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

WEI-SHENG CHIN, National Taiwan University
YONG ZHUANG, National Taiwan University
YU-CHIN JUAN, National Taiwan University
CHIH-JEN LIN, National Taiwan University

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 paper, 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.

Categories and Subject Descriptors: G.4 [**Mathematics of Computing**]: Mathematical Software

General Terms: Parallel and vector implementations

Additional Key Words and Phrases: Recommender system, Matrix factorization, Stochastic gradient descent, Parallel computing, Shared memory algorithm

**ACM Reference Format:**
Wei-Sheng Chin, Yong Zhuang, Yu-Chin Juan, and Chih-Jen Lin, 2013. A Fast Parallel Stochastic Gradient Method for Matrix Factorization in Shared Memory Systems. *ACM Trans. Intel. Sys. and Tech.* 0, 0, Article 0 ( 0), 24 pages.
DOI:http://dx.doi.org/10.1145/0000000.0000000

## 1. 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 $u$th user on the $v$th item at the $(u,v)$ entry, $r_{u,v}$, matrix

Acknowledgments: This work was supported in part by the National Science Council of Taiwan under Grants NSC 101-2221-E002-199-MY3, National Taiwan University under Grants NTU 102R7827, and a MediaTek Fellowship.
Authors' addresses: Wei-Sheng Chin, Yong Zhuang Yu-Chin Juan, and Chih-Jen Lin, Department of Computer Science, National Taiwan University;
Authors' email: {d01944006, r01922139, r01922136, cjlin}@csie.ntu.edu.tw

DOI:http://dx.doi.org/10.1145/0000000.0000000

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 $u$th column of $P$ and the $v$th column of $Q$. The optimization problem is

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

where $\|\cdot\|$ is the Euclidean norm, $(u, v) \in R$ indicates that rating $r_{u,v}$ is available, $\lambda_P$ and $\lambda_Q$ 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) is a difficult optimization problem. Many past studies have proposed optimization methods to solve (1), e.g., [Koren et al. 2009; Zhou et al. 2008; Pilászy et al. 2010]. 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), it randomly selects a ($u$,$v$) entry from the summation and calculates the corresponding gradient [Robbins and Monro 1951; Kiefer and Wolfowitz 1952]. Once $r_{u,v}$ is chosen, the objective function in (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

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

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

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 $\gamma$ is the learning rate. The overall procedure of SG is to iteratively select an instance $r_{u,v}$ , apply update rules (2)-(3), and may adjust the learning rate.

Although SG has been successfully applied to matrix factorization, it is not applicable to handle large-scale data. The iterative process of applying (2)-(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., [Zinkevich et al. 2010; McDonald et al. 2010; Mann et al. 2009; Hall et al. 2010; Gemulla et al. 2011; Niu et al. 2011]), 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 (2)-(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.

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

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 (2)-(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 (2)-(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 paper, 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 paper is organized as follows. We give details of HogWild and DSGD in Section 2. Another parallel matrix factorization method CCD++ is also discussed in this section. Then Section 3 discusses difficulties in parallelizing SG for matrix factorization. Our proposed method FPSG (Fast Parallel SG) is introduced in Section 4. We compare our method with state-of-the-art algorithms using root mean square error (RMSE) as the evaluation measure in Section 5. RMSE is defined as

$$\sqrt{\frac{1}{\text{number of ratings}} \sum_{(u,v)\in R} (r_{u,v} - \hat{r}_{u,v})^2}, \tag{4}$$

where $R$ is the rating matrix of the test set and $\hat{r}_{u,v}$ is the predicted rating value. In Section 6, we discuss some miscellaneous issues related to our proposed approach. Finally, Section 7 summarizes our work and gives future directions.

A preliminary version of this work appears in a conference paper [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.

## 2. EXISTING PARALLELIZED STOCHASTIC GRADIENT DESCENT ALGORITHMS AND COORDINATE DESCENT METHODS

Following the discussion in Section 1, in this section, 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 Section 5. 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 (2)-(3) are likely to be independent. The reason is that the selected ratings to be updated almost never share

**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 (2)-(3), respectively
5: **end while**
6: **end for**

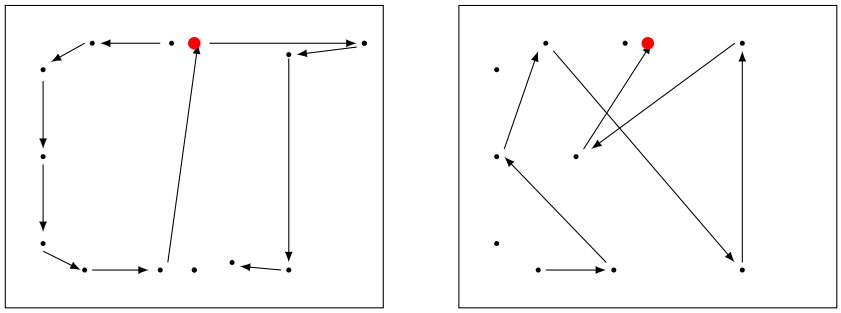


Fig. 1: An example shows updating sequences of two threads in HogWild.

the same user identity and item identity. Then, iterations of SG, (2)-(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 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 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 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 1, the operations include

— reading $r_{i,j}$, $\boldsymbol{p}_i$ and $\boldsymbol{q}_j$,
— evaluating the right-hand sides of (2)-(3), and
— assigning values to the left-hand sides of (2)-(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.

### 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-

**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 × s blocks B and generate s patterns covering all blocks
2: for t = {1, ..., 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, ..., s} parallellly do
7:             randomly sample ratings from block b
8:             apply (2)-(3) on all sampled ratings
9:         end for
10:    end for
11: end for
```

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, 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 $b$th 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 3a, we show two consecutive iterations of the innermost for-loop in Figure 3b. The left iteration assigns 4 diagonal 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 3a to Figure 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 $Q$ corresponding 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

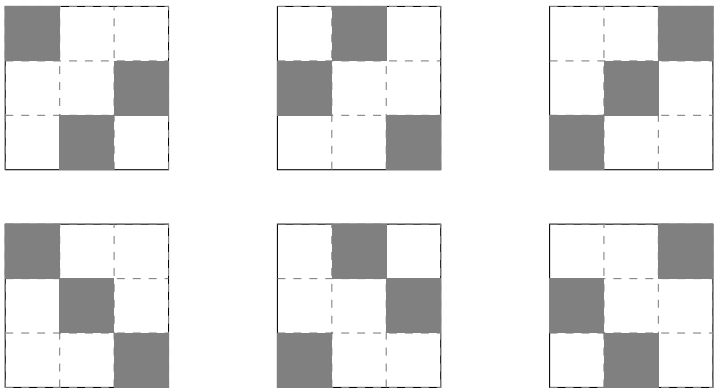
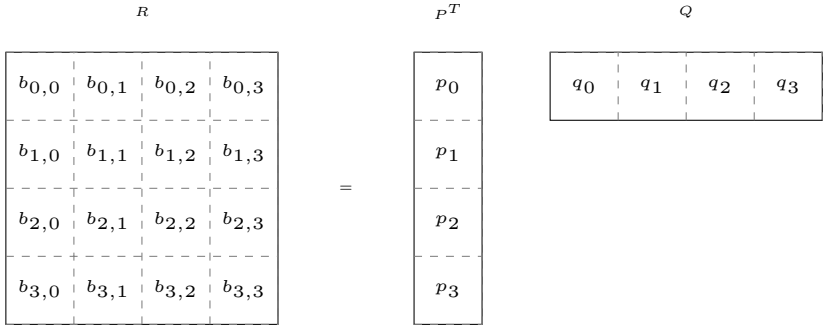
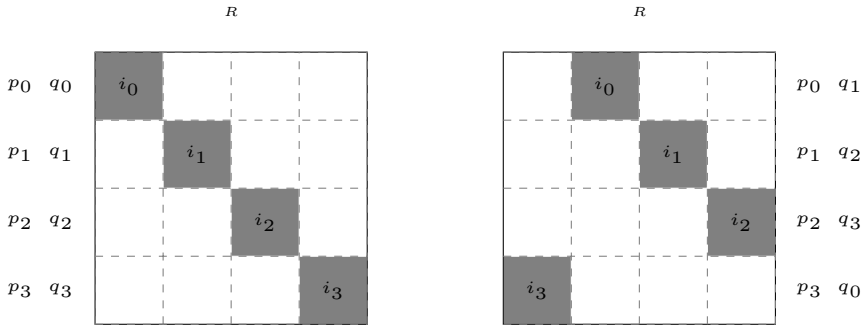


Fig. 2: Patterns of independent blocks for a 3 by 3 grided matrix.




(a) 4 by 4 grided rating matrix $R$ and corresponding segments of $P$ and $Q$. Note that $p_i$ is the $i$th segment of P and $q_j$ is the $j$th 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.

Fig. 3: An illustration of the DSGD algorithm.

other rows are fixed, the objective function in (1) can be converted to

$$\sum_{(u,v)\in R} \left(r_{u,v} - \boldsymbol{p}_u^T \boldsymbol{q}_v + \hat{p}_u \hat{q}_v - w_u h_v\right)^2 + \lambda_P \left(\sum_{u=1}^{m} \|\boldsymbol{p}_u\|^2 - \sum_{u=1}^{m} \hat{p}_u^2 + \sum_{u=1}^{m} w_u^2\right) + \lambda_Q \left(\sum_{v=1}^{n} \|\boldsymbol{q}_v\|^2 - \sum_{v=1}^{n} \hat{q}_v^2 + \sum_{v=1}^{n} h_v^2\right) \tag{5}$$

**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 (8) 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 (9) 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**

or

$$\sum_{(u,v)\in R} (e_{u,v} + \hat{p}_u\hat{q}_v - w_u h_v)^2 + \lambda_P \sum_{u=1}^{m} w_u^2 + \lambda_Q \sum_{v=1}^{n} h_v^2 \tag{6}$$

by dropping terms that do not depend on $\boldsymbol{w}$ or $\boldsymbol{h}$. If $\boldsymbol{w}$ (or $\boldsymbol{h}$) is fixed, the minimization of (6) 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 $\hat{\boldsymbol{q}}$, (6) becomes

$$\sum_{u=1}^{m} \left( \sum_{v:\,(u,v)\in R} (e_{u,v} + \hat{p}_u\hat{q}_v - w_u\hat{q}_v)^2 + \lambda_P w_u^2 \right) + \text{constant}. \tag{7}$$

It can be decomposed into $m$ independent problems

$$\min_{w_u} \sum_{v:\,(u,v)\in R} (e_{u,v} + \hat{p}_u\hat{q}_v - w_u\hat{q}_v)^2 + \lambda_P w_u^2, \quad \forall u = 1, \ldots, m. \tag{8}$$

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 $\hat{\boldsymbol{q}}$.

$$\min_{h_v} \sum_{u:\,(u,v)\in R} (e_{u,v} + \hat{p}_u\hat{q}_v - \hat{p}_u h_v)^2 + \lambda_Q h_v^2, \quad \forall v = 1, \ldots, n. \tag{9}$$

The parallelism of CCD++ is achieved by solving those independent problems in (8) and (9) simultaneously. See Algorithm 3 for the whole procedure of CCD++.

## 3. PROBLEMS IN PARALLEL SG METHODS FOR MATRIX FACTORIZATION

In this section, we point out that parallel SG methods discussed in Section 2 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.

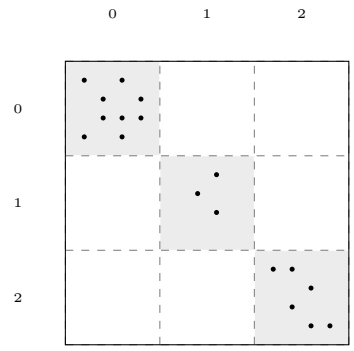



Fig. 4: An example of the locking problem in DSGD. Each dot represents a rating; gray blocks indicate a set of independent blocks. Ratings in white blocks are not shown.

### 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 4, 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.

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 Section 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 5. Though the random method generally enjoys good convergence, it suffers from 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 Section 4.2, we can easily change the update order to mitigate the memory discontinuity.

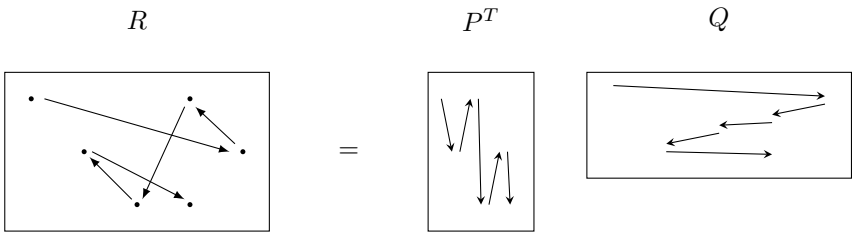



Fig. 5: A random method to select rating instances for update.

## 4. OUR APPROACHES

In this paper, we propose two techniques, *lock-free scheduling* and *partial random method*, to respectively solve the locking problem mentioned in Section 3.1 and the memory discontinuity mentioned in Section 3.2. We name the new parallel SG method as *fast parallel SG* (FPSG). In Section 4.1, we discuss how FPSG flexibly assigns blocks to threads to avoid the locking problem. In Section 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 Section 4.3, we overview the complete design of FPSG. Finally, in Section 4.4, we introduce our implementation techniques to accelerate the computation.

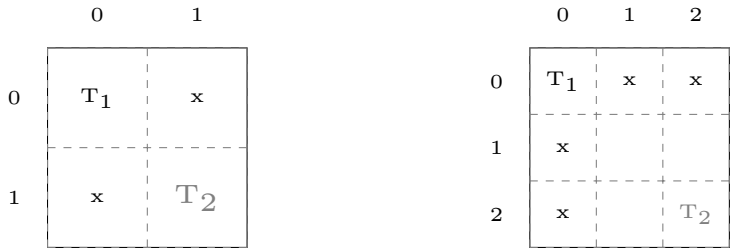
### 4.1. Lock-Free Scheduling

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.

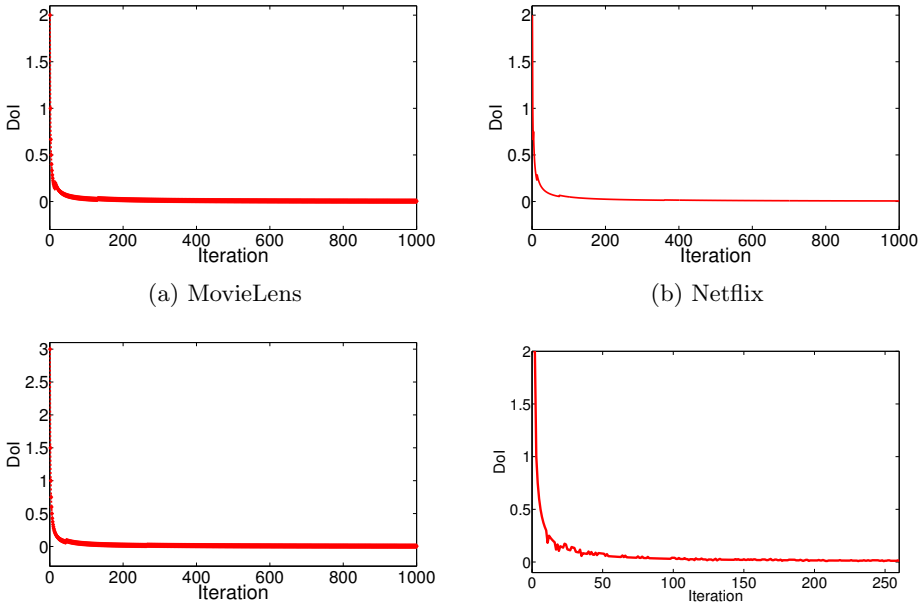
The number of updates of a block indicates how many times it has been processed. The second criterion is applied because we want to keep a similar number of updates for each block. If two or more blocks meet the above two criteria, then we randomly select one. Given $s$ threads, we show that FPSG should grid $R$ into at least $(s+1) \times (s+1)$ blocks. Take two threads as an example. Let $\mathrm{T}_1$ be a thread that is updating certain block and $\mathrm{T}_2$ be a thread that just finished updating a block and is getting a new job from the scheduler. If we grid $R$ into $2 \times 2$ blocks shown in Figure 6a, then $\mathrm{T}_2$ has only one choice: the block it just processed. A similar situation happens when $\mathrm{T}_1$ gets its new job. Because $\mathrm{T}_1$ and $\mathrm{T}_2$ always process the same block, the remaining two blocks are never processed. In contrast, if we grid $R$ into $3 \times 3$ blocks like Figure 6b, $\mathrm{T}_2$ 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 Section 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




(a) $2 \times 2$ blocks (b) $3 \times 3$ blocks

Fig. 6: An illustration of how the split of $R$ to blocks affects the job scheduling. $\mathrm{T}_1$ is the thread that is updating block $b_{0,0}$. $\mathrm{T}_2$ 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 $T_2$.




(a) MovieLens (b) Netflix

(c) Yahoo!Music (d) Hugewiki

Fig. 7: DoI on four data sets. $R$ is grided into $13 \times 13$ blocks after being randomly shuffled and 12 threads are used.

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

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

A small DoI indicates that the number of updates is similar across all blocks. In Figure 7, 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 7, please refer to Section 5.1.

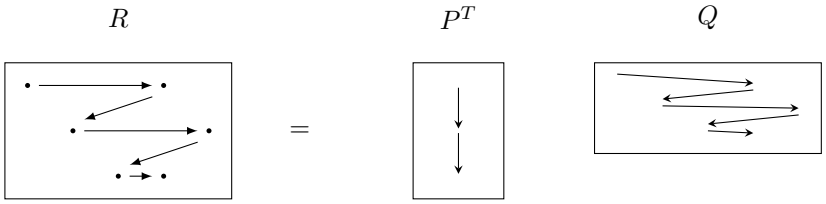


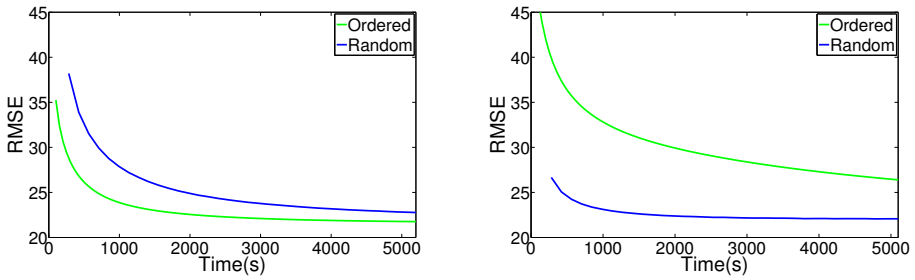

Fig. 8: Ordered method to select rating instances for update.

### 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 8 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 9 gives an example showing that under two slightly different learning rates for SG, the ordered method can be 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 10 illustrates how the partial random method works using three threads. Figure 11 extends the running time comparison in Figure 9 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 Section 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 12. In the beginning, we make the rating matrix more balanced by randomly shuffling all ratings; see the middle figure in Figure 12. However, user identities and item identities become not ordered, so we cannot achieve memory continuity by using the update strategy shown in Figure 8. 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 12.




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

Fig. 9: A comparison between the random method and the ordered method using the Yahoo!Music data set. One thread is used.

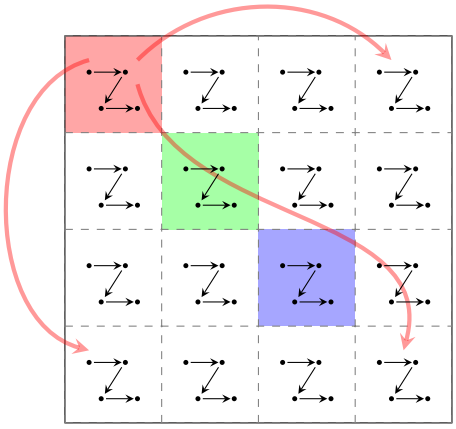
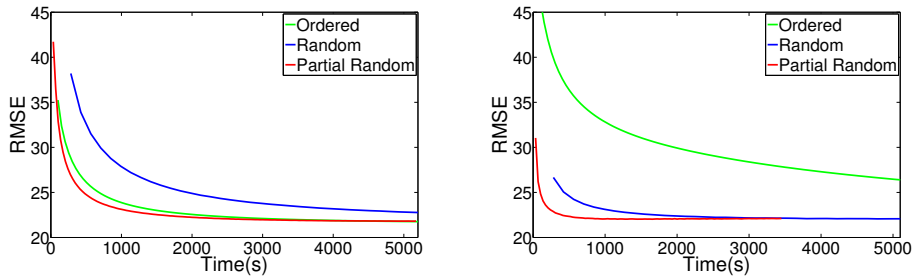


Fig. 10: An illustration of the partial random method. Each color indicates the block being processed by a thread. Within each block, the update sequence is ordered like that in Figure 8. If block (1,1) is finished first, three candidates independent of the two other running blocks (2,2) and (3,3) are (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.

### 4.3. Overview of FPSG

Algorithm 4 gives the overall procedure of FPSG. Based on the discussion in Section 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 Section 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 Section 4.1. After a working thread gets a new block, it processes ratings in the block in an ordered manner (see Section 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.




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

Fig. 11: A comparison between the ordered method, the random method, and the partial random method on the set Yahoo!Music. One thread is used.

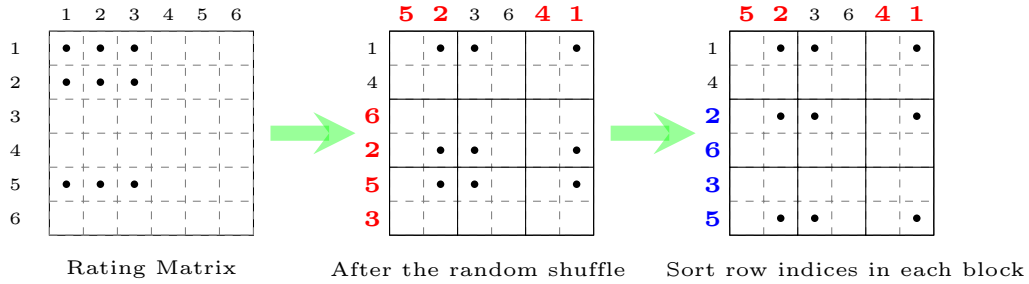



Fig. 12: An illustration of the partial random method. After the random shuffle of data, some indices (in red color) are not ordered within each block. We make the row indices ordered (in blue color) by a sorting procedure.

**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

### 4.4. Implementation Issues

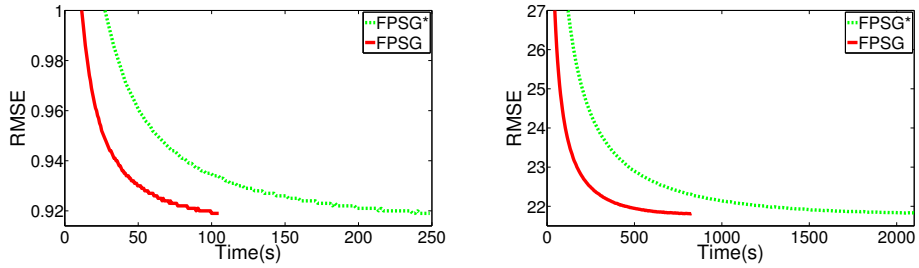
FPSG uses the standard thread class in C++ implemented by pthread to do the parallelization. For the data set Yahoo!Music of about 250M ratings, using a typical machine (details specified in Section 5.1), FPSG finishes processing all ratings once in 6 seconds and takes only about 8 minutes to converge 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.

**Algorithm 5** Scheduler of FPSG

1: **procedure** GET_JOB
2: Randomly select a block $b$ from the free blocks that have the smallest number of updates
3: **return** $b$
4: **end procedure**
5: **procedure** PUT_JOB($b$)
6: increase $b$'s update times by one
7: **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**




(a) Netflix (b) Yahoo!Music

Fig. 13: A comparison between two implementations of FPSG in Netflix and Yahoo!Music. FPSG implements the two techniques discussed in Section 4.4, while FPSG* does not.

For Yahoo!Music data set, the speed up is 2.4 times. Figure 13 shows the speedup after these techniques are applied in two data sets.[2]

Now we discuss the implementation of the scheduler. A naive way is to first identify all blocks having the smallest number of updates, and then randomly select one. This implementation must go through all blocks, so for a fine grid of the rating matrix, the running time can be large. Therefore, we should store and maintain block information in a structure so that blocks having the smallest number of updates can be quickly extracted. An example is the priority queue according to blocks' numbers of updates. However, two issues must be addressed. First, a non-free block cannot be selected. Second, when two or more free blocks simultaneously have the smallest number of updates, we must randomly select one. To solve the first problem, we keep popping non-free blocks from the priority queue to a list until a free block is obtained. Then we push these non-free blocks back into the priority queue, and then return the free block. Note that the number of popped blocks is no more than $s$, the number of threads, so we avoid going through the $O(s^2)$ number of all blocks. For the second problem, a trick can be employed. Instead of using the number

[2]For experimental settings, see Section 5.1.

**Algorithm 7** An implementation of the scheduler using priority queue

```
1: procedure GET_JOB
2:     Create an empty list NFBS to store non-free blocks
3:     while true do
4:         b = pq.pop()                          ▷ pq: the priority queue
5:         if b is non-free then
6:             Append b to NFBS
7:         else
8:             Push all blocks in NFBS back to pq
9:             return b
10:        end if
11:    end while
12: end procedure
13: procedure PUT_JOB(b)
14:    b.ut = b.ut + 1                           ▷ ut: number of updates
15:    b.utj = b.ut + rand()
16:    pq.push(b)                                ▷ pq uses utj as the priority
17: end procedure
```

of updates as the "priority" of the queue, we consider

$$\#\text{updates} + \text{a value } \in [0.0, 1.0).$$

This trick makes blocks with the same number of updates have "local" random priorities, but does not influence the relative priority between blocks with different number of updates. The pseudo code is given in Algorithm 7.

## 5. EXPERIMENTS

In this section, we provide the details about our experimental settings, and compare FPSG with other parallel matrix factorization algorithms mentioned in Section 2.

### 5.1. Settings

**Data Sets:** Four data sets, MovieLens,[3] Netflix, Yahoo!Music, and Hugewiki,[4] 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 I.

**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 I lists the parameters used for each data set. The parameters $k$, $\lambda_P$, $\lambda_Q$ may be chosen by a validation procedure although here we mainly borrow 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 I. 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

[3]http://www.grouplens.org/node/73
[4]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 |
| $\lambda_P$ | 0.05 | 0.05 | 1 | 0.01 |
| $\lambda_Q$ | 0.05 | 0.05 | 1 | 0.01 |
| $\gamma$ | 0.003 | 0.002 | 0.0001 | 0.004 |

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

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 paper, 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 Section 4 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 $32 \times 32$ blocks; for MovieLens, $R$ is grided into $16 \times 16$ 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. (4). In addition, the time in each figure refers to the training time.

**Implementations:** Among methods included for comparison, HogWild[5] and CCD++[6] 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 Section 4.4, for a fair comparison, we obtain a singles-precision version of CCD++ from its authors. Note that OpenMP[7] is used in their implementation.

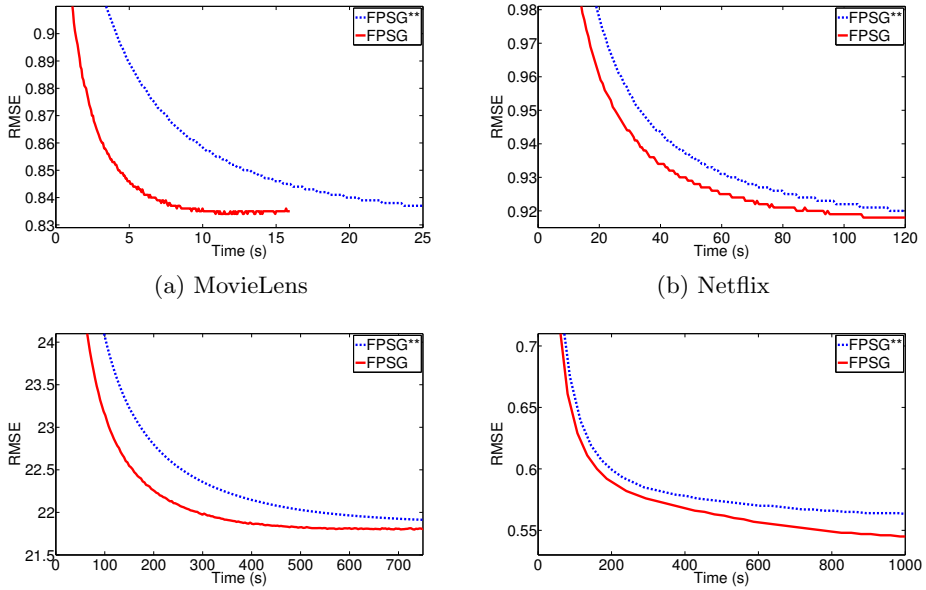
### 5.2. Comparison of Methods on Training Time versus RMSE

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 Section 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 14, 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.

*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

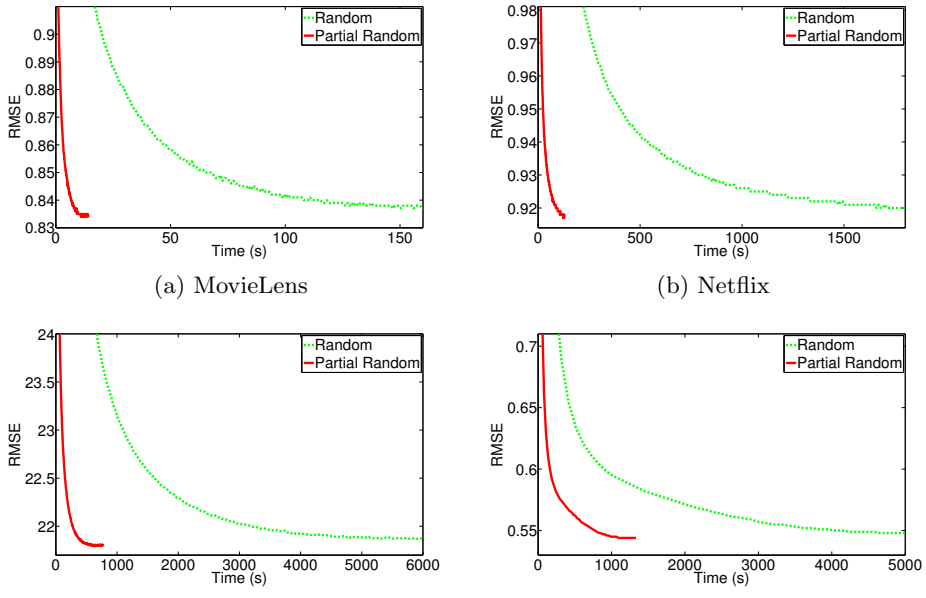
[5]http://hazy.cs.wisc.edu/hazy/victor/
[6]http://www.cs.utexas.edu/~rofuyu/libpmf/
[7]http://openmp.org/




(a) MovieLens (b) Netflix

(c) Yahoo!Music (d) Hugewiki

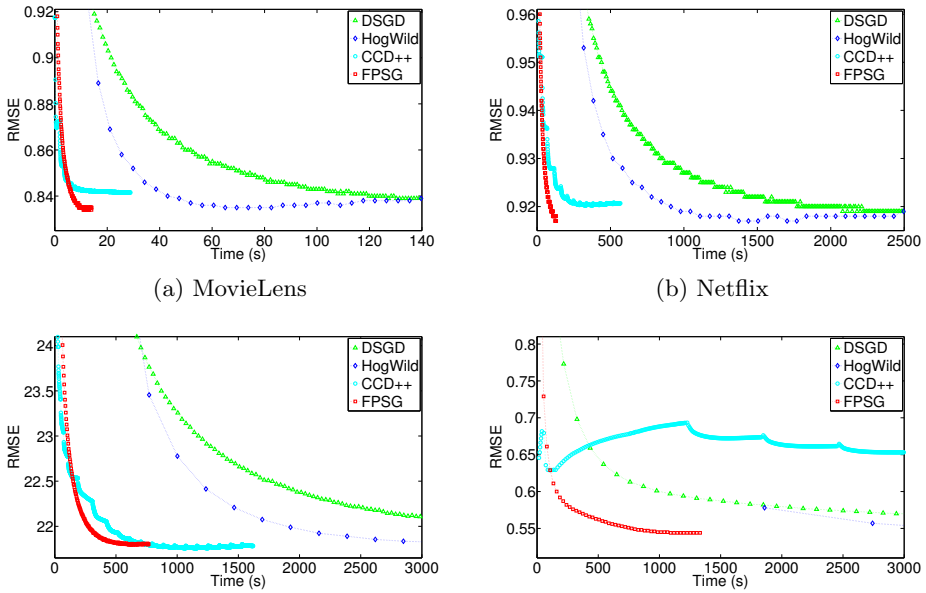
Fig. 14: A comparison between FPSG** and FPSG.




(a) MovieLens (b) Netflix

(c) Yahoo!Music (d) Hugewiki

Fig. 15: A comparison between the partial random method and the random method.




(a) MovieLens (b) Netflix

(c) Yahoo!Music (d) Hugewiki

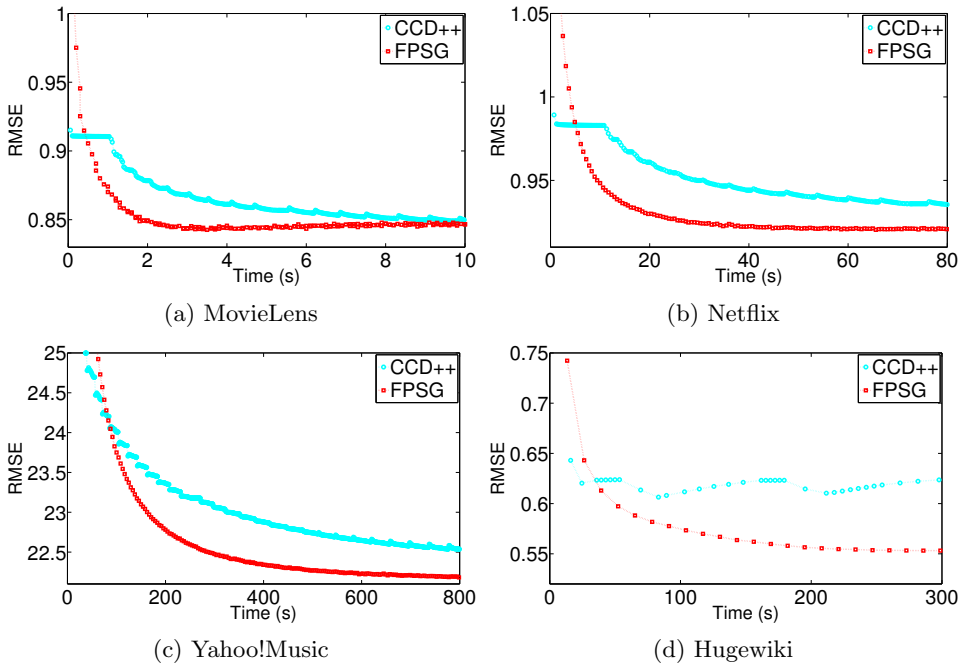
Fig. 16: A comparison among the state-of-the-art parallel matrix factorization methods.

keep good convergence. In Figure 15, 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 16 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 Section 3. However, we must note that for DSGD, it is also easy to incorporate similar techniques (e.g., the partial random method) to improve its performance.

As shown in Figure 16, 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 16, 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 other matrix factorization problems. We consider non-negative matrix factorization (NMF) that requires the non-negativity of




(a) MovieLens (b) Netflix

(c) Yahoo!Music (d) Hugewiki

Fig. 17: A comparison between CCD++ and FPSG for non-negative matrix factorization.

$P$ and $Q$.

$$\min_{P,Q} \quad \sum_{(u,v)\in R} \left((r_{u,v} - \boldsymbol{p}_u^T \boldsymbol{q}_v)^2 + \lambda_P \left\|\boldsymbol{p}_u\right\|^2 + \lambda_Q \left\|\boldsymbol{q}_v\right\|^2\right), \tag{10}$$

$$\text{subject to} \quad P_{iu} \geq 0, \quad Q_{iv} \geq 0, \quad \forall i \in \{1,\ldots,k\}, \quad \forall u \in \{1,\ldots,m\}, \quad \forall v \in \{1,\ldots,n\}.$$

It is straightforward to warp FPSG for solving (10) with a simple projection [Gemulla et al. 2011], and the corresponding update rules are

$$\begin{aligned} \boldsymbol{p}_u &\leftarrow \max(\boldsymbol{0}, \boldsymbol{p}_u + \gamma\,(e_{u,v}\boldsymbol{q}_v - \lambda_P \boldsymbol{p}_u)) \\ \boldsymbol{q}_v &\leftarrow \max(\boldsymbol{0}, \boldsymbol{q}_v + \gamma\,(e_{u,v}\boldsymbol{p}_u - \lambda_Q \boldsymbol{q}_v)), \end{aligned} \tag{11}$$

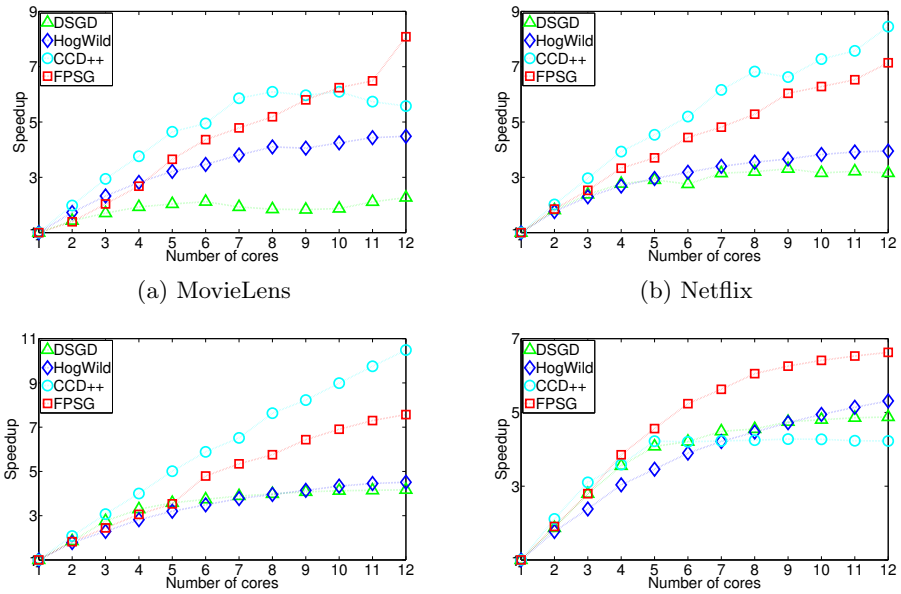
where $\max(\cdot,\cdot)$ is an element-wise maximum operation.

For CCD++, a modification for NMF has been proposed in [Hsieh and Dhillon 2011]. Like (11), it projects negative values back to zero during the coordinate descent method. Our experimental comparison on CCD++ and FPSG is presented in Figure 17. Similar to Figure 16, 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 section, we compare the speedup of FPSG with other methods. From Figure 18, 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




(a) MovieLens (b) Netflix

(c) Yahoo!Music (d) Hugewiki

Fig. 18: Speedup of different matrix factorization methods.

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 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 16. Thus parallel SG remains a compelling method for matrix factorization.

## 6. DISCUSSION

We discuss some miscellaneous issues in this section. Section 6.1 demonstrates that taking the advantage of data locality can further improve the proposed FPSG method. In Section 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 8, 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, $P$ may have to be swapped out to lower-level memory several times. Thus the cost for data movements is higher. Based on this discussion,

| 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 II: Execution time (in seconds) of 50 iterations of FPSG

| | | | | | |
|---|---|---|---|---|---|
| # item blocks | 16 | 16 | 16 | 16 | 16 |
| # user blocks | 16 | 32 | 64 | 128 | 256 |
| iterations | 47 | 47 | 47 | 48 | 47 |
| time | 3.60 | 3.59 | 3.10 | 3.19 | 3.50 |
| # item blocks | 32 | 32 | 32 | 32 | 32 |
| # user blocks | 16 | 32 | 64 | 128 | 256 |
| iterations | 47 | 49 | 48 | 47 | 48 |
| time | 3.59 | 3.24 | 3.28 | 3.40 | 4.01 |
| # item blocks | 64 | 64 | 64 | 64 | 64 |
| # user blocks | 16 | 32 | 64 | 128 | 256 |
| iterations | 48 | 48 | 48 | 48 | 47 |
| time | 3.18 | 3.67 | 3.46 | 3.92 | 5.39 |
| # item blocks | 128 | 128 | 128 | 128 | 128 |
| # user blocks | 16 | 32 | 64 | 128 | 256 |
| iterations | 47 | 47 | 48 | 47 | 48 |
| time | 3.23 | 3.60 | 4.09 | 5.18 | 9.50 |
| # item blocks | 256 | 256 | 256 | 256 | 256 |
| # user blocks | 16 | 32 | 64 | 128 | 256 |
| iterations | 48 | 47 | 47 | 47 | 47 |
| time | 3.83 | 4.52 | 5.86 | 9.58 | 17.94 |

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

we conjecture that

$$\begin{aligned} m \gg n &\Rightarrow \text{user-oriented access should be used,} \\ m \ll n &\Rightarrow \text{item-oriented access should be used.} \end{aligned} \tag{12}$$

We compare the two update orders in Table II. For Netflix and Yahoo!Music, the user-wise approach is much faster. From Table I, 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 (12).

### 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 Tables III-V. In these tables, "iterations" and "time" respectively mean the number of iterations and time used to achieve a target RMSE value.[8] 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 increases. Taking MovieLens as an example, FPSG takes only 3.60 seconds if $16 \times 16$ blocks are used, while 17.94 seconds are

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

| | | | | | |
|---|---|---|---|---|---|
| # item blocks | 16 | 16 | 16 | 16 | 16 |
| # user blocks | 16 | 32 | 64 | 128 | 256 |
| iterations | 48 | 48 | 48 | 48 | 48 |
| time | 30.43 | 26.72 | 28.79 | 27.33 | 29.35 |
| # item blocks | 32 | 32 | 32 | 32 | 32 |
| # user blocks | 16 | 32 | 64 | 128 | 256 |
| iterations | 49 | 48 | 48 | 48 | 48 |
| time | 30.57 | 29.86 | 31.71 | 31.97 | 32.22 |
| # item blocks | 64 | 64 | 64 | 64 | 64 |
| # user blocks | 16 | 32 | 64 | 128 | 256 |
| iterations | 49 | 48 | 49 | 49 | 48 |
| time | 33.64 | 34.97 | 35.18 | 37.06 | 38.53 |
| # item blocks | 128 | 128 | 128 | 128 | 128 |
| # user blocks | 16 | 32 | 64 | 128 | 256 |
| iterations | 49 | 49 | 49 | 48 | 48 |
| time | 58.83 | 44.57 | 46.05 | 50.67 | 41.47 |
| # item blocks | 256 | 256 | 256 | 256 | 256 |
| # user blocks | 16 | 32 | 64 | 128 | 256 |
| iterations | 48 | 48 | 48 | 48 | 48 |
| time | 81.86 | 72.27 | 67.36 | 64.62 | 76.69 |

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

| | | | | | |
|---|---|---|---|---|---|
| # item blocks | 16 | 16 | 16 | 16 | 16 |
| # user blocks | 16 | 32 | 64 | 128 | 256 |
| iterations | 49 | 49 | 49 | 49 | 49 |
| time | 199.59 | 204.56 | 205.96 | 204.66 | 202.11 |
| # item blocks | 32 | 32 | 32 | 32 | 32 |
| # user blocks | 16 | 32 | 64 | 128 | 256 |
| iterations | 49 | 50 | 49 | 49 | 49 |
| time | 173.23 | 181.84 | 190.59 | 196.78 | 189.09 |
| # item blocks | 64 | 64 | 64 | 64 | 64 |
| # user blocks | 16 | 32 | 64 | 128 | 256 |
| iterations | 50 | 49 | 49 | 49 | 49 |
| time | 168.26 | 170.41 | 169.96 | 171.63 | 205.93 |
| # item blocks | 128 | 128 | 128 | 128 | 128 |
| # user blocks | 16 | 32 | 64 | 128 | 256 |
| iterations | 50 | 49 | 49 | 49 | 49 |
| time | 184.93 | 178.21 | 200.42 | 193.64 | 234.01 |
| # item blocks | 256 | 256 | 256 | 256 | 256 |
| # user blocks | 16 | 32 | 64 | 128 | 256 |
| iterations | 49 | 49 | 49 | 49 | 49 |
| time | 206.93 | 203.07 | 219.30 | 235.04 | 289.15 |

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

required if $256 \times 256$ 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 $128 \times 128$ setting is too many for MovieLens, but seems adequate for Yahoo!Music. Therefore, the selection of the number of blocks is not easy. From the experimental results, using around $2s \times 2s$ blocks may be a suitable choice.

## 7. 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.[9] 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 Section 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 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 paper 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.

## REFERENCES

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

Kai-Wei Chang, Cho-Jui Hsieh, and Chih-Jen Lin. 2008. Coordinate Descent Method for Large-scale L2-loss Linear SVM. *Journal of Machine Learning Research* 9 (2008), 1369–1398. http://www.csie.ntu.edu.tw/~cjlin/papers/cdl2.pdf

Gideon Dror, Noam Koenigstein, Yehuda Koren, and Markus Weimer. 2012. The Yahoo! Music Dataset and KDD-Cup 11. In *JMLR Workshop and Conference Proceedings: Proceedings of KDD Cup 2011*, Vol. 18. 3–18.

[9] In their paper, 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.

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

Keith B. Hall, Scott Gilpin, and Gideon Mann. 2010. MapReduce/Bigtable for Distributed Optimization. In *Neural Information Processing Systems Workshop on Leaning on Cores, Clusters, and Clouds*.

Cho-Jui Hsieh and Inderjit S. Dhillon. 2011. 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*.

Jack Kiefer and Jacob Wolfowitz. 1952. Stochastic Estimation of the Maximum of a Regression Function. *The Annals of Mathematical Statistics* 23, 3 (1952), 462–466.

Yehuda Koren, Robert M. Bell, and Chris Volinsky. 2009. Matrix Factorization Techniques for Recommender Systems. *Computer* 42, 8 (2009), 30–37.

Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi: large-scale graph computation on just a PC. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

Gideon Mann, Ryan McDonald, Mehryar Mohri, Nathan Silberman, and Dan Walker. 2009. Efficient Large-Scale Distributed Training of Conditional Maximum Entropy Models. In *Advances in Neural Information Processing Systems 22*, Y. Bengio, D. Schuurmans, J. Lafferty, C. K. I. Williams, and A. Culotta (Eds.). 1231–1239.

Ryan McDonald, Keith Hall, and Gideon Mann. 2010. Distributed Training Strategies for the Structured Perceptron. In *Proceedings of the 48th Annual Meeting of the Association of Computational Linguistics (ACL)*. 456–464.

Feng Niu, Benjamin Recht, Christopher Ré, and Stephen J. Wright. 2011. HOGWILD!: a lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems 24*, J. Shawe-Taylor, R.S. Zemel, P. Bartlett, F.C.N. Pereira, and K.Q. Weinberger (Eds.). 693–701.

István Pilászy, Dávid Zibriczky, and Domonkos Tikk. 2010. Fast ALS-based matrix factorization for explicit and implicit feedback datasets. In *Proceedings of the Fourth ACM Conference on Recommender Systems*. 71–78.

Herbert Robbins and Sutton Monro. 1951. A Stochastic Approximation Method. *The Annals of Mathematical Statistics* 22, 3 (1951), 400–407.

Hsiang-Fu Yu, Cho-Jui Hsieh, Si Si, and Inderjit S. Dhillon. 2012. Scalable Coordinate Descent Approaches to Parallel Matrix Factorization for Recommender Systems. In *Proceedings of the IEEE International Conference on Data Mining*. 765–774.

Hyokun Yun, Hsiang-Fu Yu, Cho-Jui Hsieh, S.V.N. Vishwanathan, and Inderjit S. Dhillon. 2014. NOMAD: Non-locking, stochastic Multi-machine algorithm for Asynchronous and Decentralized matrix completion. In *International Conference on Very Large Data Bases (VLDB)*.

Yunhong Zhou, Dennis Wilkinson, Robert Schreiber, and Rong Pan. 2008. Large-Scale Parallel Collaborative Filtering for the Netflix Prize. In *Proceedings of the Fourth International Conference on Algorithmic Aspects in Information and Management*. 337–348.

Yong Zhuang, Wei-Sheng Chin, Yu-Chin Juan, and Chih-Jen Lin. 2013. A fast parallel SGD for matrix factorization in shared memory systems. In *Proceedings of the ACM Recommender Systems*. http://www.csie.ntu.edu.tw/~cjlin/papers/libmf.pdf

Martin Zinkevich, Markus Weimer, Alex Smola, and Lihong Li. 2010. Parallelized Stochastic Gradient Descent. In *Advances in Neural Information Processing Systems 23*, J. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R.S. Zemel, and A. Culotta (Eds.). 2595–2603.