Jacobi Method Implementation In Spark

Azam Asl
New York University
aa2821@nyu.edu

Abstract

In this project we implemented three different versions of Jacobi method to solve a Laplace equation in Spark/Scala and observed that the second implementation scaled at least 10 times the number of discretizations in compare to our first implementation, running on a 44-node YARN cluster. Our third implementation by taking advantage of caching intermediate variables as well as broadcast join runs faster than the second version, when testing on the same problem size and the same Spark+YARN configurations.

1 Introduction

Implementing a fast and scalable optimization function in Spark requires a deep understating of the essential RDD transformations, Spark actions as well as optimal way of partitioning RDDs among the nodes. Seeking minimum shuffling and maximum locality is the key to success. In this project we explored the trade-off of developing optimization functions in Spark and compared a naive implementation of Jacobi method to solve a Laplace equation versus improved implementations. We also implemented the same algorithm in C+OMP as well as C+MPI.

We tested our OMP implementation on our YARN cluster and observed that for our example which is considered an 'embarrassingly parallel' algorithm, OMP can solve the problem up to the number of discretizations $N \sim 10^{12}$. However beyond that we ran into memory issues and could not test further. OMP is a memory sharing framework and therefore comparing Spark agains OMP would not be fare. Due to system administration limitations we were not able to test our MPI code on the cluster. Therefore due to the lake of fare comparison environment we are not providing any further result on the performance of these HPC frameworks versus Spark.

2 Background and Related Work

Optimization function are iterative methods and usually involve costly computation in every iteration. Traditionally, people have used C+MPI/OMP to implement optimization libraries and run them on HPC platforms. Programming in MPI/OMP involves thread level programming and many hours of debugging and testing.

The emergence of the cloud computing in recent years, which provides a homogeneous operating environment and usually available for free or in low cost, has motivated the development of new libraries which run on large cluster of commodity machines. This has lead to the creation of new open-source cloud computing frameworks such as Apache Hadoop (MapReduce) and Apache Spark. These frameworks are designed with idea of Big Data and fault tolerance in mind.

Nonetheless, cloud computing frameworks are still considered to be very limited and in general slow for linear algebra and optimization libraries. [2] and [3] recently compared Spark and MPI/OMP for Machine Learning algorithms and they both reported MPI/OMP outperforms Spark more than one order of magnitude in terms of speed.

2.1 Brief introduction into Spark.

Apache Spark is mainly written in Scala and has APIs in other programming languages. Spark, similar to MapRedus, is a cluster-computing framework but unlike MapRedus, Spark uses in-memory storage. Every Spark application consists of a driver program that manages the execution of the application on a cluster. The worker nodes on a Spark enabled cluster are referred to as executors. The basic unit of work is called a task.

Resilient Distributed Dataset (aka RDD) is the primary data abstraction in Spark and the core of Spark. RDD is immutable and lazy evaluated. RDD operations are compiled into a Direct Acyclic Graph of RDD objects, where each RDD points to the parent it depends on. Shuffle operation (such as sortByKey, groupByKey, reduceByKey, join) means to re-distributing data which may involves new co-grouping across partitions on different executors and machines. At shuffle boundaries, the DAG is partitioned into stages that are going to be executed in order. Operations within a stage are pipelined into tasks that can run in parallel [6].

2.2 Running Spark on YARN

One can run a Spark application as a single local process, Spark standalone cluster manager or an external cluster manager such as YARN. There are two deploy modes that can be used to launch Spark applications on YARN. In the client mode, YARN client program starts the default Application Master and then Spark application will be run as a child thread of Application Master. The client periodically polls the Application Master for status updates and exits once

the Spark application has finished running. In cluster mode, the Spark driver runs inside an Application Master process which is managed by YARN on the cluster, and the client can go away after initiating the application. We tested both modes and did not observe any difference in terms of speed.

3 Problem Statement

3.1 Jacobi in one dimension

Jacobi is an iterative method to solved a diagonally dominant system of linear equations. Assuming we would like to find u such that

$$Au = f \tag{1}$$

where $A \in \mathbb{R}^{N\times N}$. We start form an initial vector $u^0 \in \mathbb{R}^N$ and compute the approximate solutions u^k iteratively until we converge. A is decomposed into a diagonal component D, and the remainder R and u^{k+1} is computed per below:

$$u^{k+1} = D^{-1}(f - Ru^k) \tag{2}$$

3.2 Solving Laplace equation with Jacobi in Spark

Laplace equation in one space dimension, which is also called the boundary value problem is as follows: given a function $f : [0, 1] \to \mathbb{R}$ we would like to find function u such that its second derivative within the interval $(0, 1)$ is equal to $-f$ and its boundary values are 0. More precisely:

$$u'' = -f \text{ in } (0, 1) \text{ and } u(0) = 0,\ u(1) = 0. \tag{3}$$

This problem can be solved analytically by integrating f twice. In higher dimensions, though, the analogous problem often cannot be solved analytically and one must rely on numerical approximations for u, using a finite number of grid points in $[0, 1]$ and finite-difference approximations for the second derivative to approximate the solution to (3). We choose the uniformly spaced points$\{x_i = ih : i = 0, 1, \ldots, N, N + 1\} \subset [0, 1]$, with $h = \frac{1}{N+1}$, and approximate $u(x_i) \approx u_i$ and $f(x_i) \approx f_i$ for $i = 0, 1, \ldots, N, N + 1$. Using Taylor expansions of $u(x_i - h)$ and $u(x_i + h)$ about $u(x_i)$ results in

$$-u''(x_i) = \frac{-u(x_i - h) + 2u(x_i) - u(x_i + h)}{h^2} + \text{h.o.t}$$

where h.o.t. stands for a remainder term that is of higher order in h, i.e., becomes small as h becomes small. Then we can approximate the second derivative at the point x_i as follows:

$$-u''(x_i) \approx \frac{-u_{i-1} + 2u_i - u_{i+1}}{h^2}.$$

This results in the following finite-dimensional approximation of (3):

ALGORITHM 1: Jacobi algorithm

Data: compute $hsq \leftarrow h$ square , set max_iter, set tol
Result: updated u and residual

1 Initialization square matrix A, column vector u, set maximum number of iterations;
2 **while** *cur_iter* $\leq$ *max_iter and residual* $\geq$ *tol* **do**
3 Compute vector $v_1 = Au$;
4 Update each element of $u = u + hsq * 0.5 * \mathbf{1} - 0.5v_1$;
5 Compute $v_2 = Au$;
6 Set residual to L2 norm of $(\frac{1}{hsq}v_2 - f)$;
7 **end**

$$\frac{1}{h^2}\begin{bmatrix} 2 & -1 & 0 & \ldots & 0 \\ -1 & 2 & -1 & \ldots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & & -1 & 2 & -1 \\ 0 & \ldots & 0 & -1 & 2 \end{bmatrix}\begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_{N-1} \\ u_N \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_{N-1} \\ f_N \end{bmatrix} \tag{4}$$

which is now in the form of (1). Using (2) we drive the following update in each iteration:

$$u_i^{k+1} = \frac{1}{a_{ii}}(f_i - \sum_{j\neq i} a_{ij}u_j^k). \tag{5}$$

Algorithm 1 applies (element-wise) equation (5) to current u^k to estimate u^{k+1} in every iteration. After it updates u in line 4 it checks the residual $||Au^k - f||_2$ to check wether it has converged or not. The coefficient 0.5 in line 4 and the vector $\mathbf{1}$ in line 6 come from the fact that we initialize f to the vector of ones. Further, we initialize u to the vector of zeros.

4 Implementation

4.1 Implementation with Distributed Matrices

MLlib is Spark's Machine Learning library and comes with two general type of Matrices, local matrix and distributed matrix. Local matrices are stored on a single machine and distributed matrices are stored distributively in one or more RDDs. Since we aim for scalable matrix/vector operation in Jacobi method we chose to implement our vectors and matrices as distributed matrix. Among the available distributed matrix classes BlockMatrix is the only one which supports matrix multiplication where both left and right matrix are distributed matrix [1].

Every iteration in Algorithm 1 involves a matrix-vector multiplication (line 3), a vector-vector addition (line 4), a matrix-vector multiplication (line 5), vector norm computation(line 6). Among these operations, matrix-vector multiplication is the most expensive operation. We define each row of the matrix A as a single block. These led us to define each single element of the vector u as a block, otherwise we cannot perform Au operation.

```
1       while (k < max_iters && (res > tol)){
2 //          Computing A*u_k
3           var Auk = A.multiply(u)
4 //          Computing 0.5*hsq-0.5A_i*u_k
5           val Aukcoor = new CoordinateMatrix(
6           Auk.toCoordinateMatrix().entries
7           .map (entry => MatrixEntry(entry.i,
    entry.j, hsqHalf - 0.5 * entry.value)))
8 //          Computing u_{k+1}
9           u = Aukcoor.toBlockMatrix(u_rowsPerBlock
    , u_colsPerBlock).add(u)
10 //         Computing A*u_{k+1}
11          val Aukplus1 = A.multiply(u)
12 //         Computing the residual
13          res= math.sqrt( Aukplus1.
    toCoordinateMatrix().entries
14                  .map(entry => math.pow(invhsq*
    entry.value - 1, 2))
15                  .reduce(_ + _))
16           ress(k) = res
17           k+=1
18      }
```

Code 1. Part of code for implementation with MLlib Distributed Matrix

4.2 Improved implementation

4.2.1 Inefficiency of the previous implementation

According to [1] it is very important to choose the right class to store distributed matrices. Converting a distributed matrix to a different format may require a global shuffle, which is quite expensive. Line 6 in Code 1 shows conversion from BlockMatrix to CoordinateMatrix and the reverse conversion in line 9. This conversions were unavoidable due to BlockMatrix not supporting scaler-matrix multiplication which is required in line 4 in Algorithm 1.

Another major drawback when using BlockMatrix is that it only allows its blocks being defined as a dense matrix, while matrix A in Jacobi method (4) is a tridiagonal matrix and defining it as a dense matrix wastes a lot of memory. Overall, the existing implementation of the BlockMatrix is very limited and comes with major bugs. Zero elements are being ignored and this issue leads to wrong size of the matrix. To overcome this issue we needed to initialize zero element in A and u to a small number (10^{-12} is our code).

In the reminder of this section we explain our second and third implementations of the Algorithm 1, which is implemented with bearing in mind to preserve the locality of the operations and exploiting the sparsity of matrix A.

4.2.2 Scalable Matrix-Vector multiplication in Spark

Line 3 of Algorithm 1 shows that we need to compute matrix-vector multiplication Au. To do so we created a RDD A_rdd which is a sequence of the format (column_id, (row_id, value)) for matrix A and another RDD u_rdd for u in the format of (row_id, value). In A_rdd and in each tuple column_id is the key and (row_id, value) is the value. For u_rdd the row_id is the key and value is the value. Lines 3-5 in Code 2 shows that we first create a joint RDD of A and u, in which each column of A and the corresponding element in u are grouped together. Then we do a flatMap transformation, where we multiply every column with the u element. We also change the key to row_id. Finally we perform a reduceByKey to aggregate elements which are in the same row.

```
1       while (k < max_iters && (res > tol)){
2 //          Computing A*u_k
3           val Auk_rdd = A_rdd.join(u_rdd)
4                   .flatMap{case(k, v) => v._1.map(
    mv => (mv._1, mv._2 * v._2))}
5                   .reduceByKey(_ + _)
6 //          Computing u_{k+1} = 0.5*hsq-0.5A*u_k+u_k
7           u_rdd = Auk_rdd.mapValues(hsqHalf - 0.5
    * _)
8                   .join(u_rdd)
9                   .mapValues{ s =>s._1 + s._2}
10 //         Computing A*u_{k+1}
11          val Aukplus1_rdd = A_rdd.join(u_rdd)
12                  .flatMap{case(k, v) => v._1.map(
    mv => (mv._1, mv._2 * v._2))
13                  .reduceByKey(_ + _)
14 //         Computing residual
15          res = Math.sqrt( Aukplus1_rdd .mapValues
    (invhsq*_ - 1)
16                  .map{ case(k, v) => v * v}
17                  .reduce(_ + _))
18           ress(k) = res
19           k+=1
20      }
```

Code 2. Part of code for implementation with pure RDDs

4.3 Improving the second implementation

In our second implementation in every iteration we join the current vector u with matrix A twice, once for computing the new u and once for computing the residual. Each join operation triggers a new re-partitioning of A and u among the nodes which is a costly operation. An alternative approach is to fix A and broadcast u. When broadcasting a variable, the driver sends it to each node via a P2P connection rather than shipping a copy of it with each task and as a result

broadcasting is very efficient. Once each node has a copy of u, using mapPartition transformation we can join A with u without requiring a re-partition operation. Lines 21-26 in Code 3 shows how we do this.

As we mentioned above in each iteration we compute Au twice (i.e. Au^k and Au^{k+1} respectively). A better solution is to caches Au^{k+1} and use it in the next iteration. This way we only compute one Au operation in each iteration. Further, we use accumulator to compute the residual. Accumulators are variables that are only 'added' to through an associative and commutative operation and can therefore be efficiently supported in parallel. Finally, we increased 'numTasks' for reduceByKey operations which by default splits to 8 tasks. Code 3 shows part of our third implementation.

```
1
2     var u_recie = sc.broadcast(u_rdd.collect().
    toMap).value
3 //   Computing A*u_0
4     var Auk_rdd = A_rdd.mapPartitions({ iter =>
5     for {
6        (ak, av) <- iter
7      } yield (ak, (av, u_recie.get(ak).get))
8     }, preservesPartitioning = true)
9           .flatMap{case(k, v) => v._1.map(Av => (
    Av._1, Av._2 * v._2))}
10          .reduceByKey(_ + _).cache()//reduce on
    rows
11
12    while (k < max_iters && (res > tol)) {
13 //         Computing u_{k+1} = 0.5*hsq-0.5A*u_k+u_k
14          u_rdd = Auk_rdd.mapValues(hsqHalf - 0.5
    * _)
15                  .mapPartitions({ iter =>
16                    for {(ak, av) <- iter
17                      } yield (ak, (av, u_recie.get
    (ak).get))
18                    },preservesPartitioning = true)
19                  .mapValues{ s =>s._1 + s._2}.
    cache()
20          u_recie = sc.broadcast(u_rdd.collect().
    toMap).value
21 //         Computing A*u_{k+1}
22          Auk_rdd = A_rdd
23                  .mapPartitions({ iter =>
24                    for {(ak, av) <- iter
25                      } yield (ak, (av, u_recie.
    get(ak).get))
26                    },preservesPartitioning = true
    )
27                  .flatMap{case(k, v) => v._1.map(
    Av => (Av._1, Av._2 * v._2))}
28                  .reduceByKey(_ + _).cache()//
    reduce on rows
29
30 //         Computing residual
31          val accum = sc.doubleAccumulator("
    Accumulating residual")
32          Auk_rdd.mapValues(invhsq*_ - 1)
33                 .map{ case(k, v) => v * v}
34                 .foreach(x => accum.add(x))
35          ress(k) = Math.sqrt(accum.value)
36          k+=1
```

Code 3. Improving previous implementation

4.4 Cluster specification

We ran our codes on NYU's Hadoop cluster which is called DUMBO. DUMBO has 48 nodes running Cloudera CDH 5.9.0 (Hadoop 2.6.0 with YARN). Two of the nodes are the Master nodes (called babar and hathi). Two other nodes are the login nodes (called dumbo0 and dumbo1). The remaining 44 nodes are the compute nodes that we used them to run our applications. Each node has 128 GB memory and two CPUs. Each CPU is a 8-core Intel 'Haswell' (c 2014).

4.5 Spark environment settings

To run a Spark application one needs to take care of certain configuration in order to fully exploit the available resources and more importantly to avoid memory issues. Hadoop Daemons take up to one core and approximately one GB per node in the cluster. From the available 44 nodes, one takes the rule of the driver, therefore we have 43 worker nodes, 15 cores per nodes and 127 GB of RAM for each node.
'–executor-cores' : Tells Spark to assign each executor 5 corse. Each executor is a JVM instance and will run 5 concurrent tasks. It's been observed that any application with more than 5 concurrent tasks, would lead to bad practice.
'–num-executors' : This is the number of nodes in abstract. With 5 as cores per executor, and 15 as total available cores in each node in DUMBO we conclude 3 executors per node: $3 \times 43 = 129$. So we conclude with 129 total executors.
'–executor-memory': So far we have 127 GB RAM and 3 executors per node. That is, each executor gets $\sim$ 42 GB of RAM. However we needed to increase the default overhead memory to $\sim$ 10% of the memory per executor. As a result each executor get 38 GB of RAM [8].

4.6 Number of partitions

Spark can only run one concurrent task for every partition of an RDD, up to the number of cores. In the first RDD transformation, (for example: sc.textFile(path, partition)), the partition parameter will be applied to all further transformations and actions on this RDD [7]. Setting number of the partitions too small or too large has negative impact on performance. However, setting it to $\sim$ 4 times the number of CPUs in the cluster is a good practice. This way the longest running task won't keep the rest waiting. Since we have two CPUs per node, we set the number of partitions to be $4 \times 2 \times 43 = 344$.

4.7 Spark UI

Spark provides a UI service running on the driver node which allows user to monitor stages, tasks and shuffle writes and reads for the running job. On DUMBO this service is available from http://babar.es.its.nyu.edu:8088/proxy/[application-ID] and only for running jobs. We could also access Spark UI from http://localhost:4040 when running on our local machine.

5 Results

We refer to the above implementations as Jacobi1, Jacobi2 and Jacobi3, respectively. Although we successfully tested each one of the above algorithms on DUMBO for $N \leq 5 * 10^4$ in case of Jacobi1 and $N \leq 5 * 10^5$ for Jacobi2 and Jacobi3, testing for larger problem size were not successful, possibly due to the *unresolved issue* [9] that Spark application hangs when dynamic allocation is enabled in Hadoop environment.

Furthermore, we are unable to provide the running time of the above tests. We asked DUMBO system administration multiple times to enable Spark log history server so that we can access those information, however the log server is not yet fully available. We only emphasis that using Spark's default setting (i.e. executor-cores =1, num-executors=2 executor-memory =2GB and driver-memory=5GB) for number of iterations =10 and $N = 10^5$, our Jacobi1 job was killed by the system after hanging for a few hours, however Jacobi2 and Jacobi2 jobs finished successfully.

Here we present the test results on our local machine which has a single 8-core Intel CPU and 8 GB RAM. We set the number of iterations to 10 in all of our following tests. Similarly, number of partitions of data is set to 8. Table1 compares the three codes for $N = 5000$ and we can see that Jacobi1 and Jacobi2 run in the same amount of time however Jacobi2 shuffle much less data around. Jacobi3 outperforms both of them in terms of running time and shuffle size.

Table2 shows the result of testing for $N = 50000$. Jacobi1 was unable to continue continue due to the lack of enough memory space. On the other hand and similar to above, Jacobi3 outperforms Jacobi2 by a factor of 2 in both running time the shuffle size.

Implementation	Total Time (GC Time)	Shuffle Write
Jacobi1	59.3 s (7.4 s)	28.9 MB
Jacobi2	60.0 s (0.7 s)	1.4 MB
Jacobi3	29.4 s (.3 s)	0.8 MB

Table 1. Test results for Jacobi1,2 and 3 for $N = 5 * 10^3$.

Implementation	Total Time (GC Time)	Shuffle Write
Jacobi1	*NA*	*NA*
Jacobi2	138 s (10.5 s)	13.2 MB
Jacobi3	66 s (7.6 s)	7.5 MB

Table 2. Test results for Jacobi1,2 and 3 for $N = 5 * 10^4$.

References

[1] Spark 2.1 Documentation https://spark.apache.org/docs/2.1.0/mllib-data-types.html

[2] Jorge L. Reyes-Ortiz, Luca Oneto, and Davide Anguita *Big Data Analytics in the Cloud: Spark on Hadoop vs MPI/OpenMP on Beowulf.* Procedia Computer Science,Volume 53, 2015, Pages 121-130, 2015.

[3] Alex Gittens, et al *Matrix Factorizations at Scale: a Comparison of Scientific Data Analytics in Spark and C+MPI Using Three Case Studies*, 2016 IEEE International Conference on Big Data (Big Data).

[4] MathWork R2016b *Apache Spark Basics.* http://fr.mathworks.com/help/compiler/spark/apache-spark-basics.html

[5] U Kang, Charalampos E. Tsourakakis, and Christos Faloutsos, *PEGASUS: A Peta-Scale Graph Mining System - Implementation and Observations,* In IEEE International Conference on Data Mining (ICDM 2009).

[6] Roberto Agostino Vitillo *Spark best practices.* https://robertovitillo.com/2015/06/30/spark-best-practices/

[7] Jacek Laskowski *Aastering Apache Spark.* https://jaceklaskowski.gitbooks.io/mastering-apache-spark

[8] Ramprasad Pedapatnam *Understanding Resource Allocation configurations for a Spark application.* http://site.clairvoyantsoft.com/understanding-resource-allocation-configurations-spark-application/

[9] https://issues.apache.org/jira/browse/SPARK-16441