Distributed implementation of the Jacobi's algorithm in MapReduce

Marco Barbone
Università di Pisa
Pisa, Italia
m.barbone@outlook.com

1 INTRODUCTION

The Jacobi's algorithm is an iterative method based on the fixed point. This means that given an iteration matrix M (with ρ(M)<1) and a vector $\boldsymbol{x}_0$ it will generate a succession of $x^{(k)}$ until x^k converges to the solution of the linear system.

Algorithm 1 Jacobi algorithm

Input:
A linear system in matrix form Ax=b of size MxM
$\boldsymbol{\epsilon}$ the maximum accepted error
K the maximum number of iterations
Output:
$\boldsymbol{x^{(k)}}$ the solution vector

1: **function** JACOBI(A, b, ϵ, K)
2: $x^{(0)} \leftarrow$ initial guess
3: $k \leftarrow 0$
4: **repeat**
5: $error \leftarrow 0$
6: **for** i=0 to M **do**
7: $\beta \leftarrow 0$
8: **for** j=0 to M **do**
9: **if** $i \neq j$ **then**
10: $\beta \leftarrow \beta + a_{ij}x_j^{(k)}$
11: $x_i^{(k+1)} \leftarrow \frac{b_i - \beta}{a_{ii}}$
12: $error \leftarrow \frac{\left\|x^{(k+1)} - x^{(k)}\right\|_1}{N}$
13: $k \leftarrow k + 1$
14: **until** $error > \epsilon$ and k < K
15: **return** $x^{(k)}$

This algorithm is embarrassing parallel and there are a lot of frameworks implementing parallel versions of it. From the code in the algorithm (1) can be noted that the computation of a component of the new solution needs the solution of the previous iteration, and the corresponding row of the iteration matrix. This means that different components can be calculated in parallel.
But, what if the matrix does not fit in memory?
It is possible to implement it in a streaming way: loading some rows at the time and calculating the corresponding components of the solution vector. While this algorithm can still be parallel, it is very slow.
The bottleneck is the disk because at each iteration the matrix instead of sitting in memory must be loaded from it.
The situation gets even worse if one single vector does not fit in memory. In this case, the vector can be divided in chunks that fit in memory, each chunk can be loaded and processed, end then the partial results can be aggregated.
In this situations is useful to implement this algorithm in a distributed cluster.
Dividing the data between several computers allows loading different partition of the matrix from several disks in parallel, reducing or completely removing the bottleneck given by the bandwidth of a single disk.
But there is no free launch and the price paid here is the communication overhead. In particular, the partial results need to be collected in a single place in order to perform the aggregation and the solution must be broadcasted to every node in the cluster at the beginning of each iteration.
With respect to the original algorithm presented in 1, I have performed a minor optimization. The original algorithm performs a global error checking and every time computes each component of the solution. My version instead computes only the components that are not converged while freezing the others.
There is also the support for sparse and dense matrices. The only difference between them is that the sparse version compress the data using the RLE0 compression.

2 EXISTING SOLUTIONS

There are a lot of parallel implementations of the Jacobi algorithm. I personally implemented two versions of it. They can be found on Github:

- **parallelIterativeMethods** (w.i.p.) is a library that contains different parallel iterative methods for sparse and dense matrices. Including some versions of Jacobi's method.
- **Jacobi** is a program used to confront the performance of different parallel programming environments using the Jacobi's algorithm.

Both of them assume that the matrix fits in memory.
Things change in the distributed domain. There are not a lot of distributed implementations of this algorithm.
These are the two that I found:

- **Elemental** is a linear algebra distributed framework written in c++. This does not contain directly the Jacobi method but provides the tool for implementing it in few lines of code.
- **BLASpark** is a linear algebra framework built on top of Apache Spark which is built on top of Hadoop.

3 IMPLEMENTATION DETAILS

When I started implementing this algorithm my idea was to produce a generic library that could work with any instance of

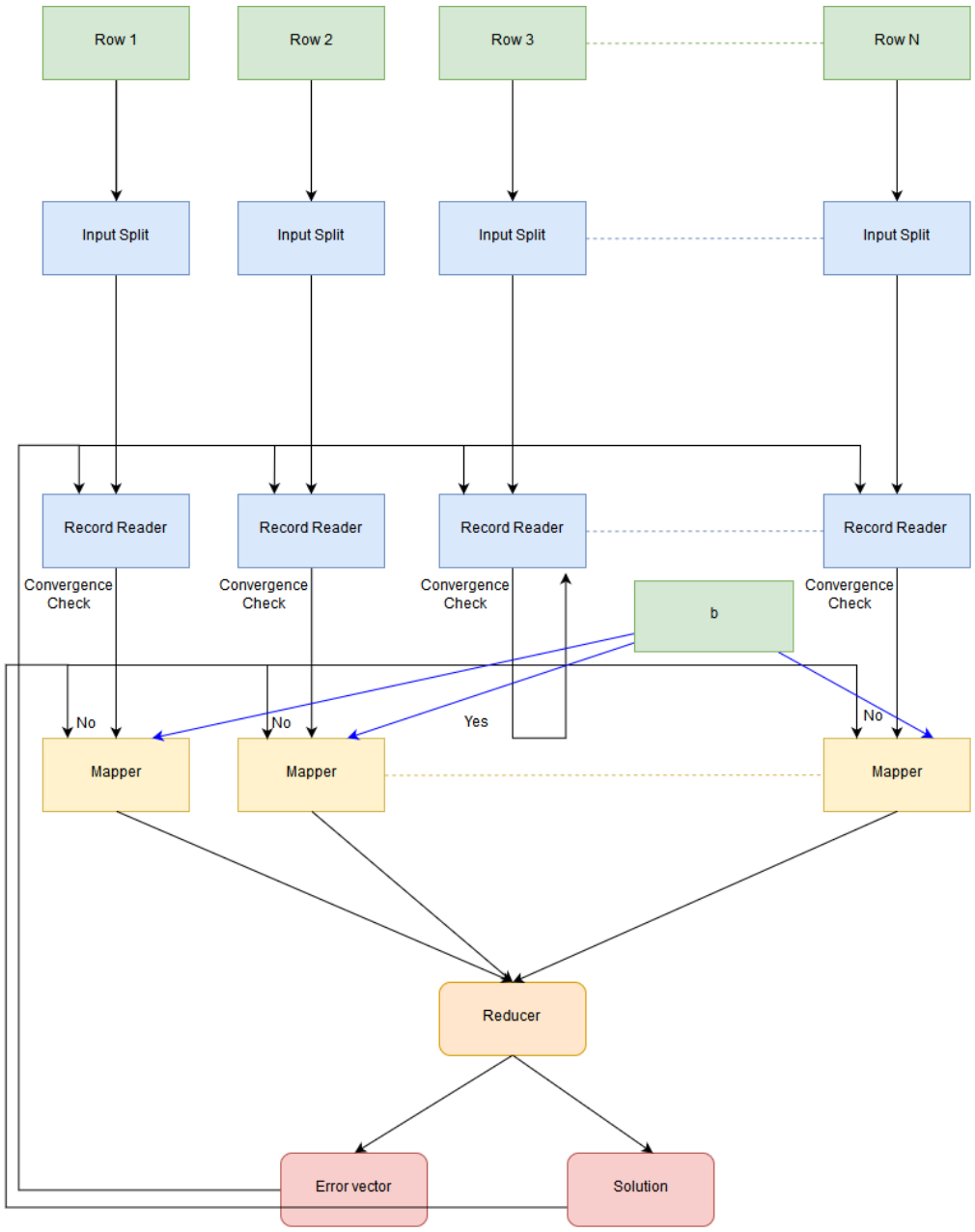

Figure 1: Jacobi Algorithm data-flow

the class Number. This would allow users to instantiate it with the desiderata data type (e.g single or double precision floating points or complex number if needed) but this approach has a major drawback. The use of generics implies a lot of class loading and object creation and destruction. Instantiating a double[] is very different than a Double[] in terms of performance and memory requirements. The use of the generics is not advisable in tight memory situations like this.

The project structure is over-designed. I could fit everything in one or two packages but in order to preserve future expandability I decided to put everything in small packages. Notably:

(1) **dataStructures** contains the supporting data structures of the algorithms (Vectors and SparseVectors).
(2) **generics** contains abtract classes that wraps the common logic of all RecordReader and SparseVectors needed it the project.
(3) **utils** contains utility functions. In this case the only one needed is a matrix generator in order to generate test data.

(4) **dense** contains the tools needed to read dense matrices and perform some basic operations such as matrix-vector multiplications and "dense" jacobi algorithm.
(5) **sparse** similar to dense contains the tools to read "sparse" matrices, stored using the RLE0 compression, and perform matrix-vector multiplications and "sparse" jacobi algorithm.

Two main classes are provided: Jacobi and SparseJacobi which respectively test the 'dense' and the 'sparse' version of the Jacobi's algorithm.
I implemented my own data structures instead of using the hadoop ArrayWritable because I did not want to be forced using generics (for the memory reasons stated above) and I wanted to wrap marshaling and un-marshaling logic inside them.
I provided the Matrix-vector multiplication in order to test the Jacobi algorithm. Performing $b - Ax = 0$ it is possible to check the correctness of the algorithms. Also part of it can be reused in other algorithms.
The first decision that I took in order to implement the project was the input format. The format that I chose is: one file contains only one row of the matrix.
As shown in the figure (2) the first line of the file contains 'row x' where x is the number of the row and the successive lines contain the list of all the values separated by one or more spaces. This is NOT the best way to structure the input because as stated in [1] hadoop encourages to use large files. Because mappers in a MapReduce job use individual files as a basic unit for splitting input data. At present, there is no default mechanism in Hadoop that allows a mapper to process multiple files.

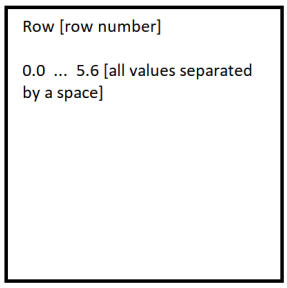

Figure 2: Input file example

Having only one row in a file implies the creation of a mapper for each row of the matrix and it is not efficient if the number of computers in the cluster is minor respect to the size of the matrix.
After deciding the input format I had to implement the RecordReader in order to parse the input and generate <key,value> pairs to be processed by the mappers.
The RecordReader logic is shown in pseudocode 2.
From the pseudocode 2 can be noted that the file is read and parsed only if needed.
The mapper is very simple and implements the code from line 8 to 11 of the pseudocode 1. In order to do so, it loads the known terms vector and the old solution from HDFS and performs the computation. In the end, it emits only a key-value pair that consist of <coordinate index, value>.
All these values then are passed to one single reducer. This reducer receives all the values, reconstructs the vector, updates the error vector, and emits the new solution.
The output produced is a file containing all the components of the solution vector separated by a space.

Algorithm 2 RecordReader logic

Input:
split portion of the input space to process
ϵ the maximum accepted error
errorVector vector containing the error computed at the previous iteration
Output:
True if a key value pair is found false otherwise

```
1: function NEXTKEYVALUE(split, b, ϵ, errorVector)
2:     k ← getKeyFromSplit(split)
3:     error ← errorVector[k]
4:     if error ≤ ϵ then
5:         return False
6:     row ← getRowFromSplit(split)
7:     return True
```

4 CONCLUSIONS

The algorithm that I implemented works but it does not perform very well. The input data that I chose is not optimal because it has to be parsed at the beginning of each iteration. It is better running a MapReduce job that parses, validate, and normalize the input data only one time. In order to avoid wasting time doing these operations over and over again.
Another problem is the output. Right now the output is in plain text and this is not the best way to emit the output. Converting the binary representation into a textual one is inefficient both in reading and writing. Since the solution vector must be broadcasted to every mapper it is better to keep it as smaller as possible.
The precision of the algorithm is very low. It produces a solution that is close to the correct one but with a not negligible error. Probably the problem is in the input/output format. Converting the binary to plain text and vice-versa produces some rounding errors. Numbers are rounded in order to keep the plain text representation short and this causes precision issues. In particular, this is a problem with denormalized numbers which, usually, they lose a lot of precision in rounding.
In conclusion, I think that implementing the Jacobi's algorithm in MapReduce is a viable option. The communication overhead introduced is not so high. The intermediate and the final key-value space is not so big, actually, is a fraction of the initial space and the broadcast of the solution can be done in parallel while reading the row of the matrix.

REFERENCES

[1] Jimmy Lin and Chris Dyer. 2010. Data-Intensive Text Processing with MapReduce. *Synthesis Lectures on Human Language Technologies* 3, 1

(2010), 1–177. https://doi.org/10.2200/S00274ED1V01Y201006HLT007
arXiv:https://doi.org/10.2200/S00274ED1V01Y201006HLT007