Under consideration for publication in Knowledge and Information
Systems

Parallel Matrix Factorization for Recommender Systems

Hsiang-Fu Yu, Cho-Jui Hsieh, Si Si, and Inderjit S. Dhillon

Department of Computer Science, University of Texas at Austin, Austin, TX 78712, USA

Abstract. Matrix factorization, when the matrix has missing values, has become one of the leading techniques for recommender systems. To handle web-scale datasets with millions of users and billions of ratings, scalability becomes an important issue. Alternating Least Squares (ALS) and Stochastic Gradient Descent (SGD) are two popular approaches to compute matrix factorization. There has been a recent flurry of activity to parallelize these algorithms. However, due to the cubic time complexity in the target rank, ALS is not scalable to large-scale datasets. On the other hand, SGD conducts efficient updates but usually suffers from slow convergence that is sensitive to the parameters. Coordinate descent, a classical optimization approach, has been used for many other large-scale problems, but its application to matrix factorization for recommender systems has not been explored thoroughly. In this paper, we show that coordinate descent based methods have a more efficient update rule compared to ALS, and have faster and more stable convergence than SGD. We study different update sequences and propose the CCD++ algorithm, which updates rank-one factors one by one. In addition, CCD++ can be easily parallelized in both multi-core and distributed systems. We empirically show that CCD++ is much faster than ALS and SGD in both settings. As an example, with a synthetic dataset containing 14.6 billion ratings, on a distributed memory cluster with 32 processors, CCD++ is 24 times faster than SGD, while on a cluster with 256 processors, CCD++ is 18 times faster than SGD and 9.6 times faster than ALS.

Keywords: Recommender systems; Matrix factorization; Low rank approximation; Parallelization; Distributed Computing

1. Introduction

In a recommender system, we want to learn a model from past incomplete rating data such that each user's preference over all items can be estimated with the

Received xxx
Revised xxx
Accepted xxx

model. Matrix factorization was empirically shown to be a better model than traditional nearest-neighbor based approaches in the Netflix Prize competition. Since then there has been a great deal of work dedicated to the design of fast and scalable methods for large-scale matrix factorization problems [1, 2, 3].

Let $A \in \mathbb{R}^{m\times n}$ be the rating matrix in a recommender system, where m and n are the number of users and items, respectively. The matrix factorization problem for recommender systems is

$$\min_{\substack{W\in\mathbb{R}^{m\times k}\\ H\in\mathbb{R}^{n\times k}}} \sum_{(i,j)\in\Omega} (A_{ij} - \boldsymbol{w}_i^T \boldsymbol{h}_j)^2 + \lambda \left(\|W\|_F^2 + \|H\|_F^2\right), \tag{1}$$

where Ω is the set of indices for observed ratings, λ is the regularization parameter, and $\boldsymbol{w}_i^T$ and $\boldsymbol{h}_j^T$ are the i^{th} and the j^{th} row vectors of the matrices W and H, respectively. The goal of problem (1) is to approximate the incomplete matrix A by WH^T, where W and H are rank-k matrices. We can interpret this low-rank matrix factorization as a transformation that maps each user and each item to a feature vector (either $\boldsymbol{w}_i$ or $\boldsymbol{h}_j$) in a latent space $\mathbb{R}^k$. Then the interaction between the i^{th} user and the j^{th} item is measured by $\boldsymbol{w}_i^T\boldsymbol{h}_j$. Due to the fact that A is not fully observed, the well-known rank-k approximation by Singular Value Decomposition (SVD) cannot be directly applied to (1).

In recent recommender system competitions, we observe that alternating least squares (ALS) and stochastic gradient descent (SGD) appear to be the two most widely used methods for matrix factorization. ALS alternatively switches between updating W and updating H while fixing the other factor. Although the time complexity per iteration is $O(|\Omega|k^2 + (m+n)k^3)$, [1] shows that ALS is inherently suitable for parallelization. It is then not a coincidence that, ALS is the only parallel matrix factorization implementation for collaborative filtering in Apache Mahout.[1]

As mentioned in [2], SGD has become one of the most popular methods for matrix factorization in recommender systems due to its efficiency and simple implementation. The time complexity per iteration of SGD is $O(|\Omega|k)$, which is lower than ALS. However, as compared to ALS, SGD usually needs more iterations to obtain a good enough model, and its performance is sensitive to the choice of the learning rate. Furthermore, unlike ALS, parallelization of SGD is challenging. A variety of schemes have been proposed to parallelize SGD [4, 5, 6, 7, 8].

This paper aims to design an efficient and easily parallelizable method for matrix factorization in large-scale recommender systems. Recently, [9] and [10] have showed that coordinate descent methods are effective for nonnegative matrix factorization (NMF). This motivates us to investigate coordinate descent approaches for (1). In this paper, we propose a coordinate descent based method, CCD++, which has fast running time and can be easily parallelized to handle data of various scales. The main contributions of this paper are:

- We propose a scalable and efficient coordinate descent based matrix factorization method CCD++. The time complexity per iteration of CCD++ is lower than that of ALS, and it achieves faster convergence than SGD.
- We show that CCD++ can be easily applied to problems of various scales in both shared-memory multi-core and distributed systems.

[1] `http://mahout.apache.org/`

Notation. The following notation is used throughout the paper. We denote matrices by uppercase letters and vectors by bold-faced lowercase letters. A_{ij} denotes the (i, j) entry of the matrix A. We use Ω_i to denote the column indices of observed ratings in the i^{th} row, and $\bar{\Omega}_j$ to denote the row indices of observed ratings in the j^{th} column. We denote the i^{th} row of W by $\boldsymbol{w}_i^T$, and the t^{th} column of W by $\bar{\boldsymbol{w}}_t \in \mathbb{R}^m$:

$$W = \begin{bmatrix} \vdots \\ \boldsymbol{w}_i^T \\ \vdots \end{bmatrix} = \begin{bmatrix} \cdots & \bar{\boldsymbol{w}}_t & \cdots \end{bmatrix}.$$

Thus, both w_{it} (i.e., the t^{th} element of $\boldsymbol{w}_i$) and $\bar{w}_{ti}$ (i.e., the i^{th} element of $\bar{\boldsymbol{w}}_t$) denote the same entry, W_{it}. For H, we use similar notation $\boldsymbol{h}_j$ and $\bar{\boldsymbol{h}}_t$.

The rest of the paper is organized as follows. An introduction to ALS and SGD is given in Section 2. We then present our coordinate descent approaches in Section 3. In Section 4.3, we show the parallelization of CCD++ and conduct the scalability analysis under different parallel computing environments. We then present experimental results in Section 5. Finally, we show an extension of CCD++ to handle L1-regularization in Section 6 and conclude in Section 7.

2. Related Work

As mentioned in [2], the two standard approaches to approximate the solution of problem (1) are ALS and SGD. In this section we briefly introduce these methods and discuss recent parallelization approaches.

2.1. Alternating Least Squares

Problem (1) is intrinsically a non-convex problem; however, when fixing either W or H, (1) becomes a quadratic problem with a globally optimal solution. Based on this idea, ALS alternately switches between optimizing W while keeping H fixed, and optimizing H while keeping W fixed. Thus, ALS monotonically decreases the objective function value in (1) until convergence.

Under this alternating optimization scheme, (1) can be further separated into many independent least squares subproblems. Specifically, if we fix H and minimize over W, the optimal $\boldsymbol{w}_i^*$ can be obtained independently of other rows of W by solving the least squares subproblem:

$$\min_{\boldsymbol{w}_i} \sum_{j \in \Omega_i} (A_{ij} - \boldsymbol{w}_i^T \boldsymbol{h}_j)^2 + \lambda \|\boldsymbol{w}_i\|^2, \tag{2}$$

which leads to the closed form solution

$$\boldsymbol{w}_i^* = (H_{\Omega_i}^T H_{\Omega_i} + \lambda I)^{-1} H^T \boldsymbol{a}_i, \tag{3}$$

where $H_{\Omega_i}^T$ is the sub-matrix with columns $\{\boldsymbol{h}_j : j \in \Omega_i\}$, and $\boldsymbol{a}_i^T$ is the i^{th} row of A with missing entries filled by zeros. To compute each $\boldsymbol{w}_i^*$, ALS needs $O(|\Omega_i|k^2)$ time to form the $k \times k$ matrix $H_{\Omega_i}^T H_{\Omega_i}$ and additional $O(k^3)$ time to solve the least squares problem. Thus, the time complexity of a full ALS iteration (i.e., updating W and H once) is $O(|\Omega|k^2 + (m + n)k^3)$.

In terms of parallelization, [1] points out that ALS can be easily parallelized in a row-by-row manner as each row of W or H can be updated independently. However, parallelization of ALS in a distributed system when W or H exceeds the memory capacity of a computation node is more involved. More details are discussed in Section 4.3.

2.2. Stochastic Gradient Descent

Stochastic gradient descent (SGD) is widely used in many machine learning problems [11]. SGD has also been shown to be effective for matrix factorization [2]. In SGD, for each update, a rating (i, j) is randomly selected from Ω, and the corresponding variables $\boldsymbol{w}_i$ and $\boldsymbol{h}_j$ are updated by

$$\boldsymbol{w}_i \leftarrow \boldsymbol{w}_i - \eta(\frac{\lambda}{|\Omega_i|}\boldsymbol{w}_i - R_{ij}\boldsymbol{h}_j),$$

$$\boldsymbol{h}_j \leftarrow \boldsymbol{h}_j - \eta(\frac{\lambda}{|\bar{\Omega}_j|}\boldsymbol{h}_j - R_{ij}\boldsymbol{w}_i),$$

where $R_{ij} = A_{ij} - \boldsymbol{w}_i^T\boldsymbol{h}_j$, and η is the learning rate. For each rating A_{ij}, SGD needs $O(k)$ operations to update $\boldsymbol{w}_i$ and $\boldsymbol{h}_j$. If we define $|\Omega|$ consecutive updates as one iteration of SGD, the time complexity per SGD iteration is thus only $O(|\Omega|k)$. As compared to ALS, SGD is faster in terms of the time complexity for one iteration, but typically it needs more iterations than ALS to achieve a good enough model.

However, conducting several SGD updates in parallel directly might raise an overwriting issue as the updates for the ratings in the same row or the same column of A involve the same variables. Moreover, traditional convergence analysis of standard SGD mainly depends on its sequential update property. These issues make parallelization of SGD a challenging task. Recently, several update schemes to parallelize SGD have been proposed. For example, "delayed updates" are proposed in [4] and [12], while [7] uses a bootstrap aggregation scheme. A lock-free approach called HogWild is investigated in [8], in which the overwriting issue is ignored based on the intuition that the probability of updating the same row of W or H is small when A is sparse. The authors of [8] also show that HogWild is more efficient than the "delayed update" approach in [4]. For matrix factorization, [5] and [6] propose Distributed SGD (DSGD)[2] which partitions A into blocks and updates a set of independent blocks in parallel at the same time. Thus, DSGD can be regarded as an exact SGD implementation with a specific ordering of updates.

Another issue with SGD is that the convergence is highly sensitive to the learning rate η. In practice, the initial choice and adaptation strategy for η are crucial issues when applying SGD to matrix factorization problems. As the learning rate issue is beyond the scope of this paper, here we only briefly discuss how the learning rate is adjusted in HogWild and DSGD. In HogWild [8], η is reduced by multiplying a constant $\beta \in (0, 1)$ at each iteration. In DSGD, [5] proposes using the "bold driver" scheme, in which, at each iteration, η is increased by a small proportion (5% is used in [5]) when the function value decreases; when the

[2] In [6], the name "Jellyfish" is used

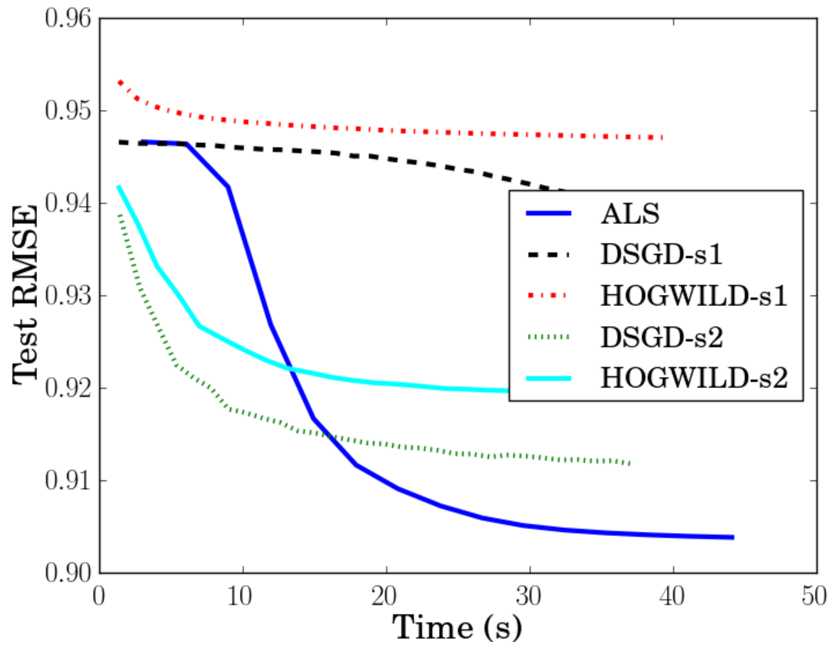


Fig. 1. Comparison between ALS, DSGD, and HogWild on the movielens10m dataset with $k = 40$ on a 8-core machine (-s1 and -s2 stand for different initial learning rates).

value increases, η is drastically decreased by a large proportion (50% is used in [5]).

We close this section with a comparison of ALS,[3] DSGD,[4] and HogWild[5] on the movielens10m dataset with $k = 40$ and $\lambda = 0.1$ (more details on the dataset are given later in Table 1 of Section 5). Here we conduct the comparison on an Intel Xeon X5570 8-core machine with 8 MB L2-cache and enough memory. All 8 cores are utilized for each method.[6] Figure 1 shows the comparison; "-s1" and "-s2" denote two choices of the initial η.[7] The reader might notice that the performance difference between ALS and DSGD is not as large as in [5]. The reason is that the parallel platform used in our comparison is different from that used in [5], which is a modified Hadoop distributed system.

From Figure 1, we first observe that the performance of both DSGD and HogWild is sensitive to the choice of η. In contrast, ALS, a parameter-free approach, is more stable, albeit it has higher time complexity per iteration than SGD. Next, we can see that DSGD converges slightly faster than HogWild with both initial η's. Given the fact that the computation time per iteration of DSGD

[3] Intel MKL is used in our implementation of ALS.
[4] We implement a multi-core version of DSGD according to [5].
[5] HogWild is downloaded from `http://research.cs.wisc.edu/hazy/victor/Hogwild/` and modified to start from the same initial point as ALS and DSGD.
[6] In HogWild, seven cores are used for SGD updates, and one core is used for random shuffle.
[7] for -s1, initial $\eta = 0.001$; for -s2, initial $\eta = 0.05$.

is similar to that of HogWild (as DSGD is also a lock-free scheme), we believe that there are two possible explanations: 1) the "bold driver" approach used in DSGD is more stable than the exponential decay approach used in HogWild; 2) the variable overwriting might slow down the convergence of HogWild.

3. Coordinate Descent Approaches

Coordinate descent is a classic and well-studied optimization technique [13, Section 2.7]. Recently it has been successfully applied to various large-scale problems such as linear SVMs [14], maximum entropy models [15], NMF problems [9, 10], and sparse inverse covariance estimation [16]. The basic idea of coordinate descent is to update a single variable at a time while keeping others fixed. There are two key components in coordinate descent methods: one is the update rule used to solve each one-variable subproblem, and the other is the update sequence of variables.

In this section, we apply coordinate descent to attempt to solve (1). We first form the one-variable subproblem and derive the update rule. Based on the rule, we investigate two sequences to update variables: item/user-wise and feature-wise.

3.1. The Update Rule

If only one variable w_{it} is allowed to change to z while fixing all other variables, we are able to formulate the following one-variable subproblem as

$$\min_z \; f(z) = \sum_{j\in\Omega_i} \left(A_{ij} - (\boldsymbol{w}_i^T\boldsymbol{h}_j - w_{it}h_{jt}) - zh_{jt}\right)^2 + \lambda z^2. \tag{4}$$

As $f(z)$ is a univariate quadratic function, the unique solution z^* to (4) can be easily found:

$$z^* = \frac{\sum_{j\in\Omega_i}(A_{ij} - \boldsymbol{w}_i^T\boldsymbol{h}_j + w_{it}h_{jt})h_{jt}}{\lambda + \sum_{j\in\Omega_i} h_{jt}^2}. \tag{5}$$

Direct computation of z^* via (5) from scratch takes $O(|\Omega_i|k)$ time. For large k, we can accelerate the computation by maintaining the residual matrix R,

$$R_{ij} \equiv A_{ij} - \boldsymbol{w}_i^T\boldsymbol{h}_j, \; \forall (i,j) \in \Omega.$$

In terms of R_{ij}, the optimal z^* can be obtained by:

$$z^* = \frac{\sum_{j\in\Omega_i}(R_{ij} + w_{it}h_{jt})h_{jt}}{\lambda + \sum_{j\in\Omega_i} h_{jt}^2}. \tag{6}$$

When R is available, computing z^* by (6) only costs $O(|\Omega_i|)$ time. After z^* is obtained, w_{it} and R_{ij} $\forall j \in \Omega_i$ can also be updated in $O(|\Omega_i|)$ time via

$$R_{ij} \leftarrow R_{ij} - (z^* - w_{it})h_{jt}, \; \forall j \in \Omega_i, \tag{7}$$

$$w_{it} \leftarrow z^*. \tag{8}$$

Therefore, if we maintain the residual matrix R, the time complexity of each single variable update is reduced from $O(|\Omega_i|k)$ to $O(|\Omega_i|)$. Similarly, the update rules for each variable in H, h_{jt} for instance, can be derived as

$$R_{ij} \leftarrow R_{ij} - (s^* - h_{jt})w_{it}, \ \forall i \in \bar{\Omega}_j, \tag{9}$$

$$h_{jt} \leftarrow s^*, \tag{10}$$

where s^* can be obtained by either:

$$s^* = \frac{\sum_{i\in\bar{\Omega}_j}(A_{ij} - \boldsymbol{w}_i^T\boldsymbol{h}_j + w_{it}h_{jt})w_{it}}{\lambda + \sum_{i\in\bar{\Omega}_j} w_{it}^2}, \tag{11}$$

or

$$s^* = \frac{\sum_{i\in\bar{\Omega}_j}(R_{ij} + w_{it}h_{jt})w_{it}}{\lambda + \sum_{i\in\bar{\Omega}_j} w_{it}^2}. \tag{12}$$

With update rules (7)-(10), we are able to apply any update sequence over variables in W and H. We now investigate two main sequences: item/user-wise and feature-wise update sequences.

3.2. Item/User-wise Update: CCD

First, we consider the item/user-wise update sequence, which updates the variables corresponding to either an item or a user at the same time.

ALS can be viewed as a method which adopts this update sequence. As mentioned in Section 2.1, ALS switches the updating between W and H. To update W when fixing H or vice versa, ALS solves many k-variable least squares subproblems. Each subproblem corresponds to either an item or a user. That is, ALS cyclically updates variables with the following sequence:

$$\overbrace{\boldsymbol{w}_1, \ldots, \boldsymbol{w}_m}^{W}, \overbrace{\boldsymbol{h}_1, \ldots, \boldsymbol{h}_n}^{H}.$$

In ALS, the update rule in (3) involves forming a $k \times k$ Hessian matrix and solving a least squares problem which takes $O(k^3)$ time. However, it is not necessary to solve all subproblems (2) exactly in the early stages of the algorithm. Thus, [17] proposed a cyclic coordinate descent method (CCD), which is similar to ALS with respect to the update sequence. The only difference is the update rules. In CCD, we update $\boldsymbol{w}_i$ by applying (8) over all elements of $\boldsymbol{w}_i$ (i.e., $w_{i1}, \ldots, w_{ik}$) with a finite number of cycles. The entire update sequence of one iteration in CCD is

$$\overbrace{\underbrace{w_{11}, \ldots, w_{1k}}_{\boldsymbol{w}_1}, \ldots, \underbrace{w_{m1}, \ldots, w_{mk}}_{\boldsymbol{w}_m}}^{W}, \overbrace{\underbrace{h_{11}, \ldots, h_{1k}}_{\boldsymbol{h}_1}, \ldots, \underbrace{h_{n1}, \ldots, h_{nk}}_{\boldsymbol{h}_n}}^{H}. \tag{13}$$

Algorithm 1 describes the CCD procedure with T iterations. Note that if we set the initial W to 0, then the initial R is exactly equal to A, so no extra effort is needed to initialize R.

As mentioned in Section 3.1, the update cost for each variable in W and H, taking w_{it} and h_{jt} for instance, is just $O(|\Omega_i|)$ or $O(|\bar{\Omega}_j|)$. If we define one iteration in CCD as updating all variables in W and H once, the time complexity

Algorithm 1 CCD Algorithm [17]

```
Input: $A$, $W$, $H$, $\lambda$, $k$, $T$
 1: Initialize $W = 0$ and $R = A$.
 2: for $iter = 1, 2, \dots, T$ do
 3:     for $i = 1, 2, \dots, m$ do                    ▷ Update $W$.
 4:         for $t = 1, 2, \dots, k$ do
 5:             Obtain $z^*$ using (6).
 6:             Update $R$ and $w_{it}$ using (7) and (8).
 7:         end for
 8:     end for
 9:     for $j = 1, 2, \dots, n$ do                    ▷ Update $H$.
10:         for $t = 1, 2, \dots, k$ do
11:             Obtain $s^*$ using (12).
12:             Update $R$ and $h_{jt}$ using (9) and (10).
13:         end for
14:     end for
15: end for
```

per iteration for CCD is thus

$$O\left(\left(\sum_i |\Omega_i| + \sum_j |\bar{\Omega}_j|\right) k\right) = O(|\Omega| k).$$

We can see that an iteration of CCD is faster than an iteration of ALS when $k > 1$, because ALS requires $O(|\Omega|k^2 + (m+n)k^3)$ time to update at each iteration. Of course, each iteration of ALS makes more progress; however, at early stages of this algorithm, it is not clear that this extra progress helps.

Instead of cyclically updating through $w_{i1}, \dots, w_{ik}$, one may think of a greedy update sequence that sequentially updates the variable that decreases the objective function the most. In [10], a greedy update sequence is applied to solve the NMF problem in an efficient manner by utilizing the property that all subproblems in NMF share the same Hessian. However, unlike NMF, each subproblem (2) of problem (1) has a potentially different Hessian as $\Omega_{i_1} \neq \Omega_{i_2}$ for $i_1 \neq i_2$ in general. Thus, if the greedy coordinate descent (GCD) method proposed by [10] is applied to (1), m different Hessians are required to update W, and n Hessians are required to to update H. The Hessian for $\boldsymbol{w}_i$ and $\boldsymbol{h}_j$ needs $O(|\Omega_i|k^2)$ and $O(|\bar{\Omega}_j|k^2)$, respectively. The total time complexity of GCD to update W and H once is $O(|\Omega|k^2)$ operations per iteration, which is the same complexity as ALS.

3.3. Feature-wise Update: CCD++

The factorization WH^T can be represented as a summation of k outer products:

$$A \approx WH^T = \sum_{t=1}^{k} \bar{\boldsymbol{w}}_t \bar{\boldsymbol{h}}_t^T, \tag{14}$$

where $\bar{\boldsymbol{w}}_t \in \mathbb{R}^m$ is the tth column of W, and $\bar{\boldsymbol{h}}_t \in \mathbb{R}^n$ is the tth column of H. From the perspective of the latent feature space, $\bar{\boldsymbol{w}}_t$ and $\bar{\boldsymbol{h}}_t$ correspond to the tth latent feature.

Algorithm 2 CCD++ Algorithm

```
Input: $A$, $W$, $H$, $\lambda$, $k$, $T$
 1: Initialize $W = 0$ and $R = A$.
 2: for $iter = 1, 2, \dots$ do
 3:     for $t = 1, 2, \dots, k$ do
 4:         Construct $\hat{R}$ by (16).
 5:         for $inneriter = 1, 2, \dots, T$ do        ▷ $T$ CCD iterations for (17).
 6:             Update $\boldsymbol{u}$ by (18).
 7:             Update $\boldsymbol{v}$ by (19).
 8:         end for
 9:         Update $(\bar{\boldsymbol{w}}_t, \bar{\boldsymbol{h}}_t)$ and $R$ by (20) and (21).
10:     end for
11: end for
```

This leads us to our next coordinate descent method, CCD++. At each time, we select a specific feature t and conduct the update

$$(\bar{\boldsymbol{w}}_t, \bar{\boldsymbol{h}}_t) \leftarrow (\boldsymbol{u}^*, \boldsymbol{v}^*),$$

where $(\boldsymbol{u}^*, \boldsymbol{v}^*)$ is obtained by solving the following subproblem:

$$\min_{\boldsymbol{u}\in\mathbb{R}^m, \boldsymbol{v}\in\mathbb{R}^n} \sum_{(i,j)\in\Omega} \left(R_{ij} + \bar{w}_{ti}\bar{h}_{tj} - u_i v_j\right)^2 + \lambda(\|\boldsymbol{u}\|^2 + \|\boldsymbol{v}\|^2). \tag{15}$$

If we define

$$\hat{R}_{ij} = R_{ij} + \bar{w}_{ti}\bar{h}_{tj}, \ \forall (i,j) \in \Omega, \tag{16}$$

(15) can be rewritten as:

$$\min_{\boldsymbol{u}\in\mathbb{R}^m, \boldsymbol{v}\in\mathbb{R}^n} \sum_{(i,j)\in\Omega} (\hat{R}_{ij} - u_i v_j)^2 + \lambda(\|\boldsymbol{u}\|^2 + \|\boldsymbol{v}\|^2), \tag{17}$$

which is exactly the rank-one matrix factorization problem (1) for the matrix $\hat{R}$. Thus we can apply CCD on (17) to obtain an approximation by alternatively updating $\boldsymbol{u}$ and updating $\boldsymbol{v}$. Note that we can simply use $(\bar{\boldsymbol{w}}_t, \bar{\boldsymbol{h}}_t)$ as the initial point of $(\boldsymbol{u}, \boldsymbol{u})$. The update sequence for $\boldsymbol{u}$ and $\boldsymbol{v}$ is

$$u_1, u_2, \dots, u_m, v_1, v_2, \dots, v_n.$$

When the rank is equal to one, (5) and (6) have the same complexity. Thus, during the CCD iterations to update u_i and v_j, z^* and s^* can be directly obtained by (5) and (11) without additional residual maintenance. The update rules for $\boldsymbol{u}$ and $\boldsymbol{v}$ in each CCD iteration become as follows.

$$u_i \leftarrow \frac{\sum_{j\in\Omega_i} \hat{R}_{ij} v_j}{\lambda + \sum_{j\in\Omega_i} v_j^2}, \ i = 1, \dots, m, \tag{18}$$

$$v_j \leftarrow \frac{\sum_{i\in\bar{\Omega}_j} \hat{R}_{ij} u_j}{\lambda + \sum_{i\in\bar{\Omega}_j} u_i^2}, \ j = 1, \dots, n. \tag{19}$$

After obtaining $(\boldsymbol{u}^*, \boldsymbol{v}^*)$, we can update $(\bar{\boldsymbol{w}}_t, \bar{\boldsymbol{h}}_t)$ and R by

$$(\bar{\boldsymbol{w}}_t, \bar{\boldsymbol{h}}_t) \leftarrow (\boldsymbol{u}^*, \boldsymbol{v}^*). \tag{20}$$

$$R_{ij} \leftarrow \hat{R}_{ij} - u_i^* v_j^*, \ \forall (i,j) \in \Omega, \tag{21}$$

The update sequence for each outer iteration of CCD++ is

$$\bar{\boldsymbol{w}}_1, \bar{\boldsymbol{h}}_1, \ldots, \bar{\boldsymbol{w}}_t, \bar{\boldsymbol{h}}_t, \ldots, \bar{\boldsymbol{w}}_k, \bar{\boldsymbol{h}}_k. \tag{22}$$

We summarize CCD++ in Algorithm 2. A similar procedure with the feature-wise update sequence is also used in [18] to avoid the over-fitting issue in recommender systems.

Each time when the t^{th} feature is selected, CCD++ consists of the following parts to update $(\bar{\boldsymbol{w}}_t, \bar{\boldsymbol{h}}_t)$: constructing $O(|\Omega|)$ entries of $\hat{R}$, conducting T CCD iterations to solve (17), updating $(\bar{\boldsymbol{w}}_t, \bar{\boldsymbol{h}}_t)$ by (20), and maintaining $|\Omega|$ residual entries by (21). Because each CCD iteration in Algorithm 2 costs only $O(|\Omega|)$ operations, the time complexity per iteration for CCD++, where all k features are updated by T CCD iterations, is $O(|\Omega|kT)$.

At first glance, the only difference between CCD++ and CCD appears to be their different update sequence. However, such difference might affect the convergence. A similar update sequence has also been considered for NMF problems. [19] observes that such feature-wise update sequence leads to faster convergence than other sequences on moderate-scale matrices. However, for large-scale NMF problems, when all entries are known, the residual matrix becomes a $m \times n$ dense matrix, which is too large to maintain. Thus [9, 10] utilize the property that all subproblems share a single Hessian because no missing values exist in NMF problems. Based on the property, they develop a technique such that variables can be efficiently updated without maintenance of the residual.

Due to the large number of missing entries in A, problem (1) does not have the nice property as NMF problems. However, as a result of the sparsity of observed entries, the residual maintenance is affordable for problem (1) with a large-scale A. Furthermore, the feature-wise update sequence might even bring faster convergence as it does for NMF problems.

3.4. Exact Memory Storage and Operation Counts

Based on the analysis in Sections 3.2 and 3.3, we know that, at each iteration, CCD, and CCD++ share the same asymptotic time complexity, $O(|\Omega|k)$. To clearly see the difference between these two methods, we do an exact count of the number of floating operations (flops) for each method.

Rating Storage. Exact counting of the number of operations depends on how the $m \times n$ residual matrix (R) is stored in the memory as almost all updating rules used in CCD and CCD++ need to access some entries of R. If the entire R including both observed and missing entries can be stored in a dense format, random access to any entry R_{ij} can be regarded as in a constant time. However, memory is usually not enough to store all $m \times n$ entries of R when m and n are large. We are only affordable to use sparse matrix formats to store only observed entries of R (i.e., Ω), which are usually sparse in real-world applications. Here we consider two commonly used formats for sparse matrices: Compressed Spares Row format (CSR), where all entries in the same row are stored adjacently in the memory, and Compressed Sparse Column format (CSC), where entries in the same column are stored adjacently.

As we can see, updating rules (6) and (18) favor CSR because both rules need to access all observed entries in the same row, while updating rules (12) and (19) prefer CSC as they require to access all observed entries in the same column. In fact, if there is only one copy of R in CSC, iterating over entries in a single row (i.e., $R_{ij}\ \forall j \in \Omega_i$) needs at least m operations to identify the location of entries, which is more than $|\bar{\Omega}_j|$ operations required by a storage in CSC. Therefore, to access both rows and columns in R efficiently, we maintain two copies of R in the memory: one is in CSR, and the other is in CSC.

As $\hat{R}$ exists only when solving each subproblem (17), there is no need to have another space to store $\hat{R}$. Here is a simple method such that $\hat{R}$ and R can share the same memory space: replacing Line 4 in Algorithm 2 by

$$R_{ij} \leftarrow R_{ij} + \bar{w}_{ti}\bar{h}_{tj},\ \forall (i,j) \in \Omega,$$

and substituting the updating rule (21) on Line 9 by

$$R_{ij} \leftarrow R_{ij} - \bar{w}_{ti}\bar{h}_{tj},\ \forall (i,j) \in \Omega.$$

In summery, both CCD and CCD++ store two copies of the observed ratings.

Exact Operation Counting. In CCD, updating rules (6) and (12) take $6|\Omega_i|$ and $6|\bar{\Omega}_j|$ flops, respectively. As the residual R is stored in two formats, updating rules (7) and (9) cost $2 \times 3|\Omega_i|$ and $2 \times 3|\bar{\Omega}_j|$ flops, respectively. As a result, one CCD iteration (i.e., $(m+n)k$ variable updates) takes

$$\left(\left(\sum_{i=1}^{m}(6+6)|\Omega_i|\right) + \left(\sum_{j=1}^{n}(6+6)|\bar{\Omega}_j|\right)\right) \times k = 24|\Omega|k \text{ flops.} \tag{23}$$

In CCD++, both the $\hat{R}$ construction (16) and the residual updating rule (21) require $2 \times 2|\Omega|$ flops due to the two copies of R. Updating rules (18) and (19) cost $4|\Omega_i|$ and $4|\bar{\Omega}_j|$ flops, respectively. Therefore, one CCD++ iteration with T inner CCD iterations (i.e., $(m+n)kT$ variable updates) takes

$$\left(4|\Omega| + T\left(\sum_{i=1}^{m}4|\Omega_i| + \sum_{j=1}^{n}4|\bar{\Omega}_j|\right) + 4|\Omega|\right) \times k = 8|\Omega|k(T+1) \text{ flops.} \tag{24}$$

Based on the results of the exact counting, if $T = 1$, where one iteration of both CCD and CCD++ updates the same number of variables, $(m+n)k$, CCD++ is 1.5 faster than CCD. If $T > 1$, the ratio between the flops required by CCD and CCD++ to update the same number of variables, $\frac{3T}{T+1}$, can be even larger.

3.5. An Adaptive Technique to Accelerate CCD++

In this section, we investigate how to accelerate CCD++ by controlling T, the number of CCD iterations, for each subproblem (17). The approaches [9, 19], which apply the feature-wise update sequence to solve NMF problems, consider only one iteration for each subproblem. However, CCD++ can be slightly more efficient when $T > 1$ due to the benefit brought by the "delayed residual update."

Note that R and $\hat{R}$ are fixed during CCD iterations for each rank-one approximation (17). Thus, (16), the construction of $\hat{R}$, and (21), the residual update, are only conducted only once for each subproblem. Based on the exact operation counting conducted in (24), to update $(m+n)kT$ variables, (16) and (21) contribute $8|\Omega|k$ flops, while (18) and (19) contribute $8|\Omega|kT$ flops. Therefore, for CCD++, the ratio of the computation effort spend on the residual maintenance over that spent on real variable updating is $\frac{1}{T}$. As a result, given the same number of variable updates, CCD++ with T CCD iterations is

$$\frac{\text{flops of } T \text{ CCD++ iterations with 1 CCD iteration}}{\text{flops of 1 CCD++ iterations with } T \text{ CCD iterations}} = \frac{8|\Omega|k(1+1)T}{8|\Omega|k(T+1)} = \frac{2T}{T+1}$$

times faster than CCD++ with only one CCD iteration. Moreover, the more CCD iterations we use, the better the approximation to subproblem (17). Hence, a direct approach to accelerate CCD++ is to increase T. On the other hand, a large and fixed T might result in too much effort on a single subproblem.

We propose a technique to adaptively determine when to stop CCD iterations based on the relative function value reduction at each CCD iteration. At each outer iteration of CCD++, we maintain the maximal function value reduction from past CCD iterations, $d^{\max}$. Once the function value reduction at the current CCD iteration is less than $\epsilon d^{\max}$, we stop CCD iterations, update the residual by (21), and switch to the next subproblem, where $\epsilon \leq 1$ is a small positive ratio such as 10^{-3}. It is not hard to see that the function value reduction at each CCD iteration for subproblem (17) can be efficiently obtained by accumulating reductions from the update of each single variable. For example, updating u_i to the optimal u_i^* of

$$\min_{u_i} f(u_i) = \sum_{j\in\Omega_i} (\hat{R}_{ij} - u_i v_j)^2 + \lambda u_i^2,$$

decreases the function by

$$f(u_i) - f(u_i^*) = (u_i^* - u_i)^2 \left(\lambda + \sum_{j\in\Omega_i} v_j^2\right),$$

where the second term is exact the denominator of the updating rule (18). As a result, the function value reduction can be obtained without extra effort.

We close this section by a comparison between CCD and CCD++. Here we include four settings with the netflix dataset on a machine with enough memory:

– CCD: iter/user-wise CCD,
– CCD++T1: CCD++ with fixed $T = 1$,
– CCD++T5: CCD++ with fixed $T = 5$,
– CCD++F: CCD++ with our adaptive approach to control T based on the function value reduction ($\epsilon = 10^{-3}$ is used).

From Figure 2, we clearly observe that the feature-wise update approach CCD++, even when $T = 1$, is faster than CCD, which confirms our analysis above and the observation for NMF in [19]. We also observe that larger T improves CCD++ in the early stages, though it also results in too much effort during some periods (e.g., the period from 100s to 180s in Figure 2). Such periods suggest that an

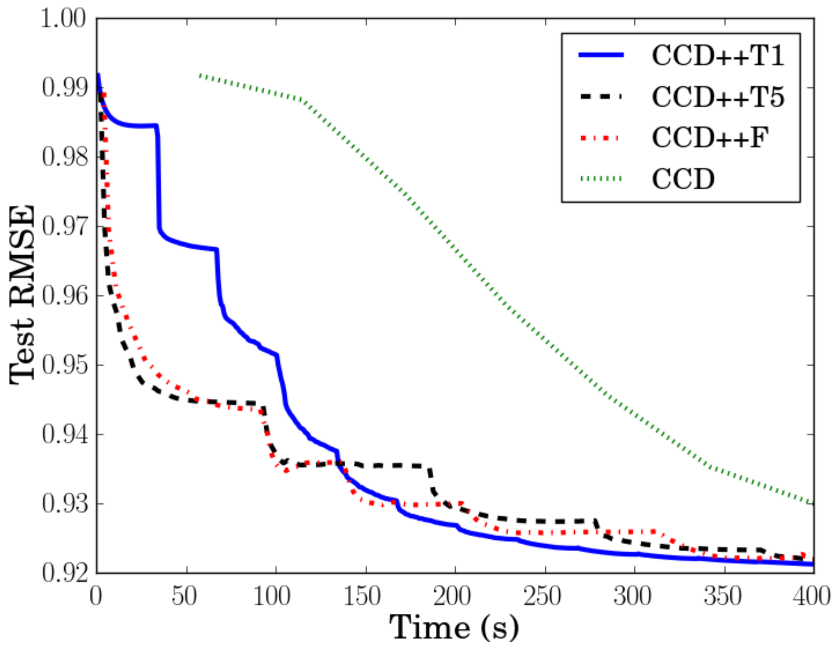


Fig. 2. Comparison between CCD and CCD++ on netflix dataset. Clearly, CCD++, the feature-wise update approach, is seen to have faster convergence than CCD, the item/user-wise update approach.

early termination might help. We also notice that our technique to adaptively control T can slightly shortens such periods and improve the performance.

4. Parallelization of CCD++

With the exponential growth of dyadic data on the web, scalability becomes an issue when applying state-of-the-art matrix factorization approaches to large-scale recommender systems. Recently, there has been a growing interest on addressing the scalability problem by using parallel and distributed computing for existing matrix factorization algorithms. Both CCD and CCD++ can be easily parallelized. Due to the similarity with ALS, CCD can be parallelized in the same way as ALS in [5]. For CCD++, we propose two versions: one version for multi-core shared memory systems and the other for distributed systems.

It is important to select the appropriate parallel environment based on the scale of the recommender system. Specifically, when the matrices A, W, and H can be loaded in the memory of a single machine, and we consider a distributed system as the parallel environment, the communication among machines dominates the entire procedure. In this case, a multi-core shared memory system is a better parallel environment. However, when the data/variables exceed the memory capacity of a single machine, a distributed system, in which data/variables are distributed across different machines, is required to handle problems of this scale. In the following sections, we demonstrate how to parallelize CCD++ under both these parallel environments.

4.1. CCD++ in Multi-core Systems

In this section we discuss the parallelization of CCD++ under a multi-core shared memory setting. If the matrices A, W, and H fit in a single machine, CCD++ can achieve significant speedup by utilizing all cores available in the machine.

The key component in CCD++ that requires parallelization is the computation to solve subproblem (17). In CCD++, the approximate solution to the subproblem is obtained by updating $\boldsymbol{u}$ and $\boldsymbol{v}$ alternately. When $\boldsymbol{v}$ is fixed, from (18), each variable u_i can be updated independently. Therefore, updating $\boldsymbol{u}$ can be divided into m independent jobs which can be handled by different cores in parallel.

Given a machine with p cores, we define $S = \{S_1, \ldots, S_p\}$ as a partition of row indices of W, $\{1, \ldots, m\}$. We decompose $\boldsymbol{u}$ into p vectors $\boldsymbol{u}^1, \boldsymbol{u}^2, \ldots, \boldsymbol{u}^p$, where $\boldsymbol{u}^r$ is the sub-vector of $\boldsymbol{u}$ corresponding to S_r. A simple strategy to make a equal-sized partition (i.e., $|S_1| = |S_2| = \cdots = |S_p| = m/p$). Due to the variance of the computation cost to update different u_i, the loading of r^{th} core is $\sum_{i \in S_r} 4|\Omega_i|$, which is not the same for all core. As a result, the equal-sized partition leads to load imbalance, which reduces the core utilization. An ideal partition can be obtained by solving

$$\min_S \left(\max_{r=1}^{p} \sum_{i \in S_r} |\Omega_i| \right) - \left(\min_{r=1}^{p} \sum_{i \in S_r} |\Omega_i| \right),$$

which is a known NP-hard problem. Hence, for multi-core parallelization, instead of being assigned to a fixed core, jobs are dynamically assigned based on the availability of each core. When a core finishes a small job, it can always conduct a new job without waiting other cores. Such dynamic assignment usually achieves good load balance on multi-core machines. Most multi-core libraries (e.g., OpenMP[8] and Intel TBB[9]) provide a simple interface to conduct this dynamic job assignment. Such approach can be also applied to update $\boldsymbol{v}$ and the residual R. For simplicity, we still use the notation of the fixed partition to describe parallel CCD++ on a shared-memory multi-core machine.

At the beginning for each subproblem, each core c constructs $\hat{R}$ by

$$\hat{R}_{ij} \leftarrow R_{ij} + \bar{w}_{ti}\bar{h}_{tj}, \ \forall (i,j) \in \Omega_{S_r}, \tag{25}$$

where $\Omega_{S_r} = \bigcup_{i \in S_r} \{(i,j) : j \in \Omega_i\}$. Each core r then

$$\textit{updates } u_i \leftarrow \frac{\sum_{j \in \Omega_i} \hat{R}_{ij} v_j}{\sum_{j \in \Omega_i} v_j^2} \ \forall i \in S_r. \tag{26}$$

Updating H can be parallelized in the same way with $G = \{G^1, \ldots, G^p\}$, which

8 http://openmp.org/
9 `http://threadingbuildingblocks.org/`

Algorithm 3 Parallel CCD++ in multi-core systems

```
Input: $A$, $W$, $H$, $\lambda$, $k$, $T$
1:  Initialize $W = 0$ and $R = A$.
2:  for $iter = 1, 2, \ldots,$ do
3:      for $t = 1, 2, \ldots, k$ do
4:          Parallel: core $r$ construct $\hat{R}$ using (25).
5:          for $inneriter = 1, 2, \ldots, T$ do
6:              Parallel: core $r$ updates $\boldsymbol{u}^r$ using (26).
7:              Parallel: core $r$ updates $\boldsymbol{v}^r$ using (27).
8:          end for
9:          Parallel: core $r$ updates $\bar{\boldsymbol{w}}_t^r$ and $\bar{\boldsymbol{h}}_t^r$ using (28).
10:         Parallel: core $r$ updates $R$ using (29).
11:     end for
12: end for
```

is a partition of row indices of H, $\{1, \ldots, n\}$. Similarly, each core r

$$\textit{updates } v_j \leftarrow \frac{\sum_{i \in \bar{\Omega}_j} \hat{R}_{ij} u_i}{\sum_{i \in \Omega_i} u_i^2} \; \forall j \in G_r. \tag{27}$$

As all cores on the machine share the common memory space, no communication is required for each core to access the latest $\boldsymbol{u}$ and $\boldsymbol{v}$. After obtaining $(\boldsymbol{u}^*, \boldsymbol{v}^*)$, we can also update the residual R and $(\bar{\boldsymbol{w}}_t^r, \bar{\boldsymbol{h}}_t^r)$ in parallel by assigning core r to perform the update:

$$(\bar{\boldsymbol{w}}_t^r, \bar{\boldsymbol{h}}_t^r) \leftarrow (\boldsymbol{u}^r, \boldsymbol{v}^r). \tag{28}$$

$$R_{ij} \leftarrow \hat{R}_{ij} - \bar{w}_{ti} \bar{h}_{tj}, \; \forall (i,j) \in \Omega_{S_r}, \tag{29}$$

We summarize the parallel CCD++ in Algorithm 3.

4.2. CCD++ in Distributed Systems

In this section, we investigate the parallelization of CCD++ when the matrices A, W, and H exceed the memory capacity of a singe machine. To avoid frequent access from disk, we consider handling these matrices with a distributed system, which connects several machines with their own computing resources (e.g., CPUs and memory) via a network. The algorithm to parallelize CCD++ in a distributed system is similar to the multi-core version of parallel CCD++ introduced in Algorithm 3. The common idea is to enable each machine/core to solve subproblem (17) and update a subset of variables and residual in parallel.

When W and H are too large to fit in memory of a single machine, we have to divide them into smaller components and distribute them to different machines. There are many ways to divide W and H. In the distributed version of parallel CCD++, assuming that the distributed system is composed of p machines, we consider p-way row partitions for W and H: $S = \{S_1, \ldots, S_p\}$ is a partition of the row indices of W; $G = \{G_1, \ldots, G_p\}$ is a partition of the row indices of H. We further denote the sub-matrices corresponding to S_r and G_r by W^r and H^r, respectively. In the distributed version of CCD++, machine r is responsible for the storage and the update of W^r and H^r. Note that the dynamic approach to assign jobs in Section 4.1 cannot be applied here because not all variables and

Algorithm 4 Parallel CCD++ in distributed systems

```
Input: $A$, $W$, $H$, $\lambda$, $k$, $T$
 1: Initialize $W = 0$ and $R = A$.
 2: for $iter = 1, 2, \ldots$ do
 3:     for $t = 1, 2, \ldots, k$ do
 4:         Broadcast: machine $r$ broadcasts $\bar{\boldsymbol{w}}_t^r$ and $\bar{\boldsymbol{h}}_t^r$.
 5:         Parallel: machine $r$ constructs $\hat{R}$ using (30).
 6:         for $inneriter = 1, 2, \ldots T$ do
 7:             Parallel: machine $r$ updates $\boldsymbol{u}^r$ using (26).
 8:             Broadcast: machine $r$ broadcasts $\boldsymbol{u}^r$.
 9:             Parallel: machine $r$ updates $\boldsymbol{v}^r$ using (27).
10:             Broadcast: machine $r$ broadcasts $\boldsymbol{v}^r$.
11:         end for
12:         Parallel: machine $r$ updates $\bar{\boldsymbol{w}}_t^r$, $\bar{\boldsymbol{h}}_t^r$ using (28).
13:         Parallel: machine $r$ updates $R$ using (31).
14:     end for
15: end for
```

ratings are available on all machines. Partitions S and G should be determined before any computation.

Typically, the residual R is much larger than W and H, thus we should avoid communication of R. Here we describe an arrangement of R on a distributed system such that all updates in CCD++ can be done without any communication of the residual. As mentioned above, machine r is in charge of updating variables in W^r and H^r. From the update rules of CCD++, we can see that values $R_{ij}\ \forall (i,j) \in \Omega_{S_r}$, are required to update variables in W^r, while $R_{ij}\ \forall (i,j) \in \bar{\Omega}_{G_r}$, are required to update H^r, where $\bar{\Omega}_{G_r} = \bigcup_{j \in G_r} \{(i,j) : i \in \bar{\Omega}_j\}$. Thus, the following entries of R should be easily accessible from machine r:

$$\Omega^r = \Omega_{S_r} \cup \bar{\Omega}_{G_r} = \{(i,j) : i \in S_r \text{ or } j \in G_r\}.$$

Thus, only $R_{ij}\ \forall (i,j) \in \Omega^r$, are stored in machine r. Assuming that the latest R_{ij}'s corresponding to Ω^r are available in machine r, the entire $\bar{\boldsymbol{w}}_t$ and $\bar{\boldsymbol{h}}_t$ are still required to construct the $\hat{R}$ in subproblem (17). As a result, we need to broadcast $\bar{\boldsymbol{w}}_t$ and $\bar{\boldsymbol{h}}_t$ in the distributed version of parallel CCD++ such that a complete local copy of the latest $\bar{\boldsymbol{w}}_t$ and $\bar{\boldsymbol{h}}_t$ is locally available for each machine to construct $\hat{R}$ distributively by

$$\hat{R}_{ij} \leftarrow R_{ij} + \bar{w}_{ti}\bar{h}_{tj}\ \forall (i,j) \in \Omega^r. \tag{30}$$

During T CCD iterations, machine r needs to broadcast the latest copy of $\boldsymbol{u}^r$ to other machines before updating $\boldsymbol{v}^r$ and broadcast the latest $\boldsymbol{v}^r$ before updating $\boldsymbol{u}^r$.

After T alternating iterations, each machine r has a complete and latest copy of $(\boldsymbol{u}^*, \boldsymbol{v}^*)$, which can be used to update $(\bar{\boldsymbol{w}}_t^r, \bar{\boldsymbol{h}}_t^t)$ by (28). The residual R can also be updated without extra communication by

$$R_{ij} \leftarrow \hat{R}_{ij} + \bar{w}_{ti}\bar{h}_{tj}\ \forall (i,j) \in \Omega^r, \tag{31}$$

as $(\bar{\boldsymbol{w}}_t^r, \bar{\boldsymbol{h}}_t^r)$ is also locally available in each machine r.

Our parallel CCD++ method in a distributed system is described in Algorithm 4. In summary, in distributed CCD++, each machine r only stores W^r and H^r and residual matrices $R_{S_r:}$ and $R_{:G_r}$. In an ideal case, where $|S_r| = m/p, |G_r| = n/p, \sum_{i\in S_r} |\Omega_i| = |\Omega|/p$, and $\sum_{j\in G_r} |\bar{\Omega}_j| = |\Omega|/p$, the memory consumption on each machine is mk/p variables of W, nk/p variables of H, and $2|\Omega|/p$ entries of R. As all communication in Algorithm follows the same scenario: each machine r broadcasts the $|S_r|$ (or $|G_r|$) local variables to others machines and gathers other $m - |S_r|$ latest variables from others. Such communication can be achieved efficiently by an Allgather operation, which is a collective operation defined by Message Passing Interface (MPI) standard.[10] With a recursive-doubling algorithm, Allgather operation can be done in

$$\alpha \log p + \frac{p-1}{p} M\beta, \tag{32}$$

where M is the message size in bytes, α is the startup time per message, independent of the message size, and β is transfer time per byte [20]. Based on Eq. (32), the total communication time of Algorithm 4 per iteration is

$$\left(\alpha \log p + \frac{8(m+n)(p-1)\beta}{p}\right) k(T+1),$$

where we assume that each entry of W and H is a double-precision floating-point number.

4.3. Scalability Analysis of Other Methods

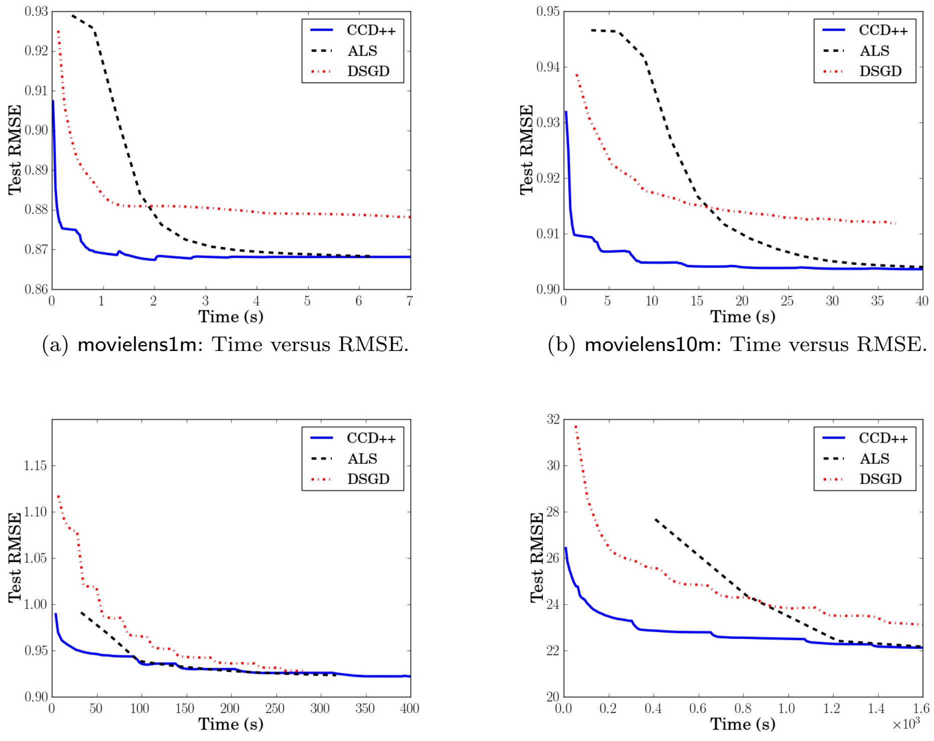
As mentioned in Section 2.1, ALS can be easily parallelized. However, it is hard to be scaled up to very large-scale recommender systems when W or H cannot fit in the memory of a single machine. When ALS updates $\boldsymbol{w}_i$, H_{Ω_i} is required to compute the Hessian matrix $(H_{\Omega_i}^T H_{\Omega_i} + \lambda I)$ in Eq. (3). In parallel ALS, even though each machine only updates a subset of rows of W or H at a time, [1] proposes that each machine should gather the entire latest H or W before the updates. However, when W or H is beyond the memory capacity of a single machine, it is not feasible to gather entire W or H and store them in the memory before the updates. Thus, each time when some rows of H or W are not available locally but are required to form the Hessian, the machine has to initiate communication with other machines to fetch those rows from them. Such complicated communication could severely reduce the efficiency of ALS. Furthermore, the higher time complexity per iteration of ALS is unfavorable when dealing with large W and H. Thus, ALS is not scalable to handle recommender systems with very large W and H.

Recently, [5] proposed a distributed SGD approach, DSGD, which partitions A into blocks and conducts SGD updates with a particular ordering. Similar to our approach, DSGD stores W, H, and A in a distributed manner such that each machine only needs to store $(n+m)k/p$ variables and $|\Omega|/p$ rating entries. Each communication scenario in DSGD is that each machine sends m/p (or n/p) variables to a particular machine, which can be done by done by a Sendreceive operation. As a result, the communication time per iteration of DSGD is $\alpha p +$

[10] `http://www.mcs.anl.gov/research/projects/mpi/`

Table 1. The statistics and parameters for each dataset

dataset	movielens1m	movielens10m	netflix	yahoo-music	synthetic-u	synthetic-p
m	6,040	71,567	2,649,429	1,000,990	3,000,000	20,000,000
n	3,952	65,133	17,770	624,961	3,000,000	1,000,000
$\|\Omega\|$	900,189	9,301,274	99,072,112	252,800,275	8,999,991,830	14,661,239,286
$\|\Omega^{\text{Test}}\|$	100,020	698,780	1,408,395	4,003,960	90,001,535	105,754,418
k	40	40	40	100	10	30
λ	0.1	0.1	0.05	1	0.001	0.001



(a) movielens1m: Time versus RMSE. (b) movielens10m: Time versus RMSE.

(c) netflix: Time versus RMSE. (d) yahoo-music: Time versus RMSE.

Fig. 3. RMSE versus computation time on a 8-core system for different methods (time is in seconds). Due to non-convexity of the problem, different methods may converge to different values.

$8mk\beta$. Thus, both DSGD and CCD++ can handle recommender systems with very large W and H.

5. Experimental Results

In this section, we compare parallel CCD++, parallel ALS, and parallel SGD in large-scale datasets under both multi-core and distributed platforms. For CCD++, we use the implementation with our adaptive technique based on the function value reduction. We implement parallel ALS with the Intel Math Kernel Library.[11] Based on the observation in Section 2, we choose DSGD as an example of the parallel SGD methods because of its faster and more stable convergence than other variants. In this paper, all algorithms are implemented in C++ to make a fair comparison. Similar to [1], all of our implementations use the weighted λ regularization.[12]

Datasets. We consider four public datasets for the experiment: movielens1m, movielens10m, netflix, and yahoo-music. The original training/test split is used for reproducibility.

To conduct experiments in a distributed environment, we follow the procedure used to create the Jumbo dataset in [8] to generate the synthetic-u dataset, a 3M by 3M sparse matrix with rank 10. We first build the ground truth W and H with each variable uniformly distributed over the interval $[0, 1)$. We then sample about 9 billion entries uniformly at random from WH^T and adding a small noise as our training set and sample about 90 million other entries without noise as the test set.

Since the observed entries in real-world datasets usually follow power-law distributions, we further construct a dataset synthetic-p with the unbalanced size 20M by 1M and rank 30. The power-law distributed observed set Ω is generated using the Chung-Lu-Vu (CLV) model proposed in [21]. More specifically, we first sample the degree sequence $a_1, \cdots, a_m$ for all the rows following the power-law distribution $p(x) \propto x^{-c}$ with $c = -1.316$ (the parameter c is selected to control the number of nonzeros). The edge potential for column nodes $b_1, \cdots, b_n$ are sampled from the same distribution. We then sample each edge (i, j) with the probability $\frac{a_i b_j}{\sum_k b_k}$. The value of the observed entries are generated by the same way as in synthetic-u. For training/test split, we randomly select about 1% observed entries as test set and the test observed entries as the training set. See Table 1 for more information about the statistics and parameters used (k and λ) for each dataset.

5.1. Experiments on a Multi-core Environment

In this section, we compare the multi-core version of parallel CCD++ with other methods on a multi-core shared-memory environment.

Experimental platform. We use an 8-core Intel Xeon X5570 processor with 8 MB L2-cache and enough memory for the comparison. The OpenMP library is used for the multi-core parallelization.

Results. We ensure that eight cores are fully utilized for each method. Figure 3 shows the comparison of the running time versus RMSE for the four real-world datasets. We observe that the performance of CCD++ is generally better than parallel ALS and DSGD for each dataset.

Speedup. Another important measurement in parallel computing is the speedup – how much faster a parallel algorithm is when we increase the number of cores. To test the speedup, we run each parallel method on yahoo-music with various numbers of cores, from 1 to 8, and measure the running time for

[11] Our C implementation is 6x faster than the MATLAB version in [1].

[12] $\lambda \left(\sum_i |\Omega_i| \|\boldsymbol{w}_i\|^2 + \sum_j |\bar{\Omega}_j| \|\boldsymbol{h}_j\|^2 \right)$ is used to replace the regularization term in (1).

one iteration. The speedup comparison is shown in Figure 4 and the result of HogWild is included as well. We can clearly see that all methods have nearly linear speedup. However, the slopes of CCD++ and ALS are steeper than DSGD and HogWild. This can be explained by the cache-miss rate for each method. Due to the fact that CCD++ and ALS access variables in contiguous memory spaces, both of them enjoy better locality. In contrast, due to the randomness, two consecutive updates in SGD usually access non-contiguous variables in W and H, which increases the cache-miss rate. Given the fixed size of cache, time spent on loading data from memory to cache becomes the bottleneck for DSGD and HogWild to achieve better speedup when the number of cores increases.

5.2. Experiments on a Distributed Environment

In this section, we conduct experiments to show that distributed CCD++ is faster than DSGD and ALS for handling large-scale data on a distributed system.

Experimental platform. The following experiments are conducted in a large-scale parallel platform at the Taxas Advanced Computing Center (TACC), Stampede[13]. Each computing node in Stampede is an Intel Xeon E5-2680 2.7GHz CPU machine with 32 GB memory and communicates by FDR 56 Gbit/s cable. For a fair comparison, we implement a distributed version with MPI in C++ for all the methods. The reason we do not use Hadoop is that almost all operations in Hadoop need to access data and variables from disks, which is quite slow and thus not suitable for iterative methods. It is reported in [22] that ALS implemented with MPI is 40 to 60 times faster than its Hadoop implementation in the Mahout project.

Results on yahoo-music. First we show comparisons on the yahoo-music dataset, which is the largest real-world dataset we used in this paper. Figure 5 shows the result with 4 computing nodes – we can make similar observations as in Figure 3.

Results on synthetic datasets. When data is large enough, we believe that the benefit of distributed environments would be obvious.

For the scalability comparison, we vary the number of computing nodes, ranging from 32 to 256, and compare the time and speedup for these three algorithms on the synthetic-u and synthetic-p datasets. Note that ALS requires more than 32GB memory for each node when using all 32 nodes on synthetic-p dataset, so we run each algorithm with at least 64 nodes for this dataset. Here we set the Here we calculate the training time as the time taken to achieve 0.01 test RMSE on synthetic-u and 0.02 test RMSE on synthetic-p respectively. The results are shown in Figure 6a and 7a. We can see clearly that CCD++ is more than 8 times faster than both DSGD and ALS on synthetic-u and synthetic-p datasets with the number of computing nodes varying from 32 to 256. We also show the speedup of ALS, DSGD, and CCD++ on both datasets in Figure 6b and 7b. Note that since the data cannot be loaded in memory of a single machine, the speedup using p machines is T_p/T_{32} on synthetic-u and T_p/T_{64} on synthetic-p respectively, where T_p is the time taken on p machines. We observe that DSGD achieves super linear speedup on both datasets. For example, on synthetic-u dataset, the training time for DSGD is 2768 seconds using 32 machines and 218 seconds using 256

[13] `http://www.tacc.utexas.edu/user-services/user-guides/stampede-user-guide\#compenv`

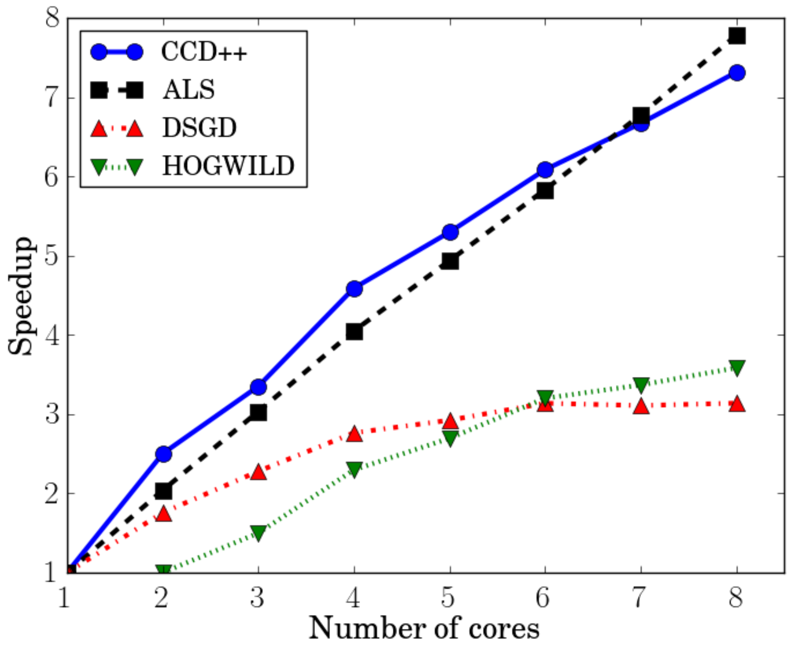


Fig. 4. Speedup comparison among four algorithms with the yahoo-music dataset in a shared-memory multi-core environment. All of them have nearly linear speedup. CCD++ and ALS have better performance than DSGD and HogWild because of better locality.

machines, while it achieves $2768/218 \approx 12.7$ times speedup with only 8 times the number of machines. This super linear speedup is due to the cache effect as well. In DSGD, each machine stores one block of W and one block of H. When the number of machines is large enough, these blocks can fit into caches, which causes the dramatic reduce of the memory access time. In contrast, when the number of machines is not enough, these blocks cannot fit into caches, thus resulting in severe cache miss in DSGD. However, even if W and H cannot fit into caches, the cache miss is not severe in CCD++ and ALS. This is because the memory is accessed sequentially in both methods.

Though that the speedups are smaller than in a multi-core setting, CCD++ takes the least time to achieve the desired RMSE. This shows that CCD++ is not only fast but also scalable for large-scale matrix factorization on distributed systems.

6. Extension to L1-regularization

Besides the L2-regularization, L1-regularization is used in many applications to achieve a more sparse solution of W and H. In the following we show that CCD and CCD++ can be easily modified to handle (1) with L1-regularization:

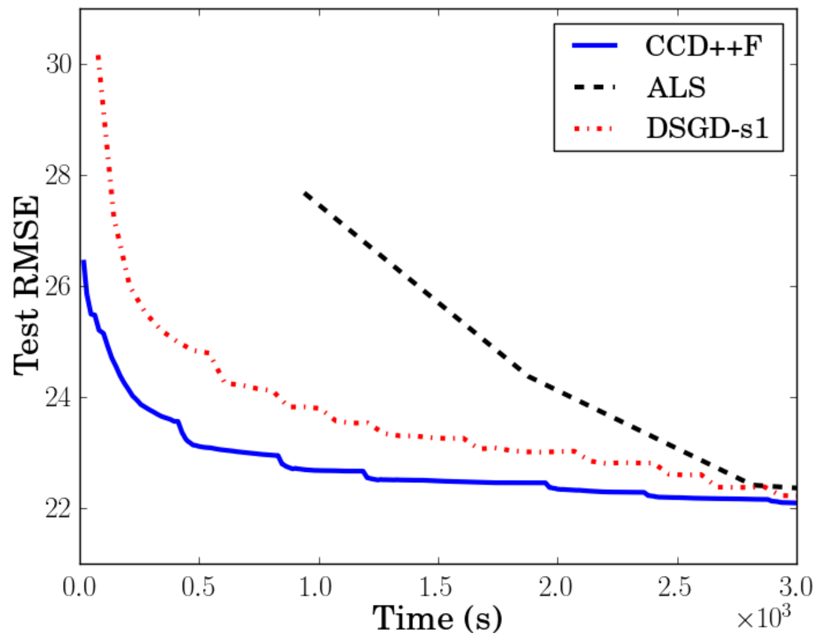
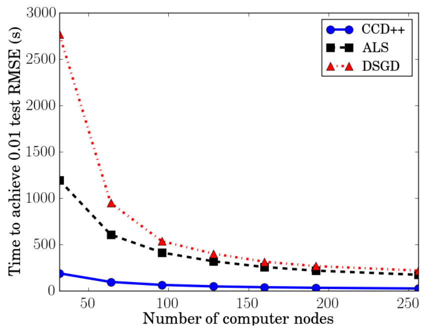
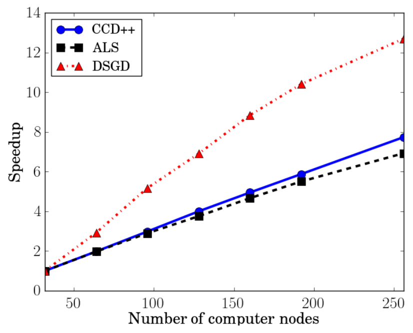


Fig. 5. Comparison among CCD++, ALS, and DSGD with the yahoo-music dataset on a MPI distributed system with 4 computing nodes.

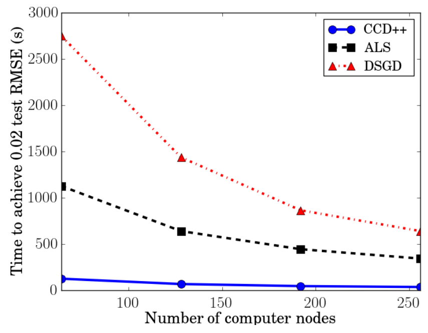


(a) Number of computation nodes versus training time.

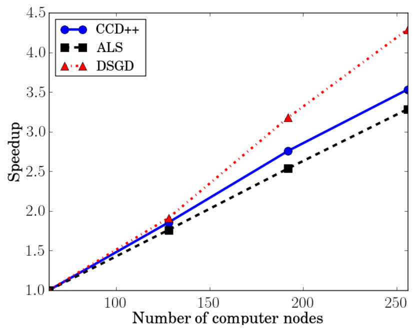


(b) Number of computation nodes versus speedup.

Fig. 6. Comparison among CCD++, ALS and DSGD on the synthetic-u dataset (9 billion ratings) on a MPI distributed system with varying number of computing nodes. The vertical axis in the left panel is the time for each method to achieve 0.01 test RMSE, while the right panel shows the speedup for each method. Note that, as discussed in Section 5.2, speedup is T_p/T_{32}, where T_p is the time taken on p machines.



(a) Number of computation nodes versus training time.



(b) Number of computation nodes versus speedup.

Fig. 7. Comparison among CCD++, ALS and DSGD on the synthetic-p dataset (14.5 billion ratings) on a MPI distributed system with varying number of computing nodes. The vertical axis in the left panel is the time for each method to achieve 0.02 test RMSE, while the right panel shows the speedup for each method. Note that, as discussed in Section 5.2, speedup is T_p/T_{64}, where T_p is the time taken on p machines.

$$\min_{\substack{W\in\mathbb{R}^{m\times k}\\ H\in\mathbb{R}^{n\times k}}} \sum_{(i,j)\in\Omega} (A_{ij} - \boldsymbol{w}_i^T\boldsymbol{h}_j)^2 + \lambda\left(\sum_i^m \|\boldsymbol{w}_i\|_1 + \sum_j^n \|\boldsymbol{h}_j\|_1\right). \tag{33}$$

For CCD and CCD++, the L1-regularized one-variable optimization problem in (4) becomes

$$\min_z \; f(z) = f_0(z) + \lambda|z|, \tag{34}$$

where $f_0(z) = \sum_{j\in\Omega_i}\left(R_{ij} + w_{it}h_{jt}) - zh_{jt}\right)^2$. Since $f_0(z)$ is a quadratic function, the solution z^* to (34) can uniquely obtained by the following soft thresholding operation:

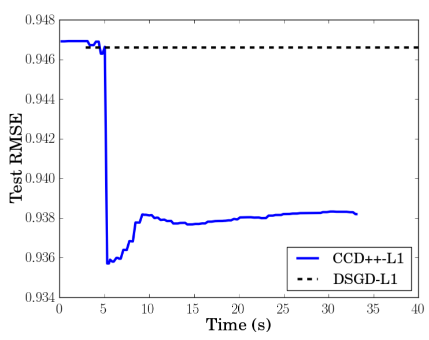
$$z^* = \frac{-\operatorname{sgn}(g)\max\left(|g|-\lambda, 0\right)}{d}, \tag{35}$$

where

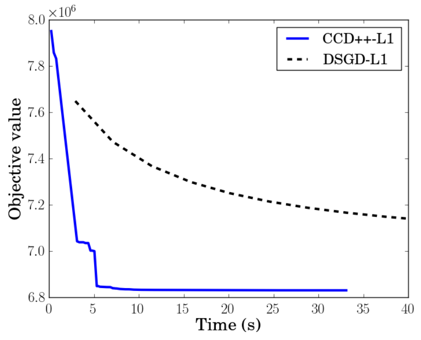
$$g = f_0'(0) = -2\sum_{j\in\Omega_i}(R_{ij} + w_{it}h_{jt})h_{jt}, \text{ and } d = f_0''(0) = 2\sum_{j\in\Omega_i} h_{jt}^2.$$

We can maintain the residual matrix R to reduce the time complexity for each single variable update to $O(|\Omega_i|)$, which is the same with that of L2-regularization case. Similarly, we can use Algorithm 3 to parallelize CCD++ with L1-regularization.

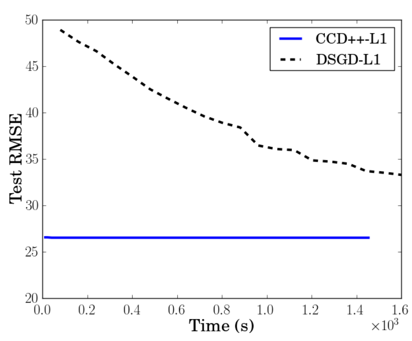
In ALS, with the L1 penalty, the second term in each row-subproblem (2) will be replaced by a non-smooth term $\lambda\|\boldsymbol{w}_i\|$, thus the resulting problem does not have a closed form solution. Therefore, ALS cannot be applied to solve the L1-regularized matrix factorization problem. When SGD is applied to solve a non-smooth function, the gradient in the update rule has to be replaced by the



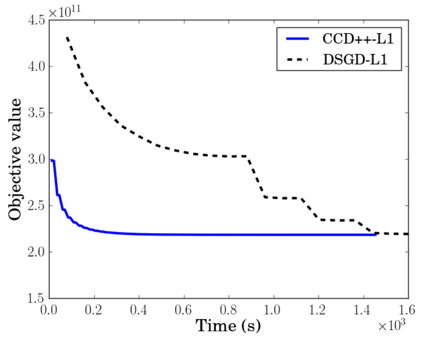
(a) movielens10m: Time versus RMSE.



(b) movielens10m: Time versus Obj.



(c) yahoo-music: Time versus RMSE.



(d) yahoo-music: Time versus Obj.

Fig. 8. RMSE and objective function values versus computation time(in seconds) for different methods on the matrix factorization problem with L1-regularization. Due to non-convexity of the problem, different methods may converge to different values.

subgradient, thus the update rule for the (i, j) element becomes

$$w_{it} = \begin{cases} w_{it} - \eta \left(\operatorname{sgn}(w_{it}) \frac{\lambda}{|\Omega_i|} - 2R_{ij}h_j \right) & \text{if } w_{it} \neq 0 \\ w_{it} - \eta \left(-\operatorname{sgn}(2R_{ij}h_j) \max(|2R_{ij}h_j| - \frac{\lambda}{|\Omega_i|}, 0) \right) & \text{if } w_{it} = 0 \end{cases}$$

$$h_{jt} = \begin{cases} h_{jt} - \eta \left(\operatorname{sgn}(h_{jt}) \frac{\lambda}{|\bar{\Omega}_i|} - 2R_{ij}w_i \right) & \text{if } h_{jt} \neq 0 \\ h_{jt} - \eta \left(-\operatorname{sgn}(2R_{ij}w_i) \max(|2R_{ij}w_i| - \frac{\lambda}{|\bar{\Omega}_i|}, 0) \right) & \text{if } h_{jt} = 0. \end{cases}$$

The time complexity for each update is the same with the L2-regularized one. Similarly, the same trick in DSGD and HogWild can also be used to parallelize SGD with L1-regularization as well.

Figure 8 presents the comparison of multi-core version of parallel CCD++ and DSGD with L1-regularization on two large-scale datasets, movielens10m and yahoo-music. The experiment settings and platform are the same with that used in Section 4.1.

First, we compare the solution of L2-regularized matrix factorization problem (1) versus the L1-regularized one (33) in Table 2. Although L1-regularized form

Table 2. The best test RMSE for each model (the lower, the better). We run both CCD++ and DSGD with a large number of iterations to obtain the best test RMSE for each model.

	movielens10m	yahoo-music
L1-regularization	0.9381	24.49
L2-regularization	**0.9035**	**21.92**

achieves worse test RMSE comparing to L2-regularized form, it can successfully recover sparse models W and H, which is important for interpretation in many applications.

We then compare the convergence speed of CCD++ and SGD for solving the L1-regularized problem (33). Figure 8b and 8d present the objective function values versus computation time on movielens10m and yahoo-music datasets. In both figures, we can clearly see that the objective function values gradually decrease with respect to time with much faster convergence in CCD++ than that of DSGD. This shows the superiority of CCD++ to solve (33). Meanwhile, Figure 8a and 8c show the test RMSE versus computation time on both datasets. Similar to L2-regularized case, in both datasets, CCD++ achieves better test RMSE than DSGD. However, since (33) is not initially designed to improve the generalization of the model or in other words, to reduce the test RMSE, but to achieve the sparse pattern of W and H, when updating W and H, we sacrifice the test RMSE in sake of sparse pattern of W and H. This can explain the increase of test RMSE in some parts of the curves in both datasets.

7. Conclusions

In this paper, we have shown that the coordinate descent method is efficient and scalable for solving large-scale matrix factorization problems in recommender systems. The proposed method CCD++ not only has lower time complexity per iteration than ALS, but also achieves faster and more stable convergence than SGD in practice. We also explore different update sequences and show that the feature-wise update sequence (CCD++) gives better performance. Moreover, we show that CCD++ can be easily parallelized in both multi-core and distributed environments and thus can handle large-scale datasets where both ratings and variables cannot fit in the memory of a single machine. Empirical results demonstrate the superiority of CCD++ under both parallel environments. For instance, running with a large-scale synthetic dataset (14.6 billion ratings) on a distributed memory cluster, CCD++ is 24 times faster to achieve the desired test accuracy than DSGD when we use 32 processors, and when we use 256 processors, CCD++ is 18 times faster than DSGD and 9.6 times faster than ALS.

8. Acknowledgments

This research was supported by NSF grants CCF-0916309, CCF-1117055, and DOD Army grant W911NF-10-1-0529. We also thank the Texas Advanced Computer Center (TACC) for providing computing resources required to conduct experiments in this work.

References

[1] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan, "Large-scale parallel collaborative filtering for the Netflix prize," in *Proceedings of the 4th international conference on Algorithmic Aspects in Information and Management*, 2008.
[2] Y. Koren, R. M. Bell, and C. Volinsky, "Matrix factorization techniques for recommender systems," *IEEE Computer*, vol. 42, pp. 30–37, 2009.
[3] G. Takács, I. Pilászy, B. Németh, and D. Tikk, "Scalable collaborative filtering approaches for large recommender systems," *JMLR*, vol. 10, pp. 623–656, 2009.
[4] J. Langford, A. Smola, and M. Zinkevich, "Slow learners are fast," in *NIPS*, 2009.
[5] R. Gemulla, P. J. Haas, E. Nijkamp, and Y. Sismanis, "Large-scale matrix factorization with distributed stochastic gradient descent," in *ACM KDD*, 2011.
[6] B. Recht, C. Re, and S. J. Wright, "Parallel stochastic gradient algorithms for large-scale matrix completion," 2011. Submitted for publication.
[7] M. Zinkevich, M. Weimer, A. Smola, and L. Li, "Parallelized stochastic gradient descent," in *NIPS*, 2010.
[8] F. Niu, B. Recht, C. Re, and S. J. Wright, "Hogwild!: A lock-free approach to parallelizing stochastic gradient descent," in *NIPS*, 2011.
[9] A. Cichocki and A.-H. Phan, "Fast local algorithms for large scale nonnegative matrix and tensor factorizations," *IEICE Transactions on Fundamentals of Electronics Communications and Computer Sciences*, vol. E92-A, no. 3, pp. 708–721, 2009.
[10] C.-J. Hsieh and I. S. Dhillon, "Fast coordinate descent methods with variable selection for non-negative matrix factorization," in *ACM KDD*, 2011.
[11] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proceedings of the 19th International Conference on Computational Statistics*, 2010.
[12] A. Agarwal and J. C. Duchi, "Distributed delayed stochastic optimization," in *NIPS 24*, 2011.
[13] D. P. Bertsekas, *Nonlinear Programming.* Belmont, MA 02178-9998: Athena Scientific, second ed., 1999.
[14] C.-J. Hsieh, K.-W. Chang, C.-J. Lin, S. S. Keerthi, and S. Sundararajan, "A dual coordinate descent method for large-scale linear SVM," in *ICML*, 2008.
[15] H.-F. Yu, F.-L. Huang, and C.-J. Lin, "Dual coordinate descent methods for logistic regression and maximum entropy models," *Machine Learning*, vol. 85, no. 1-2, pp. 41–75, 2011.
[16] C.-J. Hsieh, M. Sustik, I. S. Dhillon, and P. Ravikumar, "Sparse inverse covariance matrix estimation using quadratic approximation," in *NIPS*, 2011.
[17] I. Pilászy, D. Zibriczky, and D. Tikk, "Fast ALS-based matrix factorization for explicit and implicit feedback datasets," in *ACM RecSys*, 2010.
[18] R. M. Bell, Y. Koren, and C. Volinsky, "Modeling relationships at multiple scales to improve accuracy of large recommender systems," in *ACM KDD*, 2007.
[19] N.-D. Ho and P. V. D. V. D. Blondel, "Descent methods for nonnegative matrix factorization," in *Numerical Linear Algebra in Signals, Systems and Control*, pp. 251–293, Springer Netherlands, 2011.
[20] R. Thakur and W. Gropp, "Improving the performance of collective operations in MPICH," in *Proceedings of the tenth European PVM/MPI Users' Group Meeting*, 2003.
[21] F. Chung, L. Lu, and V. Vu, "Spectral of random graphs with given expected degrees," *Internet Mathematics*, vol. 1, 2004.
[22] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Distributed GraphLab: A framework for machine learning in the cloud," *PVLDB*, vol. 5, no. 8, 2012.