Final Project INF 583

Adam Chader, Jana Ismail

March 2022

Abstract

This report details our work on the project by explaining the algorithms that we developed, covering the problems and thoughts that we had and showing some relevant results of our implementation

A Lists Of Integers

A.1 Spark

We decided to do the first exercise using Spark, as we found it more intuitive to use. The pipelining aspect makes it easy to compose jobs of several maps and reduces, with less constraints than with Hadoop.

A.1.1 The largest integer.

To find the largest integer in the sequence, we simply mapped every integer as a value with the same key, 1. thus the reduce only does one operation, with the key 1, and the value a list of all integers. We then `reduceByKey` by calculating the maximum value.

A.1.2 The average of all integers.

Similarly than for finding the largest integer, to find the average, we mapped all integers as values with the same key of 1. The only subtlety is that we did not put only the integers as values, rather, the values were pairs of the integers with their coefficient for the average computation (meaning 1 at this stage). Then, for the reduce, we could just sum the integers together, and the coefficient together. We only had left to divide the sum with the coefficient to get the average, which we did with a final map.

A.1.3 The same set of integers, but with each integer appearing only once.

To do this, we used a key property of the map, which is the key data structure behind mapReduce, which is that keys are unique in the structure. Then what we did is we mapped all integers as keys with the same value (of 1 for example) and we reduced by just assigning a value of 1. Then we collect only the keys. This ensures that all keys are unique, and thus that each integer appears only once.

A.1.4 The count of the number of distinct integers in the input.

We reused the result of the last question here. We just have to use the last RDD containing only distinct integers, and get the sum by mapping them to values with the same key and reducing.

A.2 Spark Streaming

The logic behind our answers to the questions using streaming is pretty similar to those just using Spark. We used the `javaStreamingSparkContext` to build streams out of the file containing the integers, so that we could apply stream logic to our computations.

We choose to load 10 lines each one minute with a window size of three minutes. Each minute the largest integer in the window is displayed. Since we're working on a file with a fixed size, we can adjust the window duration and the number of lines that we load during a batch to have the largest integer in the whole file, however we decided to to keep the initial parameters to see how actually spark streaming works in real life were we do not necessarily take the whole history of the data into consideration. The same reasoning goes to exercise 2 as the average of the numbers loaded in each window is displayed.

We did not have the time to come back to exercise 4 as we faced some difficulties in section B which took more time than we expected. The Flajolet–Martin algorithm estimates the number of distinct integers in the input for each window but the implementation seems more complicated that we expect.

B Ranking Wikipedia Web pages with a centrality measure

B.1 EigenVector Centrality

B.1.1 Using threads

In order to implement the eigenvector centrality using five threads we divided the edgelist file into 5 equal chunks (edgelist0 to edgelist4) and we implemented two classes. The EigenVectorCentrality class creates the initial vector r0 of type vector that is thread-safe, so many threads can access the vector with no conflicts. At each iteration (500 in our case) we create five threads and start them. Each thread is given an id and the shared vector. Once the threads are joined, we normalize the vector before going on to the next iteration. The EigenVectorCentralityThread class opens the file corresponding to its id. If the thread has the id 0 the line number is set to 0. The other threads set their line number according to the previous thread id. The lines of a chunk will be read in a loop by each thread in order to compute the result of a row of the vector (`vector.set(line_nb, sum);`). The line number in incremented at the end of every iteration until reaching the end of the file.

B.1.2 Using Spark

In order to implement the eigenvector centrality with Spark we started by mapping each node to its neighbors so we can have an RDD of tuples containing the edges of the graph. We also created the RDD of tuples that represents the vector r0 where a tuple is composed of the node id and its value in the vector. The idea is to join the two RDDs on the destination nodes of the RDD that contains the edges and the nodes id of the RDD representing the vector, that is why we needed to reverse the edges in the first RDD. After the join we will have an RDD of tuples containing the source node id and the value of each of the corresponding destination nodes id. We finally sum the values of each source node with a reduce operation before performing a last map operation to normalize the vector. We repeat this process many times (30 in our case) to calculate the eigenvector centrality. We added some operations at the end to retrieve the name of the page with the highest value in the vector.

B.1.3 Using Hadoop

In order to implement the eigenvector centrality with Hadoop we used two map reduce phase with the second one executed many times. The first map-reduce phase is composed of two maps. The EdgeMapper maps each line of the edgelist file into tuples with the destination node being the key and the source node being the value. The VectMapper maps the lines of the vector file into tuples with the node id being the key and the value of the vector being the value. An "_" is added before the vector value to distinguish it from the nodes ids during the second map-reduce phase. The first reducer writes each node with its sources and its value in the vector in a first output file. The second map-reduce phase is executed many times to compute the eigenvector centrality.

This second phase starts with ResultMapper, which gathers the source nodes for each destination node. It writes, for each source node the value of the vector at the index of the destination node, such that at the end of the map phase, we have two types of tuples : all destination nodes with their source node, and all nodes with the values of the vector at the index of their destination nodes. Then, we go through the ResultReducer, which reduces all these informations by building the exact same layout as the one outputed by the MultReducer, so that we can use the ResultMapper once again. For each index, we sum all values that are vectors values (identified by a "_"). We then concatenate all source nodes in the same fashion as with the MultReducer. This ensures that we can easily loop with the ResultMapper and ResultReducer, while incrementing the value of the vector.

We realized at the end that we forgot to normalize the vector at the end of each iteration, this might require a lot of changes in our algorithm, we need to add a map-reduce phase in order to do this properly. We decided to leave it as it as as the final result was not affected.

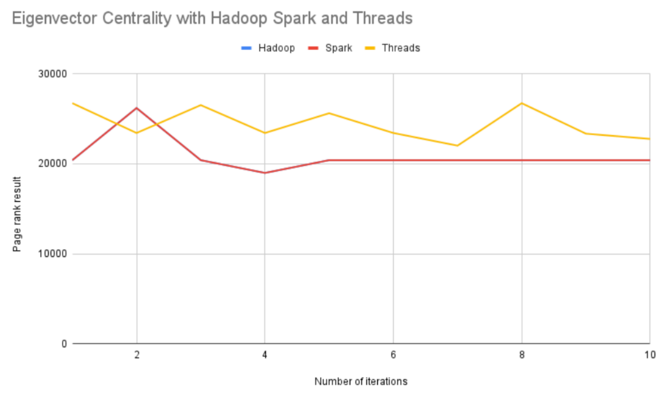

Figure 1: Convergence of the *Eigenvector Centrality* algorithm as a function of the number of iterations for Hadoop Spark, and using Threads.

B.2 Most important page in Wikipedia

According to the Spark and Hadoop implementation, the most important page in Wikipedia is "Category:Rivers in Romania" with the id of 20409.

B.3 Matrix Multiplication

For the matrix multiplication with two map-reduce phases, we used the algorithm of the eigenvector centrality with one iteration.

We could not figure out a way to implement the matrix multiplication with one map-reduce phase with Hadoop as it is impossible to use the two input files in the same operation, we always have to have a first map-reduce to combine them. We decided to implement this with Spark as the operations are easier to manipulate and we can map many inputs in one "mapper". In the map phase, we flatten the input file to have the tuples of edges and we map each destination edges to its value in the vector so the final result would be an RDD if tuples containing each source node with the value of each of its destination nodes in the vector. For the reduce phase we perform a simple sum of the values for each node.

B.4 Algorithm cost

For the eigenvector centrality computation with Spark the cost of the algorithm is calculated as follows: The load operation and the first mapping to pairs of source-destinations have input size of n each.The input size of the flatten operation that returns the RDD of the edges is of m. We also have an input size of m to reverse the edges. The operation that creates the vector has an input size of n. The join operation has an input size of n+m as it takes performs the join on the RDDs containing the vector and the reversed edges. We have an additional mapToPair operation to recompute the vector with an input size of m as we map each node to the value of each of its destination nodes. The reduce operation takes as input the RDD of size m that contains each source node with the value of each of its destination nodes. We add two last operations with input size of n for each to compute the norm and normalize the vector. Every thing from the join is repeated k times. To total cost would be: n + n + m + m + n + k(n + m + m + m + n + n) = 3n(k + 1) + m(3k + 2).

For the eigenvector centrality computation with Hadoop the cost of the algorithm is calculated as follows: In the first map-reduce phase, the two maps have each an input size of n (one input for every node of the graph, we do not take into consideration the nodes that have no outgoing edges to simplify the calculation and to keep it general). The first reducer receives the tuples of the reversed edges from the first mapper and the vector from the second mapper so it has an input size of n+m. In the second map-reduce phase, the input of the mapper has a size of n as the first reduce writes a line for each node in the graph. The input size of the last reducer is m+m as it receives a file from the second mapper containing the tuples of the reversed edges and the tuples of the value in the vector for each destination node. If we repeat the second map-reduce phase k times, the total cost of the algorithm would be 2n + (n + m) + k(n + m + m) = n(k + 3) + m(2k + 1). Of course the cost is even higher as the normalization would add another map-reduce phase to the algorithm.

The eigenvector centrality computation with Hadoop costs more in terms of computation and time.

We can find the cost of the matrix multiplication with two map-reduce phase by setting k to 1 in the previous result, so the cost would be 4n + 3m.

For the matrix multiplication with one map-reduce, the cost is calculated as follows: The operation to create the vector has an input size of n. In the map phase, the first mapToPair has also an input size of n as it maps each line to a tuple of node id and a list of its destination nodes. The flatten operation receives n these tuples and outputs the tuples of m edges, so the last two mapValues operations have each an input size of m. The reduce phase also has an input size of m The total cost would be n + n + n + m + m + m = 3n + 3m, which is less costly than a two map-reduce with Hadoop.

Conclusion

In conclusion, this project helped us to understand the point of MapReduce much better. It requires a different way of thinking than for traditional parallel algorithm. When wanting to solve complex problem, it can quickly become overwhelming to use the primitives of MapReduce, but the first exercise and the previous labs allowed us to familiarize ourselves with it such that it could become second nature for us to think in the MapReduce way.

The first exercise was really interesting, and many of the ideas we used for it we reused in the second, harder exercise. Unfortunately, we weren't able to finish the SparKStreaming portion, especially the last question, as we went on to work on the second exercise and didn't have time to go back.

The second exercise around *eigenvector centrality*, was much more complicated for us, and forced us to think in creative and sometimes convoluted ways in order to solve it. We especially had a lot of trouble using Hadoop, because to perform the necessaty computations, we needed to be able to map to separate files, in such a way that the data could be used in the reduce. We managed to find a solution, but we are not really satisfied as we came up with a really complicated algorithm, and we are convinced that there exists a simpler one.

We also did not manage to find a solution using only one mapreduce iteration for matrix multiplication using hadoop, so we did it with spark, and thus we were not able to propose a performance comparision between one and two mapreduces iterations as it was asked.