluated).

However, when we talk about multiprocessing, we see that one of the main advantages is that it helps to speed up CPU-bound computations, taking advantage of multiple cores and multiple processors. It also can help to reduce I/O time by overlapping computation and by reading chunks of files. This is clearly reflected in the execution time, which shows that the multiprocessing is the best option running this program.

[2]Our results were tested using a machine with 8 cores (i7-8550U) running Ubuntu on Hyper-V

In conclusion, we obtain the best performance when applying the **multiprocessing** program. Our program is optimized when using it, and it is reflected in the execution time. The second-best performance, and somehow surprisingly, is given by the **serial** program, that give us a really good result. However, the **threading** program doesn't work well in this case, since it is not the best choice for the conditions we had in our task. This doesn't mean that threading programs are not optimal in other cases, but they are not when facing a problem like this one.

Finally, we show a speed-up table that compares these three different programs and reflects what we are commenting in this report.

SPEED UP	**50k**	**500k**	**5Million**
MULTIPROCESSING	1.9523	2.7242	2.9019
OPTIMAL THREADING	0.744	0.7481	0.7197