Explore and Optimize Matrix Factorization with Spark

Linquan Chen, *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 Apache Spark Programming Framework to optimize the sequential matrix factorization using Stochastic Gradient Descent(SGD) algorithm. Based on the sequential program, I implemented a basic program using spark and analyzed the disadvantages of this version. Then, I optimize the program using parallel updating the blocks, partition and group the blocks, and also tuning the spark configuration parameters to improve the performance.

Keywords: Matrix Factorization, SGD, Spark, RDD

1. Introduction

Low-rank matrix factorization has recently been applied with great success on matrix completion problems for application like 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 bottle-neck. In this paper, I utilized Apache Spark parallel programming framework to optimize a complete, functional(Python) implementation of collaborative filtering using a matrix factorization model and stochastic gradient descent optimization(written by Jinliang Wei)[2].

This paper is organized as follows. Section 1 describes the background of matrix factorization and Spark's programming model and RDDs. Section 2 shows the basic spark implementation and explore the problem. In Section 3, I optimize the Spark implementation using parallel updating blocks, partition and tuning the spark configuration. And end with a discussion and future plan in Section4.

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 a 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.

1.2. Spark

As we know, MapReduce and its variants have been highly successful in implementing large-scale data-intensive applications on commodity clusters[3]. 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 if 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. Spark Implementation – Basic

Based on the sequential complete, functional(Python) implementation, I utilized the spark programming to implement the algorithms. I read the data file and used RDD to store the data, also using cache. As for H and W matrices, I used two RDDs to store the values. At last, the most important part-iterative, I just followed the implementation provided and updated one block at one time. Also, because updating a part of the RDD is very expensive, so I 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, I found this implement is not very efficient. Because I was bounded by the sequential implementation. In fact, although I 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, I optimized the programing in three aspects: Firstly, I updated N(the partition number) blocks at same time instead of one block; Secondly, I used partitions on specific RDDs to distribute them across slaves, and grouped Z, W and H blocks; Thirdly, I tuning the spark configuration parameters to improve the performance.

3. Spark Implementation – Optimization

3.1 Updating blocks parallel

As shown in Figure 1, I updated the blocks in the diagonal order, which means that I 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.

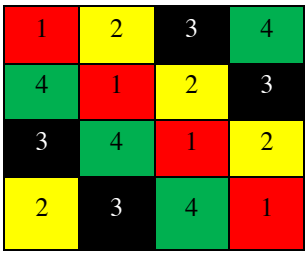

Figure 1. Updating blocks in diagonal

3.2 Partition blocks

Considering the characteristic of RDD, I divided the Z, W and H into different blocks based on core num. Because the Z matrix will be used frequently and I 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 I divided them into N blocks respectively.

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

3.3 Spark Configuration

At the process of optimization, I 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. I noticed that maybe there is a bug for the new version of EMR. When I set the --num-executors equals to 2, but I always only found one executor working. I think there may have a bug for calculating the executors. Therefore, I need to lower the --executor-memory. For example, when the memory of slave is 15GB, I 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, I will set the --driver-memory to 8GB and --driver-cores to 2.

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

3.4 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. Future Plan

Based on the above analyze, the most challenge of my program is about memory. I think there are two steps to solve the problem: Firstly, I need to optimize my program. Because, I have created many large RDDs during the iterative processes. I need to discard them when no longer need them. Therefore, I need to re-analyze the process and optimize the memory utilization. Secondly, I 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.

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]https://theproject.zone/student/writeup/15/107#section_1

[3] Zaharia, Matei, et al. "Spark: Cluster Computing with Working Sets."*HotCloud* 10 (2010): 10-10.