MAP543 - Database Management

Matrix Multiplication

Alexis Laks - Elliot Stocker

Contents

1 - Introduction : Problem Motivation

Matrix multiplication. Consider the problem of multiplying two big matrices A n,m and B m,n . Design MapReduce algorithms for this problem, knowing that up to two jobs may be needed. Consider the simple textual representation of a matrix where the value v for A i,j is represented by a text line 'i j v'. Only non-zero values will have a line in the textual file representing a matrix. Provide at least two non-trivial different algorithms, and implement them in both Python with Hadoop streaming and in Spark. Perform experimental evaluation by considering 5 couples of matrices of increasing sizes, by doubling the sizes from a pair of matrix to the subsequent pair. You can use Spark to generate matrices.

2 - Algorithms Description

Before jumping into the details of our algorithms we would like to point out that we assume that we know the size of our matrices, which is essential to make our programs work. We assign these indices at the beginning of our programs when they are needed. We also assume that the matrices are sorted by line then by column for the second algorithm. The input file we work with is a pure text file containing the information for matrix A then for matrix B in the form: Provenance, row, column, value. We have used a matrix generator which yields matrices in that order and does not print values that are equal to 0. You will find all our test matrices and code on a dropbox (link in appendix) :

We decided to show you in order of development the ideas we had to run matrix multiplication in MapReduce. Note that the spark implementation may vary slightly from the original MapReduce program as the format of Spark limits the way we can develop our algorithm.

A - Algorithm 1 (Replicating)

We consider two matrices A of size (n x p) and B of size (p x m) that will produce a final matrix C of size (n x m). We know that for every element of a line i of C all the elements of a same line i of A will be needed, similarly for every element of a column j of C all the elements of column j of B will be needed. Given this need, our first thought was to replicate as many times as needed every element of a line in A for every line of A and every element of a column in B for every column in B. For the replicates of lines of A, we will need as many replicates as there are columns in our final matrix C (or as many columns as there are in B) and vice-versa for the replicates of columns of B which will be needed as many times as there are rows in C (or A). Once these replicates are yielded, we then group them according to the element in C where they will be useful, this is our (key, value) pair output of our mapper. For example for the first replicates of the first line of A, they will be attributed the key (0,0). These replicates will be used again and again for as many elements there are in line 0 of C, namely m times or the number of columns in C so the same values will be attributed a key (0, i) for i in range(0,m). The same goes for the first replicates of the first column in B which will be attributed to a key (j,0) for j in range (0,n), n being the number of rows there are in C. We designed a 1 MapReduce job for this task that you will find in the following (Note: Be careful when you wget … from our dropbox, we often get bizarre end statements when downloading, if this happens with you as well just use *mv bad_format good_format*):

wget https://www.dropbox.com/s/ppy7krfkdne5pdg/mapper.py

Given that stdin reads a text line by line, the order in which we print the elements of a line of A or a column of B changes. For a line of A the task is easy and will print all the replicates in order. For B this does not happen, it will retrieve one element for each of its columns before going to the second element of a given column. Thus the map task is as follows:

```
If provenance of element is matrix A:
      for k in range (number of columns of B):
            key = row(element), k
            print (key, column(element), value(element))
if provenance of element is matrix B:
      for j in range(number of row of A):
            key = j, colum(element)
            print (key, row(element), value(element))
```

The element i,j of C is a sum of the product between the first element of the line i of A and the first element of the column j of B and this until the last element in both line and column of A and B respectively. This mapper was designed to output after S&S in the order mentioned beforehand:

((i,j) element(1) of line i of A

(i,j) element(1) of column j of B

(i,j) element(2) of line i of A

(i,j) element(2) of column j of B

Consider elements of either matrices with value 0 are printed, then for each key you just do the product between the values of A and B while incrementing by 2 and sum when you've gone through the product of all elements associated to a same key. Unfortunately elements having value 0 are not printed, and you would end up mixing up products and getting false results. This intermediate value is there to control for such sparsity, if an element in line i of A does not have its homologue in column j of B because that value is 0 then you would have the following map output:

```
(0,0), 0, -2
(0,0), 0, 1
(0,0), 1, 4          # Missing value of either A or B!
(0,0), 2, 2
(0,0), 2, 1
```

This intermediate index allows to find these “holes” due to sparsity, by controlling this index by an updated (current_index) we tell our reducer that the homologue of this lonely value is a 0 and it does not require a computation (since in any case the result would be 0 and would not change anything to the sum) and can therefore move on to the next double pair. I values are missing in both matrices it acts just as it would normally do if there wasn’t sparsity. Here’s the code for our reducer:

wget https://www.dropbox.com/s/t05g6u9rmm5fx7n/reducer.py

We initialize a current_key at (0,0) as the mapper output after S&S will always output that key first. As long as stdin reads the same current_key it will do the product of the pair of elements from A and B as mentioned before as long as the intermediate index agrees and append it to a temporary list. If it sees two distinct indexes but for the same key it will reatribute the second value with a different index to the first element of the next pair. When stdin finally reaches a different key it will sum the products within the temporary list and assign it to the key it had read until then. The reducer does this for every key retrieved from the map output.

Spark implementation:

For the first command just put your pwd to the matrix you want to test, and don’t forget to manually change the length and size parameters in the mapper !

```
data = sc.textFile("")
```

#split data into pairs

```
pairs = data.map(lambda x: x.strip().split(','))
```

# define a map function similar to the one described in our previous mapreduce algorithm, remember to update the range depending on the size of the matrix!

```
def mapper1(x):
   map = []
   if x[0] == "A":
      for i in range (4):
          map.append([(int(x[1]),int(i),int(x[2])),int(x[3])])
```

```
    else:
        for j in range (4):
            map.append([(int(j),int(x[2]),int(x[1])),int(x[3])])
    return(Y)
```

# Apply our previous map function

```
pairs2 = pairs.flatMap(lambda x: mapper1(x))
```

# Transform Pipeline into an RDD !

```
pairs3 = sc.parallelize(pairs2.collect())
```

# Deal with sparsity with intermediate index and do the element wise product of all values of A and B necessary for the element of C

```
pairs4= pairs3.groupByKey().mapValues(list).mapValues(lambda x: x[0]*x[1] if len(x) > 1 else 0)
```

# Create output with key being the index of the element of C computed and its respective value:

```
output = pairs4.map(lambda x: ((x[0][0],x[0][1]),x[1])).reduceByKey(lambda x, y: x + y)
output.collect()
```

Benchmarking:

In terms of performance, the limit of this MapReduce algorithm is for two 600 x 600 matrices. Locally (Mac - OSX 10.11.6 - 2,8 GHz Intel Core i5 - 8 Go 1600 MHz DDR3) it takes up to 50 minutes to retrieve values, the main time taken is by the sort operation which has to deal with 600 replicates for each of the 600 x 600 elements in each matrix. Thus the sort operation must be done on 600 x 600 x 600 x 2 (432 million) elements for an operation of complexity O(nlog(n)). Although if we downgrade this matrix size by 3 (two 200x200 matrices) the operation takes only 8 minutes (with print operation from terminal). The major drawback of this algorithm is in the replicates it creates that may be unnecessary, although this allows the program to be very highly parallelizable and can be useful for very large matrix multiplication on big hadoop clusters.

- A (600 x 600) & B (600 x 600) :
    - Local: ~50m
    - Hadoop: 5m50s
- A (300 x 300) & B (300 x 300) :
    - Local: ~23m
    - Hadoop: 1m30s
- A (150 x 150) & B (150 x 150) :
    - Local: ~2m40
    - Hadoop: 55s
- A (75 x 75) & B (75 x 75) :
    - Local: ~16s
    - Hadoop: 35s
- A (35 x 35) & B (35 x 35) :
    - Local: ~2s
    - Hadoop: 25s

We can see the from our results that hadoop greatly improves speed of execution for higher dimension matrices, which is greatly allowed by the parallelism of this procedure. Although there is a certain limit where running locally is faster given that communication, splitting and redistribution can take time and may be un-optimal for smaller matrices.

B - Algorithm 2 (Slicing)

Given the drawback in of our previous algorithm in the replicates it needed to generate we thought of a way to do matrix multiplication without using replicates. The idea is that instead of grouping elements according to the final element in C for which they will be needed we create two lists of length (n*p for A and p*m for B) that will contain all the elements of A and B respectively. Given that we know the dimension of both A and B, we can retrieve the elements of a line i of A using slicing, same goes for B where its elements are assembled according to its columns. For example if we have a 2x2 matrix for A and B, we know that the first 2 elements in our A_list containing all elements of A will correspond to the first line. Same goes for B, first two elements of B will correspond two column 1 of B. Given this information we can thus use slicing in order to retrieve the needed elements of A and B for every element in C. For example element (2,2) of C will be done by : np.dot(A_list[4:6],B_list[4:6]). Here is the code for our mapper:

wget https://www.dropbox.com/s/on04uam92ir18xv/mapper.py

Note that here the key is no longer the indexes of the element of C it will be needed for but it's provenance (A or B) and the row (or column for B) it apparteined to in the first place. We are basically printing out every element of A reading left to right and B up to down. The intermediate index is used for the indexing within either A list or B list, it's to keep track of what order an element has within a row of A or a column of B. This output is then moved to the reducer:

wget https://www.dropbox.com/s/f8y3xx023v43kct/reducer.py

To deal with sparsity a list filled with only zeros is originated before filling the list, as if a (key,value) pair is missing overall it will not affect the dimension of either A_list or B_list and will stay a 0 at the specific placement it should have been if it had been any value except 0. For values that were indeed printed, their position within the list they appartein to will be the n x i + intermediate_index for i in range len(A_list) for an element of A and m x i + intermediate_index for i in range len(B_list) for an element of B. To illustrate this say we take A (2x2) and B(2x2), the output of the mapper for the 2nd element of the 2nd line of A

(A,1,1,-4) will be (A1,1,-4) and thus its position in A_list will be A_list[2*1 +1] = -4. This scheme is repeated for all existing (non-zero) elements of A and B.

Once all the elements of A and B are attributed in the right position in A_list and B_list respectively, we then slice each list according to the element in C we wish to calculate. So say for element C(2,2) (Still in the case of A (2x2) and B(2x2)) we know we need the 2nd line of A and 2nd column of B, we thus retrieve A_list[2:4] and B_list[2:4] for which we sum the element-wise product of both. We then print the result for C(2,2) with the according value. To go through all elements of C (2x2) we use a double for loop ranging from 0 to 2(=n) and 0 to 2(=m).

Spark implementation :

In order to run our spark code for this algorithm you just need to copy paste

#Get the data:

```
data = sc.textFile("file:///pwd_on_your_computer")
```

# Strip and split all elements in the data:

```
pairs = data.map(lambda x: x.strip().split(','))
```

# Same map scheme as in previous mapper, remember to reset the number in the for loops depending on the size of the matrix taken (since we are in python you must add 1 to the actual size of the matrix) :

```
def mapper(x):
    map = []
    if x[0] == "A":
        for i in range(4):
            key = "A" + str(i)
            if x[1] == str(i):
                map.append([(key,x[2]),x[3]])
    else:
        for i in range(4):
            key = "B" + str(i)
            if x[2] == str(i):
                map.append([(key,x[1]),x[3]])
    return(map)
```

# Apply map function on all pairs:

```
pairs2 = pairs.flatMap(lambda x: mapper(x))
```

# Now we want to recover our two lists A and B in the order we explained previously, first we need to sort to be sure that we have elements of matrix A in order of left to right (row wise) and elements of B from up to down (column wise)

```
pairs3 = pairs2.sortByKey().map(lambda x: [x[0][0][0], x[1]])
```

# Now that they are sorted we only need their provenance and their value (either A or B) thus we group by to be able to slice afterwards:

```
lists = pairs3.groupByKey()
```

# We need to go through another map procedure in order to be able to apply our slicing procedure, again pay attention to the range of the for loops which depend on the size of the matrices:

```
import numpy as np
def mapper2(x,y):
    for i in range(0,4):
        for j in range(0,4):
            C_A = np.array(x[length*i : length*(i+1)])
            C_B = np.array(y[width*j : width*(j+1)])
            C_temp = C_A * C_B
```

# Unfortunately we weren't able to go through this procedure, our idea was to apply the as a map and do the product of elements of A and B and list them with a corresponding key,value pair. Then we would have just applied a .reduce(lambda x: sum(x)) to get our values.

Benchmarking:

This algorithm proved to be much much faster as the shuffle and sort does not have to deal with as many values as previously, for the same matrix for which we reached our limit in the previous algorithm (two 400x400 matrices) this new algorithm need just below 3 minutes to compute all the elements of C. The computation time improvement is drastic, although this comes at a certain price. This algorithm isn't as parallelizable as the previous one as it must yield list_A and list_B before being able to go further. The trade-off locally here goes in favor or this second algorithm but this might not be the case for even bigger matrices on big hadoop clusters.

- <u>A (600 x 600) & B (600 x 600) :</u>
    - Local: ~10m
    - Hadoop: 3m33s
- <u>A (300 x 300) & B (300 x 300) :</u>
    - Local: ~2m10
    - Hadoop: 1m30s
- <u>A (150 x 150) & B (150 x 150) :</u>
    - Local: ~23s
    - Hadoop: 1m
- <u>A (75 x 75) & B (75 x 75) :</u>
    - Local: ~5s
    - Hadoop: 43s
- <u>A (35 x 35) & B (35 x 35) :</u>
    - Local: ~1s
    - Hadoop: 29s

Although as mentioned before this second algo outperforms our first one locally, the difference is not that noticeable once on hadoop, to be more precise the most drastic gain in efficiency through distribution of files/execution tasks is for the first algorithm, surely due to its high parallelism. What we can also notice is that hadoop becomes an essential tool for both algorithms as further we go in matrix dimensions. Operations like S&S can be very heavy for a single processor computer (local computation), this where very large matrices and operations on them should be considered with services like hadoop who are based on partitioning and parallelism.

C - Algorithm 3 (Block Matrix Multiplication Attempt)

Given our initial lack of total success with the two previous Algorithms (full implementation in Spark and Hadoop), we decided to explore a third route: Block Matrix Multiplication. The intuition behind it is to split the original matrices into blocks (we focussed on 2x2 for simplicity) and the final matrix C as well. The theory behind it is that we only need certain blocks of the original matrix for a given block of the final Matrix. By doing this split, in theory, we reduce the computational complexity because we reduce the amount of lines the system has to read through that are not relevant to the computation. However, in our implementation this does mean that the output of the Map phase is twice as long as in the algorithms above. We are essentially assuming that increasing the Transformation function will cost less (in computational complexity) than the gain obtained from reducing the Action function (please excuse the crossover with Spark terminology).

For example, assume all 3 matrices (A, B and C) are square matrices for simplicity of illustration. Let us split each matrix into 4 equal sized blocks (again for simplicity, they do not need to be equal - only number of rows of A must match with number of columns of B). Let us call the upper left corner block of C : "C00" (with the other blocks "C01" - upper right, "C10" - lower left and "C11" - lower right).

Our intuition was for the map phase to emit (for each value in the original matrices) : the destination block ("C00" for example), the provenance (original block eg: "A00"), the specific provenance within that block to be used for element-wise multiplication in the reducer (A,35,53 - corresponding to row 35 and column 52 of matrix A) and finally the Value of the cell (eg: 7). However; this exact same cell (A,35,52,7) would be used in the computation of C01 as well. Thus we had to print the exact same line with the key "C01" instead of "C00" thus doubling [doubling because of 2x2 block partition] the output of the mapper). Thus there are some concerns about scalability.

In order to compute C00, we need information from A00, A01, B00 and B10. Thus we only need half (we don't need blocks {A10, A11, B01, B11}) of the information from the original matrices. It is easy to observe that with more granular partitions (ie. more blocks), we require less and less information from the original matrices to compute the final matrix (we mean that we reduce the amount of redundant information). Thus we have decreased the amount of information needed in the reducer for each cell of the final matrix. On the other hand, our mapper so far has had to output the block A00 twice: once with a key allocating it to C00, and once more with a key allocating it to C01. It is thus easy to see that increasing the number of blocks increases the number of (key,value) pairs output by the mapper. This latter issue is a weakness of our algorithm which, in its trivial state, requires the repetition of the map output. One way to move forward would be to dynamically incorporate the number of partitions into a function in our Reducer that could then reuse cell blocks from the original matrices as many times as required without needing to be re-printed by the mapper. For example, instead of printing "C00" and "C01" as keys we would just print the origin block "A00" and the reducer would automatically know how many times it needed to access that unique line in order to compute C. This is an issue that could theoretically be fixed with two MapReduce jobs.

Please find the commented code for the mapper and the reducer at the following link :

wget https://www.dropbox.com/s/6muk786071b19xm/BlockMatrix_Mapper.py

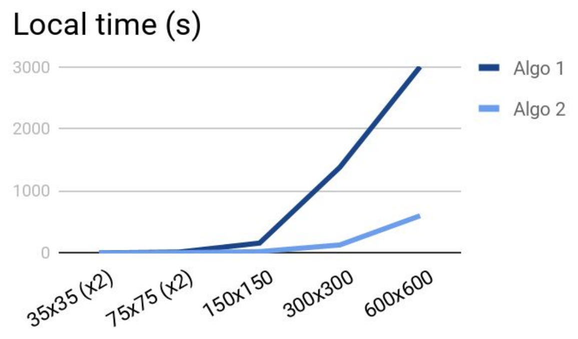
wget https://www.dropbox.com/s/tzp5quztyusmkxy/BlockMatrix_Reducer.py

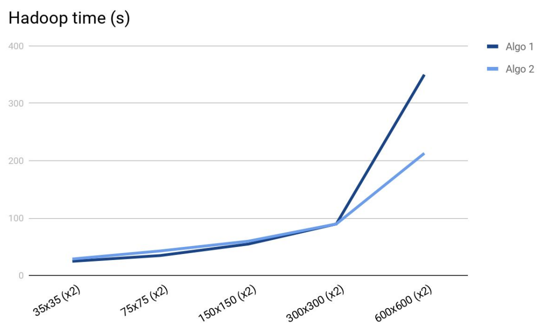
3 - Conclusion

To conclude, we would prefer our second algorithm if running on a local computer, given that the size of the file containing the info of the matrices fits in the disk memory. Although this gain in efficiency comes at the cost of a lack of parallelism, for extremely large matrices we would prefer the first algorithm if the use of hadoop streaming on large clusters would be necessary. The main difference between both algorithms would be that instead of replicating elements as in the first one we simply call them through indexing in the second algorithm.

Benchmark comparison:

(approximate - in seconds)	**600x600 (x2)**	**300x300 (x2)**	**150x150 (x2)**	**75x75 (x2)**	**35x35 (x2)**
Algo 1 local	3000.0	1380.0	160.0	16.0	2.0
Algo 2 local	600.0	130.0	23.0	5.0	1.0
Algo 1 hadoop	350.0	90.0	55.0	35.0	25.0
Algo 2 hadoop	213.0	90.0	60.0	43.0	29.0





We can see here more clearly how run time explodes with the replication algorithm due to the sort operation. We can also see that this drastic difference is not as big on Hadoop clusters, and we can expect that the curves change rank as we go into even bigger matrices.

For our exploration into block matrix multiplication, unfortunately we weren't able to get the Mapper and Reducer to both fully function simultaneously. The main issue we encountered was with iterations when creating the lists required for the dot product computation to obtain to the final matrix. In terms of pushing further in the exploration, one could explore the various impacts on computation time of the number and size of blocks we choose to partition the matrices into. It turns out that the number of rows per block has very little impact on algorithm efficiency but that efficiency has a negative correlation with the number of rows (cf presentation in appendix).

4 - Appendix

If any of the following links do not work please contact us in the briefest delays so that you can access our code freely.

Inputs:

https://www.dropbox.com/sh/s1ugz5731tsgwg9/AABq6FefTuDHTpyymmztmB_Ba

Algo 1:

https://www.dropbox.com/sh/pja02m0lu1g8fcw/AADqK1d9eDA7fupx8HWI4-SWa

Algo 2:

https://www.dropbox.com/sh/fjjrgop5vriqqmb/AABjocu8V7xfqP9XnT5QPN2ja

Block matrix multiplication algo:

https://www.dropbox.com/sh/1f7c5q6ug6p2use/AACaG0ln9DFXQv1UiOjfnhbja

Block matrix multiplication study:

http://dc-vis.irb.hr/repository/2014/Optimal%20Block%20Size%20for%20Matrix%20Multiplication%20Using%20Blocking.pdf