Nasri Chiheb

UNIVERSITAT ROVIRA I VIRGILI

Secure Distributed Systems Architectures 2020/21

Assignment#3 (Practical)

Introduction:

The aim of this assignment is to implement a python function for matrix multiplication. The implementation must be designed in a distributive way. In next steps of the implementation the second matrix can be homomorphically encrypted with Paillier cryptosystem. In this report we will describe the implementation characteristics, then describe the provided solution and finally, discuss the results. In addition, a how to run part is added in the end of this report. The fully commented code for more details is attached to this report.

Implementation description:

The implementation was made under python 3.7 by using different libraries. Here we used NumPy, Lithops , Pickle and Phe libraries. We use NumPy library for several reason. First it is easier to manipulate matrices thanks to NumPy arrays as we can access data per column. Furthermore, when encrypting the data one of the only ways to make block multiplication is by passing with NumPy arrays. NumPy also implements function to multiply row blocks and column blocks but also matrices so we cam compare our implementation with the one of NumPy

Lithops is used to add parallelism to our solution. It permits to create workers and make them work simultaneously. Lithops also provides storage function to make data transfer easy and less time consuming between the workers.

Pickle is used to deal with high size data. That goes through the storage. With pickle we can convert it in bytes and not in string which permits to keep data in the same size (approximately).

Phe library is used to encrypt one of the matrices. It has implemented the Paillier cryptosystem that crypts data with a public-private key pair. With this we can crypt the data send it to the server to make the multiplication and take decrypt it locally. Thanks to the homomorphically encryption we can multiply encrypted and raw data without having to decrypt it.

Solution:

The implemented solution will can handle two matrices with respective size m*n and n*l. Given these two matrices we will divide it in a*n blocks to compute each bloc of final solution independently. The figure 1 extracted from the subject illustrates the main Idea.

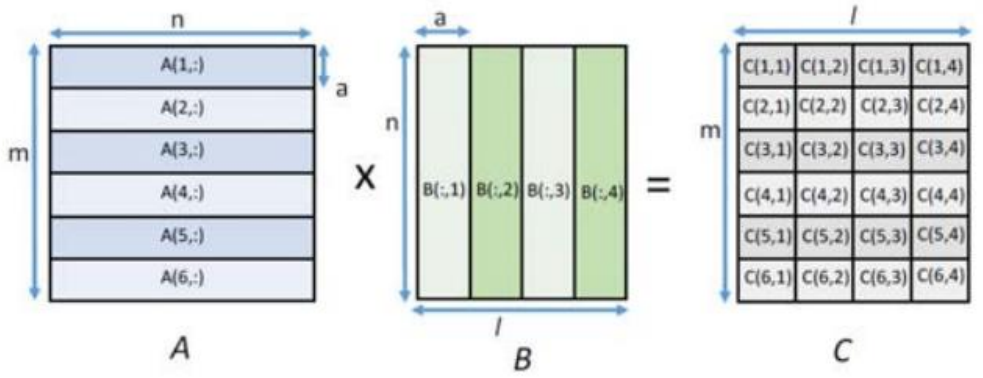

Figure 1. Graphical representation of matrix multiplication.

For our solution starting from two matrices and the desired size of a we will get the indices of all the blocks. Additionally, to the indice,s we also provide the storage name space. Then we will instantiate an empty C matrix of size m*l. Those indices will be passed to the map function. The map function will multiply the row blocks and the column blocks and store it in C in right place as the indices are provided. Finally, we store the Ci block in the storage with the worker id for i.

To do so we will create a storage object, that represents the server for local version. This storage will have the matrices A and B. Then we will use Lithops to create as many workers as we have tuple indices, in other words we will have as many workers as we have blocks. Each worker will use the map function and one tuple of indices with the name space for storage. All the workers will work simultaneously and store the results in the storage. Then we wait until all workers finish the job and request the result from the matrix. Finally, we have two choices either reduce locally the result to get final result or reduce using a reduce function. In both cases we will call all blocks and sum them.

The crypted version of this code is quite the same. The difference is when we create matrix B for example, we ill encrypt it with Paillier algorithm and transform it back to a NumPy matrix. This crypted matrix will do the same as explained before. Once the final solution is composed, we get it in local machine and decrypt the matrix with the private key. Another difference stands for the zeros matrix when we instantiate C. We have to define the elements as objects otherwise Python returns a problem of type compatibility. We cannot define Object as floats.

Results:

In order to compare the results, we will use execution time by varying matrices sizes , block sizes and either reducing in local or with a reduce function. Note that all the tests have been made in local and not in cloud.

First test will be with : m = 8 / n = 4 / l = 12 / a = 2

```
--- 2.112208127975464 seconds for uncrypted---
--- 0.0 seconds  for numpy function---
```

```
--- 7.022959232330322 seconds for crypted without decrypt---
--- 9.592934846878052 seconds for crypted with decyrpt---
```

With reduce function :

```
--- 2.90632700920105 seconds for plain---
--- 0.0 seconds  for numpy function---
```

```
--- 10.830482006072998 seconds for crypted without decrypt---
--- 13.473206043243408 seconds for crypted with decyrpt---
```

As we can see for this size of matrices the NumPy function is still faster. Adding encryption multiplies execution time by 7 but it grows more with bigger matrices . Also using reduce function increases time but not that much. It is because again it is not designed to work locally and for matrices that small

Now we will increase the matrices size but for time execution problems we will not perform encryption for these tests.

Second Test = m = 3000 / n = 1500 / l = 3000 / a = 1000

```
--- 21.41427779197693 seconds for uncrypted---
--- 29.017608165740967 seconds  for numpy function---
```

Here we can see that when going on bigger matrices our function is faster than the NumPy function. As we give higher matrices the difference will be bigger but my SSD is running out place (128gb SSD) so I cannot work on higher values.

Third test we will see the difference by varying the size of blocks.

a = 500 / a = 1500 with same last parameters

```
--- 44.41523766517639 seconds for uncrypted---
--- 26.630103588104248 seconds  for numpy function---
```

A = 500

```
--- 12.933658838272095 seconds for uncrypted---
--- 26.496633052825928 seconds  for numpy function---
```

A = 15000

We can that choosing too many blocks when you have not many available cores will increase the execution time. Also choosing better block size value gives better time.Final remark will be that result depends also on the hardware that might have more cores, mor ram, use SSD…

Conclusion:

As we saw in this report computing matrix multiplication can go computationally expensive very fast. The presented presentation is not implemented in cloud version which reduce the results scope as it was designed for cloud use. Even though using cloud with this version is quite easy. Working on new libraries is quite interesting, I have already worked with multiprocessing in python but not with Lithops. Adding encryption increases highly computational time, this is explained by the fact of transforming a simple integer into bigger hashed number.

How to run:

Prerequisite :

- Python(Python 3.7 used for the implementation)
- Packages : Numpy , Lithops , pickle , phe (all of them in last version)

Run :

- Launch main.py for classic version
- mainRed.py for reducer version