A Fast Distributed Stochastic Gradient Descent Algorithm for Matrix Factorization

Fanglin Li FANGLINLI@BUPT.EDU.CN
Bin Wu WUBIN@BUPT.EDU.CN
Liutong Xu XLIUTONG@BUPT.EDU.CN
Chuan Shi SHICHUAN@BUPT.EDU.CN
Jing Shi SHIJING_917@126.COM
Beijing Key Lab of Intelligent Telecommunication Software and Multimedia
Beijing University of Posts and Telecommunications
Beijing 100876, China

Editors: Wei Fan, Albert Bifet, Qiang Yang and Philip Yu

Abstract

The accuracy and effectiveness of matrix factorization technique were well demonstrated in the Netflix movie recommendation contest. Among the numerous solutions for matrix factorization, Stochastic Gradient Descent (SGD) is one of the most widely used algorithms. However, as a sequential approach, SGD algorithm cannot directly be used in the Distributed Cluster Environment (DCE). In this paper, we propose a fast distributed SGD algorithm named FDSGD for matrix factorization, which can run efficiently in DCE. This algorithm solves data sharing problem based on independent storage system to avoid data synchronization which may cause a big influence to algorithm performance, and synchronous operation problem in DCE using a distributed synchronization tool so that distributed cooperation threads can perform in a harmonious environment.

Keywords: Matrix factorization, SGD, Recommender system, Distributed computing

1. Introduction

With the development of Internet, especially the mobile internet, information gets explosive increasing. More than 90% data in the world is created in recent years. With the inflation of information, people access to useful information with more difficulties, which is called information overload problem. The main way to solve the problem of information overload is through search engine and recommender system. For search engine, people search information with a couple of corresponding keywords, which has strong objective. However, as a more intelligent and active according to user information, like purchase history and basic personal information, recommender system is another kind of solutions to information overload. According to different users, it will recommend the most possible interested information or products, which has strong personalization.

To recommender systems, the key point is how to help users find their desired products precisely. As demonstrated in KDD Cup 2011 Dror et al. and Netflix competition Bell and Koren (2007), a collaborative filter using low-rank matrix factorization has been considered as one of the best models for recommender systems. The low-rank matrix decomposition aims to divide rating matrix into two low-rank factor matrices, and uses the product of these two factor matrices to approximate the original rating matrix. The approximation is to minimize the error between the predicted and the original rating matrix. More specifically, considerate the user number is m, the number of items is n, and rating matrix R (m × n) that records the preference of the u_{th} user on the v_{th} item at the (u,v) entry, $r_{u,v}$. The key to the problem is to find two low-rank matrices, P (m × d) and Q (n × d), that makes PQ^T approximate the rating matrix R. The optimization problem is equivalent to the following Formula (1):

$$\min_{P,Q} \sum_{(u,v)\in \mathrm{R}} \left(r_{u,v} - \mathbf{p}_u \mathbf{q}_v^T\right)^2 + \lambda_P ||\mathbf{p}_u||^2 + \lambda_Q ||\mathbf{q}_v||^2 \tag{1}$$

where $\mathbf{p}_u$ means the u_{th} line of matrix P, $\mathbf{q}_v^T$ means the v_{th} line of matrix Q. Simply considering $\mathbf{p}_u \mathbf{q}_v^T$ is the predicted score to users. While the decomposition is easy to generate over-fitting problem, so add the normalization factors λ_P and λ_Q into the objective function, which try to avoid overfitting effect. Scholars have proposed many methods for solving Formula (1), e.g. Zhou et al. (2008); Koren et al. (2009); Pilászy et al. (2010). And the most popular are SGD, ALS (alternative least squares) and etc. The core idea of SGD is to select a $r_{u,v}$ in R, then finds out the corresponding factor vector $\mathbf{p}_u$ from the user factor matrix P, $\mathbf{q}_v$ from the item factor matrix Q, calculates the predicted score $\mathbf{p}_u \mathbf{q}_v^T$, and updates parameters according to the following two rules Zhuang et al. (2013):

$$\mathbf{p}_u \leftarrow \mathbf{p}_u + \gamma \left(e_{u,v} \mathbf{q}_v - \lambda_P \mathbf{p}_u\right) \tag{2}$$

$$\mathbf{q}_v \leftarrow \mathbf{q}_v + \gamma \left(e_{u,v} \mathbf{p}_u - \lambda_Q \mathbf{q}_v\right) \tag{3}$$

where $e_{u,v} = r_{u,v} - \mathbf{p}_u \mathbf{q}_v^T$, which represents the difference between predicted score and actual score and γ is the learning rate.

Stand-alone SGD algorithm for solving matrix factorization has been widely applied. With the growth of data volume, the requirement for implementation of parallel SGD algorithm comes out. However, because of the sequential future of SGD algorithm, it is difficult to parallelize SGD under advanced architectures such as GPU, multi-core CPU or distributed Zhuang et al. (2013). Though, many parallel SGD algorithms have been raised, such as PSGD Zinkevich et al. (2010), DSGD Gemulla et al. (2011) and FPSGD Zhuang et al. (2013). Due to having synchronous operation after each rule being processed, DSGD has to confront the locking problem that may occurs if the rating matrix is not gridded evenly. And DSGD randomly picks instances from a block of R for updating operations, it also suffers from the problem of memory discontinuity. However, FPSGD, based on shared memory system, can avoid the synchronous operation problem briefly. It grids rating matrix into at least $(s+1) \times (s+1)$ blocks, s is the number of worker threads, to settle the locking problem. As to memory discontinuity problem, FPSGD adopts a solution called partial random method which randomly chooses a free block and accesses the block sequentially.

In order to settle the locking problem that DSGD faces in DCE, based on DSGD Gemulla et al. (2011) and FPSGD Zhuang et al. (2013), we put forward a fast distributed SGD algorithm called FDSGD. And it mainly needs to solve the following two problems among threads in DCE, data sharing and synchronous operation.

Data Sharing: Data needing to be shared among worker threads could store in one public storage system so as to avoid data synchronization. We chose Memcached mem which

is a high-performance, distributed memory system as our storage system and our storage strategy is described in section 3.1.
Synchronous Operation: Conflicts occur frequently when worker threads need to work together at critical resources. To realize synchronous operation of cooperating threads in DCE, we adopt ZooKeeper zoo to manage a distributed sharing lock. Worker threads that want to access critical resources needing to get the lock first and our strategy to control a distributed sharing lock is described in section 3.2.

This paper is organized as follows. We will introduce DSGD and FPSGD in section 2. Section 3 introduces the difficulties of parallelized SGD in DCE and our solutions. Section 4 presents our experiment environment and algorithm parameters. And section 5 will use RMSE of test dataset to verify the performance of our algorithms. Finally, section 6 summarizes our work and gives future directions.

2. Related Work

Researchers have proposed many parallelized SGD algorithms for matrix factorization, such as PSGD Zinkevich et al. (2010), DSGD Gemulla et al. (2011) and FPSGD Zhuang et al. (2013). We will introduce two of them in detail, DSGD, parallelized in DCE, and FPSGD, parallelized in shared memory environment.

2.1. DSGD Algorithm

Although the traditional SGD algorithm is a sequential approach when updating the factor matrices and other parameters, for the independent blocks in the rating matrix, the corresponding factor vectors of these independent blocks are independent of each other too. DSGD just uses this feature to achieve the parallelization of SGD in DCE. It grids rating matrix into d×d blocks, and two blocks are mutually independent if they are both not in the same line and column. Figure 1 shows a rating matrix gridded into 3×3 blocks and its all six rules of each rule involves three independent blocks Gemulla et al. (2011).

Algorithm 1 DSGD's Process.

```
Require: rating matrix R, user latent-factor matrix P, item latent-factor matrix Q, maximum iterations K, number of worker threads t.
1: grid R into t×t, P into t×1, Q into t×1 blocks;
2: generate t rules covering all the rating matrix blocks and each rule covering t mutually independent blocks;
3: for $k \in [1, K]$ do
4:     order the t rules sequentially or randomly;
5:     for each rule do
6:         assign the corresponding t blocks to t workers;
7:         for each block $b \in [1, t]$ distributed do
8:             run SGD algorithm on b;
9:         end for
10:    end for
11: end for
```

$$
\begin{array}{cccccc}
(a) & (b) & (c) & (d) & (e) & (f) \\
\begin{array}{|c|c|c|}\hline \underline{\mathbf{b}_{00}} & b_{01} & b_{02} \\ \hline b_{10} & \underline{\mathbf{b}_{11}} & b_{12} \\ \hline b_{20} & b_{21} & \underline{\mathbf{b}_{22}} \\ \hline\end{array} &
\begin{array}{|c|c|c|}\hline b_{00} & b_{01} & \underline{\mathbf{b}_{02}} \\ \hline \underline{\mathbf{b}_{10}} & b_{11} & b_{12} \\ \hline b_{20} & \underline{\mathbf{b}_{21}} & b_{22} \\ \hline\end{array} &
\begin{array}{|c|c|c|}\hline b_{00} & \underline{\mathbf{b}_{01}} & b_{02} \\ \hline b_{10} & b_{11} & \underline{\mathbf{b}_{12}} \\ \hline \underline{\mathbf{b}_{20}} & b_{21} & b_{22} \\ \hline\end{array} &
\begin{array}{|c|c|c|}\hline \underline{\mathbf{b}_{00}} & b_{01} & b_{02} \\ \hline b_{10} & b_{11} & \underline{\mathbf{b}_{12}} \\ \hline b_{20} & \underline{\mathbf{b}_{21}} & b_{22} \\ \hline\end{array} &
\begin{array}{|c|c|c|}\hline b_{00} & \underline{\mathbf{b}_{01}} & b_{02} \\ \hline \underline{\mathbf{b}_{10}} & b_{11} & b_{12} \\ \hline b_{20} & b_{21} & \underline{\mathbf{b}_{22}} \\ \hline\end{array} &
\begin{array}{|c|c|c|}\hline b_{00} & b_{01} & \underline{\mathbf{b}_{02}} \\ \hline b_{10} & \underline{\mathbf{b}_{11}} & b_{12} \\ \hline \underline{\mathbf{b}_{20}} & b_{21} & b_{22} \\ \hline\end{array}
\end{array}
$$

Figure 1: Six rules of a 3 × 3 gridded matrix

Algorithm 1 gives the overall procedure of DSGD, we will take an in-depth analysis of why the 7th step can be executed in parallel. Figure 2a gives a 3×3 gridded rating matrix and 3 workers as an example and we list three rules that cover all blocks of the 3×3 gridded matrix in Figure 2b to explain the whole process.

(a) 3 × 3 gridded rating matrix R and corresponding segments of P and Q, p_i is the i_{th} segment of P and q_j^T is the j_{th} segment of Q

$$
\begin{array}{|c|c|c|}\hline b_{00} & b_{01} & b_{02} \\ \hline b_{10} & b_{11} & b_{12} \\ \hline b_{20} & b_{21} & b_{22} \\ \hline\end{array}
=
\begin{array}{|c|}\hline p_0 \\ \hline p_1 \\ \hline p_2 \\ \hline\end{array}
\begin{array}{|c|c|c|}\hline q_0^T & q_1^T & q_2^T \\ \hline\end{array}
$$

(b) An example of an iteration contains three rules (rule1, rule2 and rule3) covering all the blocks of rating matrix R

$$
\begin{array}{l|c|c|c|c|c|c|c|c|c|c|c|c|}
\hline
\text{worker1} & \underline{\mathbf{b}_{00}} & b_{01} & b_{02} & p_0\,q_0^T & b_{00} & \underline{\mathbf{b}_{01}} & b_{02} & p_0\,q_1^T & b_{00} & b_{01} & \underline{\mathbf{b}_{02}} & p_0\,q_2^T \\ \hline
\text{worker2} & b_{10} & \underline{\mathbf{b}_{11}} & b_{12} & p_1\,q_1^T & b_{10} & b_{11} & \underline{\mathbf{b}_{12}} & p_1\,q_2^T & \underline{\mathbf{b}_{10}} & b_{11} & b_{12} & p_1\,q_0^T \\ \hline
\text{worker3} & b_{20} & b_{21} & \underline{\mathbf{b}_{22}} & p_2\,q_2^T & \underline{\mathbf{b}_{20}} & b_{21} & b_{22} & p_2\,q_0^T & b_{20} & \underline{\mathbf{b}_{21}} & b_{22} & p_2\,q_1^T \\ \hline
\end{array}
$$

Figure 2: A graphical display of DSGD's process.

In rule1, containing three independent blocks b_{00}, b_{11}, b_{22} and its corresponding independent factor vectors $p_0q_0^T$, $p_1q_1^T$, $p_2q_2^T$, it can be executed parallel without conflicts when updating factor vectors. Similarly, rule2 and rule3, also can be processed in parallel. It is called an iteration after these three rules being performed one by one. From the iteration process in 2b, we can notice that worker1, worker2 and worker3 respectively uses p_0, p_1, p_2 of factor matrix P when processing each rule. This example shows that, after the execution of one rule, DSGD only needs to synchronize factor matrix Q before move to next rule. However, if we choose different rule strategy, factor matrix P may also need to be synchronized. And because of the synchronous operation, DSGD may runs into the locking problem that all threads have to wait for the slowest thread if the blocks are gridded unevenly.

2.2. FPSGD Algorithm

Zhuang et al. (2013) analyzed the locking and memory discontinuity problem of the existing parallel SGD algorithms, HogWild Niu et al. (2011) and DSGD Gemulla et al. (2011), and proposed FPSGD algorithm running in shared memory environment. Algorithm 2 gives the overall procedure of FPSGD.

FPSGD uses the thought of gridding matrix into blocks just as DSGD. Because of FPSGD based on shared memory system, there is no data synchronous problem that DSGD

Algorithm 2 FPSGD's Process.

```
Require: rating matrix R, user latent-factor matrix P, item latent-factor matrix Q, maximum iterations K, number of worker threads T.
 1: shuffle R randomly;
 2: grid R into a set B with at least (T+1)×(T+1) blocks;
 3: sort each block by user and item identities;
 4: construct a scheduler and launch t worker threads;
 5: for $k \in [1, K]$ do
 6:   for each thread $t \in [1, T]$ parallel  do
 7:     while if it is not reach next iteration do
 8:       randomly get a block b with the minimum times of being executed on SGD
          algorithm among the free blocks from the scheduler;
 9:       run SGD algorithm on b;
10:     end while
11:   end for
12: end for
```

has. FPSGD needs to grid R into at least (t + 1)×(t + 1) blocks, so it can always assign a free block which is independent from all the blocks that are being processed to a thread and its scheduler does not suffer from the locking problem Zhuang et al. (2013). At the same time, FPSGD also can deal with rating matrix with random shuffling operation to balance the distribution of block size as far as possible.

The continuity of memory access has significant effects to the performance of a program. DSGD randomly picks a rating record from the chosen block, which is called as random method. So it may causes memory access discontinuity because identities of the chosen users and items are not continuous. FPSGD proposed a partial random method which randomly chooses a block and accesses the block in order, and proved its effectiveness. To balance the number of being executed on SGD algorithm of each block, FPSGD records the number for each block. The blocks with the minimum number have the maximum priority to be choosen by the scheduler. FPSGD has no iteration concept in the traditional sense, so it calls every cycle of processing $(t + 1)^2$ blocks as an iteration Zhuang et al. (2013). FPSGD has the limitation that the full rating matrix and factor matrices need to be stored in memory and this limitation can be a serious drawback for large factorization problem.

3. Our FDSGD Algorithm

Based on the two kinds of parallel SGD algorithm we introduced in section 2, we propose a fast distributed stochastic gradient descent algorithm, FDSGD, described in Algorithm 3. This algorithm runs in DCE compared with FPSGD and avoids the locking problem that DSGD has to face, which is a kind of fast DSGD in some degree. To implement FDSGD, we need to solve the following two challenges, data sharing in DCE and synchronous operation among distributed cooperation threads.

Algorithm 3 FDSGD's Process.

```
Require: rating matrix R, user latent-factor matrix P, item latent-factor matrix Q, maximum iterations K, number of worker threads T.
 1: initialize P and Q with random numbers between [0, 1];
 2: grid R into a set B with at least (T+1)×(T+1) blocks as well as grid the factor matrices
    P and Q at least (T+1)×1 blocks;
 3: store all the blocks produced from the former operations into a high speed public storage
    system, such as memcached or redis;
 4: for each thread $t \in [1, T]$ parallel do
 5:   for $k \in [1, K]$ do
 6:     while if it is not reach next iteration do
 7:       get a block b with the minimum times of being executed on SGD algorithm
          among the free block sets; //needs synchronization
 8:       get corresponding segments of P and Q and other parameters from the public
          storage system;
 9:       run SGD algorithm on b;
10:       update the new value of the related factors and parameters into the public storage
          system; //needs synchronization
11:     end while
12:   end for
13: end for
```

3.1. Data Sharing

After DSGD performing a rule, it has a data synchronous process before processing the next rule. However, there is no data synchronous problem to FPSGD which runs in a shared memory system. In DCE, in order to avoid the synchronous process, data needing to be shared can be stored in a high speed public storage system. In our experiment, we choose Memcached, a high performance storage system based on memory, and all the workers need to store and update the sharing data in the Memcached cluster.

As described in Algorithm 3, we grid rating and factor matrices into blocks and assign a number to each block. Then respectively encapsulate a block number and its content into a key-value pair and store it into Memcached Cluster. Each thread attempt to get a block that is free and with the minimum number of being executed on SGD algorithm. And according to the block number, thread will access Memcached to get the content of corresponding rating blocks, factor blocks and other parameters. After the execution of SGD on each block, the new value should be updated into Memcached Cluster again.

According to the limitation of Memcached that the maximum size of its storing value is 1M map; Fitzpatrick (2004), we will compress those exceeded value to put them into Memcached. And we build a first-level index to those value size over 1M after compression so that they can be stored properly. Though this method solves the limited, it needs more than once to get those exceeded value that still need to use first-level index after the compression operation.

3.2. Synchronous Operation

In order to avoid the conflict problems when workers need to obtain the next free block or updating the sharing data, it needs synchronous operation (Algorithm 4) among workers. ZooKeeper Hunt et al. (2010), based on Paxos algorithm, is a high available distributed data management framework. It guarantees strong consistency of data in a distributed environment. We adopt ZooKeeper to realize distributed sharing lock, achieving the synchronous operation among workers. For the purpose of calculating the entire RMSE of test sets, we also use ZooKeeper to realize a producer-consumer queue and to calculate each block's RMSE parallel using its watcher registration and asynchronous notification mechanism.

Although worker threads may encounter conflicts when they attempt to obtain the sharing lock and those failing to get the lock have to wait for the lock to be released, the execution procedure of SGD algorithm on each block which occupies most of the time can perform without keeping the sharing lock, which largely avoids the conflicts among workers. And the worker that occupies the lock will release it in a very short period of time, so the waiting time for those attempting to get the sharing lock will not to be long even when conflict occurs. It means that our strategy to those failing to get the sharing lock is to have the thread sleep for a while before attempting to obtain the lock again.

Algorithm 4 Process of Synchronous Operation for Attemping to Get a Free Block

```
1: while (true) do
2:   if (success to get the sharing lock) then
3:     get a block with the minimum times of being executed on SGD algorithm among
       the free block sets and its relevant parameters;
4:     release the lock;
5:     break;
6:   else
7:     wait a short time break and continue;
8:   end if
9: end while
```

4. Implementation of FDSGD

In this section, we will introduce our selected datasets, loss function, cluster environment and other relevant parameters of our FDSGD algorithm.

DataSets: We use two datasets, MovieLens 10M(Part B) and Netflix. We split both datasets into two parts, training sets and test sets. The following form (Table 1) shows the statistics information and some algorithm parameters of the two datasets.

Table 1: Parameters of FDSGD for different datasets.

Dataset	User Num	Item Num	Training Set	Test Set	Factor Num	Blocks	Learning Rate
MovieLens	71,567	65,133	9,301,274	698,780	40	32×32	0.003
Netflix	2,649,429	17,770	99,072,112	1,408,395	40	32×32	0.002

Cluster Environment: We use 4 linux servers to build our cluster environment and each server is with Intel(R) Xeon(R) E56202.40GHz processor and 16GB memory.
Loss Function: We adopt Formula (4) that considered user and item bias and average score as our loss function.

$$\min_{P,Q,a,b} \sum_{(u,v)\in \mathrm{R}} \left(r_{u,v} - \mathbf{p}_u\mathbf{q}_v^T - a_u - b_v - avg\right)^2 + \lambda_P||\mathbf{p}_u||^2 + \lambda_Q||\mathbf{q}_v||^2 + \lambda_{\mathbf{a}}||\mathbf{a}||^2 + \lambda_{\mathbf{b}}||\mathbf{b}||^2 \tag{4}$$

where **a**, **b** respectively represents user bias vector and item bias vector, $\lambda_{\mathbf{a}}, \lambda_{\mathbf{b}}$ are the penalty of regularized term of **a** and **b**, a_u, b_v represent the u_{th} element of **a** and the v_{th} element of **b**, avg is the average score of rating records and the other parameters are the same with Formula (1).
Parameters: We follow how parameters are chosen in Yu et al. (2012) and the value of factor matrices P and Q are initialized between 0 and 1 randomly. We also shuffle rating matrix to see how the balance of block size influences the performance of our algorithm.
Evaluation: As most researchers do, we adopt RMSE of the test sets to evaluate our algorithms. RMSE is defined as Zhuang et al. (2013):

$$\sqrt{\frac{1}{number\ of\ ratings} \sum_{(u,v)\in \mathrm{R}} \left(r_{u,v} - p_{u,v}\right)^2} \tag{5}$$

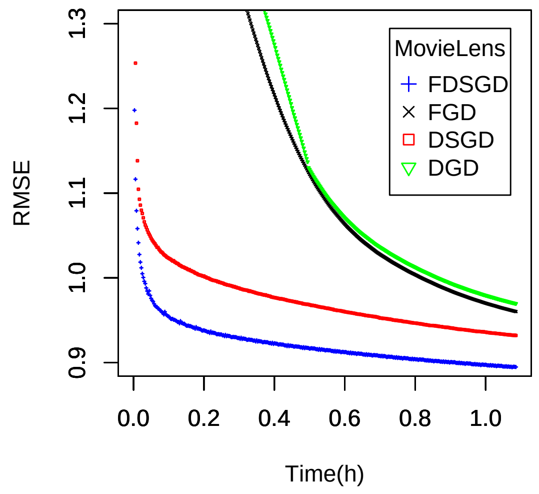
where R is the rating matrix of the test set and $p_{u,v}$ is the predicted rating value.
Implementations: We choose Hadoop had, an open source MapReduce implementation, as our distributed computing platform, Memcached as the public storage system for sharing data, ZooKeeper as a synchronous operation tool. We also adopt Magent mag as the management master of Memcached cluster and choose xmemcached xme, a java memcached client, to communicate with Magent. As the concept of iteration, we adopt the same definition with FPSGD.

5. Experiments

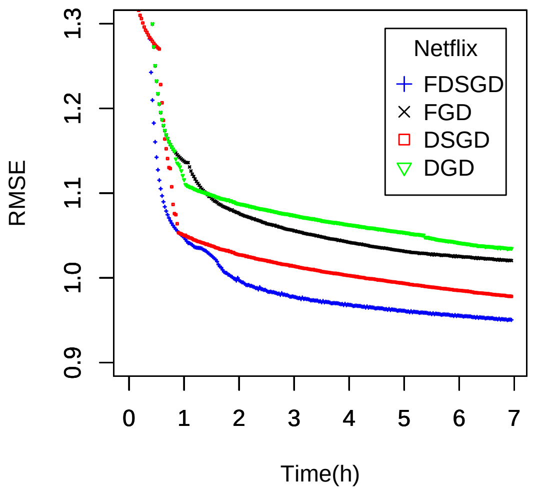
Gemulla et al. (2011) compared DSGD with other methods for solving matrix factorization problem, such as ALS, DGD and PSGD, and proved its better performance. Our experiments mainly focus on the comparison between FDSGD and DSGD and based on FDSGD and DSGD, we also implement FDGD and DGD which apply gradient descent algorithm to update factor matrices and parameters. All of these algorithms are implemented on Hadoop.

5.1. Comparison Between FDSGD and DSGD

As we mentioned in section 2.1, DSGD relates to data synchronous operation after a rule being processed, which causes all of the threads must wait for the slowest thread before moving to process the next rule. With Memcached and Zookeeper, FDSGD solves the data sharing and operation synchronization problems so that there is no data synchronization that DSGD has to confront. We adopt RMSE of the test sets to evaluate the performance of FDSGD.



(a) A comparison among FDSGD, FDGD, DSGD and DGD on MovieLens dataset



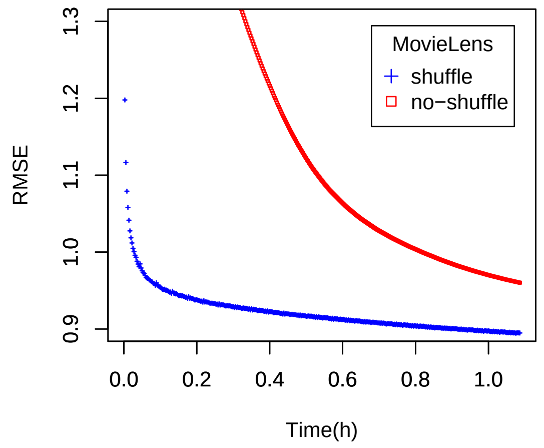
(b) A comparison among FDSGD, FDGD, DSGD and DGD on Netflix dataset

Figure 3: Performance Comparison of Algorithms.

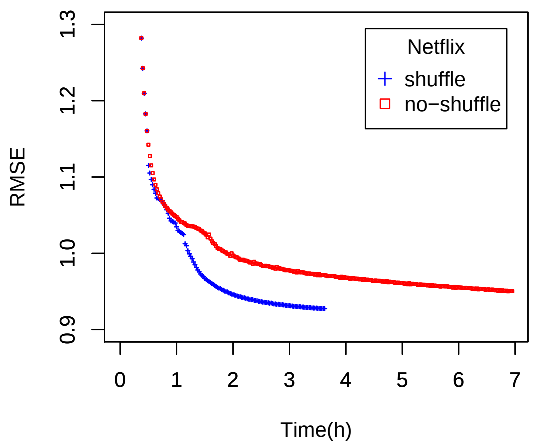
Figure 3 presents the performance of our algorithms on MovieLens and Netflix datasets. Apparently, FDSGD has better speed of convergence than DSGD without the process of data synchronization and algorithms based on SGD get better performance than that based on GD.

5.2. Influence of Shuffling

For a parallel algorithm, in order to achieve the maximum performance, it should ensure that all threads are in busy state as far as possible. However, threads needing cooperation often have to confront locking problem because of different running time.



(a) A comparison between no-shuffle and shuffle operation on MovieLens dataset



(b) A comparison between no-shuffle and shuffle operation on Netflix dataset

Figure 4: Influence of shuffling operation.

In order to balance the size of each block, a simple way is to shuffle the rating matrix randomly. We run FDSGD on no-shuffle and shuffle datasets of MovieLens and Netflix to verify the influence of shuffling to performance. Through Figure 4, we figure out that FDSGD gets better speed of convergence after shuffling operation. Because of the shuffling operation, rating matrix is gridded into blocks evenly and it balances the time of each block being executed on SGD algorithm.

6. Conclusion

We implement a fast distributed stochastic gradient descent algorithm named FDSGD running in DCE for matrix factorization. It provides a method to solve matrix factorization of web-scale. In order to improve the performance of our algorithm, we plan to adopt a changeable learning rate based on the variation of RMSE and to investigate some more effective shuffling methods. In our experiment, we figure out that Memcached is not stable as we expected and it has the limitation that the maximum size of its storing value object is 1M. So we plan to investigate a more stable and flexible storage system for sharing data, such as Redis red.

Acknowledgments

This work is supported by the National Key Basic Research and Department(973) Program of China (No.2013CB329603), the National Science Foundation of China (Nos.61375058, and 71231002), China Mobile Research Fund (MCM 20130351), the Ministry of Education of China and the Special Co-construction Project of Beijing Municipal Commission of Education.

References

An open-source software for reliable, scalable and distributed computing. Project URL: http://hadoop.apache.org/.

A simple but useful proxy program for memcached servers. Project URL: http://code.google.com/p/memagent/.

An unpublished paper named A Dynamic Caching Mechanism for Hadoop using Memcached. URL: https://wiki.engr.illinois.edu/download/attachments/197297260/ClouData-3rd.pdf?version=1&modificationDate=1336549179000.

A free & open source, high-performance, distributed memory object caching system. Project URL: http://www.memcached.org/.

An open source, BSD licensed, advanced key-value store. Project URL: http://redis.io/.

A high performance, easy to use multithreaded memcached client in java. Project URL: http://code.google.com/p/xmemcached/.

Provide a fast, highly available, fault tolerant, and distributed coordination service. Project URL: http://zookeeper.apache.org/.

Robert M Bell and Yehuda Koren. Lessons from the netflix prize challenge. *ACM SIGKDD Explorations Newsletter*, 9(2):75–79, 2007.

Gideon Dror, Noam Koenigstein, Yehuda Koren, and Markus Weimer. The yahoo! music dataset and kdd-cup'11.

Brad Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004(124):5, 2004.

Rainer Gemulla, Erik Nijkamp, Peter J Haas, and Yannis Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 69–77. ACM, 2011.

Patrick Hunt, Mahadev Konar, Flavio P Junqueira, and Benjamin Reed. Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, volume 8, pages 11–11, 2010.

Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, 2009.

Feng Niu, Benjamin Recht, Christopher Ré, and Stephen J Wright. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. *Advances in Neural Information Processing Systems*, 24:693–701, 2011.

István Pilászy, Dávid Zibriczky, and Domonkos Tikk. Fast als-based matrix factorization for explicit and implicit feedback datasets. In *Proceedings of the fourth ACM conference on Recommender systems*, pages 71–78. ACM, 2010.

Hsiang-Fu Yu, Cho-Jui Hsieh, Si Si, and Inderjit S Dhillon. Scalable coordinate descent approaches to parallel matrix factorization for recommender systems. In *ICDM*, pages 765–774, 2012.

Yunhong Zhou, Dennis Wilkinson, Robert Schreiber, and Rong Pan. Large-scale parallel collaborative filtering for the netflix prize. In *Algorithmic Aspects in Information and Management*, pages 337–348. Springer, 2008.

Yong Zhuang, Wei-Sheng Chin, Yu-Chin Juan, and Chih-Jen Lin. A fast parallel sgd for matrix factorization in shared memory systems. In *Proceedings of the 7th ACM conference on Recommender systems*, pages 249–256. ACM, 2013.

Martin Zinkevich, Markus Weimer, Alexander J Smola, and Lihong Li. Parallelized stochastic gradient descent. In *NIPS*, volume 4, page 4, 2010.