A Fast Parallel SGD for Matrix Factorization in Shared Memory Systems

Yong Zhuang, Wei-Sheng Chin, Yu-Chin Juan, and Chih-Jen Lin
Dept. of Computer Science, National Taiwan University
Taipei 106, Taiwan
{r01922139, d01944006, r01922136, cjlin}@csie.ntu.edu.tw

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 descent (SGD) is one of the most popular algorithms for matrix factorization. However, as a sequential approach, SGD is difficult to be parallelized for handling web-scale problems. In this paper, we develop a fast parallel SGD method, FPSGD, for shared memory systems. By dramatically reducing the cache-miss rate and carefully addressing the load balance of threads, FPSGD is more efficient than state-of-the-art parallel algorithms for matrix factorization.

Categories and Subject Descriptors

G.4 [**Mathematics of Computing**]: Mathematical Software—*Parallel and vector implementations*

Keywords

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

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 [3] and Netflix competition [1], 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 [7] 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 $r_{u,v} \simeq \mathbf{p}_u^T\mathbf{q}_v$, where $\mathbf{p}_u \in \mathbb{R}^k$ and $\mathbf{q}_v \in \mathbb{R}^k$ are respectively the uth column of P and the vth column of Q. The optimization problem is

$$\min_{P\in\mathbb{R}^{k\times m},Q\in\mathbb{R}^{k\times n}} \sum_{(u,v)\in R} (r_{u,v}-\mathbf{p}_u^T\mathbf{q}_v)^2+\lambda_P \|P\|_F^2+\lambda_Q \|Q\|_F^2 , \tag{1}$$

where k is the pre-specified number of latent factors, $\|\cdot\|_F$ is the Frobenius 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. Because $\sum_{(u,v)\in R} \left(r_{u,v} - \mathbf{p}_u^T\mathbf{q}_v\right)^2$ 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. [7] [14] [11]. Among them, stochastic gradient descent (SGD) is popularly used. For example, all of the top three teams in KDD Cup 2011 (track 1) employed SGD in their winning approaches.

The basic idea of SGD 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 [12] [6]. Once $r_{u,v}$ is chosen, the objective function in (1) is reduced to

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

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

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

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

where

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

is the error between the real and predicted ratings for the (u,v) entry, and γ is the learning rate. The overall procedure of SGD is to iteratively select an instance $r_{u,v}$, apply update rules (2)-(3), and may adjust the learning rate.

Although SGD 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 SGD under advanced architectures such as GPU, multi-core CPU or distributed clusters. Several parallel SGD approaches have been proposed (e.g., [15] [9] [8] [5] [4] [10]), although their focuses may be on other machine learning techniques rather than matrix factorization. In this work, we aim at developing an

RecSys'13, October 12–16, 2013, Hong Kong, China.

effective parallel SGD method for matrix factorization in a shared memory environment. Although for huge data a distributed system must be used, in many situations running SGD on a data set that can fit in memory is still very time consuming. For example, the data set of KDD Cup 2011 is less than 4GB and can be easily stored in the memory of one computer, but a single SGD iteration of implementing (2)-(3) takes more than 30 seconds. The overall SGD procedure can take hours. Therefore, an efficient parallel SGD to fully take the power of multi-core CPU can be very useful in practice.

Among existing parallel-SGD 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 [10] 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 $r_{u,v}$, $\mathbf{p}_u$ and $\mathbf{q}_v$ at the same time. That is, one thread is allowed to over-write another's work. DSGD [4] is another popular parallel SGD 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 SGD 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, and then Section 3 discusses difficulties in parallel SGD for matrix factorization. Our proposed method 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

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

where R is the rating matrix of the test set and $\hat{r}_{u,v}$ is the predicted rating value. Finally, Section 6 summarizes our work and gives future directions.

2. EXISTING PARALLELIZED STOCHASTIC GRADIENT DESCENT ALGORITHMS

Following the discussion in Section 1, in this section, we present two parallel SGD methods, HogWild [10] and DSGD [4], in detail.

Algorithm 1 HogWild's Algorithm

Require: number of threads s, R, P, and Q
1: **for each** thread i parallelly **do**
2: **while** true **do**
3: randomly select an instance $r_{u,v}$ from R
4: update corresponding $\mathbf{p}_u$ and $\mathbf{q}_v$ using (2)-(3), respectively
5: **end while**
6: **end for**

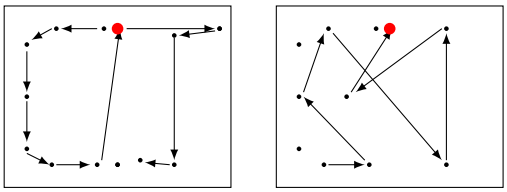


Figure 1: An example shows updating sequences of two threads in HogWild

2.1 HogWild

HogWild [10] 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 the same user identity and item identity. That is to say, iterations of SGD, (2)-(3), can be in parallel executed in different threads. HogWild drops the synchronization that prevents concurrent variable access via atomic operations, each of which is a series of CPU instructions that can not be interrupted. Even through the potential over-writing may occur, they 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 SGD 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 SGD updates using the same rating value $r_{i,j}$. From Algorithm 1, the operations include

- reading $r_{i,j}$, $\mathbf{p}_i$ and $\mathbf{q}_j$,
- 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 $r_{i,j}$, $\mathbf{p}_i$ and $\mathbf{q}_j$. 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 SGD is a sequential process, DSGD takes the property that some blocks of the rating matrix are mutually independent and their corresponding variables can be updated in parallel [4]. DSGD uniformly grids the rating matrix R into many sub-matrices (also called blocks), and

Algorithm 2 DSGD's Algorithm

Require: number of threads s, maximum iterations T, R, P, and Q

1: grid R into $s \times s$ blocks B and generate s patterns covering all blocks
2: **for** $t = \{1, \ldots, T\}$ **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** $b = \{1, \ldots, s\}$ 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**

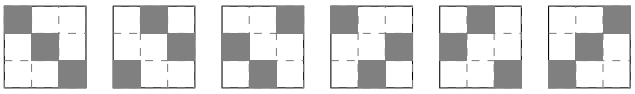
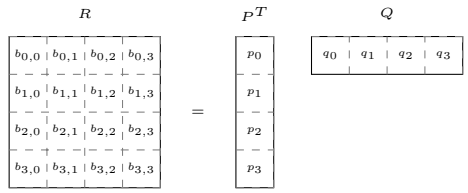


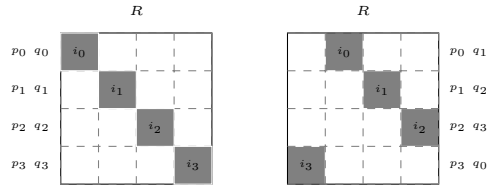
Figure 2: Patterns of independent blocks for a 3 by 3 grided matrix

applies SGD 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 [4] restricts 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 SGD 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.



(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 3: An illustration of the DSGD algorithm

3. PROBLEMS IN PARALLEL SGD METHODS FOR MATRIX FACTORIZATION

In this section, we point out that parallel SGD 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.

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.

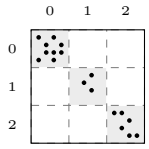


Figure 4: An example of the locking problem in DSGD. Each dot represents a rating; gray blocks indicate a set of independent blocks. Ratings in white blocks are not shown.

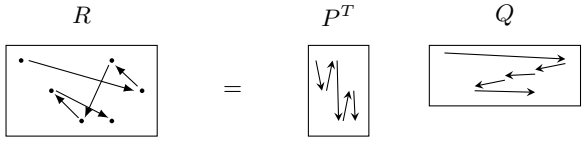


Figure 5: Random method to select rating instances for update.

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 SGD solvers for matrix factorization including HogWild and DSGD randomly pick instances from R (or from a block of R) for update. 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.

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 SGD method as *fast parallel SGD* (FPSGD). In Section 4.1, we discuss how FPSGD 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 FPSGD. 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,

(a) 2×2 blocks (b) 3×3 blocks

Figure 6: 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.

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:

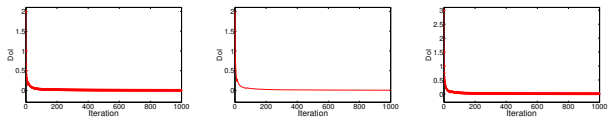
1. It is a free block.
2. Its number of past updates is the smallest among all free blocks.

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 FPSGD 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 6a, 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 6b, 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. (FPSGD does not have the concept of iterations. Here we call 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 three 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 set used in Figure 7, please refer to Section 5.1.



(a) MovieLens (b) Netflix (c) Yahoo!Music

Figure 7: DoI on three data sets. Eight threads are used, and R is grided into 9×9 blocks after being randomly shuffled.

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 very stable. Figure 9 gives an example showing that under two slightly different learning rates for SGD, 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, [2] shows 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 FPSGD, 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. The overall update sequence is neither user-by-user nor item-by-item, so some randomness is achieved. Figure 11 extends the running time comparison in Figure 9 to include FPSGD. We can see that FPSGD enjoys both fast convergence and excellent RMSE.

Some related methods have been investigated in [4], 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 FPSGD 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 (or item) identities before processing each block because an ordered method is applied within the block.

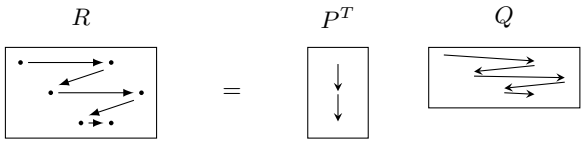
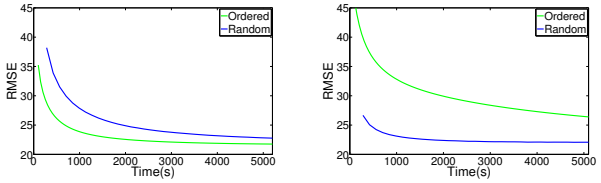


Figure 8: Ordered method to select rating instances for update.



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

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

4.3 Overview of FPSGD

Algorithm 3 gives the overall procedure of FPSGD. It first randomly shuffles R to avoid data imbalance as we mentioned in Section 4.1. Then it grids R into at least $(s+1) \times (s+1)$ blocks and sorts each block by user (or item) identities so that the partial random method discussed in Section 4.2 can be applied. 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 4 and Algorithm 5, 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.

4.4 Implementation Issues

FPSGD is implemented in C++. For the data set Yahoo!Music of about 250M ratings, using a typical machine (details specified in Section 5.1), FPSGD 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 Yahoo!Music, 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 12 shows the speedup after these techniques are applied in two data sets.

5. EXPERIMENTS

In this section, we compare FPSGD with other parallel

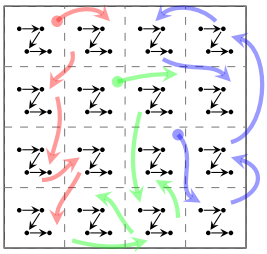
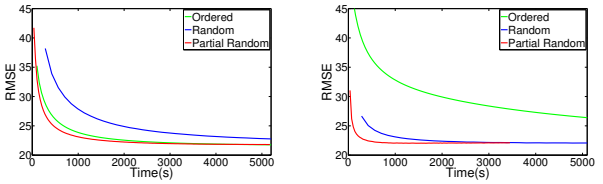


Figure 10: An illustration of the partial random method. Different colors indicate different threads. Each colored dot represents the initial block of a thread.



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

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

SGD methods mentioned in Section 2. Besides, we also compared with CCD++ [13], which is a parallel coordinate descent method for matrix factorization. It considers a feature-wise update sequence based on coordinate descent. At each iteration, one row of P and the corresponding row of Q are selected for update. Their parallelization relies on splitting the update of one row to several independent sub-tasks. CCD++ is currently considered the best parallel implementation in a shared-memory environment.

5.1 Settings

Data Sets: Three data sets, MovieLens,[1] Netflix and Yahoo!Music, are used for the experiments. For reproducibility, we consider the original training/test sets in our experiments (as 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 statistics of each data set is in Table 1.

Platform: We use a server with two Intel Xeon E5620 2.4GHz processors and 64GB memory. There are four cores in each processor.

Parameters: Table 1 lists the parameters used for each data set. We let $\lambda_P = \lambda_Q$ and follow the parameters chosen in [13] for Netflix and Yahoo!Music. For MovieLens, we set $\lambda_P = \lambda_Q = 0.05$. 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. From empirical experiences, the influence of the initial value of P and Q cannot be ignored, so we carefully set these values to be identical to [13].

In our platform, 8 physical cores are available, so we use 8 threads in all experiments. For FPSGD, Section 4 shows that $(s+1)\times(s+1)$ blocks are already enough for s threads,

[1] `http://www.grouplens.org/node/73`

Algorithm 3 The overall procedure of FPSGD

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

Algorithm 4 Scheduler of FPSGD

```
1: procedure GET_JOB
2:     b_x = NULL
3:     for all b in B do
4:         if b is non-free then
5:             continue
6:         else if b.ut ≤ b_x.ut then        ▷ ut: number of updates
7:             b_x=b
8:         end if
9:     end for
10:    return b_x
11: end procedure
12: procedure PUT_JOB(b)
13:    b.ut = b.ut + 1
14: end procedure
```

we use more blocks to ensure the randomness of blocks that are simultaneously processed. For Yahoo!Music and Movielens, R is grided into 32×32 blocks; for Netflix, R is grided into 16×16 blocks because the number of items 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[2] and CCD++[3] are publicly available. We reimplement HogWild under the same framework of our FPSGD 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 experiment.

5.2 Results

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 FPSGD.

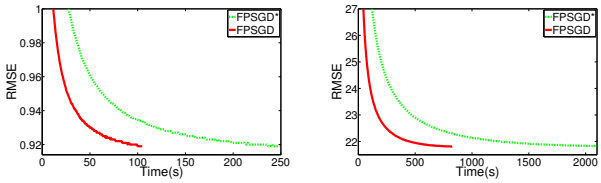
5.2.1 *The effectiveness to address the locking problem*

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 FPSGD, 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 FPSGD**.

[2] `http://hazy.cs.wisc.edu/hazy/victor/`
[3] `http://www.cs.utexas.edu/~rofuyu/libpmf/`

Algorithm 5 Working thread of FPSGD

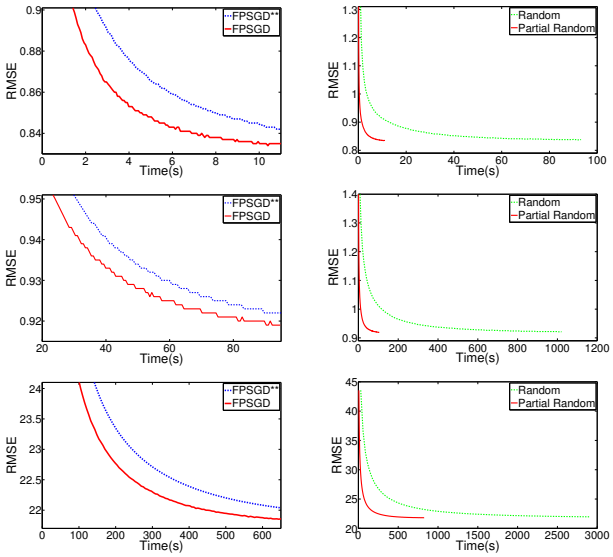
```
1: while true do
2:     get a block b from scheduler->get_job()
3:     process elements orderly in this block
4:     scheduler->put_job(b)
5: end while
```



(a) Netflix (b) Yahoo!Music

Figure 12: A comparison between two implementations of FPSGD in Netflix and Yahoo!Music. FPSGD implements the two techniques discussed in Section 4.4, while FPSGD* does not.

In Figure 13a, we compare the running time of FPSGD and FPSGD** on three data sets. As we can see, FPSGD runs much faster than FPSGD** because it does not suffer from the locking problem. In Table 2, we find that FPSGD runs 1.9 times, 1.3 times, and 1.6 times faster than FPSGD** on MovieLens, Netflix, and Yahoo!Music, respectively.



(a) (b)

Figure 13: (a) A comparison between FPSGD** and FPSGD. (b) A comparison between the partial random method and the random method. The data set from top to bottom are: MovieLens, Netflix, and Yahoo!Music.

5.2.2 *The effectiveness for discontinuous memory access*

We conduct experiments to investigate if the proposed partial random method can not only avoid memory discontinuity, but also keep good convergence.

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

Table 1: The statistics and parameters for each data set

Data Set	Method	Time (s)	RMSE
MovieLens	FPSGD**	18.71	0.835
	FPSGD	9.72	0.835
Netflix	FPSGD**	116.60	0.919
	FPSGD	90.37	0.919
Yahoo!Music	FPSGD**	728.23	21.985
	FPSGD	462.55	21.985

Table 2: A comparison between FPSGD** and FPSGD. See Section 5.2.1 for the details of FPSGD**.

In Figure 13b, we select rating instances in each block orderly (the partial random method) or randomly (the random method). Both methods converge to a similar RMSE. However, the training time of the partial random method is obviously faster then the random method.

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

Figure 14 presents the test RMSE and training time of various parallel matrix factorization methods.

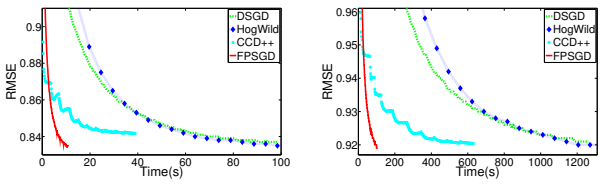
Among the three parallel SGD methods, FPSGD is faster than DSGD and HogWild. We believe that this result is because FPSGD is designed to effectively address issues mentioned in Section 3. However, we must note that for DSGD, it is also easy to incorporate similar techniques (e.g., the partial random method) to improve the performance.

From Figure 14, CCD++ is the fastest in the beginning, but becomes slower than FPSGD. Because the optimization problem of matrix factorization is non-convex and CCD++ is a more greedy setting than SGD 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, SGD-based methods may be able to escape from a local minimum because of the randomness.

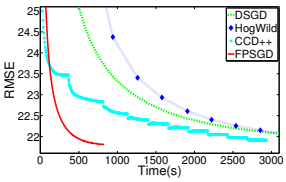
6. CONCLUSIONS AND FUTURE WORKS

To provide a better SGD solver for recommender systems, we will extend our algorithm to solve variants of matrix-factorization problems. To further reduce the cache-miss rate, we plan to investigate non-uniform splits of the rating matrix or other permutation methods such as Cuthill-McKee ordering.

In conclusion, we point out some computational bottlenecks in existing parallel SGD methods for shared-memory systems. We propose FPSGD to address these issues and confirm its effectiveness by experiments. The comparison shows that FPSGD is more efficient than state-of-the-art methods.



(a) MovieLens (b) Netflix



(c) Yahoo!Music

Figure 14: A comparison among the state-of-the-art parallel matrix factorization methods.

Based on this study, we develop the papckage `LIBMF`. It is available at `http://www.csie.ntu.edu.tw/~cjlin/libmf`

7. REFERENCES

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

[2] 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.

[3] 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.

[4] 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*, KDD '11, pages 69–77, New York, NY, USA, 2011. ACM.

[5] 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.

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

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.

[14] 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.

[15] 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.