國立臺灣大學電機資訊學院資訊工程學研究所

碩士論文

Department of Computer Science and Information Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Master Thesis

在共享記憶體系統的快速平行隨機梯度下降法矩陣分解

A Fast Parallel Stochastic Gradient Method for Matrix Factorization in Shared Memory Systems

阮毓欽

Yu-Chin Juan

指導教授：林智仁 博士

Advisor: Chih-Jen Lin, Ph.D.

中華民國 103 年 7 月

July, 2014

國立臺灣大學碩士學位論文

口試委員會審定書

在共享記憶體系統的快速平行隨機梯度下降法矩陣分解

A Fast Parallel Stochastic Gradient Method for Matrix Factorization in Shared Memory Systems

本論文係阮毓欽君（學號 R01922136）在國立臺灣大學資訊工程學系完成之碩士學位論文，於民國 103 年 7 月 11 日承下列考試委員審查通過及口試及格，特此證明

口試委員：

林智仁

（指導教授）

李育杰　　林軒田

系 主 任　許永真

中文摘要

在推薦系統上，矩陣分解是一個非常有效的技術。對於矩陣分解問題，隨機梯度下降法是一個高效的演算法。然而，這個演算法並不容易被平行。這篇論文，在共享記憶體系統中，我們開發一個新的平行演算法叫做FPSG。藉由解決負載不平衡問題及快取失效問題，我們開發的平行演算法比現有的平行演算法更加有效。

關鍵詞: 推薦系統，矩陣分解，隨機梯度下降法，平行計算，共享記憶體演算法。

ABSTRACT

Matrix factorization is known to be an effective method for recommender systems that are given only the ratings from users to items. Currently, stochastic gradient (SG) method is one of the most popular algorithms for matrix factorization. However, as a sequential approach, SG is difficult to be parallelized for handling web-scale problems. In this thesis, we develop a fast parallel SG method, FPSG, for shared memory systems. By dramatically reducing the cache-miss rate and carefully addressing the load balance of threads, FPSG is more efficient than state-of-the-art parallel algorithms for matrix factorization.

KEYWORDS: Recommender system, Matrix factorization, Stochastic gradient descent, Parallel computing, Shared memory algorithm.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER I

Introduction

Many customers are overwhelmed with the choices of products in the e-commerce activities. For example, Yahoo!Music and GrooveShark provide a huge number of songs for on-line audiences. An important problem is how to let users efficiently find items meeting their needs. Recommender systems have been constructed for such a purpose. As demonstrated in KDD Cup 2011 (Dror et al., 2012) and Netflix competition (Bell and Koren, 2007), a collaborative filter using latent factors has been considered as one of the best models for recommender systems. This approach maps both users and items into a latent feature space. A latent factor, though not directly measurable, often contains some useful abstract information. The affinity between a user and an item is defined by the inner product of their latent-factor vectors. More specifically, given m users, n items, and a rating matrix R that encodes the preference of the uth user on the vth item at the (u,v) entry, $r_{u,v}$, matrix factorization (Koren et al., 2009) is a technique to find two dense factor matrices $P \in \mathbb{R}^{k\times m}$ and $Q \in \mathbb{R}^{k\times n}$ such that $r_{u,v} \simeq \boldsymbol{p}_u^T \boldsymbol{q}_v$, where k is the pre-specified number of latent factors, and $\boldsymbol{p}_u \in \mathbb{R}^k$ and $\boldsymbol{q}_v \in \mathbb{R}^k$ are respectively the uth column of P and the vth column of Q. The optimization problem is

$$\min_{P,Q} \sum_{(u,v)\in R} \Big((r_{u,v} - \boldsymbol{p}_u^T \boldsymbol{q}_v)^2 + \lambda_P \left\|\boldsymbol{p}_u\right\|^2 + \lambda_Q \left\|\boldsymbol{q}_v\right\|^2 \Big), \tag{1.1}$$

where $\|\cdot\|$ is the Euclidean norm, $(u, v) \in R$ indicates that rating $r_{u,v}$ is available, λ_P and λ_Q are regularization coefficients for avoiding over-fitting.[1] Because $\sum_{(u,v)\in R}\left(r_{u,v} - \boldsymbol{p}_u^T\boldsymbol{q}_v\right)^2$ is a non-convex function of P and Q, (1.1) is a difficult optimization problem. Many past studies have proposed optimization methods to solve (1.1), e.g., (Koren et al., 2009; Pilászy et al., 2010; Zhou et al., 2008). Among them, stochastic gradient (SG) is popularly used. For example, all of the top three teams in KDD Cup 2011 (track 1) employed SG in their winning approaches.

The basic idea of SG is that, instead of expensively calculating the gradient of (1.1), it randomly selects a (u,v) entry from the summation and calculates the corresponding gradient (Kiefer and Wolfowitz, 1952; Robbins and Monro, 1951). Once $r_{u,v}$ is chosen, the objective function in (1.1) is reduced to

$$\left(r_{u,v} - \boldsymbol{p}_u^T\boldsymbol{q}_v\right)^2 + \lambda_P\boldsymbol{p}_u^T\boldsymbol{p}_u + \lambda_Q\boldsymbol{q}_v^T\boldsymbol{q}_v.$$

After calculating the sub-gradient over $\boldsymbol{p}_u$ and $\boldsymbol{q}_v$, variables are updated by the following rules

$$\boldsymbol{p}_u \leftarrow \boldsymbol{p}_u + \gamma\left(e_{u,v}\boldsymbol{q}_v - \lambda_P\boldsymbol{p}_u\right), \tag{1.2}$$

$$\boldsymbol{q}_v \leftarrow \boldsymbol{q}_v + \gamma\left(e_{u,v}\boldsymbol{p}_u - \lambda_Q\boldsymbol{q}_v\right), \tag{1.3}$$

where

$$e_{u,v} = r_{u,v} - \boldsymbol{p}_u^T\boldsymbol{q}_v$$

is the error between the real and predicted ratings for the (u,v) entry, and γ is the learning rate. The overall procedure of SG is to iteratively select an instance $r_{u,v}$, apply update rules (1.2)-(1.3), and may adjust the learning rate.

[1]The regularization terms can be rewritten in an alternative form, $\sum_{u,v}\lambda_P\|\boldsymbol{p}_u\|^2 = \lambda_P\sum_{u=1}^m|\Omega_u|\,\|\boldsymbol{p}_u\|^2$ and $\sum_{u,v}\lambda_Q\|\boldsymbol{q}_v\|^2 = \lambda_Q\sum_{v=1}^n|\bar{\Omega}_v|\,\|\boldsymbol{q}_v\|^2$, where $|\Omega_u|$ and $|\bar{\Omega}_v|$ indicate the number of non-zero ratings associated with the uth user and the vth item, respectively.

Although SG has been successfully applied to matrix factorization, it is not applicable to handle large-scale data. The iterative process of applying (1.2)-(1.3) is inherently sequential, so it is difficult to parallelize SG under advanced architectures such as GPU, multi-core CPU or distributed clusters. Several parallel SG approaches have been proposed (e.g., (Gemulla et al., 2011; Hall et al., 2010; Mann et al., 2009; McDonald et al., 2010; Niu et al., 2011; Zinkevich et al., 2010)), although their focuses may be on other machine learning techniques rather than matrix factorization. In this work, we aim at developing an effective parallel SG method for matrix factorization in a shared memory environment. Although for huge data a distributed system must be used, in many situations running SG on a data set that can fit in memory is still very time consuming. For example, the size of the KDD Cup 2011 data is less than 4GB and can be easily stored in the memory of one computer, but a single SG iteration of implementing (1.2)-(1.3) takes more than 30 seconds. The overall SG procedure may take hours. Therefore, an efficient parallel SG to fully take the power of multi-core CPU can be very useful in practice.

Among existing parallel-SG methods for matrix factorization, some are directly designed or can be adapted for shared-memory systems. We briefly discuss two state-of-the-art methods because our method will improve upon them. HogWild (Niu et al., 2011) randomly selects a subset of $r_{u,v}$ instances and apply rules (1.2)-(1.3) in all available threads simultaneously without synchronization between threads. The reason why they can drop the synchronization is that their algorithm guarantees the convergence when factorizing a highly sparse matrix with the rare existence of the over-writing problem where different threads access the same data or variables such as $r_{u,v}$, $\boldsymbol{p}_u$ and $\boldsymbol{q}_v$ at the same time. That is, one thread is allowed to over-write another's work. DSGD (Gemulla et al., 2011) is another popular parallel SG approach although it is mainly designed for cluster environments. Given s computation nodes and a rating matrix R, DSGD uniformly grids R into s by s blocks first. Then DSGD assigns s different blocks to the s nodes. On each node, DSGD performs (1.2)-(1.3) on all ratings of the block in a random order. As expected, DSGD can be adapted for shared-memory systems if we replace a computational node with a thread.

In this thesis, we point out that existing parallel SG methods may suffer from the following issues when they are applied in a shared-memory system.

- Data discontinuity: the algorithm may randomly access data or variables so that a high cache-miss rate is endured.
- Block imbalance: for approaches that split data to blocks and utilize them in parallel, cores/CPUs for sparser blocks (i.e., a block contains fewer ratings) must wait for those assigned to denser blocks.

Our main contribution is to design an effective method to alleviate these issues. This thesis is organized as follows. We give details of HogWild and DSGD in Chapter II. Another parallel matrix factorization method CCD++ is also discussed in this chapter. Then Chapter III discusses difficulties in parallelizing SG for matrix factorization. Our proposed method FPSG (Fast Parallel SG) is introduced in Chapter IV. We compare our method with state-of-the-art algorithms using root mean square error (RMSE) as the evaluation measure in Chapter V. RMSE is defined as

$$\sqrt{\frac{1}{\text{number of ratings}} \sum_{(u,v)\in R} (r_{u,v} - \hat{r}_{u,v})^2}, \tag{1.4}$$

where R is the rating matrix of the test set and $\hat{r}_{u,v}$ is the predicted rating value. In Chapter VI, we discuss some miscellaneous issues related to our proposed approach. Finally, Chapter VII summarizes our work and gives future directions.

A preliminary version of this work appears in a conference thesis (Zhuang et al., 2013). The major extensions in this journal version include first we add more experiments to show the effectiveness of FPSG, second we compare the speedup of state-of-the-art methods, and third many detailed descriptions are now given.

CHAPTER II

Existing Parallelized Stochastic Gradient Descent Algorithms and Coordinate Descent Methods

Following the discussion in Chapter I, in this chapter, we present two parallel SG methods, HogWild (Niu et al., 2011) and DSGD (Gemulla et al., 2011), in detail. We also discuss a non-SG method CCD++ (Yu et al., 2012) because it is included for comparison in Chapter V. CCD++ is a parallel coordinate descent method that is considered state-of-the-art for matrix factorization.

2.1 HogWild

HogWild (Niu et al., 2011) assumes that the rating matrix is highly sparse and deduces that for two randomly sampled ratings, the two serial updates via (1.2)-(1.3) are likely to be independent. The reason is that the selected ratings to be updated almost never share the same user identity and item identity. Then, iterations of SG, (1.2)-(1.3), can be parallely executed in different threads. With the assumption of independent updates, HogWild does not synchronize the state of each thread for preventing concurrent variable access. Instead, HogWild employs atomic operations, each of which is a series of CPU instructions that can not be interrupted. Therefore, as a kind of asynchronous methods, HogWild saves the time for synchronization. Although the potential over-writing may occur (i.e., the ratings to be updated share the same

Algorithm 1 HogWild's Algorithm

```
Require: number of threads $s$, $R \in \mathbb{R}^{m\times n}$, $P \in \mathbb{R}^{k\times m}$, and $Q \in \mathbb{R}^{k\times n}$
1: for each thread $i$ parallelly do
2:     while true do
3:         randomly select an instance $r_{u,v}$ from $R$
4:         update corresponding $\boldsymbol{p}_u$ and $\boldsymbol{q}_v$ using (1.2)-(1.3), respectively
5:     end while
6: end for
```

user identity or item identity), Niu et al. (2011) prove the convergence under some assumptions such as the rating matrix is very sparse.

Algorithm 1 shows the whole process of HogWild. We use Figure 2.1 to illustrate how two threads run SG updates simultaneously. The left matrix and the right matrix are the updating sequences of two threads, where black dots are ratings randomly selected by a thread and arrows indicate the order of processed ratings. The red dot, which is simultaneously accessed by two threads in their last iterations in Figure 2.1, indicates the occurrence of the over-writing problem. That is, two threads conduct SG updates using the same rating value $r_{i,j}$. From Algorithm 2.1, the operations include

- reading $r_{i,j}$, $\boldsymbol{p}_i$ and $\boldsymbol{q}_j$,
- evaluating the right-hand sides of (1.2)-(1.3), and
- assigning values to the left-hand sides of (1.2)-(1.3)

The second operation does not change shared variables because it is a series of arithmetic operations on local variables $r_{i,j}$, $\boldsymbol{p}_i$ and $\boldsymbol{q}_j$. However, for the first and the last operations, we use atomic instructions that are executed without considering the situation of other threads. All available threads would continuously execute the above-mentioned procedure until achieving the user-defined number of iterations.

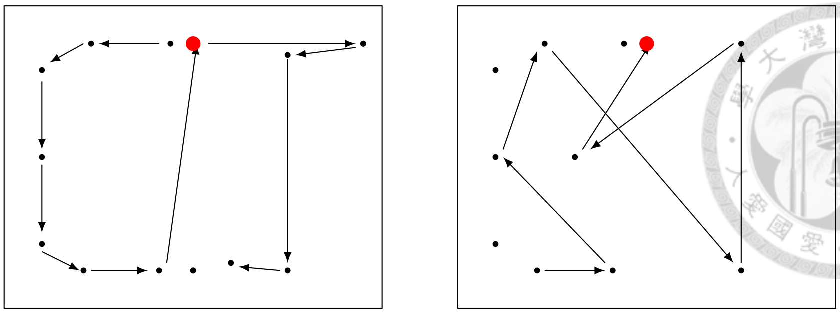


Figure 2.1: An example shows updating sequences of two threads in HogWild.

2.2 DSGD

Although SG is a sequential process, DSGD (Gemulla et al., 2011) takes the property that some blocks of the rating matrix are mutually independent and their corresponding variables can be updated in parallel. DSGD uniformly grids the rating matrix R into many sub-matrices (also called blocks), and applies SG to some independent blocks simultaneously. In the following discussion, we say two blocks are independent to each other if they share neither any common column nor any common row of the rating matrix. For example, in Figure 2.2, the six patterns of gray blocks in R cover all possible patterns of independent blocks. Note that Gemulla et al. (2011) restrict the number of blocks in each patten to be s, the number of available computational nodes, for reducing the data communication in distributed systems; see also the explanation below.

The overall algorithm of DSGD is shown in Algorithm 2, where T is the maximal number of iterations. In line 2, R is grided into $s \times s$ uniform blocks, and the intermediate for-loop continuously assigns s independent blocks to computation nodes until all blocks in R have been processed once. The bth iteration of the innermost for-loop updates P and Q by performing SG on ratings in the block b. Given a 4-by-4 divided rating matrix and 4 threads as an example in Figure 2.3a, we show two consecutive iterations of the innermost for-loop in Figure 2.3b. The left iteration assigns 4 diagonal

Algorithm 2 DSGD's Algorithm

```
Require: number of threads $s$, maximum iterations $T$, $R \in \mathbb{R}^{m \times n}$, $P \in \mathbb{R}^{k \times m}$, and
    $Q \in \mathbb{R}^{k \times n}$
 1: grid $R$ into $s \times s$ blocks $B$ and generate $s$ patterns covering all blocks
 2: for $t = \{1, \dots, T\}$ do
 3:     Decide the order of $s$ patterns sequentially or by random permutation
 4:     for each pattern of $s$ independent blocks of $B$ do
 5:         assign $s$ selected blocks to $s$ threads
 6:         for $b = \{1, \dots, s\}$ parallellly do
 7:             randomly sample ratings from block $b$
 8:             apply (1.2)-(1.3) on all sampled ratings
 9:         end for
10:     end for
11: end for
```

blocks to 4 nodes (i_0, i_1, i_2, i_3); node i_0 updates p_0 and q_0, node i_1 updates p_1 and q_1, and so on. In the next (right) iteration, each node updates the same segment of P, but for Q, q_1, q_2, q_3 and q_0 are respectively updated by nodes i_0, i_1, i_2 and i_3. This example shows that we can keep p_k in node i_k to avoid the communication of P. However, nodes must exchange their segments of Q, which are alternatively updated by different nodes in different iterations. For example, from Figure 2.3a to Figure 2.3b, node i_0 must send node i_3 the segment q_0 after finishing its computation. Consequently, the total amount of data transferred in one iteration of the intermediate loop is the size of Q because each of s nodes sends $|Q|/s$ and receives $|Q|/s$ entries of Q from another node, where $|Q|$ is the total number of entries in Q.

2.3 CCD++

CCD++ (Yu et al., 2012) is a parallel method for matrix factorization in both shared-memory and distributed environments. Based on the concept of a coordinate descent method, CCD++ sequentially updates one row of P and one row of Q corre-

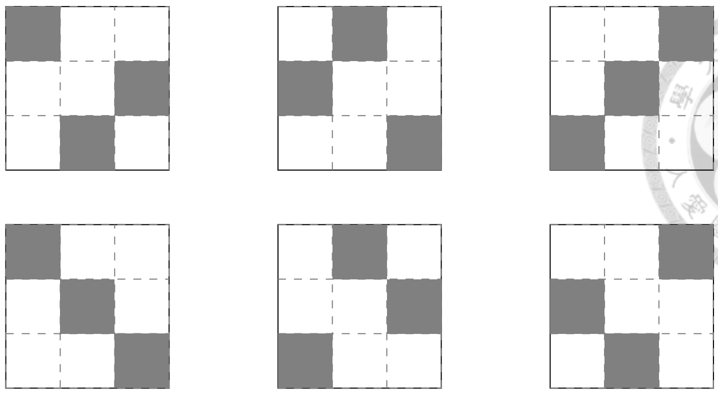


Figure 2.2: Patterns of independent blocks for a 3 by 3 grided matrix.

sponding to the same latent dimension while fixing other variables. Let

$$\hat{\boldsymbol{p}}_1, \ldots, \hat{\boldsymbol{p}}_k \text{ be } P\text{'s rows and}$$

$$\hat{\boldsymbol{q}}_1, \ldots, \hat{\boldsymbol{q}}_k \text{ be } Q\text{'s rows.}$$

CCD++ cyclically updates $(\hat{\boldsymbol{p}}_1, \hat{\boldsymbol{q}}_1)$ until $(\hat{\boldsymbol{p}}_k, \hat{\boldsymbol{q}}_k)$. Let $(\hat{\boldsymbol{p}}, \hat{\boldsymbol{q}})$ be the current values of the selected row and denote $(\boldsymbol{w}, \boldsymbol{h})$ as the corresponding variables to be determined. Because other rows are fixed, the objective function in (1.1) can be converted to

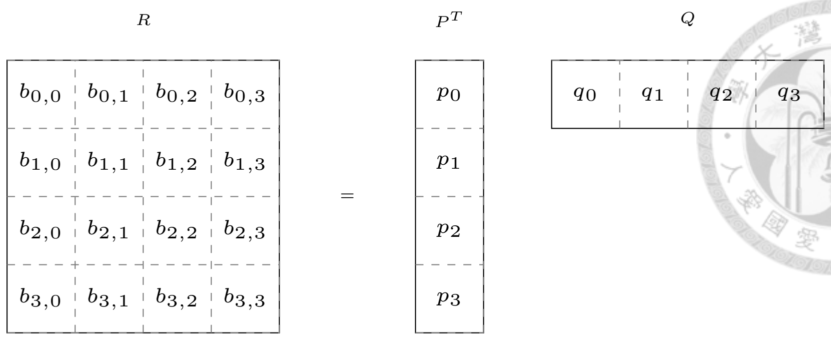
$$\begin{aligned}&\sum_{(u,v)\in R} \left(r_{u,v} - \boldsymbol{p}_u^T \boldsymbol{q}_v + \hat{p}_u \hat{q}_v - w_u h_v\right)^2 + \lambda_P \left(\sum_{u=1}^m \|\boldsymbol{p}_u\|^2 - \sum_{u=1}^m \hat{p}_u^2 + \sum_{u=1}^m w_u^2\right) \\ &+ \lambda_Q \left(\sum_{v=1}^n \|\boldsymbol{q}_v\|^2 - \sum_{v=1}^n \hat{q}_v^2 + \sum_{v=1}^n h_v^2\right)\end{aligned} \tag{2.1}$$

or

$$\sum_{(u,v)\in R} (e_{u,v} + \hat{p}_u \hat{q}_v - w_u h_v)^2 + \lambda_P \sum_{u=1}^m w_u^2 + \lambda_Q \sum_{v=1}^n h_v^2 \tag{2.2}$$

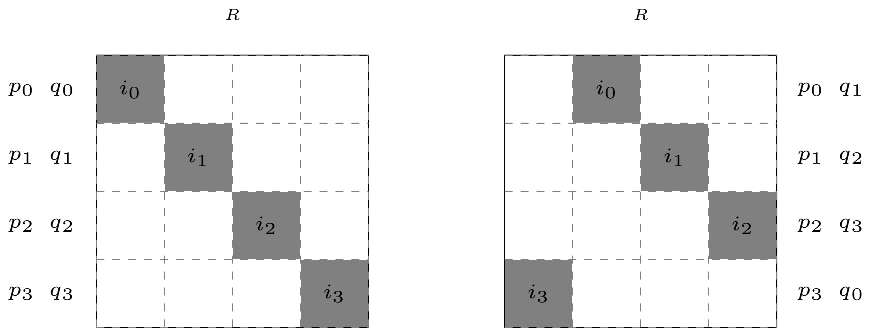
by dropping terms that do not depend on $\boldsymbol{w}$ or $\boldsymbol{h}$. If $\boldsymbol{w}$ (or $\boldsymbol{h}$) is fixed, the minimization of (2.2) becomes a least square problem. Yu et al. (2012) alternatively update $\boldsymbol{w}$ and $\boldsymbol{h}$ several times (called inner iterations in CCD++). In the case where $\boldsymbol{h}$ is fixed as the current $\hat{\boldsymbol{q}}$, (2.2) becomes

$$\sum_{u=1}^m \left(\sum_{v:\,(u,v)\in R} (e_{u,v} + \hat{p}_u \hat{q}_v - w_u \hat{q}_v)^2 + \lambda_P w_u^2 \right) + \text{constant.} \tag{2.3}$$



R
P^T
Q
$b_{0,0}$ $b_{0,1}$ $b_{0,2}$ $b_{0,3}$
$b_{1,0}$ $b_{1,1}$ $b_{1,2}$ $b_{1,3}$
$b_{2,0}$ $b_{2,1}$ $b_{2,2}$ $b_{2,3}$
$b_{3,0}$ $b_{3,1}$ $b_{3,2}$ $b_{3,3}$
=
p_0 p_1 p_2 p_3
q_0 q_1 q_2 q_3

(a) 4 by 4 grided rating matrix R and corresponding segments of P and Q. Note that p_i is the ith segment of P and q_j is the jth segment of Q.



(b) An example of two consecutive iterations (the left is before the right) of the innermost for-loop of Algorithm 2. Each iteration considers a set of 4 independent blocks.

Figure 2.3: An illustration of the DSGD algorithm.

It can be decomposed into m independent problems

$$\min_{w_u} \sum_{v:\,(u,v)\in R} (e_{u,v} + \hat{p}_u\hat{q}_v - w_u\hat{q}_v)^2 + \lambda_P w_u^2, \quad \forall u = 1,\ldots,m. \tag{2.4}$$

Each involves a quadratic function of a single variable, so a closed-form solution exists. Then for any u, $e_{u,v}$ can be updated by

$$e_{u,v} \leftarrow e_{u,v} + (\hat{p}_u - w_u)\hat{q}_v, \quad \forall v \text{ with } (u,v) \in R.$$

Similarly, by fixing $\boldsymbol{w}$, we solve the following n independent problems to find $\boldsymbol{h}$ for updating $\hat{\boldsymbol{q}}$.

$$\min_{h_v} \sum_{u:\,(u,v)\in R} (e_{u,v} + \hat{p}_u\hat{q}_v - \hat{p}_u h_v)^2 + \lambda_Q h_v^2, \quad \forall v = 1,\ldots,n. \tag{2.5}$$

The parallelism of CCD++ is achieved by solving those independent problems in (2.4) and (2.5) simultaneously. See Algorithm 3 for the whole procedure of CCD++.

Algorithm 3 CCD++'s Algorithm

```
Require: maximum outer iterations $T$, $R \in \mathbb{R}^{m \times n}$, $P \in \mathbb{R}^{k \times m}$, and $Q \in \mathbb{R}^{k \times n}$
 1: Initialize $P$ as a zero matrix
 2: Calculate rating error $e_{u,v} = r_{u,v}$ for all $(u, v) \in R$
 3: for $t = \{1, \ldots, T\}$ do
 4:     for $t_k = \{1, \ldots, k\}$ do
 5:         Let $\hat{\boldsymbol{p}}$ and $\hat{\boldsymbol{q}}$ be the $t_k$th row of $P$ and $Q$, respectively.
 6:         for $u = \{1, \ldots, m\}$ parallely do
 7:             Solve (2.4) under the given $u$, and then update $\hat{p}_u$ and $e_{u,v}$, $\forall v$ with $(u, v) \in R$
 8:         end for
 9:         for $v = \{1, \ldots, n\}$ parallely do
10:             Solve (2.5) under the given $v$, and then update $\hat{q}_v$ and $e_{u,v}$, $\forall u$ with $(u, v) \in R$
11:         end for
12:         Copy $\hat{\boldsymbol{p}}$ and $\hat{\boldsymbol{q}}$ back to the $t_k$th row of $P$ and $Q$, respectively.
13:     end for
14: end for
```

CHAPTER III

Problems in Parallel SG Methods for Matrix Factorization

In this chapter, we point out that parallel SG methods discussed in Chapter II may suffer some problems when they are applied in a shared-memory environment. These problems are *locking problem* and *memory discontinuity*. We introduce what these problems are, and explain how they result in performance degradation.

3.1 Locking Problem

For a parallel algorithm, to maximize the performance, keeping all threads busy is important. The *locking problem* occurs if a thread idles because of waiting for other threads. In DSGD, if s threads are used, then according to Algorithm 2, s independent blocks are updated in a batch. However, if the running time for each block varies, then a thread that finishes its job earlier may need to wait for other threads.

The locking problem may be more serious if R is unbalanced. That is, available ratings are not uniformly distributed across all positions in R. In such a case, the thread updating a block with fewer ratings may need to wait for other threads. For example, in Figure 3.1, after all ratings in block $b_{1,1}$ have been processed, only one third of ratings in block $b_{0,0}$ have been handled. Hence the thread updating $b_{1,1}$ idles most of the time.

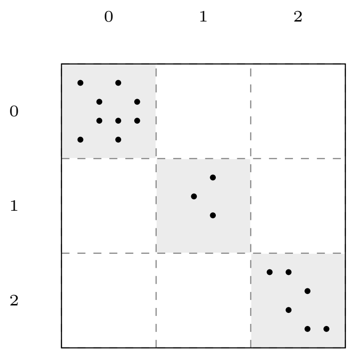


Figure 3.1: An example of the locking problem in DSGD. Each dot represents a rating; gray blocks indicate a set of independent blocks. Ratings in white blocks are not shown.

A simple method to make R more balanced is *random shuffling*, which randomly permutes user identities and item identities before processing. However, the amount of ratings in each block may still not be exactly the same. Further, even if each block contains the same amount of ratings, the computing time of each code can still be slightly different. Therefore, other techniques are needed to address the locking problem.

Interestingly, DSGD has a reason to ensure that s blocks are processed before moving to the next s. As mentioned in Chapter 2.2, it is designed for distributed systems, so minimizing the communication cost between computing nodes may be more important than reducing the idle time of nodes. However, in shared memory systems the locking problem becomes an important issue.

3.2 Memory Discontinuity

When a program accesses data in memory discontinuously, it suffers from a high cache-miss rate and performance degradation. Most SG solvers for matrix factorization including HogWild and DSGD randomly pick instances from R (or from a block of R) to be updated. We call this setting as the *random method*, which is illustrated in Figure 3.2. Though the random method generally enjoys good convergence, it suffers from

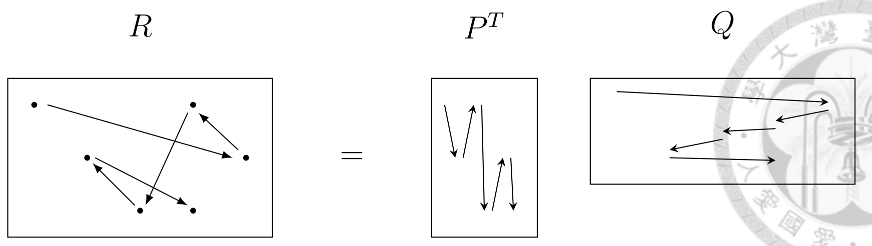


Figure 3.2: A random method to select rating instances for update.

the memory discontinuity seriously. The reason is that not only are rating instances randomly accessed, but also user/item identities become discontinuous.

The seriousness of the memory discontinuity varies in different methods. In HogWild, each thread randomly picks instances among R independently, so it suffers from memory discontinuity in R, P, and Q. In contrast, for DSGD, though ratings in a block are randomly selected, as we will see in Chapter 4.2, we can easily change the update order to mitigate the memory discontinuity.

CHAPTER IV

Our Approaches

In this thesis, we propose two techniques, *lock-free scheduling* and *partial random method*, to respectively solve the locking problem mentioned in Chapter 3.1 and the memory discontinuity mentioned in Chapter 3.2. We name the new parallel SG method as *fast parallel SG* (FPSG). In Chapter 4.1, we discuss how FPSG flexibly assigns blocks to threads to avoid the locking problem. In Chapter 4.2, we observe that a comprehensive random selection may not be necessary, and show that randomization can be applied only among blocks instead of within blocks to maintain both the memory continuity and the fast convergence. In Chapter 4.3, we overview the complete design of FPSG. Finally, in Chapter 4.4, we introduce our implementation techniques to accelerate the computation.

4.1 Lock-Free Scheduling

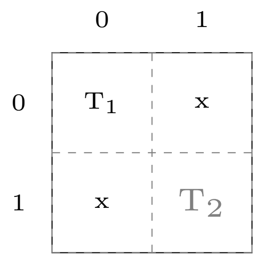
We follow DSGD to grid R into blocks and design a scheduler to keep s threads busy in running a set of independent blocks. For a block $b_{i,j}$, if it is independent from all blocks being processed, then we call it as a *free* block. Otherwise, it is a *non-free* block. When a thread finishes processing a block, the scheduler assigns a new block that meets the following two criteria:

1. It is a free block.

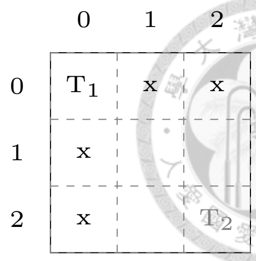
2. Its number of past updates is the smallest among all free blocks.

The number of updates of a block indicates how many times it has been processed. The second criterion is applied because we want to keep a similar number of updates for each block. If two or more blocks meet the above two criteria, then we randomly select one. Given s threads, we show that FPSG should grid R into at least $(s+1) \times (s+1)$ blocks. Take two threads as an example. Let T_1 be a thread that is updating certain block and T_2 be a thread that just finished updating a block and is getting a new job from the scheduler. If we grid R into 2×2 blocks shown in Figure 4.1a, then T_2 has only one choice: the block it just processed. A similar situation happens when T_1 gets its new job. Because T_1 and T_2 always process the same block, the remaining two blocks are never processed. In contrast, if we grid R into 3×3 blocks like Figure 4.1b, T_2 has three choices $b_{1,1}$, $b_{1,2}$ and $b_{2,1}$ when getting a new block.

As discussed above, because we can always assign a free block to a thread when it finishes updating the previous one, our scheduler does not suffer from the locking problem. However, for extremely unbalanced data sets, where most available ratings are in certain blocks, our scheduler is unable to keep the number of updates in all blocks balanced. In such a case blocks with many ratings are updated only very few times. A simple remedy is the random shuffling technique introduced in Chapter 3.1. In our experience, after random shuffling, the number of ratings in the heaviest block is smaller than twice of the lightest block. We then experimentally check how serious the imbalance problem is after random shuffling. Here we define *degree of imbalance* (DoI) to check the number of updates in all blocks. Let $\mathrm{UT}_M(t)$ and $\mathrm{UT}_m(t)$ be the maximal and the minimal numbers of updates in all blocks, respectively, where t is the iteration index. (FPSG does not have the concept of iterations. Here we call every



(a) 2×2 blocks



(b) 3×3 blocks

Figure 4.1: An illustration of how the split of R to blocks affects the job scheduling. T_1 is the thread that is updating block $b_{0,0}$. T_2 is the thread that is getting a new block from the scheduler. Blocks with "x" are dependent on block $b_{0,0}$, so they cannot be updated by T_2.

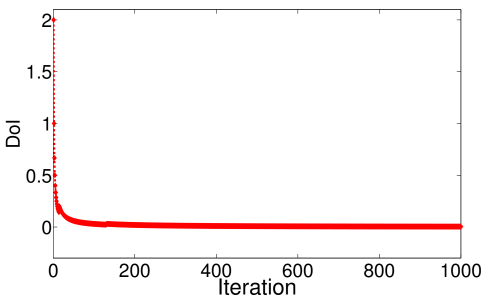
cycle of processing $(s+1)^2$ blocks as an iteration.) DoI is defined as

$$\text{DoI} = \frac{\text{UT}_M(t) - \text{UT}_m(t)}{t}.$$

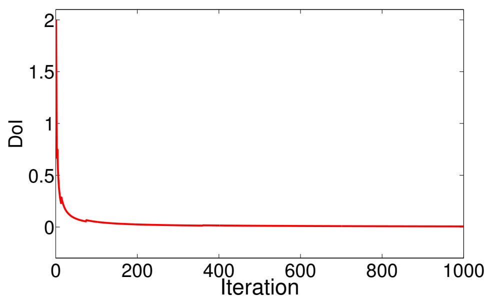
A small DoI indicates that the number of updates is similar across all blocks. In Figure 4.2, we show DoI for four different data sets. We can see that our scheduler reduces DoI to be close to zero in just a few iterations. For details of the data sets used in Figure 4.2, please refer to Chapter 5.1.

4.2 Partial Random Method

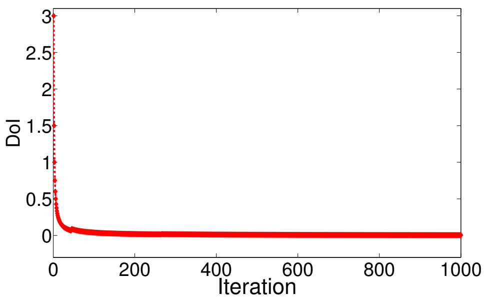
To achieve memory continuity, in contrast to the random method, we can consider an *ordered method* to sequentially select rating instances by user identities or item identities. Figure 4.3 gives an example of following the order of users. Then matrix P can be accessed continuously. Alternatively, if we follow the order of items, then the continuous access of Q can be achieved. For R, if the order of selecting rating instances is fixed, we can store R into memory with the same order to ensure its continuous access. Although the ordered method can access data in a more continuous manner, empirically we find that it is not stable. Figure 4.4 gives an example showing that under two slightly different learning rates for SG, the ordered method can be



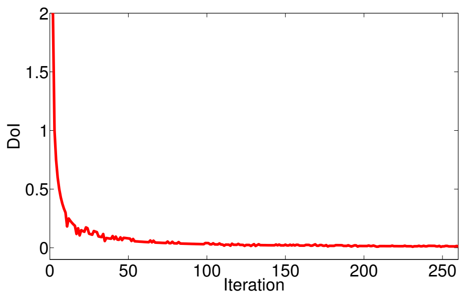
(a) MovieLens



(b) Netflix



(c) Yahoo!Music



(d) Hugewiki

Figure 4.2: DoI on four data sets. R is grided into 13×13 blocks after being randomly shuffled and 12 threads are used.

either much faster or much slower than the random method.

The above experiment indicates that a random access of data/variables may be useful for the convergence. This property has been observed in related optimization techniques. For example, in coordinate descent methods to solve some optimization problems, Chang et al. (2008) show that a random rather than a sequential order to update variables significantly improves the convergence speed. To compromise between data continuity and convergence speed, in FPSG, we propose a *partial random method*, which selects ratings in a block orderly but randomizes the selection of blocks. Although our scheduling is close to deterministic by choosing blocks with the smallest numbers of accesses, the randomness can be enhanced by griding R into more blocks. Then at any time point, some blocks have been processed by the same number of times, so the scheduler can randomly select one of them. Figure 4.5 illustrates how

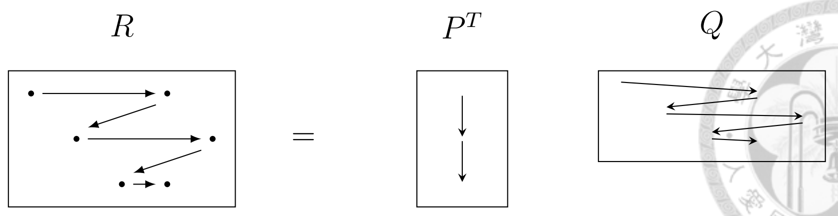
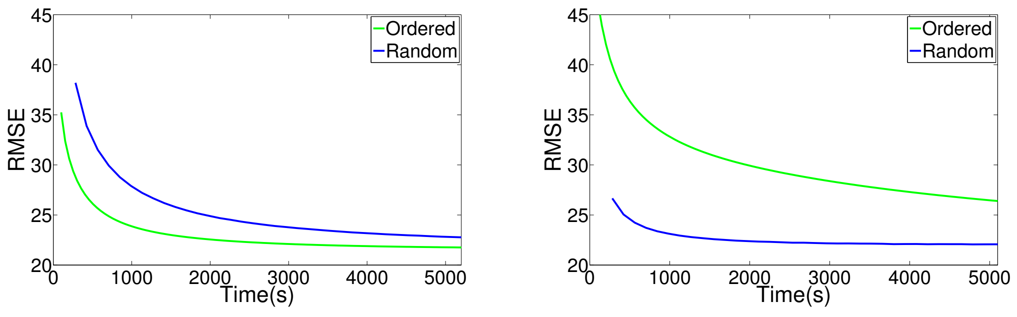


Figure 4.3: Ordered method to select rating instances for update.

the partial random method works using three threads. Figure 4.6 extends the running time comparison in Figure 4.4 to include FPSG. We can see that FPSG enjoys both fast convergence and excellent RMSE. Some related methods have been investigated in (Gemulla et al., 2011), although they showed that the convergence on the ordered method in terms of training loss is worse than the random method. Their observation is opposite to our experimental results. A possible reason is that we consider RMSE on the testing set while they consider the training loss.

Some subtle implementation details must be noted. We discussed in Chapter 4.1 that FPSG applies random shuffling to avoid the unbalanced number of updates of each block. However, after applying the random shuffling and griding R in to blocks, the ratings in each block are not sorted by user (or item) identities. To apply the partial random method we must sort user identities before processing each block because an ordered method is applied within the block. We give an illustration in Figure 4.7. In the beginning, we make the rating matrix more balanced by randomly shuffling all ratings; see the middle figure in Figure 4.7. However, user identities and item identities become not ordered, so we cannot achieve memory continuity by using the update strategy shown in Figure 4.3. Therefore, we must rearrange ratings in each block so that their row indices (i.e., user identities) are ordered; see the last figure in Figure 4.7.



(a) $\gamma = 0.0001$ (b) $\gamma = 0.0005$

Figure 4.4: A comparison between the random method and the ordered method using the Yahoo!Music data set. One thread is used.

Algorithm 4 The overall procedure of FPSG

```
1: randomly shuffle $R$
2: grid $R$ into a set $B$ with at least $(s + 1) \times (s + 1)$ blocks
3: sort each block by user (or item) identities
4: construct a scheduler
5: launch $s$ working threads
6: wait until the total number of updates reaches a user-defined value
```

4.3 Overview of FPSG

Algorithm 4 gives the overall procedure of FPSG. Based on the discussion in Chapter 4.1, FPSG first randomly shuffles R to avoid data imbalance. Then it grids R into at least $(s + 1) \times (s + 1)$ blocks and applies the partial random method discussed in Chapter 4.2 by sorting each block by user (or item) identities. Finally it constructs a scheduler and launches s working threads. After the required number of iterations is reached, it notifies the scheduler to stop all working threads. The pseudo code of the scheduler and each working thread are shown in Algorithm 5 and Algorithm 6, respectively. Each working thread continuously gets a block from the scheduler by invoking `get_job`, and the scheduler returns a block that meets criteria mentioned in Chapter 4.1. After a working thread gets a new block, it processes ratings in the block in an ordered manner (see Chapter 4.2). In the end, the thread invokes `put_job` of the scheduler to update the number of times that the block has been processed.

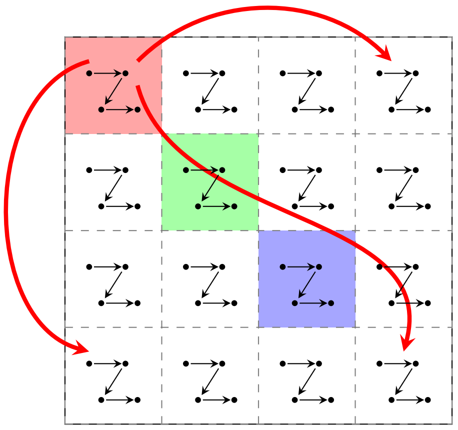
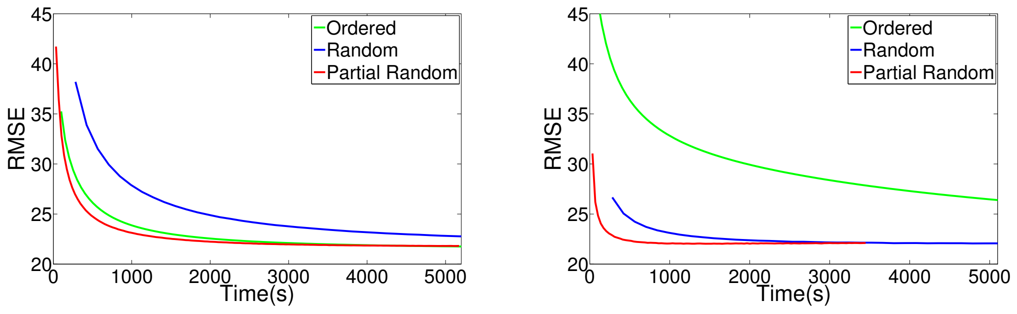


Figure 4.5: An illustration of the partial random method. Each color indicates the block being processed by a thread. Within each block, the update sequence is ordered like that in Figure 4.3. If block (1,1) is finished first, three candidates independent of the two other running blocks (2,2) and (3,3) are (1,4), (4,1), and (4,4), which are indicated by red arrows. If these three candidates have been accessed by the same number of times, then one is randomly chosen. This example explains how we achieve the random order of blocks.

4.4 Implementation Issues

FPSG uses the standard thread class in C++ implemented by pthread to do the parallelization. For the data set Yahoo!Music of about 250M ratings, using a typical machine (details specified in Chapter 5.1), FPSG finishes processing all ratings once in 6 seconds and takes only about 8 minutes to converge to a reasonable RMSE. Here we describe some techniques employed in our implementation. First, empirically we find that using single-precision floating-point computation does not suffer from numerical error accumulation. For the data set Netflix, using single precision runs 1.1 times faster then using double precision. Second, modern CPU provides SSE instructions that can concurrently run floating-point multiplications and additions. We apply SSE instructions for vector inner products and additions. For Yahoo!Music data set, the speed up is 2.4 times. Figure 4.8[1] shows the speedup after these techniques are applied in two data sets.

[1]For experimental settings, see Chapter 5.1.



(a) $\gamma = 0.0001$ (b) $\gamma = 0.0005$

Figure 4.6: A comparison between the ordered method, the random method, and the partial random method on the set Yahoo!Music. One thread is used.

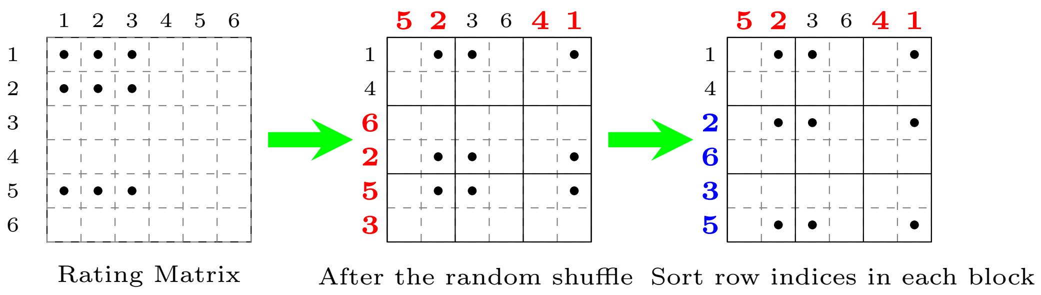
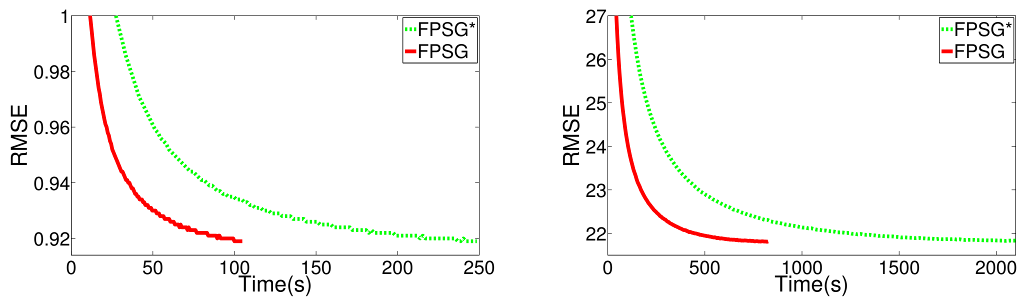


Figure 4.7: An illustration of the partial random method. After the random shuffle of data, some indices (in red color) are not ordered within each block. We make the row indices ordered (in blue color) by a sorting procedure.



(a) Netflix (b) Yahoo!Music

Figure 4.8: A comparison between two implementations of FPSG in Netflix and Yahoo!Music. FPSG implements the two techniques discussed in Chapter 4.4, while FPSG* does not.

Algorithm 5 Scheduler of FPSG

```
1: procedure GET_JOB
2:     Initialize an empty list $\boldsymbol{b}$ and $\boldsymbol{b}.ut_{min} = \infty$        $\triangleright$ $ut$: number of updates
3:     for all $b$ in $B$ do
4:         if $b$ is non-free then
5:             continue
6:         else
7:             if $b.ut == \boldsymbol{b}.ut_{min}$ then
8:                 Add $b$ into $\boldsymbol{b}$
9:             else if $b.ut < \boldsymbol{b}.ut_{min}$ then
10:                $\boldsymbol{b}.ut_{min} = b.ut$
11:                Make $\boldsymbol{b}$ empty and add $b$ into $\boldsymbol{b}$
12:            end if
13:        end if
14:    end for
15:    Randomly select an element denoted by $b_x$ from $\boldsymbol{b}$
16:    return $b_x$
17: end procedure
18: procedure PUT_JOB($b$)
19:    $b.ut = b.ut + 1$
20: end procedure
```

Algorithm 6 Working thread of FPSG

```
1: while true do
2:     get a block $b$ from scheduler $\rightarrow$ get_job()
3:     process elements orderly in this block
4:     scheduler $\rightarrow$ put_job($b$)
5: end while
```

CHAPTER V

Experiments

In this chapter, we provide the details about our experimental settings, and compare FPSG with other parallel matrix factorization algorithms mentioned in Chapter II.

5.1 Settings

Data Sets: Four data sets, MovieLens,[1] Netflix, Yahoo!Music, and Hugewiki,[2] are used for the experiments. For reproducibility, we consider the original training/test sets in our experiments if they are available (for MovieLens, we use Part B of the original data set generated by the official script). Because the test set of Yahoo!Music is not available, we consider the last four ratings of each user for testing, while the remaining ratings for training set. The data set Hugewiki is too large to fit in our machines, so we sample one quarter of the data randomly, and split them into training/test sets. The statistics of each data set is in Table 5.1.

Platform: We use a server with two Intel Xeon E5-2620 2.0GHz processors and 64 GB memory. There are six cores in each processor.

Parameters: Table 5.1 lists the parameters used for each data set. The parameters k, λ_P, λ_Q may be chosen by a validation procedure although here we mainly borrow

[1]`http://www.grouplens.org/node/73`
[2]`http://graphlab.org/downloads/datasets/`

Data Set	MovieLens	Netflix	Yahoo!Music	Hugewiki
m	71,567	2,649,429	1,000,990	39,706
n	65,133	17,770	624,961	25,034,863
#Training	9,301,274	99,072,112	252,800,275	761,429,411
#Test	698,780	1,408,395	4,003,960	100,000,000
k	40	40	100	100
λ_P	0.05	0.05	1	0.01
λ_Q	0.05	0.05	1	0.01
γ	0.003	0.002	0.0001	0.004

Table 5.1: The statistics and parameters for each data set. Note that the Hugewiki set used here contains only one quarter of the original set.

values from earlier works to obtain comparable results. For Netflix and Yahoo!Music, we use the parameters in (Yu et al., 2012); see values listed in Table 5.1. Although (Yu et al., 2012) have considered MovieLens, we use a different setting of $\lambda_P = \lambda_Q = 0.05$ for a better RMSE. For Hugewiki, we consider the same parameters as in (Yun et al., 2014). The initial values of P and Q are chosen randomly under a uniform distribution. This setting is the same as that in (Yu et al., 2012). The learning rate is determined by an ad hoc parameter selection. Because we focus on the running speed rather than RMSE in this thesis, we do not apply an adaptive learning rate.

In our platform, 12 physical cores are available, so we use 12 threads in all experiments. For FPSG, even though Chapter IV shows that $(s + 1) \times (s + 1)$ blocks are already enough for s threads, we use more blocks to ensure the randomness of blocks that are simultaneously processed. For Netflix, Yahoo!Music and Hugewiki, R is grided into 32×32 blocks; for MovieLens, R is grided into 16×16 blocks because the number of non-zeros is smaller.

Evaluation: As most recommender systems do, the metric adopted as our evaluation is RMSE on the test set, which is disjoint with the training set; see Eq. (1.4). In addition, the time in each figure refers to the training time.

Implementations: Among methods included for comparison, HogWild[3] and CCD++[4] are publicly available. We reimplement HogWild under the same framework of our FPSG and DSGD implementations for a fairer comparison. In the official HogWild package, the formulation includes the average value of training ratings. After trying different settings, the program still fails to converge. Therefore, we present only results of our HogWild implementation in the experiments.

The publicly available CCD++ code uses double precision. Because ours uses single precision following the discussion in Chapter 4.4, for a fair comparison, we obtain a singles-precision version of CCD++ from its authors. Note that OpenMP[5] is used in their implementation.

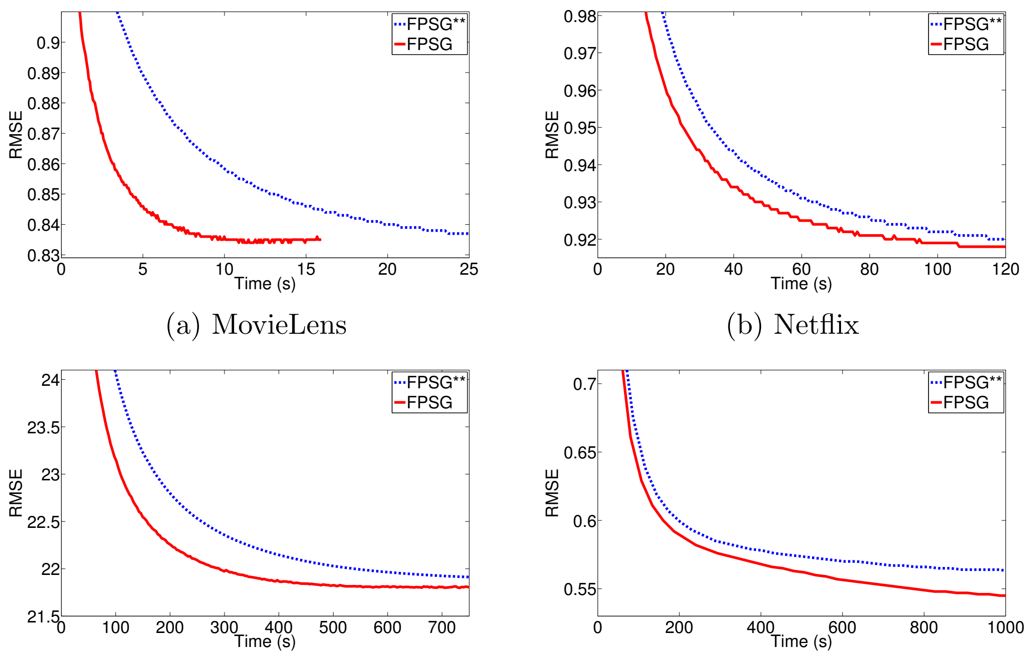
5.2 Comparison of Methods on Training Time versus RMSE

We first illustrate the effectiveness of our solutions for data imbalance and memory discontinuity. Then, we compare parallel matrix factorization methods including DSGD, CCD++, HogWild and our FPSG.

5.2.1 The effectiveness of addressing the locking problem

In Chapter 3.1, we mentioned that updating several blocks in a batch may suffer from the locking problem if the data is unbalanced. To verify the effectiveness of FPSG, in Figure 5.1, we compare it with a modification where the scheduler processes a batch of independent blocks as DSGD (Algorithm 2) does. We call the modified algorithm as FPSG**. It can be clearly seen that FPSG runs much faster than FPSG** because it does not suffer from the locking problem.

[3]`http://hazy.cs.wisc.edu/hazy/victor/`
[4]`http://www.cs.utexas.edu/~rofuyu/libpmf/`
[5]`http://openmp.org/`



(a) MovieLens (b) Netflix

(c) Yahoo!Music (d) Hugewiki

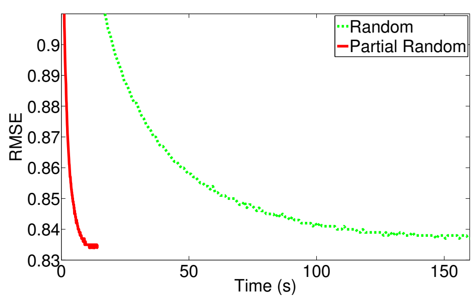
Figure 5.1: A comparison between FPSG** and FPSG.

5.2.2 The effectiveness of having better memory locality

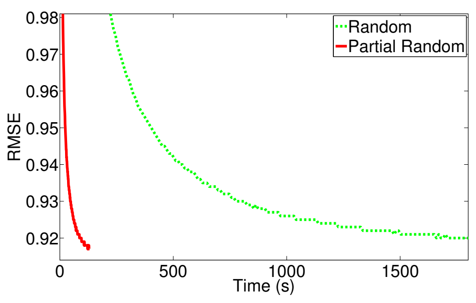
We conduct experiments to investigate if the proposed partial random method can not only avoid memory discontinuity, but also keep good convergence. In Figure 5.2, we select rating instances in each block orderly (the partial random method) or randomly (the random method). Both methods converge to a similar RMSE, but the training time of the partial random method is obviously shorter than that of the random method.

5.2.3 Comparison with the state-of-the-art methods

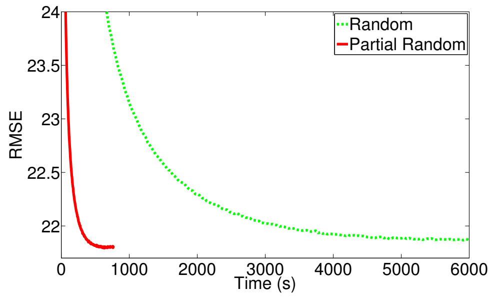
Figure 5.3 presents the test RMSE and training time of various parallel matrix factorization methods. Among the three parallel SG methods, FPSG is faster than DSGD and HogWild. We believe that this result is because FPSG is designed to effectively address issues mentioned in Chapter III. However, we must note that for DSGD, it is also easy to incorporate similar techniques (e.g., the partial random method) to



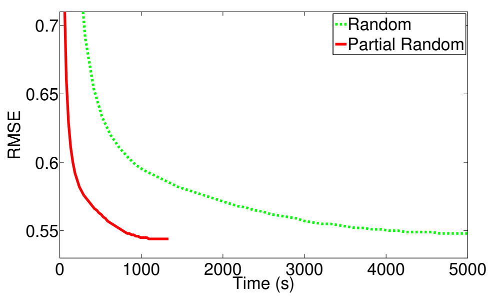
(a) MovieLens



(b) Netflix



(c) Yahoo!Music



(d) Hugewiki

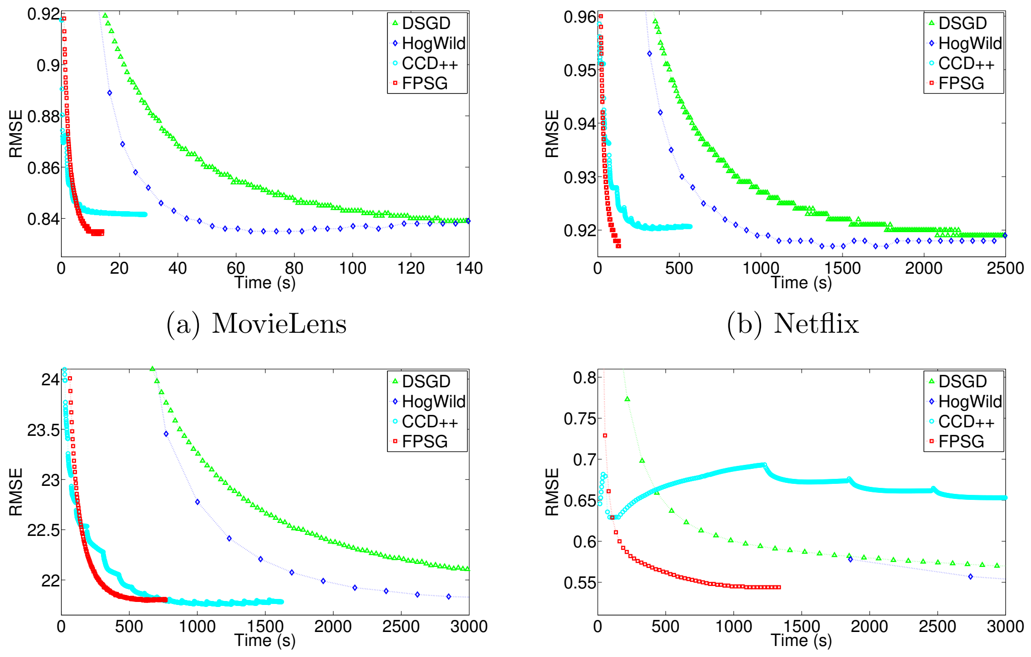
Figure 5.2: A comparison between the partial random method and the random method.

improve its performance.

As shown in Figure 5.3, CCD++ is the fastest in the beginning, but becomes slower than FPSG. Because the optimization problem of matrix factorization is non-convex and CCD++ is a more greedy setting than SG by accurately minimizing the objective function over certain variables at each step, we suspect that CCD++ may converge to some local minimum pre-maturely. On the contrary, SG-based methods may be able to escape from a local minimum because of the randomness. Furthermore, for the Hugewiki in Figure 5.3, CCD++ does not give a satisfactory RMSE. Note that in addition to the regularization parameter used in this experiment, (Yun et al., 2014) have applied larger parameters for Hugewiki. The resulting RMSE can be improved.

5.2.4 Comparison with CCD++ for Non-negative Matrix Factorization

We have seen that FPSG and CCD++ are two state-of-the-art algorithms for standard matrix factorization. It is interesting to see if FPSG can be extended to solve



(a) MovieLens (b) Netflix

(c) Yahoo!Music (d) Hugewiki

Figure 5.3: A comparison among the state-of-the-art parallel matrix factorization methods.

other matrix factorization problems. We consider non-negative matrix factorization (NMF) that requires the non-negativity of P and Q.

$$\min_{P,Q} \sum_{(u,v)\in R} \left((r_{u,v} - \boldsymbol{p}_u^T \boldsymbol{q}_v)^2 + \lambda_P \left\|\boldsymbol{p}_u\right\|^2 + \lambda_Q \left\|\boldsymbol{q}_v\right\|^2 \right), \tag{5.1}$$

$$\text{subject to} \quad P_{iu} \geq 0, \quad Q_{iv} \geq 0, \quad \forall i \in \{1, \cdots, k\}.$$

It is straightforward to warp FPSG for solving (5.1) with a simple projection (Gemulla et al., 2011), and the corresponding update rules are

$$\begin{aligned} \boldsymbol{p}_u &\leftarrow \max(\boldsymbol{0}, \boldsymbol{p}_u + \gamma\left(e_{u,v}\boldsymbol{q}_v - \lambda_P \boldsymbol{p}_u\right)) \\ \boldsymbol{q}_v &\leftarrow \max(\boldsymbol{0}, \boldsymbol{q}_v + \gamma\left(e_{u,v}\boldsymbol{p}_u - \lambda_Q \boldsymbol{q}_v\right)), \end{aligned} \tag{5.2}$$

where $\max(\cdot,\cdot)$ is an element-wise maximum operation.

For CCD++, a modification for NMF has been proposed in (Hsieh and Dhillon, 2011). Like (5.2), it projects negative values back to zero during the coordinate descent

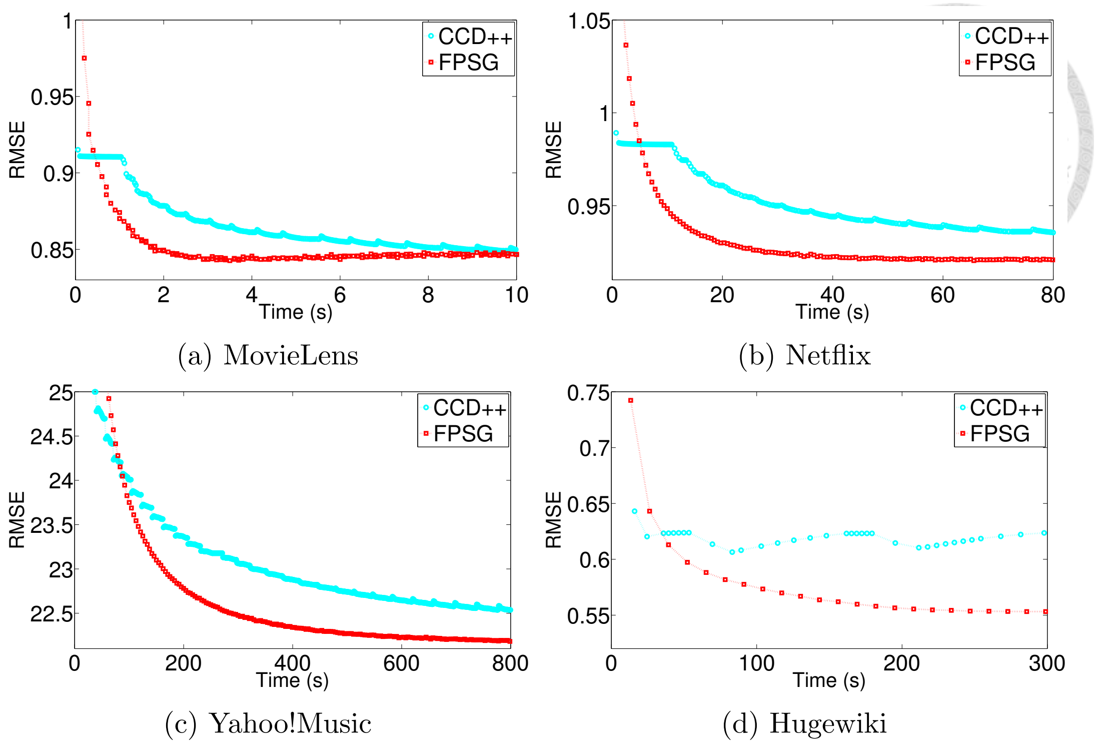


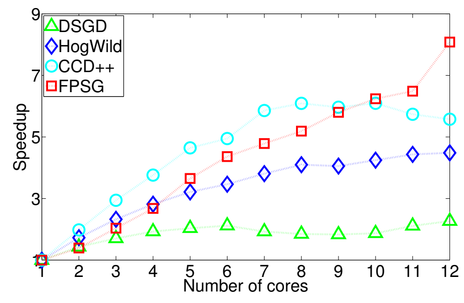
Figure 5.4: A comparison between CCD++ and FPSG for non-negative matrix factorization.

method. Our experimental comparison on CCD++ and FPSG is presented in Figure 5.4. Similar to Figure 5.3, FPSG outperforms CCD++ on NMF.

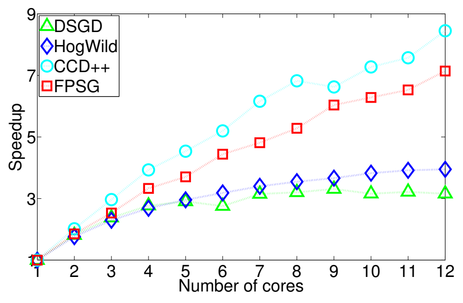
5.3 Speedup of FPSG

Speedup is an indicator on the effectiveness of a parallel algorithm. On a shared memory system, it refers to the time reduction from using one core to several cores. In this chapter, we compare the speedup of FPSG with other methods. From Figure 5.5, FPSG outperforms DSGD and HogWild. This result is expected because FPSG aims at improving some shortcomings of these two methods.

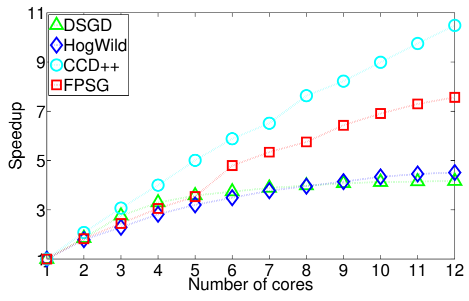
Compared with CCD++, FPSG is better on two data sets, while CCD++ is better on the others. As Algorithm 3 and Algorithm 4 show, FPSG and CCD++ are parallelized with very different ideas. Because speedup is determined by many factors in their respective parallel algorithms, it is difficult to explain why one is better than



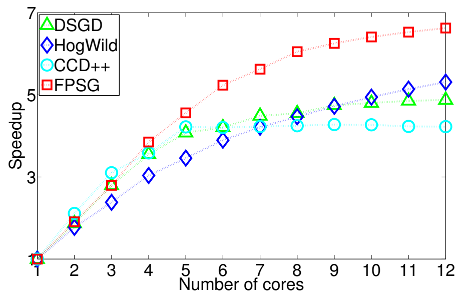
(a) MovieLens



(b) Netflix



(c) Yahoo!Music



(d) Hugewiki

Figure 5.5: Speedup of different matrix factorization methods.

the other on some problems. Nevertheless, even though CCD++ gives better speedup in some occasions, its overall performance (running time and RMSE) is still worse than FPSG in Figure 5.3. Thus parallel SG remains a compelling method for matrix factorization.

CHAPTER VI

Discussion

We discuss some miscellaneous issues in this chapter. Chapter 6.1 demonstrates that taking the advantage of data locality can further improve the proposed FPSG method. In Chapter 6.2, the selection of the number of blocks is discussed.

6.1 Data Locality and the Update Order

In our partial random method, ratings in each block are ordered. We can consider a user-oriented or item-oriented ordering. Interestingly, these two ways may cause different costs on the data access. For example, in Figure 4.3, we consider a user-oriented setting, so under a given u

$$R_{u,v} \text{ and } \boldsymbol{q}_v,\ \forall(u,v) \in R$$

must be accessed. While $R_{u,v}$ is a scalar, $\boldsymbol{q}_v$, $\forall(u,v) \in R$ involve many columns of the dense matrix Q. Therefore, for going through all users, Q is needed many times. Alternatively, if an item-oriented setting is used, for every item, P^T is needed. Now if $m \gg n$, P's size ($k \times m$) is much larger than Q ($k \times n$). Under the user-oriented setting, it is possible that Q (or a significant portion of Q) can be stored in a higher layer of the memory hierarchy because of its small size. Thus we do not waste time to frequently load Q from a lower layer. In contrast, under the item-oriented setting, P

Order \ Data set	MovieLens	Netflix	Yahoo!Music	Hugewiki
User	2.27	22.50	173.34	1531.14
Item	2.91	43.26	294.19	1016.19

Table 6.1: Execution time (in seconds) of 50 iterations of FPSG

# item blocks	16	16	16	16	32	32	32	32
# user blocks	16	32	48	64	16	32	48	64
iterations	50	48	49	48	48	49	49	49
time	3.25	3.37	3.66	3.94	3.28	3.86	4.74	6.98
# item blocks	48	48	48	48	64	64	64	64
# item blocks	16	32	48	64	16	32	48	64
iterations	49	49	49	49	49	49	48	49
time	3.43	4.83	8.15	13.30	3.67	6.94	12.87	21.16

Table 6.2: The performance of FPSG on MovieLens with different number of blocks. The target RMSE is 0.858. Time is in seconds.

may have to be swapped out to lower-level memory several times. Thus the cost for data movements is higher. Based on this discussion, we conjecture that

$$\begin{aligned} m \gg n &\Rightarrow \text{user-oriented access should be used,} \\ m \ll n &\Rightarrow \text{item-oriented access should be used.} \end{aligned} \tag{6.1}$$

We compare the two update orders in Table 6.1. For Netflix and Yahoo!Music, the user-wise approach is much faster. From Table 5.1, these two data sets have $m \gg n$. On the contrary, because $n \gg m$, the item-oriented approach is much better for Hugewiki. This experiment fully confirms our conclusion in (6.1).

6.2 Number of Blocks

Recall that in FPSG, R is separated to at least $(s+1) \times (s+1)$ blocks, where s is the number of threads. We conduct experiments to see how the number of blocks affects the performance of FPSG. The results on three data sets are listed in Table 6.2, Table 6.3, and Table 6.4. In these tables, “iterations” and “time” respectively mean

# item blocks	16	16	16	16	16	16	16
# user blocks	16	32	48	64	80	96	112
iterations	50	50	49	50	49	50	50
time	28.92	28.39	30.73	30.69	29.66	32.34	30.61
# item blocks	32	32	32	32	32	32	32
# user blocks	16	32	48	64	80	96	112
iterations	50	50	50	50	50	50	50
time	33.45	32.10	32.58	31.38	33.51	32.56	34.73
# item blocks	48	48	48	48	48	48	48
# user blocks	16	32	48	64	80	96	112
iterations	50	50	50	49	50	50	50
time	39.29	33.99	37.12	31.91	39.08	39.36	43.25
# item blocks	64	64	64	64	64	64	64
# user blocks	16	32	48	64	80	96	112
iterations	50	50	50	50	50	50	50
time	34.25	39.99	35.37	35.92	47.98	49.11	59.18
# item blocks	80	80	80	80	80	80	80
# user blocks	16	32	48	64	80	96	112
iterations	49	50	50	50	50	50	50
time	45.54	35.98	43.60	51.92	53.88	66.30	87.09
# item blocks	96	96	96	96	96	96	96
# user blocks	16	32	48	64	80	96	112
iterations	50	49	50	50	50	50	50
time	49.62	56.06	40.43	48.70	67.52	91.23	124.01
# item blocks	112	112	112	112	112	112	112
# user blocks	16	32	48	64	80	96	112
iterations	50	50	50	50	50	50	50
time	54.21	39.97	50.68	65.97	87.69	123.57	170.75

Table 6.3: The performance of FPSG on Netflix with different number of blocks. The target RMSE is 0.941. Time is in seconds.

# item blocks	32	32	32	32	32	32
# user blocks	32	64	96	128	160	192
iterations	50	50	50	50	50	50
time	188.13	180.21	184.57	183.51	186.66	187.02
# item blocks	64	64	64	64	64	64
# user blocks	32	64	96	128	160	192
iterations	50	50	50	50	50	50
time	172.74	182.90	218.67	196.95	189.23	211.93
# item blocks	96	96	96	96	96	96
# user blocks	32	64	96	128	160	192
iterations	50	50	50	50	50	50
time	188.57	179.78	205.17	203.49	227.86	254.71
# item blocks	128	128	128	128	128	128
# user blocks	32	64	96	128	160	192
iterations	50	50	49	50	50	50
time	192.20	188.69	218.00	242.98	379.63	565.25
# item blocks	160	160	160	160	160	160
# user blocks	32	64	96	128	160	192
iterations	50	50	50	50	50	50
time	195.24	201.39	254.64	374.44	606.47	842.62
# item blocks	192	192	192	192	192	192
# user blocks	32	64	96	128	160	192
iterations	50	50	50	50	50	50
time	196.41	217.05	246.15	563.86	842.81	1170.79

Table 6.4: The performance of FPSG on Yahoo!Music with different number of blocks. The target RMSE is 22.40. Time is in seconds.

the number of iterations and time used to achieve a target RMSE value.[1] We use 12 threads for the experiments.

On each data set, different numbers of blocks achieve the target RMSE in a similar number of iterations. Clearly, the number of blocks does not seem to affect the convergence. However, when many blocks are used, the running time dramatically increases. Taking MovieLens as an example, FPSG takes only 3.25 seconds if 16×16 blocks are used, while 21.16 seconds are required if 64×64 blocks are used. To explain this result, let us check what happens when the number of blocks increases. First, the overhead of getting a job increases because the selection is from a pool of more blocks. Second, the execution time per block decreases as a block contains less ratings. Third, the scheduler is executed more frequently because the execution time per block decreases. The overall impact is that the scheduling becomes cost-ineffective. That is, we spend innegligible time to select a block, but the block is quickly processed. Further, we explain that the CPU utilization may be lowered when too many blocks are used. In this situation, the scheduler takes more time to assign a block to a thread, but during this process, another thread that needs to get a block must wait.

The above discussion suggests that we should avoid splitting R to too many blocks. However, whether the number of blocks is too many or not depends on the data set. The 64×64 setting is too many for MovieLens, but seems adequate for Netflix. Therefore, the selection of the number of blocks is not easy. From the experimental results, using $(s+1) \times (s+1)$ to $2s \times 2s$ blocks may be a suitable choice.

[1]The cost of experiments would be very high if the best RMSE is considered, so we use a moderate one.

CHAPTER VII

Conclusions and Future Works

To provide a more complete SG solver for recommender systems, we will extend our algorithm to solve variants of matrix-factorization problems. In addition, to further reduce the cache-miss rate, we will investigate non-uniform splits of the rating matrix or other permutation methods such as Cuthill-McKee ordering. Very recently a new parallel matrix factorization method NOMAD (Yun et al., 2014) has been released. It uses an asynchronization setting to reduce the waiting time at any thread. This technique is related to our non-blocking scheduling. Another parallel solver for matrix factorization is in GraphChi (Kyrola et al., 2012), which is a framework for graph computation.[1] Their method divides R into m blocks, where each block contains the ratings of a particular user, and these blocks are updated in parallel. An important difference between ours and theirs is that they do not require blocks being processed are mutually independent. Therefore, the over-writing problem discussed in Chapter 2.1 may occur. We plan to conduct comparisons between our method, NOMAD, and GraphChi.

In conclusion, we point out some computational bottlenecks in existing parallel SG methods for shared-memory systems. We propose FPSG to address these issues and

[1]In their thesis, they use alternative least square method (ALS) as the solver. However, an SG implementation has been included in their latest release available at `https://github.com/GraphChi/graphchi-cpp`. We discuss their SG-based method in the context.

confirm its effectiveness by experiments. The comparison shows that FPSG is more efficient than state-of-the-art methods. Programs used for experiments in this thesis can be found at

`http://www.csie.ntu.edu.tw/~cjlin/libmf/exps/`

Further, based on this study, we develop an easy-to-use package `LIBMF` available at

`http://www.csie.ntu.edu.tw/~cjlin/libmf`

for public use. Appendix A describes the formulation used in `LIBMF`.

Acknowledgement

This is a joint work with Wei-Sheng Chin and Yong Zhuang.

BIBLIOGRAPHY

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

K.-W. Chang, C.-J. Hsieh, and C.-J. Lin. Coordinate descent method for large-scale L2-loss linear SVM. *Journal of Machine Learning Research*, 9:1369–1398, 2008. URL `http://www.csie.ntu.edu.tw/~cjlin/papers/cdl2.pdf`.

G. Dror, N. Koenigstein, Y. Koren, and M. Weimer. The Yahoo! music dataset and KDD-Cup 11. In *JMLR Workshop and Conference Proceedings: Proceedings of KDD Cup 2011*, volume 18, pages 3–18, 2012.

R. Gemulla, E. Nijkamp, P. J. Haas, and Y. 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, 2011.

K. B. Hall, S. Gilpin, and G. Mann. MapReduce/Bigtable for distributed optimization. In *Neural Information Processing Systems Workshop on Leaning on Cores, Clusters, and Clouds*, 2010.

C.-J. Hsieh and I. S. Dhillon. Fast coordinate descent methods with variable selection for non-negative matrix factorization. In *Proceedings of the Seventeenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2011.

J. Kiefer and J. Wolfowitz. Stochastic estimation of the maximum of a regression function. *The Annals of Mathematical Statistics*, 23(3):462–466, 1952.

Y. Koren, R. M. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, 2009.

A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*, Hollywood, October 2012.

G. Mann, R. McDonald, M. Mohri, N. Silberman, and D. Walker. Efficient large-scale distributed training of conditional maximum entropy models. In Y. Bengio, D. Schuurmans, J. Lafferty, C. K. I. Williams, and A. Culotta, editors, *Advances in Neural Information Processing Systems 22*, pages 1231–1239. 2009.

R. McDonald, K. Hall, and G. Mann. Distributed training strategies for the structured perceptron. In *Proceedings of the 48th Annual Meeting of the Association of Computational Linguistics (ACL)*, pages 456–464, 2010.

F. Niu, B. Recht, C. Ré, and S. J. Wright. HOGWILD!: A lock-free approach to parallelizing stochastic gradient descent. In J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira, and K. Weinberger, editors, *Advances in Neural Information Processing Systems 24*, pages 693–701, 2011.

I. Pilászy, D. Zibriczky, and D. 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, 2010.

H. Robbins and S. Monro. A stochastic approximation method. *The Annals of Mathematical Statistics*, 22(3):400–407, 1951.

H.-F. Yu, C.-J. Hsieh, S. Si, and I. S. Dhillon. Scalable coordinate descent approaches to parallel matrix factorization for recommender systems. In *Proceedings of the IEEE International Conference on Data Mining*, pages 765–774, 2012.

H. Yun, H.-F. Yu, C.-J. Hsieh, S. Vishwanathan, and I. S. Dhillon. Nomad: Non-locking, stochastic multi-machine algorithm for asynchronous and decentralized matrix completion. In *International Conference on Very Large Data Bases (VLDB)*, 2014.

Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan. Large-scale parallel collaborative filtering for the Netflix prize. In *Proceedings of the Fourth International Conference on Algorithmic Aspects in Information and Management*, pages 337–348, 2008.

Y. Zhuang, W.-S. Chin, Y.-C. Juan, and C.-J. Lin. A fast parallel SGD for matrix factorization in shared memory systems. In *Proceedings of the ACM Recommender Systems*, 2013. URL `http://www.csie.ntu.edu.tw/~cjlin/papers/libmf.pdf`.

M. Zinkevich, M. Weimer, A. Smola, and L. Li. Parallelized stochastic gradient descent. In J. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems 23*, pages 2595–2603. 2010.

APPENDICES

APPENDIX A. Formulation Used in LIBMF

In `LIBMF`, in addition to P and Q, we add user bias, item bias, and average terms, which are useful for recommender systems. The formulation is described in (A.1). Table A.1 shows the dimension and meaning of symbols in the formulation.

$$\begin{aligned}\min_{P,Q,\mathbf{a},\mathbf{b}} \sum_{(u,v)\in R} &\Big((r_{u,v} - \boldsymbol{p}_u^T \boldsymbol{q}_v - a_u - b_v - \text{avg})^2 \\ &+ \lambda_P ||\boldsymbol{p}_u||^2 + \lambda_Q ||\boldsymbol{q}_v||^2 + \lambda_{\mathbf{a}} ||\mathbf{a}||^2 + \lambda_{\mathbf{b}} ||\mathbf{b}||^2\Big)\end{aligned} \tag{A.1}$$

Symbol	Dimension	Meaning
m, n	1×1	number of users and items
k	1×1	number of latent dimensions
u, v	1×1	index indicates u_{th} user and v_{th} item
R	$m \times n$	rating matrix
$r_{u,v}$	1×1	$(u, v)_{th}$ rating of R
P	$k \times m$	latent matrix
Q	$k \times n$	latent matrix
$\boldsymbol{p}_u, \boldsymbol{q}_v$	$k \times 1$	u_{th} column of P and v_{th} column of Q
$\mathbf{a}$	$m \times 1$	user bias vector
$\mathbf{b}$	$n \times 1$	item bias vector
a_u, b_v	1×1	u_{th} element of $\mathbf{a}$ and v_{th} element of $\mathbf{b}$
λ_P, λ_Q	1×1	penalty of regularized term of P and Q
$\lambda_{\mathbf{a}}, \lambda_{\mathbf{b}}$	1×1	penalty of regularized term of $\mathbf{a}$ and $\mathbf{b}$
avg	1×1	average rating in training data

Table A.1: Symbols in (A.1).