INF583 - Systems for Big Data (2021-2022)

Final Project

Big Data Processing

submitted by

Ali Haidar
Maya Awada

Institut Polytechnique de Paris

Contents

1 Introduction

Distributed data processing allows the distribution of application programs and data among interconnected sites. It diverges massive amount of data to several different nodes running in a cluster for processing in an aim to increase the speed, facilitate scalability and provide fault tolerance. This project demonstrates the efficiency of distributed data processing frameworks such as Apache Hadoop and Spark through the implementation of MapReduce jobs. On one hand, part A of the project focuses on the design and development of algorithms to perform operations on a list of integers. On the other hand, part B focuses on ranking Wikipedia Web pages with eigenvector centrality measure by implementing matrix multiplication in Apache Hadoop, Apache Spark and using only threads to then compare the different performances.

2 Part A - Operations on List of Integers

This part consists of performing a number of operations on a list of integers. These operations include finding the largest integer, the average integer, the set of distinct integers, and the number of distinct integers. We implemented these operations in each of Apache Spark, Apache Hadoop, and Spark Streaming and got the results displayed in table 1.

Algorithm	Output
Largest Integer	100
Average Integer	51.067
Set of Distinct Integers	1,2,..,99,100
Number of Distinct Integers	100

Table 1: Algorithms Results

2.1 Implementation in Apache Spark

2.1.1 Largest Integer

To find the largest integer, we implement one map reduce job. The map phase consists of creating a pair (1, number) for each number. The reduce phase consists of reducing each group of tuples (1,a) , (1,b) to (1, max(a,b)). We finally output the second element in the resulting pair, which corresponds to the largest integer in the data.

2.1.2 Average Integer

To find the average of all the integers, we also implement one map reduce job. The map phase consists of creating a pair (1, (number, 1)) for each number. The reduce phase consists of reducing each group of tuples (1, (a1,b1)) , (1, (a2,b2)) to (1, (a1+a2, b1+b2)). The value part of the resulting tuple corresponds to the pair (sum of all integers, total number of elements). We finally use the mapValues function, which only operates on the values, to divide the sum of the integers by the number of elements and therefore get the average.

2.1.3 Set of Distinct Integers

To display all the distinct integers, we also implement one map reduce job. The map phase consists of creating a pair (number, 0) for each number. The reduce phase consists of reducing each group of tuples (a,0) , (a,0) to (a,0). We finally loop over the resulting pairs, and print the keys which correspond to the distinct integers.

2.1.4 Number of Distinct Integers

To compute the number of distinct integers in the input, we implement two map reduce jobs. The first map phase consists of creating a pair (number, 0) for each number. The first reduce phase consists of reducing each group of tuples (a,0) , (a,0) to (a,0). Then, the second map phase consists of mapping each pair (number,0) to (1,1). Finally, the second reduce phase consists of reducing each group of tuples (1,a),(1,b) to (1,a+b). We finally output the second element in the resulting pair, which corresponds to the count of distinct integers.

2.2 Implementation in Apache Hadoop

2.2.1 Largest Integer

Similar to the implementation in Spark, the map phase consists of creating a pair (1, number) for each number. The reduce phase consists of aggregating all the values to then loop over them and modify the value of the 'largest' element that we defined by taking the maximum value of each pair of values at a time.

2.2.2 Average Integer

A different approach then that used in the implementation in Spark was used here. To find the average of all the integers on Hadoop, the map phase consists of creating a pair (1, number) for each number. The reduce phase consists of aggregating all the values to then loop over them and compute the sum by simply adding each value one after the other to the 'sum' element that we defined. A counter is also incremented by 1 at every iteration in the for loop to compute the total number of integers. We finally divide the sum obtained by the counter to get the average integer.

2.2.3 Set of Distinct Integers

To display all the distinct integers, the map phase consists of creating a pair (number, 1) for each number. The reduce phase consists of aggregating for each key all the values and keeping the value 1 as such: (number, 1). We are now successfully left with the distinct integers that appear in the original set of integers.

2.2.4 Number of Distinct Integers

To compute the number of distinct integers in the input, we first use the output of the previous operation to get the same set of integers, but with each integer appearing only once. Then, the map phase consists of mapping each pair (number,1), what we got as output from the reducer of exercise 3, to (1,1). The reduce phase

consists of aggregating all the values to then loop over them and compute the count of values by adding each value (ie 1) one after the other to the 'count' element that we defined. The resulting value corresponds to the count of distinct integers.

2.3 Implementation in Spark Streaming

2.3.1 Largest Integer

To find the largest integer, we first apply the map and reduce job separately on each stream or batch of data to get its largest integer value. We then use a long accumulator that goes over these values that we got from each RDD and returns the maximum integer out of them.

2.3.2 Average Integer

To find the average of all the integers, we also first apply the map and reduce job separately on each batch of data to get its sum and count of elements. We then use 2 separate long accumulators to compute the sum and count of all elements respectively by going over the values returned from each RDD. The average integer is calculated by dividing the sum of all elements by the count of all elements.

2.3.3 Number of Distinct Integers

The Flajolet–Martin algorithm, which is an algorithm for approximating the number of distinct elements in a stream in one pass, was used in this part. A stepwise pseudo code solution of this algorithm given by:

1. Initialize a bit-vector BITMAP to be of length L containing all 0s.
2. For each element x in M:
   - Calculate the index i = p(hash(x))
     where p(x) = min{k >= 0|$bit(y, k) \neq 0$}
   - Set BITMAP [i]= 1
3. Let R denote the smallest index i such that BITMAP[i]=0.
4. Estimate the number of distinct elements in M as $2^R/0.77351$

In our implementation of the algorithm on Spark Streaming, we defined the following hash function: hash(x) = (3 * x + 5) mod 128 and created a function that returns the least significant bit. Note that we chose to perform modulo of 128 because we have 100 distinct numbers so 7 bits will allow use to represent 2^7 =128 > 100 numbers. Then for each batch of data, we calculate the bitmap of 7 bits for the numbers that exist in the batch. Then, we create a bitmap accumulator in spark that will be updated after each batch. Finally, we approximate the number of distinct numbers in the whole file as $2^R/0.77351$ where R denotes the smallest index i such that bitmap[i] = 0.

3 Part B - Ranking Wikipedia Web Pages

3.1 Eigenvector Centrality Implementation

To get the most important page on Wikipedia, we implement the Eigenvector Centrality algorithm in each of Apache Hadoop, Apache Spark, and using only threads. The 3 different implementation successfully returned the same results.

3.1.1 Implementation in Apache Hadoop

In this part, we implement the matrix-vector multiplication using two map reduce jobs. Since the adjacency matrix A (nxn) is saved in the 'edgelist' text file, we also consider that the eigenvector r_t is saved in a text file.

The matrix-vector multiplication is given by: $Ar_t = r_{t+1}$
For a given index k, we have:

$$r_{t+1}[k] = \sum_{i=0}^{n} A[k][i] \cdot r_t[i]$$

We are therefore interested in grouping all elements having the same i.

1. First Map Phase: We create two mappers, one for the eigenvector and one for the adjacency matrix.
   - Mapping of the vector: For each index i, we emit a pair (i, ('E', '0', r[i])).
   - Mapping of the matrix: For each edge $< a, b >$, or in other words, for each A[a][b]=1, we emit a pair (b, ('M', a, '1')).
2. First Reduce Phase: We group the elements sharing the same key together. For each key, we have one value coming from the vector and a list of values coming from the matrix. Taking key = b as example and assuming node b has connections with nodes 0, 3 and n, we get:

   $(b, ((E, 0, r[b]), (M, 0, 1), (M, 3, 1), (M, n, 1)))$

   Then, for each tuple (M, i, 1) coming from the matrix, we emit a pair (i,1*r[j]) where 1 is the last value masked in the output value of the matrix mapper and r[j] is the last value masked in the output value of the vector mapper. Proceeding with the previous example, we get: $(0, 1 * r[b])$, $(3, 1 * r[b])$, $(n, 1 * r[b])$
3. Second Mapper Phase: We read the output of the last reducer and we emit (key,val).
4. Second Reducer Phase: For each key, we sum the values and we emit (i, sum(values)).

We finally normalize the resulting eigenvector values through an additional map-reduce job.

3.1.2 Implementation in Apache Spark

In this part, we consider that we can pass an array to the mapper of Spark. We represent the vector r_t as a array in Java.

- Map phase: We create a mapper that reads the edgelist file and for each existing edge between two nodes i and j, it will emit a pair $(i, r[j])$.
- Reduce Phase: For each key i which represents a row we sum all the values that share the same row number.

At the end, we can normalize the vector r_{t+1} using sample array traversal or using spark by creating an RDD structure.

Algorithm 1 Matrix Vector Multiplication on MapReduce

```
function Map(i,j)
Emit(i,r_t[j])
endfunction
functionReduce(key, values)
ret ← 0
for val ∈ values do
ret ← ret + val
end for
Emit(key,ret)
end function
```

3.1.3 Implementation using Threads

To multiply a matrix with a vector, we can do the computation separately for each line. Therefore, it easy to calculate this using threads by dividing the sparse matrix into several files and share a common vector result between all threads.

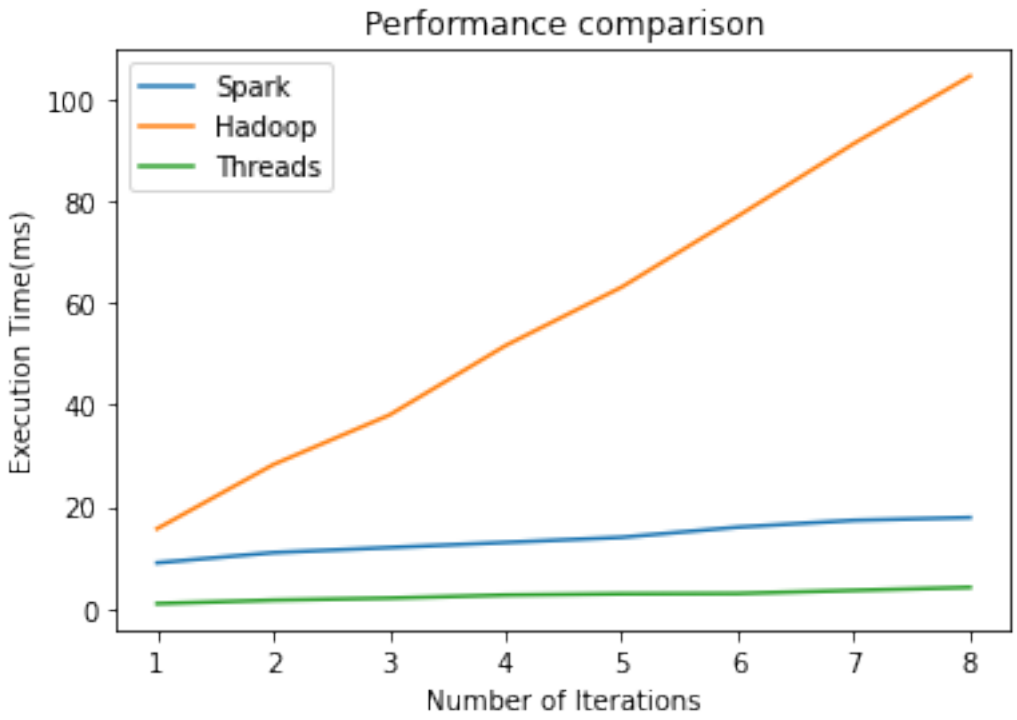
In our implementation, we divide the matrix into 4 different parts and then calculate the elements of the resulting shared vector by doing the calculation over these 4 threads at the same time.

3.1.4 Performance Comparison

To compare the performance of the 3 implementations, we measured and compare the execution times of each implementation in milliseconds for a different number of iterations.

Iterations	1	2	3	4	5	6	7	8
Spark	9	11	12	13	14	16	17.338	17.888
Hadoop	15.687	28.272	38.001	51.629	63.108	77.032	91.295	104.539
Threads	0.985	1.673	2.069	2.673	2.917	2.977	3.59	4.175

Table 2: Execution Times



We observe that the implementation using only threads is the fastest followed by the implementation in Spark and finally the implementation in Hadoop which is relatively slow. These results were expected as threads are only performing the computations over the iterations without any map-reduce job. Moreover, given that Hadoop is a distributed file system, the time needed to read large amount of data from a file - as compared to reading from RAM - is high leading to decreased performance. Note that these results are based on the performance of our machines. We expect different results if we have a larger-scale system.

3.1.5 Most Important Wikipedia Page

We ran each of the 3 previous implementations using different number of iterations. The implementations in Apache Hadoop, Apache Spark, and using only threads returned the same results, displayed in table 2, for each given number of iterations. Overall, the most important Wikipedia page appeared to be 'Category:Rivers in Romania'.

Number of Iterations	Page Number	Page Title
1	20409	Category:Rivers in Romania
2	26200	Day
3	20409	Category:Rivers in Romania
4	18990	Category:Main Belt asteroids
5	20409	Category:Rivers in Romania
6	20409	Category:Rivers in Romania

Table 3: Most Important Wikipedia Page Results

3.2 Matrix Multiplication Implementation

3.2.1 Implementation using 1 MapReduce Step

In this part, we implement matrix multiplication using only one map reduce job. We assume we have two matrices A and B.

- Map phase:
  - Mapping of Matrix A: (key, value)=((i, k), (A, j, Aij)) for all k
  - Mapping of Matrix B: (key, value)=((i, k), (B, j, Bjk)) for all i
- Reduce Phase: For each key, we generate 2 sorted lists: Alist and Blist. We then compute the summation (Aij * Bjk)) for each key and output ((i, k), sum).

3.2.2 Implementation using 2 MapReduce Steps

The Matrix Multiplication implementation using 2 map-reduce jobs follows the same procedures used in the eigenvector centrality computation. Refer to part 3.1.1.

3.3 Computational Cost

- Using threads, the computation cost will be O($m*k+n$) such that m represents the number of edges, k represents the number of iterations and n represents the number of nodes.
- Using two MapReduce, the computation cost will be :
  - First Mapper cost = O($m + n$).
  - First Reducer cost = cost to regroup the elements with the same key together (it can be O($log(n)$) if we are using B-tree or it can be O($log(\alpha)$) if we are using hash map.) + o(m).
  - Second Mapper cost = O(m).
  - Second Reducer cost = cost to regroup the elements with the same key together + O(m).

  Therefore, the overall cost of using one map reduce is O(m)+ cost to regroup the elements with the same key together.
- Using one MapReduce, the computation cost will be :
  - Mapper cost = O($n^2 + m$).
  - Reducer cost = O(n^2) + cost to regroup the elements with the same key together.

  Therefore, the overall cost of using one map reduce is O(n^2) + cost to regroup the elements with the same key together.

So, all the experimental tests approve these results and show that the performance of TwoMapReduce algorithm is better than OneMapReduce algorithm.

4 Conclusion

This course project provided great insight on the appropriate systems and tools used in Big Data applications. Distributed data processing using Apache spark and Apache Hadoop and parallel programming using threads allow the execution of code more efficiently, which can save the organization time and money.