Matrix Factorization with SGD using Spark and Redis

Linquan Chen, Yuankun Chang, Vinodh Paramesh
Electrical and Computer Engineering, CMU

Abstract

Spark is an efficient, general-purpose programming language to be used interactively to process large datasets on a cluster. This paper focuses on using Spark Programming Framework to optimize the sequential matrix factorization using Stochastic Gradient Descent(SGD) algorithm. Based on the sequential program, we implemented a basic programming using spark and analyzed the disadvantages of this version. Then, we optimize the program by updating the blocks in parallel, partition and group the blocks, and also tuning the spark configuration parameters to improve the performance.

Keywords: Matrix Factorization, SGD, Spark, RDD

1. Overview

Low-rank matrix factorization has recently been applied with great success on matrix completion problems for application likes recommendation system, link prediction for social networks, and click prediction for web search[1]. However, as this method is applied to larger and larger datasets, such as recommender systems like Amazon and YouTube, the data management quickly become challenging and facing a bottleneck. So we want to focus on this topic and try to improve the speed of matrix factorization.

This paper is organized as follows. Section 2 describes the background of matrix factorization and Spark's programming model, and also the related work. Section 3 shows the basic spark implementation and optimize the Spark implementation by updating blocks in parallel, partition and tuning the spark configuration. And end with a discussion and future plan in Section 4.

2. Background and Related Work

In this part, we did some survey on matrix factorization and related techniques. Firstly, we find that matrix factorization techniques are usually more effective in doing recommendations, such as personalized recommendations we received on Amazon. So it mainly deals with large scale datasets. Secondly, we find that there are mainly two

approaches for matrix factorization, stochastic gradient descent(SGD) and alternation least squares(ALS). Compared with ALS approach, SGD combines implementation ease with a relatively fast running time, so we choose to use SGD approach for our project.

2.1 background

2.1.1 Matrix Factorization

Matrix factorization is a factorization of a matrix into a product of matrices. For example, we have a M-by-N data matrix Z, we can use two matrices W and H, whose size are M-by-K and K-by-Q, to estimate Z, such as W * H = Z. And K is a parameter that's typically referred to as rank.

We can use sequential Stochastic Gradient Descent(SGD) algorithm for matrix factorization. This is an iterative algorithm and we can refine the values of the two matrices by iterating. And for a right result, the factor matrices will not change a lot when iterate for several times. Therefore, we can utilize Apache Spark programming framework to accelerate the speed.

2.1.2 Spark

As we know, MapReduce and its variants have been highly successful in implementing large-scale data-intensive applications on commodity clusters[2]. However, this model can't suit for all applications, such as applications that reuse a working set of data across multiple parallel operations. At these cases, Apache Spark programming framework is a good choice. It can support iterative process and maintains the scalability and fault tolerance likes MapReduce. To achieve this goal, Spark uses a dataset called Resilient distributed Datasets(RDD).

RDD represents a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. User can explicitly cache an RDD in memory across machines and reuse it in multiple MapReduce-like parallel operations. Therefore, it is an efficient, general-purpose programming language to be used interactively to process large datasets on a cluster.

2.2 related work

Researchers have explored the feasibility of using mapreduce to parallelize the sequential SGD by creating variants of SGD algorithm. Generally, the idea in these algorithms is to partition the input matrix into d1 * d2 blocks, which are distributed in

mapreduce clusters. Both row and column factors are blocked conformingly and these algorithms are designed such that each block can be processed independently in the map phase taking only corresponding blocks of factors as input.While some algorithms directly update the factors in map phase while others aggregate the factor updates in a reduce phase[4]. Such parallelized SGD algorithms for matrix factorization are on par in terms of convergence time while running-time is much faster and stable for large data sets.

Success of Distributed SGD has inspired researchers to explore Spark as a framework to solve this problem since matrix factorization inherently allows the intermittent result be re-used. Infact, [1] Sparkler is an extension to the spark framework , which works on large-scale matrix factorization. Deriving inspiration from this, we decided to use spark to solve the problem of matrix factorization.

3. Detailed techniques and results analysis

We utilized Apache Spark parallel programming framework to optimize a complete, functional(Python) implementation of collaborative filtering[3] using a matrix factorization model and stochastic gradient descent optimization.

3.1 Spark Implementation – Basic

Based on the sequential complete, functional(Python) implementation, we utilized the spark programming to implement the algorithms. we read the data file and used RDD to store the data, also using cache. As for H and W matrices, we used two RDDs to store the values. At last, the most important part-iterative, we just followed the implementation provided and updated one block at one time. Also, because updating a part of the RDD is very expensive, so we will transfer the RDD(W and H) into list at every slave, and updated this list every time. After updating, the driver will collect the updated W and H, and parallelize them into RDD again.

After doing experiment, we found this implement is not very efficient. Because we was bounded by the sequential implementation. In fact, although we used RDD to store the reused data, the program framework is still sequential and wasted lots of resource in updating process, such as transfer the RDD into two dimensional array and collect the data between slaves and drivers, also rebuilt the RDD again based on the updated W and H.

Based on the above problems, we optimized the programing in three aspects: Firstly, we updated N(the partition number) blocks at same time instead of one block; Secondly,

we used partitions on specific RDDs to distribute them across slaves, and grouped Z, W and H blocks; Thirdly, we tuning the spark configuration parameters to improve the performance.

3.2 Spark Implementation – Optimization

3.2.1 Updating blocks parallel

As shown in Figure 1, we updated the blocks in the diagonal order, which means that we updated the block 1(red), block 2(yellow), block 3(black) and block 4(green) at same time respectively. We can know from the figure 1, every color blocks correspond to different parts of H and W blocks, therefore we can update the H and W matrices parallel and not affect each other.

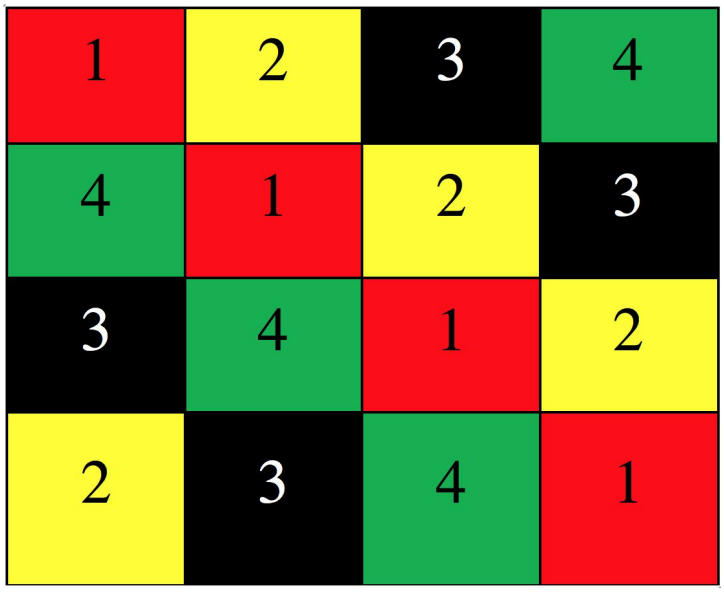

Figure 1. Updating blocks in diagonal

3.2.2 Partition blocks

Considering the characteristic of RDD, we divided the Z, W and H into different blocks based on core num. Because the Z matrix will be used frequently and we divided it into N * N blocks based on maximum user id and maximum movie id. And for W and H matrices, they correspond user and movie, and we divided them into N blocks respectively.

As for number N, we decided it based on the total cores of slaves. In order to utilize the resource in nearly 100%, we treated one core as one process and update one block. Therefore, the core number is equally to partition number. For example, we built a

cluster with one master and two slaves, every slave has four cores and the total cores is eight, so the partition number is eight.

3.2.3 Spark Configuration

At the process of optimization, we found that tuning the spark configuration is really helpful and can help me understand the Spark framework more deeply.

There are two really important parameters: --executor-memory and --num-executors. we noticed that maybe there is a bug for the new version of EMR. When we set the --num-executors equals to 2, but we always only found one executor working. we think there may have a bug for calculating the executors. Therefore, we need to lower the --executor-memory. For example, when the memory of slave is 15GB, we set the --executor-memory to 10GB. And for --executor-cores, because the slave also need resource to run the operate system, so we need to leave one to two cores for the system.

Other important parameters are relative with driver, such as --driver-memory and --drive-cores. Because the drive node not only run the driver program, but also run the YARN and Application Manager, therefore, we can allocate all resources to driver. For example, if the driver node has 15GB memory and 4 cores, we will set the --driver-memory to 8GB and --driver-cores to 2.

When we a built cluster with one master and two slaves using m4.xlarge instances, the program can run fast but the efficiency is very low. After analyzing the data, we found the cluster had wasted lots of memory when execute the program. Therefore, we change the cluster instance to c4.xlarge, whose memory is 7.5 GB. With this cluster, the program can achieve both speed and efficiency.

The most challenge is to run the 22M data with rank 500, we had account lots of problems relative to memory. Firstly, the process is out of memory, so we change the instance to r4.xlarge whose memory is 30 GB. After that, we counted a problem that result size exceed the maximum return size of driver, therefore, we tuned the parameter --driver-maxResultSize to 2 GB(the default is 1 GB). Then we come across a problem that the memory exceeds the physical memory, So we tuned the –executor-memory, and utilize the memory mostly. However, it broke the heap after running for two hours. Until now, we still can not succeed to run the program for 22M data with rank 500.

3.2.4 Final Result

a. ratings_1M.csv

```
Calculating RMSE
Results
=======
Andrew ID = linquanc
Memory =  19
Runtime = 191
Runtime-memory product = 3629
Dataset = /ratings_1M.csv
RMSE = 0.896892
```

b. ratings_10M.csv

```
Calculating RMSE
Results
=======
Andrew ID = linquanc
Memory =  19
Runtime = 1963
Runtime-memory product = 37297
Dataset = /ratings_10M.csv
RMSE = 0.859759
```

4. Conclusion

4.1 Problem

Based on the above analyze, the most challenge of my program is about memory. we think there are two steps to solve the problem: Firstly, we need to optimize my program. Because, we have created many large RDDs during the iterative processes. we need to discard them when no longer need them. Therefore, we need to re-analyze the process and optimize the memory utilization. Secondly, we can optimize the program at the aspect of tuning Spark Configuration parameters, such as cluster instance type and number, executor memory and number, driver memory and number, memory fraction and so on.

4.2 Future Plan

In the next step, we want to combine spark and redis. As we said in the abstract report, in theory, redis can accelerate Spark performance by up to 50 times. And we also know there is a Redis-Spark connector that expose the Redis data structures and API to

Spark. Since we used python to write spark in the previous work, we are working on translating it into scala at this time. Hopefully, we could work it out.

Reference

[1] Li, Boduo, Sandeep Tata, and Yannis Sismanis. "Sparkler: supporting large-scale matrix factorization." *Proceedings of the 16th International Conference on Extending Database Technology*. ACM, 2013.
[2] Zaharia, Matei, et al. "Spark: Cluster Computing with Working Sets."*HotCloud* 10 (2010): 10-10.
[3] https://en.wikipedia.org/wiki/Collaborative_filtering
[4]Large-Scale Matrix Factorization with Distributed Stochastic
Gradient Descent http://www.almaden.ibm.com/cs/people/peterh/dsgdTechRep.pdf