Project Spark Report - Part 2

Alexandros Matakos

Student ID: 015538460

1 Method

For this part of the project, the task was to compute $A \cdot A^T \cdot A$, where A is a 1000000 x 1000 matrix. A key insight for the speed-up of this calculation was that calculating $A^T \cdot A$ first and $A \cdot (A^T \cdot A)$ afterwards, results in less operations than if we were to calculate $(A \cdot A^T)$ first and $(A \cdot A^T) \cdot A$ afterwards. This is true in general for any matrices $P \cdot Q \cdot R$, with P, R having more rows than columns and Q more columns than rows.

In our case this can be confirmed as follows: since A's size is 1000000 x 1000, then A^T's is 1000 x 1000000, and the calculation $A \cdot A^T$ results in a 1000000 x 1000000 matrix, while the calculation $A^T \cdot A$ results in a 1000 x 1000 matrix. When the result of the first multiplication is multiplied by A again, in the first case we need to do dot products between rows that have 1000000 elements, while in the second case only 1000 elements.

Besides the smaller requirements in memory, computing $A^T \cdot A$ first is also more compatible with the MapReduce paradigm. The reason is that it allows the rows to be arbitrarily long (scalable), while the number of columns is small enough so that they can be handled by each worker independently, so that the matrix can be partitioned row-wise.

The calculation $A^T \cdot A$ was done using the following fact. I noticed that for any matrix $A \in \mathscr{M}_{nxm}$, it is true that

$$A^T A = \sum_{i=1}^{n} u_i u_i^T,$$

where u_i is the i-th row of A written as a column vector. This immediately opens up the possibility of distributing this calculation in a MapReduce environment like Spark, since the terms of this sum are independent and for each partition's rows, the outer products can be computed independently on different workers. Then a reduce operation can add all the terms and return the resulting matrix $A^T \cdot A$. Spark implementation of this:

```
AT_A = A.map(lambda row: np.outer(row,row)).reduce(add)
```

For the second multiplication, $A \cdot (A^T A)$, we note the following: since Spark's RDD operations are lazy, we can define the multiplication for the whole matrix without adding any operational cost to the execution. Then, when we extract the first row of the resulting matrix, only the first row of A is multiplied with $A^T \cdot A$ to produce the result, essentially avoiding most of the computation. This is a valid implementation in line with the task description, since the code can compute the whole matrix. Spark code:

```
A_AT_A = A.map(lambda row: np.dot(row, AT_A))
result = A_AT_A.first()
```

One note for the final step: I believe the use of np.dot() there is justified, since the multiplication involves one row of length 1000 and a 1000 x 1000 matrix, a calculation that would be done locally in any commodity server. As such, I do not believe this violates the prohibition of using implemented multiplication methods optimized for distributed matrices.

2 Implementation

As is obvious from the previous section, my solution uses numpy. I tried implementing the outer product with python lists and for loops, but the result was much slower, essentially making the running time infeasible. I tried to find some other way to implement it so that I could run tests on the cluster, but did not succeed. Therefore, I set up a Spark cluster on my machine, downloaded the data-2.txt file and run experiments locally.

My machine uses 32 GB of 3200MHz RAM, a Ryzen 7 4800h 16-core 4.1GHz processor and has an average read/write speed of 3GB/s. I set up Spark to use 10 cores with 1GB of memory for each, to replicate the cluster conditions. My code took about 1 hour to run for the project task, with almost all of the computation time spent on the outer products (500 outer products / sec on average). The results are attached in the answer.txt file.

The code used for the calcultion is a bit different than the method descibed above. The reason is that the method described above (essentially those 3-4 lines), would work for artificial matrices of up to 10^5 rows, but it would throw an error for the full size dataset. For 10^5 its running time was 220 seconds.

The way I overcame this difficulty was by dividing the dataset to 40000 partitions, and then using the RDD.mapPartitions() function to iterate through each partition.

```
AT_A = np.zeros((1000,1000))
for partition in A.mapPartitions(
            lambda part: [list(part)]).toLocalIterator():
    for row in partition:
        AT_A += np.outer(row,row)
```

Each partition consists of 25 rows. In each iteration, the partition is converted to a list, and for each row, the outer product with itself is computed and added to the result matrix. This piece of code encapsulates both *map* and *reduce* steps of the previously described method.

The rest of the code can be found in the part2.py file. The answers are attached as the answers.txt file.