Knowl Inf Syst (2014) 41:793–819
DOI 10.1007/s10115-013-0682-2

REGULAR PAPER

# Parallel matrix factorization for recommender systems

**Hsiang-Fu Yu · Cho-Jui Hsieh · Si Si · Inderjit S. Dhillon**

Received: 1 February 2013 / Revised: 19 July 2013 / Accepted: 17 August 2013 /
Published online: 1 September 2013
© Springer-Verlag London 2013

**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, and 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 thoroughly explored. 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 on 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 64 processors, to deliver the desired test RMSE, CCD++ is 49 times faster than SGD and 20 times faster than ALS. When the number of processors is increased to 256, CCD++ takes only 16 s and is still 40 times faster than SGD and 20 times faster than ALS.

**Keywords** Recommender systems · Missing value estimation · 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 model. Matrix factor-

H.-F. Yu (✉) · C.-J. Hsieh · S. Si · I. S. Dhillon
Department of Computer Science, The University of Texas at Austin, Austin, TX 78712, USA
e-mail: rofuyu@cs.utexas.edu

**Table 1** Comparison between CCD++ and other state-of-the-art methods for matrix factorization

| | ALS | SGD | CCD++ |
|---|---|---|---|
| Time complexity per iteration | $O(\|\Omega\|k^2 + (m+n)k^3)$ | $O(\|\Omega\|k)$ | $O(\|\Omega\|k)$ |
| Convergence behavior | Stable | Sensitive to parameters | Stable |
| Scalability on distributed systems | Not scalable | Scalable | Scalable |

ization was empirically shown to be a better model than traditional nearest neighbor-based approaches in the Netflix Prize competition and KDD Cup 2011 [1]. 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 [2–4].

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 $\Omega$ is the set of indices for observed ratings; $\lambda$ is the regularization parameter; $\|\cdot\|_F$ denotes the Frobenius norm; $\boldsymbol{w}_i^T$ and $\boldsymbol{h}_j^T$ are the $i^{\text{th}}$ and the $j^{\text{th}}$ 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. Note that, the well-known rank-$k$ approximation by singular value decomposition (SVD) cannot be directly applied to (1) as $A$ is not fully observed.

Regarding problem (1), we can interpret $\boldsymbol{w}_i$ and $\boldsymbol{h}_j$ as the length-$k$ feature vectors for user $i$ and item $j$. The interaction/similarity between the $i^{\text{th}}$ user and the $j^{\text{th}}$ item is measured by $\boldsymbol{w}_i^T \boldsymbol{h}_j$. As a result, solving problem (1) can be regarded as a procedure to find a "good" representation for each user and item such that the interaction between them can well approximate the real rating scores.

In recent recommender system competitions, we observe that alternating least squares (ALS) and stochastic gradient descent (SGD) have attracted much attention and are widely used for matrix factorization [2,5]. ALS alternatively switches between updating $W$ and updating $H$ while fixing the other factor. Although the time complexity per iteration is $O(|\Omega|k^2 + (m+n)k^3)$, [2] shows that ALS is well suited 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 [3], 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 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, and a variety of schemes have been proposed to parallelize it [6–10].

This paper aims to design an efficient and easily parallelizable method for matrix factorization in large-scale recommender systems. Recently, [11] and [12] have showed that coordinate descent methods are effective for non-negative 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. Table 1 shows a comparison between the

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

state-of-the-art approaches and our proposed algorithm CCD++. The main contributions of this paper are as follows:

- 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 on both shared-memory multi-core and distributed systems.

**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 $\Omega_i$ to denote the column indices of observed ratings in the $i^{\text{th}}$ row and $\bar{\Omega}_j$ to denote the row indices of observed ratings in the $j^{\text{th}}$ column. We denote the $i^{\text{th}}$ row of $W$ by $\boldsymbol{w}_i^T$ and the $t^{\text{th}}$ 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 $t^{\text{th}}$ element of $\boldsymbol{w}_i$) and $\bar{w}_{ti}$ (i.e., the $i^{\text{th}}$ 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 Sect. 2. We then present our coordinate descent approaches in Sect. 3. In Sect. 4, we present strategies to parallelize CCD++ and conduct scalability analysis under different parallel computing environments. We then present experimental results in Sect. 5. Finally, we show an extension of CCD++ to handle L1-regularization in Sect. 6 and conclude in Sect. 7.

## 2 Related work

As mentioned in [3], 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 regularized 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 submatrix with columns $\{\boldsymbol{h}_j : j \in \Omega_i\}$, and $\boldsymbol{a}_i^T$ is the $i^{\text{th}}$ 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, [2] 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, in a distributed system, when $W$ or $H$ exceeds the memory capacity of a computation node, the parallelization of ALS becomes more challenging. More details are discussed in Sect. 4.3.

### 2.2 Stochastic gradient descent

Stochastic gradient descent (SGD) is widely used in many machine learning problems [13], and it has also been shown to be effective for matrix factorization [3]. In SGD, for each update, a rating $(i, j)$ is randomly selected from $\Omega$, and the corresponding variables $\boldsymbol{w}_i$ and $\boldsymbol{h}_j$ are updated by

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

where $R_{ij} = A_{ij} - \boldsymbol{w}_i^T \boldsymbol{h}_j$, and $\eta$ 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 appears to be 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 [6] and [14], while [9] uses a bootstrap aggregation scheme. A lock-free approach called HogWild is investigated in [10], 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 [10] also show that HogWild is more efficient than the "delayed update" approach in [6]. For matrix factorization, [7] and [8] 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 $\eta$. In practice, the initial choice and adaptation strategy for $\eta$ 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 [10], $\eta$ is reduced by multiplying a constant $\beta \in (0, 1)$ at each iteration. In DSGD, [7] proposes using the "bold driver" scheme, in which, at each iteration, $\eta$ is increased by a small proportion (5 % is used in [7]) when the function value decreases; when the value increases, $\eta$ is drastically decreased by a large proportion (50 % is used in [7]).

[2] In [8], the name "Jellyfish" is used.

**Table 2** 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 |
| $\lambda$ | 0.1 | 0.1 | 0.05 | 1 | 0.001 | 0.001 |

### 2.3 Experimental comparison

Next, we compare various parallel matrix factorization approaches: 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 2 of Sect. 5). Here, we conduct the comparison on an 8-core machine (see Sect. 5.2 for the detailed description of the experimental environment). All 8 cores are utilized for each method.[6] Figure 1 shows the comparison; "−s1" and "−s2" denote two choices of the initial $\eta$.[7] The reader might notice that the performance difference between ALS and DSGD is not as large as in [7]. The reason is that the parallel platform used in our comparison is different from that used in [7], which is a modified Hadoop distributed system.

In Fig. 1, we first observe that the performance of both DSGD and HogWild is sensitive to the choice of $\eta$. 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 $\eta$'s. Given the fact that the computation time per iteration of DSGD 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 convergence of HogWild.

## 3 Coordinate descent approaches

Coordinate descent is a classic and well-studied optimization technique [15, Section 2.7]. Recently, it has been successfully applied to various large-scale problems such as linear SVMs [16], maximum entropy models [17], NMF problems [11,12] and sparse inverse covariance estimation [18]. 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:

[3] Intel MKL is used in our implementation of ALS.

[4] We implement a multi-core version of DSGD according to [7].

[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 $\eta = 0.05$.

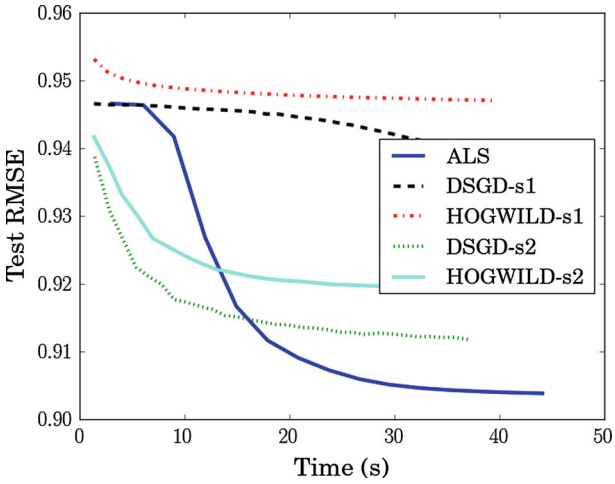



**Fig. 1** Comparison between ALS, DSGD, and HogWild on the movielens10m dataset with $k = 40$ on a 8-core machine (−s1 and −s2 stand for different initial learning rates)

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 need to solve the following one-variable subproblem:

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

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 computed 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

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

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

Note that (7) requires $O(|\Omega_i|)$ operations. 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 computed 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 a time.

ALS can be viewed as a method, which adopts this update sequence. As mentioned in Sect. 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, [19] proposed a cyclic coordinate descent method (CCD), which is similar to ALS with respect to the update sequence. The only difference lies in the update rules. In CCD, $\boldsymbol{w}_i$ is updated by applying (8) over all elements of $\boldsymbol{w}_i$ (i.e., $w_{i1}, \ldots, w_{ik}$) once. 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 residual matrix $R$ is exactly equal to $A$, so no extra effort is needed to initialize $R$.

**Algorithm 1** CCD Algorithm xx

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

As mentioned in Sect. 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 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 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 $w_{i1}, \ldots, w_{ik}$, one may think of a greedy update sequence that sequentially updates the variable that decreases the objective function the most. In [12], 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 in [12] is applied to solve (1), $m$ different Hessians are required to update $W$, and $n$ Hessians are required to update $H$. The computation of Hessian for $\boldsymbol{w}_i$ and $\boldsymbol{h}_j$ needs $O(|\Omega_i|k^2)$ and $O(|\bar{\Omega}_j|k^2)$ to compute, respectively. The total time complexity of GCD to update $W$ and $H$ once is thus $O(|\Omega|k^2)$ 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 $t^{\text{th}}$ column of $W$, and $\bar{\boldsymbol{h}}_t \in \mathbb{R}^n$ is the $t^{\text{th}}$ column of $H$. From the perspective of the latent feature space, $\bar{\boldsymbol{w}}_t$ and $\bar{\boldsymbol{h}}_t$ correspond to the $t^{\text{th}}$ latent feature.

**Algorithm 2** CCD++ Algorithm

**Input:** $A, W, H, \lambda, k, T$
Initialize $W = 0$ and $R = A$.
**for** $iter = 1, 2, \ldots$ **do**
  **for** $t = 1, 2, \ldots, k$ **do**
    Construct $\hat{R}$ by (16).
    **for** $inneriter = 1, 2, \ldots, T$ **do** ▷ $T$ CCD iterations for (17).
      Update $\boldsymbol{u}$ by (18).
      Update $\boldsymbol{v}$ by (19).
    **end for**
    Update $(\bar{\boldsymbol{w}}_t, \bar{\boldsymbol{h}}_t)$ and $R$ by (20) and (21).
  **end for**
**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:

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

where $R_{ij} = A_{ij} - \boldsymbol{w}_i^T \boldsymbol{h}_j$ is the residual entry for $(i, j)$. 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 to (17) to obtain an approximation by alternatively updating $\boldsymbol{u}$ and updating $\boldsymbol{v}$. When the current model $(W, H)$ is close to an optimal solution to (1), $(\bar{\boldsymbol{w}}_t, \bar{\boldsymbol{h}}_t)$ should be also very close to an optimal solution to (17). Thus, the current $(\bar{\boldsymbol{w}}_t, \bar{\boldsymbol{h}}_t)$ can be a good initialization for $(\boldsymbol{u}, \boldsymbol{v})$. The update sequence for $\boldsymbol{u}$ and $\boldsymbol{v}$ is

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

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 $\boldsymbol{v}$ at each CCD iteration become as follows:

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

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

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 [20] to avoid the over-fitting issue in recommender systems.

Each time when the $t^{\text{th}}$ feature is selected, CCD++ consists of the following steps 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). Since 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, and [21] observes that such a feature-wise update sequence leads to faster convergence than other sequences on moderate-scale matrices. However, for large-scale sparse NMF problems, when all entries are known, the residual matrix becomes a $m \times n$ dense matrix, which is too large to maintain. Thus, [11,12] utilize the property that all subproblems share a single Hessian, where there are no missing values, to develop techniques that allow efficient variable updates without maintenance of the residual.

Due to the large number of missing entries in $A$, problem (1) does not share the above favorable property. 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 count

Based on the analysis in Sects. 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.** An exact count of the number of operations depends on how the residual matrix $(R)$ of size $m \times n$ is stored in memory. The update rules used in CCD and CCD++ require frequent access to entries of $R$. If both observed and missing entries of $R$ can be stored in a dense format, random access to any entry $R_{ij}$ can be regarded as a constant time operation. However, when $m$ and $n$ are large, computer memory is usually not enough to store all $m \times n$ entries of $R$. As $|\Omega| \ll m \times n$ in most real-word recommender systems, storing only observed entries of $R$ (i.e., $\Omega$) in a sparse matrix format is a more feasible way to handle large-scale recommender systems. Two commonly used formats for sparse matrices are considered: compressed row storage (CRS) and compressed column storage (CCS). In CRS, observed entries of the same row are stored adjacent to each other in the memory, while in CCS, observed entries of the same column are stored adjacent to each other.

The update rules used in CCD and CCD++ access $R$ in two different fashions. Rules such as (6) and (7) in Algorithm 1 and (18) in Algorithm 2 need to access observed entries of a particular row (i.e., $\Omega_i$) fast. In this situation, CRS provides faster access than CCS as

observed entries of the same row are located next to each other. On the other hand, rules such as (12) and (9) in Algorithm 1 and (19) in Algorithm 2 require fast accesses to observed entries of a particular column (i.e., $\bar{\Omega}_j$). CCS is thus more favorable for such rules.

In fact, if only one copy $R$ is stored in either CCS or CRS format, the update rules can no longer be computed in $O(|\Omega_i|)$ or $O(|\bar{\Omega}_j|)$ time. For instance, assume only a copy of $R$ in CCS is available, locating the observed entries of a single row (i.e., $R_{ij}\ \forall j \in \Omega_i$) requires at least $n$ operations. In the worst case, it might even costs $|\Omega|$ operations to identify the locations of $|\Omega_i|$ entries. Thus, there is no way to compute rules such as (6) and (18) in $O(\Omega_i)$ time. In contrast, if a copy of $R$ in CRS is also available, the time to access the observed entries of row $i$ is only $|\Omega_i|$ operations. As a result, to efficiently access both rows and columns in $R$, in both CCD and CCD++, two copies of $R$ are maintained in the memory: one is in CRS format and the other is in CCS format.

Another concern is about the storage of $\hat{R}$ in CCD++. Since $\hat{R}$ exists only when solving each subproblem (17), there is no need to allocate extra storage for $\hat{R}$. In fact, $\hat{R}$ and $R$ can share the same memory in the following implementation of Algorithm 2:

- For rule (16) in Line 4, reuse $R$ to store $\hat{R}$:

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

- For rules (18) and (19), use $R$ to update $\boldsymbol{u}$ and $\boldsymbol{v}$.
- For rule (21) in Line 9, use the following to update the real "residual":

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

**Operation count.** In CCD, the update rules (6) and (12) take about $6|\Omega_i|$ and $6|\bar{\Omega}_j|$ flops, respectively. For update rule (7), it takes about $3|\Omega_i|$ flops to compute values $R_{ij},\ \forall (i,j) \in \Omega_i$ in CRS format and store those values to the copy of the residual in CCS format. Similarly, update rule (9) takes about $3|\bar{\Omega}_j|$flops to update the residual $R$. As a result, one CCD iteration, where $(m+n)k$ variables are updated, requires

$$\left(\left(\sum_{i=1}^{m}(6+3)|\Omega_i|\right)+\left(\sum_{j=1}^{n}(6+3)|\bar{\Omega}_j|\right)\right)\times k = 18|\Omega|k \text{ flops}. \tag{23}$$

In CCD++, the construction of $\hat{R}$ (16) and the residual (21) require $2\times 2|\Omega|$ flops due to the two copies of $R$. The update rules (18) and (19) cost $4|\Omega_i|$ and $4|\bar{\Omega}_j|$ flops, respectively. Therefore, one CCD++ iteration with $T$ inner CCD iterations, where $(m+n)kT$ variables are updated, 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 above counting results, if $T = 1$, where the same number of variables are updated in one iteration of both CCD and CCD++, CCD++ is 1.125 faster than CCD. If $T > 1$, the ratio between the flops required by CCD and CCD++ to update the same number of variables, $\frac{9T}{4(T+1)}$, 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 inner CCD iterations for each subproblem (17). The approaches [11,21], 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, the construction of $\hat{R}$ (16) and the residual update (21) are only conducted once for each subproblem. Based on the exact operation counts 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 $\epsilon d^{\max}$ for some small positive ratio $\epsilon$, such as $10^{-3}$, we stop CCD iterations, update the residual by (21) and switch to the next subproblem. 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 exactly the denominator of the update rule (18). As a result, the function value reduction can be obtained without extra effort.

Next, we show an empirical comparison between CCD and CCD++, where we include four settings with the netflix dataset on a machine with enough memory:

- CCD: item/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).

In Fig. 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 [21]. 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 100 to 180 s in Fig. 2). Such periods suggest that an early termination might help. We also notice that our technique to adaptively control $T$ can slightly shorten such periods and improve the performance.

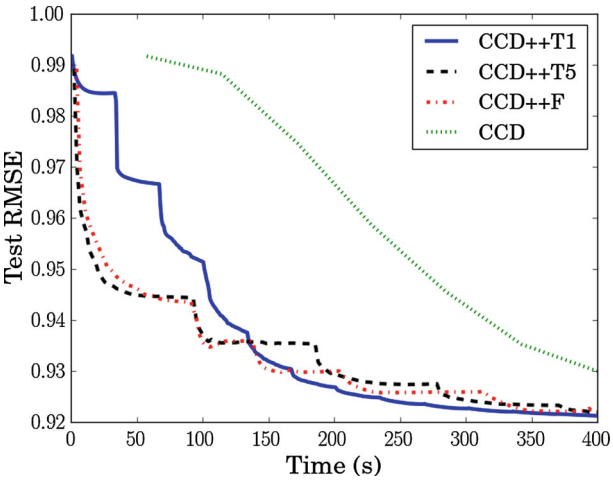



**Fig. 2** Comparison between CCD and CCD++ on netflix dataset. Clearly, CCD++, the feature-wise update approach, is seen to have faster convergence than CCD, the item/user-wise update approach.

## 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 growing interest in addressing the scalability problem by using parallel and distributed computing. 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 [7]. 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 an appropriate parallel environment based on the scale of the recommender system. Specifically, when the matrices $A$, $W$ and $H$ can be loaded in the main memory of a single machine, and we consider a distributed system as the parallel environment, the communication among machines might dominate 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++ on 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 on 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, the update to $\boldsymbol{u}$ 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 subvector of $\boldsymbol{u}$ corresponding to $S_r$. A simple strategy is to make equal-sized partitions (i.e., $|S_1| = |S_2| = \cdots = |S_p| = m/p$). The workload on the $r^{\text{th}}$ core to update $\boldsymbol{u}^r$ equals $\sum_{i \in S_r} 4|\Omega_i|$, which is not the same for all cores. As a result, this strategy leads to load imbalance, which reduces 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, we assign jobs dynamically based on the availability of each core. When a core finishes a small job, it can always start a new job without waiting for 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. Thus, from now, partition $S_r$ will refer to the indices assigned to the $r^{\text{th}}$ core as a result of this dynamic assignment. Such an approach can be also applied to update $\boldsymbol{v}$ and the residual $R$.

We now provide the details. 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

$$updatesu_i \leftarrow \frac{\sum_{j \in \Omega_i} \hat{R}_{ij} v_j}{\lambda + \sum_{j \in \Omega_i} v_j^2} \ \forall i \in S_r. \tag{26}$$

Updating $H$ can be parallelized in the same way with $G = \{G^1, \ldots, G^p\}$, which is a partition of row indices of $H$, $\{1, \ldots, n\}$. Similarly, each core $r$

$$updatesv_j \leftarrow \frac{\sum_{i \in \bar{\Omega}_j} \hat{R}_{ij} u_i}{\lambda + \sum_{i \in \Omega_i} u_i^2} \ \forall j \in G_r. \tag{27}$$

As all cores on the machine share a 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 our parallel CCD++ approach in Algorithm 3.

### 4.2 CCD++ on 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

[8] http://openmp.org/.

[9] http://threadingbuildingblocks.org/.

**Algorithm 3** Parallel CCD++ on multi-core systems

**Input:** $A$, $W$, $H$, $\lambda$, $k$, $T$
Initialize $W = 0$ and $R = A$.
**for** $iter = 1, 2, \ldots,$ **do**
  **for** $t = 1, 2, \ldots, k$ **do**
    **Parallel:** core $r$ constructs $\hat{R}$ using (25).
    **for** $inneriter = 1, 2, \ldots, T$ **do**
      **Parallel:** core $r$ updates $\boldsymbol{u}^r$ using (26).
      **Parallel:** core $r$ updates $\boldsymbol{v}^r$ using (27).
    **end for**
    **Parallel:** core $r$ updates $\bar{\boldsymbol{w}}_t^r$ and $\bar{\boldsymbol{h}}_t^r$ using (28).
    **Parallel:** core $r$ updates $R$ using (29).
  **end for**
**end for**

to parallelize CCD++ on 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 submatrices 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 Sect. 4.1 cannot be applied here because not all variables and ratings are available on all machines. Partitions $S$ and $G$ should be determined prior to any computation.

Typically, the memory required to store the residual $R$ is much larger than for $W$ and $H$, and 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 values $R_{ij}$, $\forall (i, j) \in \bar{\Omega}_{G_r}$ are required to update $H^r$, where $\Omega_{S_r} = \bigcup_{i \in S_r} \{(i, j) : j \in \Omega_i\}$, and $\bar{\Omega}_{G_r} = \bigcup_{j \in G_r} \{(i, j) : i \in \bar{\Omega}_j\}$. 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 entries $R_{ij}$, $\forall (i, j) \in \Omega^r$ are stored in machine $r$. Specifically, entries corresponding to $\Omega_{S_r}$ are stored in CRS format, and entries corresponding to $\bar{\Omega}_{G_r}$ are stored in CCS format. Thus, the entire $R$ has two copies stored on the distributed system. Assuming that the latest $R_{ij}$ corresponding to $\Omega^r$ is available on 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 CCD++ such that a complete copy of the latest $\bar{\boldsymbol{w}}_t$ and $\bar{\boldsymbol{h}}_t$ is locally available on each machine to compute $\hat{R}$:

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

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 $\boldsymbol{u}^r$.

**Algorithm 4** Parallel CCD++ on 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**

After $T$ alternating iterations, each machine $r$ has a complete 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 on each machine $r$.

The distributed version of CCD++ 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 other machines and gathers the remaining $m - |S_r|$ (or $n - |G_r|$) latest variables from other machines. Such communication can be achieved efficiently by an Allgather operation, which is a collective operation defined in the Message Passing Interface (MPI) standard.[10] With a recursive-doubling algorithm, Allgather operations can be done in

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

where $M$ is the message size in bytes, $\alpha$ is the startup time per message, independent of the message size and $\beta$ is transfer time per byte [22]. 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 Sect. 2.1, ALS can be easily parallelized when entire $W$ and $H$ can fit in the main memory of one computer. However, it is hard to be scaled up to very large-scale

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

recommender systems when $W$ or $H$ cannot fit in the memory of a single machine. When ALS updates $\boldsymbol{w}_i$, $H_{\Omega_i}$ 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, [2] 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, [7] 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 a send–receive operation. As a result, the communication time per iteration of DSGD is $\alpha p + 8mk\beta$. Thus, both DSGD and CCD++ can handle recommender systems with very large $W$ and $H$.

## 5 Experimental results

In this section, we compare CCD++, ALS and SGD in large-scale datasets under serial, multi-core and distributed platforms. For CCD++, we use the implementation with our adaptive technique based on function value reduction. We implement ALS with the Intel Math Kernel Library.[11] Based on the observation in Sect. 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 [2], all of our implementations use the weighted $\lambda$-regularization.[12]

**Datasets.** We consider four public datasets for the experiment: movielens1m, movielens10m, netflix and yahoo-music. These datasets are extensively used in the literature to test the performance of matrix factorization algorithms [3,7,23]. 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 [10] 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 add a small amount of noise to obtain our training set. We 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 $\Omega$ is generated using the Chung-Lu-Vu (CLV) model proposed in [24]. More specifically, we first sample the degree sequence $a_1, \ldots, a_m$ for all the rows following the power-law distribution $p(x) \propto x^{-c}$ with $c = -1.316$ (the parameter

[11] Our C implementation is 6x faster than the MATLAB version provided by [2].

[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).

$c$ is selected to control the number of nonzeros). We then generate another degree sequence $b_1, \ldots, b_n$ for all the columns by the same power-law distribution and normalize it to ensure $\sum_{j=1}^{n} b_j = \sum_{i=1}^{m} a_i$. Finally, each edge $(i, j)$ is sampled with probability $\frac{a_i b_j}{\sum_k b_k}$. The values of the observed entries are generated in the same way as in synthetic-u. For training/test split, we randomly select about 1 % observed entries as test set and the rest observed entries as the training set.

For each dataset, the regularization parameter $\lambda$ is chosen from $\{1, 0.5, 0.1, 0.05, 0.01, 0.005, 0.001\}$ with the lowest test RMSE. The parameter $k$ of both synthetic datasets is set according to the ground truth, and for real datasets, we choose $k$ from $\{20, 40, 60, 80, 100\}$ with the lowest test RMSE. See Table 2 for more information about the statistics and parameters used for each dataset.

### 5.1 Experiments on a single machine serial setting

We first compare CCD++ with ALS and DSGD in a serial setting.

**Experimental platform.** As mentioned in Sect. 2.3, we use an 8-core Intel Xeon X5570 processor with 32KB L1-cache, 256KB L2-cache, 8MB L3-cache and enough memory for the comparison. We only use 1 core for the serial setting in this section, while we will use multiple cores in the multi-core experiments (Sect. 5.2).

**Results on training time.** Figure 3 shows the comparison of the running time versus RMSE for the four real-world datasets in a serial setting, and we observe that CCD++ is faster than ALS and DSGD.

### 5.2 Experiments on a multi-core environment

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

**Experimental platform.** We use the same environment as in Sect. 5.1. The processor has 8 cores, and the OpenMP library is used for multi-core parallelization.

**Results on training time.** We ensure that eight cores are fully utilized for each method. Figure 4 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.

**Results on 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 one iteration. Although we have shown in Sect. 2.3 that with regard to convergence DSGD has better performance than HogWild, it remains interesting to see how HogWild performs in terms of speedup. Thus, we also include HogWild into the comparison. The results are shown in Fig. 5. Based on the slope of the curves, we observe that CCD++ and ALS have better speedup than both SGD approaches (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 the cache, time spent in loading data from memory to cache becomes the bottleneck for DSGD and HogWild to achieve better speedup when the number of cores increases.

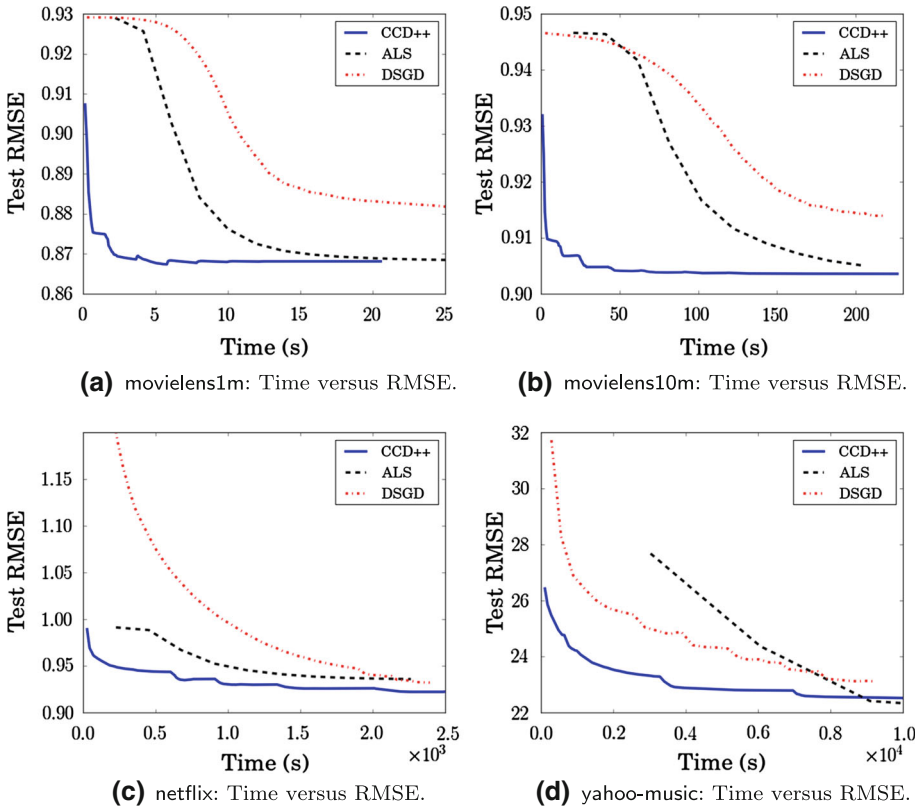



**Fig. 3** RMSE versus computation time on a serial setting for different methods (time is in seconds). Due to non-convexity of the problem, different methods may converge to different values

### 5.3 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 on a large-scale parallel platform at the Texas 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 [25] that ALS implemented with MPI is 40 to 60 times faster than its Hadoop implementation in the Mahout project. We also tried to run the ALS code provided as part of the GraphLab library[14] but in our experiments, the GraphLab code (which has an asynchronous implementation of ALS) did not converge. Hence, we developed our own implementation of ALS, using which we report all ALS results.

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

[14] We downloaded version 2.1.4679 from https://code.google.com/p/graphlabapi/.

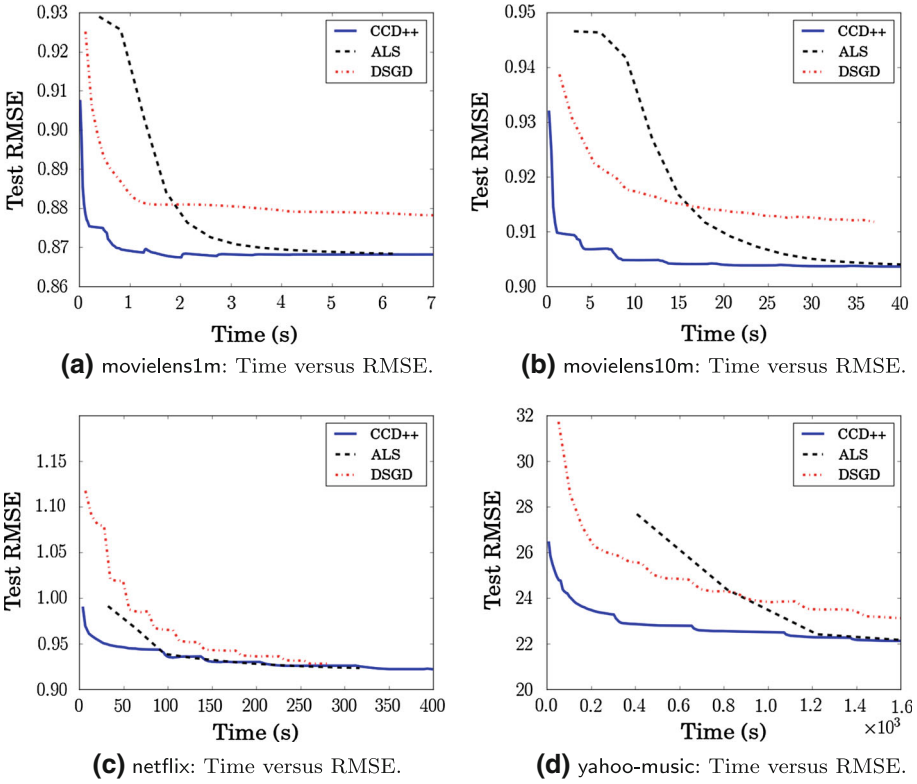



**Fig. 4** RMSE versus computation time on an 8-core system for different methods (time is in seconds). Due to non-convexity of the problem, different methods may converge to different values

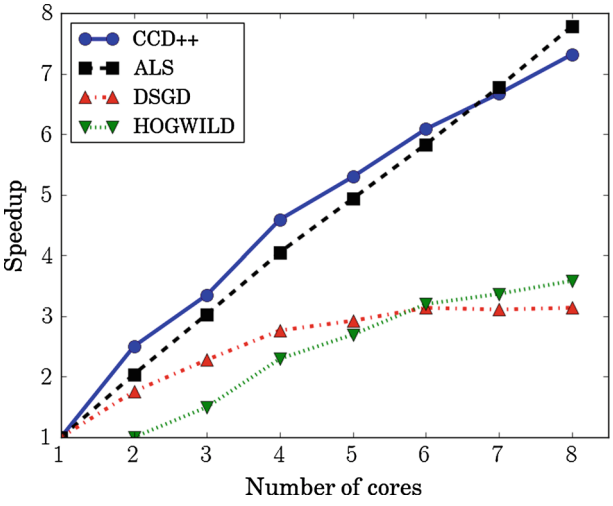



**Fig. 5** Speedup comparison among four algorithms with the yahoo-music dataset on a shared-memory multi-core machine. CCD++ and ALS have better speedups than DSGD and HogWild because of better locality

**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 6 shows the result with 4 computing nodes—we can make similar observations as in Fig. 4.

**Results on synthetic datasets.** When data is large enough, the benefit of distributed environments is obvious.

For the scalability comparison, we vary the number of computing nodes, ranging from 32 to 256, and compare the time and speedup for three algorithms on the synthetic-u and synthetic-p datasets. As discussed in Sect. 4, ALS requires larger memory on each machine. In our setting, it requires more than 32GB memory when using 32 nodes on synthetic-p dataset, so we run each algorithm with at least 64 nodes for this dataset. 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 Figs. 7a and 8a. 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 Figs. 7b and 8b. 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 s using 32 machines and 218 s using 256 machines, so 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 caching effect. 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 the L2-cache, which leads to dramatic reduction in the memory access time. On the other hand, when the number of machines is not large enough, these blocks cannot fit into cache. Thus, DSGD, which accesses entries in the block at random, suffers from frequent cache misses. In contrast, for CCD++ and ALS, the cache miss is not that severe even when the block of $W$ and $H$ cannot fit into cache since the memory is accessed sequentially in both methods.

Though 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 L2-regularization, L1-regularization is used in many applications to obtain a sparse model, such as linear classification [26]. Replacing the L2-regularization in (1), we have the following L1-regularized problem:

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

which tends to yield a more sparse $W$ and $H$.

### 6.1 Modification for each method

In this section, we explore how CCD, CCD++, ALS and SGD can be modified to solve (33).

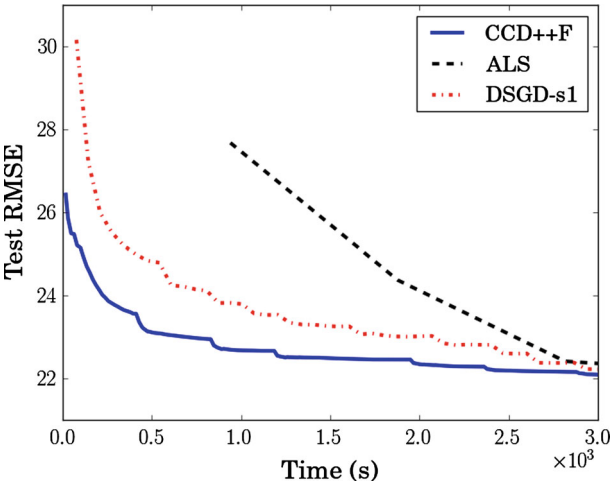
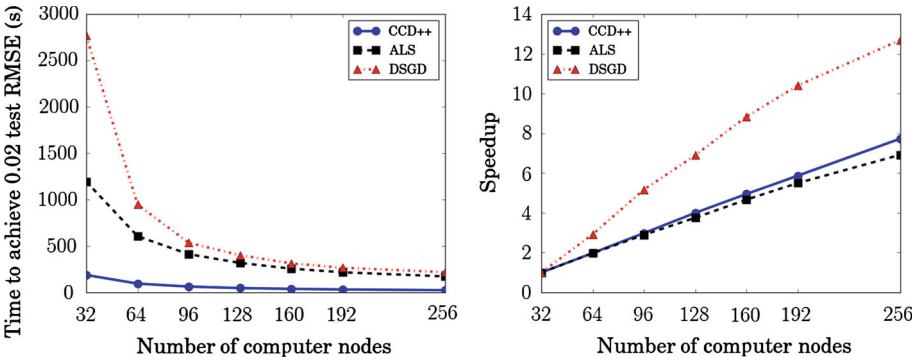



**Fig. 6** Comparison among CCD++, ALS and DSGD with the yahoo-music dataset on a MPI distributed system with 4 computing nodes




**(a)** Number of computation nodes versus training time.

**(b)** Number of computation nodes versus speedup.

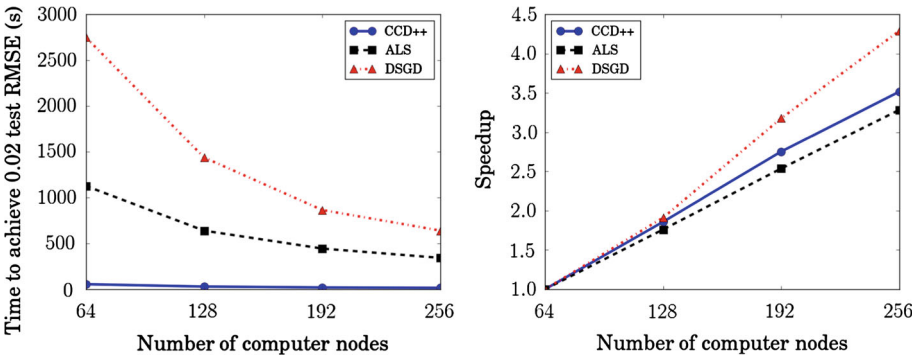
**Fig. 7** Comparison among CCD++, ALS and DSGD on the synthetic-u dataset (9 billion ratings) on a MPI distributed system with varying number of computing nodes. The *vertical axis* in the *left panel* is the time for each method to achieve 0.01 test RMSE, while the *right panel* shows the speedup for each method. Note that, as discussed in Sect. 5.3, speedup is $T_p/T_{32}$, where $T_p$ is the time taken on $p$ machines

**CCD and CCD++.** When we apply coordinate descent methods to (33), the one-variable subproblem becomes

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

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

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




**(a)** Number of computation nodes versus training time.

**(b)** Number of computation nodes versus speedup.

**Fig. 8** Comparison among CCD++, ALS and DSGD on the synthetic-p dataset (14.6 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 Sect. 5.3, speedup is $T_p/T_{64}$, where $T_p$ is the time taken on $p$ machines

where

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

Similar to the situation with L2-regularization, by maintaining the residual matrix $R$, the time complexity for each single variable update can be reduced to $O(|\Omega_i|)$. Thus, CCD and CCD++ can be applied to solve (33) efficiently.

**ALS.** When we apply ALS to (33), the second term in each subproblem (2) is replaced by a non-smooth term $\lambda\|\boldsymbol{w}_i\|_1$. The resulting problem does not have a closed form solution. As a result, an iterative method is required to solve the subproblem. If coordinate descent is applied to solve this problem, ALS and CCD become exactly the same algorithm.

**SGD.** When SGD is applied to solve the non-smooth problem, the gradient in the update rule has to be replaced by the subgradient, and thus, the update rule corresponding the $(i, j)$ rating becomes

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

where $R_{ij} = A_{ij} - \boldsymbol{w}_i^T\boldsymbol{h}_j$. The time complexity for each update is the same as the one with L2-regularization. Similarly, the same trick in DSGD and HogWild can be used to parallelize SGD with L1-regularization as well.

Figure 9 presents the comparison of the multi-core version of parallel CCD++ and DSGD with L1-regularization on two datasets: movielens10m and yahoo-music. In this comparison, we use the same experimental settings and platform in Sect. 4.1.

### 6.2 Experimental results

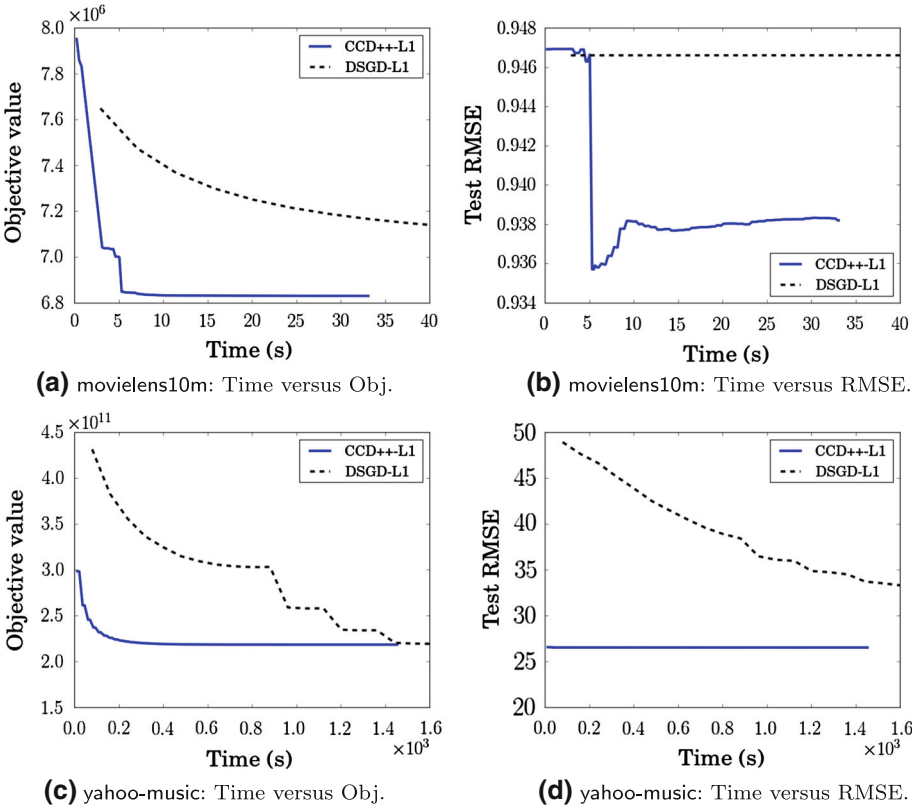
First, we compare the solution of L2-regularized matrix factorization problem (1) versus the L1-regularized one (33) in Table 3. Although L1-regularized form achieves worse test RMSE comparing to L2-regularized form, it can successfully yield sparse models $W$ and $H$, which is important for interpretation in many applications.

We then compare the convergence speed of CCD++ and DSGD for solving the L1-regularized problem (33). Figures 9a and c present the results of the objective function values versus computation time. In both figures, we clearly observe that CCD++ has faster convergence than DSGD, which demonstrates the superiority of CCD++ to solve (33).

**Table 3** The best test RMSE for each model (the lower, the better)

| | Movielens10m | Yahoo-music |
|---|---|---|
| L1-regularization | 0.9381 | 24.49 |
| L2-regularization | **0.9035** | **21.92** |

We run both CCD++ and DSGD with a large number of iterations to obtain the best test RMSE for each model




**(a)** movielens10m: Time versus Obj. **(b)** movielens10m: Time versus RMSE.

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

**Fig. 9** The results of RMSE and objective function value 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

Meanwhile, Figs. 9b and d show the results of the test RMSE versus computation time. Similar to L2-regularized case, CCD++ achieves better test RMSE than DSGD for both datasets. However, (33) is designed to obtain better sparsity of $W$ and $H$ instead of improving the generalization error of the model. As a result, the test RMSE might be sacrificed for a more sparse $W$ 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 49 times faster to achieve the desired test accuracy than DSGD when we use 64 processors, and when we use 256 processors, CCD++ is 40 times faster than DSGD and 20 times faster than ALS.

**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

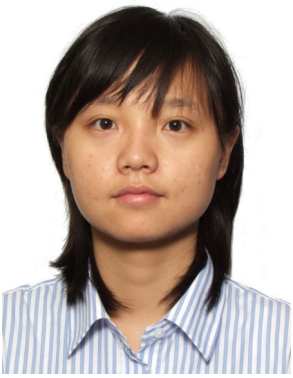
1. Dror G, Koenigstein N, Koren Y, Weimer M (2012) The Yahoo! music dataset and KDD-Cup'11. In: JMLR workshop and conference proceedings: proceedings of KDD Cup 2011 competition, vol. 18, pp 3–18
2. Zhou Y, Wilkinson D, Schreiber R, Pan R (2008) Large-scale parallel collaborative filtering for the Netflix prize. In: Proceedings of international conference on algorithmic aspects in, information and management
3. Koren Y, Bell RM, Volinsky C (2009) Matrix factorization techniques for recommender systems. IEEE Comput 42:30–37
4. Takács G, Pilászy I, Németh B, Tikk D (2009) Scalable collaborative filtering approaches for large recommender systems. JMLR 10:623–656
5. Chen P-L, Tsai C-T, Chen Y-N, Chou K-C, Li C-L, Tsai C-H, Wu K-W, Chou Y-C, Li C-Y, Lin W-S, Yu S-H, Chiu R-B, Lin C-Y, Wang C-C, Wang P-W, Su W-L, Wu C-H, Kuo T-T, McKenzie TG, Chang Y-H, Ferng C-S, Niv, Lin H-T, Lin C-J, Lin S-D (2012) A linear ensemble of individual and blended models for music. In: JMLR workshop and conference proceedings: proceedings of KDD cup 2011 competition, vol. 18, pp 21–60
6. Langford J, Smola A, Zinkevich M (2009) Slow learners are fast. In: NIPS
7. Gemulla R, Haas PJ, Nijkamp E, Sismanis Y (2011) Large-scale matrix factorization with distributed stochastic gradient descent. In: ACM KDD
8. Recht B, Re C, (2013) Parallel stochastic gradient algorithms for large-scale matrix completion. Math Program Comput 5(2): 201–226
9. Zinkevich M, Weimer M, Smola A, Li L (2010) Parallelized stochastic gradient descent. In: NIPS
10. Niu F, Recht B, Re C, Wright SJ (2011) Hogwild!: a lock-free approach to parallelizing stochastic gradient descent. In: NIPS

11. Cichocki A, Phan A-H (2009) Fast local algorithmsfor large scale nonnegative matrix and tensor factorizations. IEICE Trans Fundam Electron Commun Comput Sci, vol. E92-A, no. 3, pp 708–721
12. Hsieh C-J, Dhillon IS (2011) Fast coordinate descent methods with variable selection for non-negative matrix factorization. In: ACM KDD
13. Bottou L (2010) Large-scale machine learning with stochastic gradient descent. In: proceedings of international conference on, computational statistics
14. Agarwal A, Duchi JC (2011) Distributed delayed stochastic optimization. In: NIPS
15. Bertsekas DP (1999) Nonlinear programming. Belmont, MA 02178–9998: Athena Scientific, second ed.
16. Hsieh C-J, Chang K-W, Lin C-J, Keerthi SS, Sundararajan S (2008) A dual coordinate descent method for large-scale linear SVM. In: ICML
17. Yu H-F, Huang F-L, Lin C-J (2011) Dual coordinate descent methods for logistic regression and maximum entropy models. Mach Learn 85(1–2):41–75
18. Hsieh C-J, Sustik M, Dhillon IS, Ravikumar P (2011) Sparse inverse covariance matrix estimation using quadratic approximation. In: NIPS
19. Pilászy I, Zibriczky D, Tikk D (2010) Fast ALS-based matrix factorization for explicit and implicit feedback datasets. In: ACM RecSys
20. Bell RM, Koren Y, Volinsky C (2007) Modeling relationships at multiple scales to improve accuracy of large recommender systems. In: ACM KDD
21. Ho N-D, Blondel PVDVD (2011) Descent methods for nonnegative matrix factorization. In: numerical linear algebra in signals, systems and control. Springer: Netherlands, SA, pp 251–293
22. Thakur R, Gropp W (2003) Improving the performance of collective operations in MPICH. In: proceedings of European PVM/MPI users' group meeting
23. Low Y, Gonzalez J, Kyrola A, Bickson D, Guestrin C, Hellerstein JM (2010) Graphlab: a new framework for parallel machine learning. CoRR, vol. abs/1006.4990
24. Chung F, Lu L, Vu V (2003) The spectra of random graphs with given expected degrees. Intern Math, 1(3): 257–275
25. Low Y, Gonzalez J, Kyrola A, Bickson D, Guestrin C, Hellerstein JM (2012) Distributed graphlab: a framework for machine learning in the cloud. PVLDB 5(8): 716–727
26. Yuan G-X, Chang K-W, Hsieh C-J, Lin C-J (2010) A comparison of optimization methods and software for large-scale l1-regularized linear classification. J Mach Learn Res 11:3183–3234

## Author Biographies

**Hsiang-Fu Yu** is currently a Ph.D. student at University of Texas at Austin (UT-Austin). He received his B.S. degree in 2008 and M.S degree in 2010 from the Computer Science Department of National Taiwan University (NTU). At NTU, Hsiang-Fu conducted his research with Prof. Chih-Jen Lin and received the Best research paper award in KDD 2010. He was also a member of NTU team, which won the first prize of KDD Cup 2010 and the third place of KDD Cup 2009. At UT-Austin, Hsiang-Fu is a member of data mining group led by Prof. Inderjit Dhillon. Along with his colleague, he received the best paper award in ICDM 2010. His research interests focus on large-scale machine learning and data mining.

**Cho-Jui Hsieh** is currently a Ph.D. student at University of Texas at Austin (UT-Austin). He received his B.S. degree in 2007 and M.S degree in 2009 from the Computer Science Department of National Taiwan University (NTU). At NTU, Cho-Jui Hsieh conducted his research with Prof. Chih-Jen Lin and received the Best research paper award in KDD 2010. At UT-Austin, Cho-Jui is a member of data mining group led by Prof. Inderjit Dhillon. Along with his colleague, he received the best paper award in ICDM 2010. His research interests focus on large-scale machine learning and data mining.



**Si Si** is currently a Ph.D. student in the Computer Science Department at the University of Texas at Austin (UT-Austin) where she is working with Professor Inderjit Dhillon. Si Si received the BEng degree from the University of Science and Technology of China (USTC) in 2008 and M.Phil. degree from the University of Hong Kong(HKU) in 2010. Her research interests include data mining and machine learning, especially big data and social network analysis.

**Inderjit S. Dhillon** is a Professor of Computer Science and Mathematics at The University of Texas at Austin. Inderjit received his B.Tech. degree from the Indian Institute of Technology at Bombay, and Ph.D. degree from the University of California at Berkeley. His Ph.D. dissertation led to the fastest known numerically stable algorithm for the symmetric tridiagonal eigenvalue/eigenvector problem. Software based on this work is now part of all state-of-the-art numerical software libraries. Inderjit's current research interests are in big data, machine learning, network analysis, numerical optimization and scientific computing. Inderjit received an NSF Career Award in 2001, a University Research Excellence Award in 2005, the SIAG Linear Algebra Prize in 2006, the SIAM Outstanding Paper Prize in 2011 and the ICES Distinguished Research Award in 2013. Along with his students, he has received several best paper awards at leading data mining and machine learning conferences. Inderjit has published over 100 journal and conference papers, and serves on the Editorial Board of the Journal of Machine Learning Research, the IEEE Transactions of Pattern Analysis and Machine Intelligence, and Foundations and Trends in Machine Learning. He is a member of the ACM, SIAM, AAAS and a Senior Member of the IEEE.