Scalable Coordinate Descent Approaches to Parallel Matrix Factorization for Recommender Systems

Hsiang-Fu Yu, Cho-Jui Hsieh, Si Si, and Inderjit Dhillon
Department of Computer Science
University of Texas at Austin
Austin, Texas, USA
{*rofuyu,cjhsieh,ssi,inderjit*}*@cs.utexas.edu*

***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 are faster and have 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, on a synthetic dataset with 2 billion ratings, CCD++ is 4 times faster than both SGD and ALS using a distributed system with 20 machines.**

***Keywords*-Recommender systems, Matrix factorization, Low rank approximation, Parallelization.**

I. 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 factorization was empirically shown to be a better model than traditional nearest-neighbor approaches in the Netflix Prize competition, and 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]–[3].

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

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

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 $\boldsymbol{w}_i^T \boldsymbol{h}_j$. Because of 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 not a coincidence then 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, compared to ALS, SGD usually needs more iterations to obtain a good enough model, and the 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]–[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

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

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 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} will denote the (i,j) entry of the matrix A. We will 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 II. We then present our coordinate descent approaches in Section III. In Section IV, we give the scalability analysis under different parallel computing environments. Finally, we present experimental results in Section V and conclusions in Section VI.

II. 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 both of them and discuss recent parallelization approaches.

A. 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 alternatively switches between optimizing W while keeping H fixed, and optimizing H while keeping W fixed. Thus, ALS monotonically decreases the objective value of (1) until convergence.

Under this alternating optimization scheme, (1) can be further separated into many independent least squares subproblems. Specifically, if we fix H to 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_{Ω_i} is the sub-matrix formed by $\{\boldsymbol{h}_j : j \in \Omega_i\}$, and $\boldsymbol{a}_i^T$ is the i^{th} row of A with missing entries filled by zeros. To update 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 another $O(k^3)$ 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 from the updates of other rows. 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 IV-B.

B. 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

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

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 $O(|\Omega|k)$. Compared to ALS, it is faster in terms of the time complexity for one iteration.

However, conducting several SGD updates in parallel directly might raise the overwriting issue because 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.

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

Thus, DSGD can be regarded as exact SGD with a specific ordering of updates.

Another issue with SGD is its convergence, which 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 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 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 I of Section V). Here we conduct the comparison on a 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 the platform 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 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.

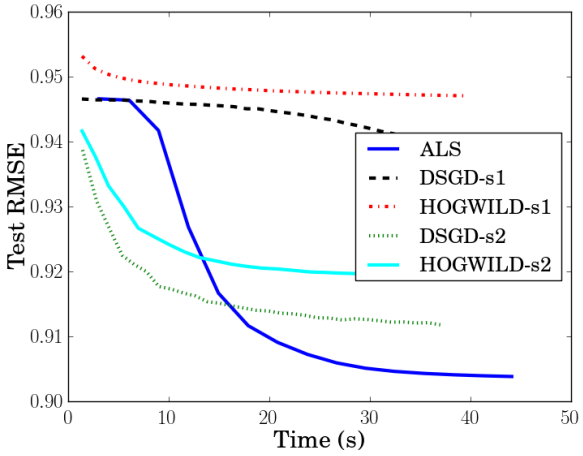


Figure 1. Comparison between ALS, DSGD, and HogWild on the movielens10m dataset with $k = 40$ on a 8-core machine (-s1 and -s2 stand for different initial learning rates).

III. 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 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.

A. 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

[3] Intel MKL is used in the 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 $\eta = 0.05$.

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.

B. 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 is a method which adopts this update sequence. As mentioned in Section II-A, 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 CCD Algorithm [17]

Input: Initial $R = A$, $W = 0$, H, λ, and k
  for $iter = 1, 2, \ldots, T$ **do**
    for $i = 1, 2, \ldots, m$ **do**
      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**
      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

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.

As mentioned in Section III-A, 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 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 $w_{i1}, \ldots, w_{ik}$, one may think of a greedy update sequence that sequentially updates the variables that decrease the objective function the most. In [10], a greedy update sequence is applied to solve the NMF problem in an efficient manner which utilizes the property that all subproblems in NMF share the same Hessian. However, unlike NMF, each subproblem (2) in problem (1) has a potentially different Hessian as $\Omega_{i_1} \neq \Omega_{i_2}$ for $i_1 \neq i_2$ in general. Thus, the greedy coordinate descent (GCD) method proposed by [10] would require $O(|\Omega|k^2)$ operations per iteration.

C. Feature-wise Update: CCD++

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

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

Algorithm 2 CCD++ Algorithm

Input: Initial $R = A$, $W = 0$, H, λ, and k
  for $iter = 1, 2, \ldots$ **do**
    for $t = 1, 2, \ldots, k$ **do**
      Get $(\boldsymbol{u}^*, \boldsymbol{v}^*)$ using T CCD iterations for (16).
      Update R and $(\bar{\boldsymbol{w}}_t, \bar{\boldsymbol{h}}_t)$ using (17) and (18).
    end for
  end for

where $\bar{\boldsymbol{w}}_t \in \mathbb{R}^m$ is the t^{th} column of W, and $\bar{\boldsymbol{h}}_t \in \mathbb{R}^n$ is the t^{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^{th} latent feature.

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 $\hat{R}_{ij} = R_{ij} + \bar{w}_{ti}\bar{h}_{tj}\ \forall (i,j) \in \Omega$, (15) can be rewritten as:

$$\min_{\boldsymbol{u}\in\mathbb{R}^m, \boldsymbol{v}\in\mathbb{R}^n} \sum_{(i,j)\in\Omega} (\hat{R}_{ij} - u_i v_j)^2 + \lambda(\|\boldsymbol{u}\|^2 + \|\boldsymbol{v}\|^2), \tag{16}$$

which is exactly the rank-one matrix factorization problem (1) for the matrix $\hat{R}$. Thus we can apply CCD on (16) to obtain an approximation by alternatively updating $\boldsymbol{u}$ and updating $\boldsymbol{v}$. 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. After obtaining $(\boldsymbol{u}^*, \boldsymbol{v}^*)$, we can update $(\bar{\boldsymbol{w}}_t, \bar{\boldsymbol{h}}_t)$ and R by

$$R_{ij} \leftarrow \hat{R}_{ij} - u_i^* v_j^*,\ \forall (i,j) \in \Omega, \tag{17}$$

$$(\bar{\boldsymbol{w}}_t, \bar{\boldsymbol{h}}_t) \leftarrow (\boldsymbol{u}^*, \boldsymbol{v}^*). \tag{18}$$

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

$$\bar{\boldsymbol{w}}_1, \bar{\boldsymbol{h}}_1, \ldots, \bar{\boldsymbol{w}}_t, \bar{\boldsymbol{h}}_t, \ldots, \bar{\boldsymbol{w}}_k, \bar{\boldsymbol{h}}_k. \tag{19}$$

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.

For the t^{th} feature, variable updating in CCD++ consists of three parts: T CCD iterations to approximate $\hat{R}$, the update of $|\Omega|$ residual entries by (17), and the update of $(\bar{\boldsymbol{w}}_t, \bar{\boldsymbol{h}}_t)$ by (18). Thus the time complexity per iteration for CCD++ is $O(|\Omega|k)$.

At first glance, the only difference between CCD++ and CCD is their update sequences. 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.

D. An Adaptive Technique to Accelerate CCD++

In this section, we investigate how to accelerate CCD++ by controlling the number of CCD iterations for each subproblem (16). The approaches [9], [19] with the feature-wise update sequence to solve NMF problems consider only one iteration for each subproblem. However, CCD++ could be slightly more efficient when $T > 1$ due to the benefit brought by the "delayed residual update." Note that R is fixed during CCD iterations for each rank-one approximation (16), and so the residual update (17) is required only when we switch to the next subproblem corresponding to another feature. The ratio of the computation spent on residual maintenance over that spent on variable updating is $O(1/T)$ for CCD++. Thus, given the same number of variable updates, CCD++ with T CCD iterations is $O(\frac{2T}{T+1})$ times faster than that with only one CCD iteration. Moreover, the more CCD iterations we use, the better the approximation to subproblem (16). 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 (17), 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 (16) can be efficiently obtained by accumulating reductions from the update of each single variable. For example, updating u_i to

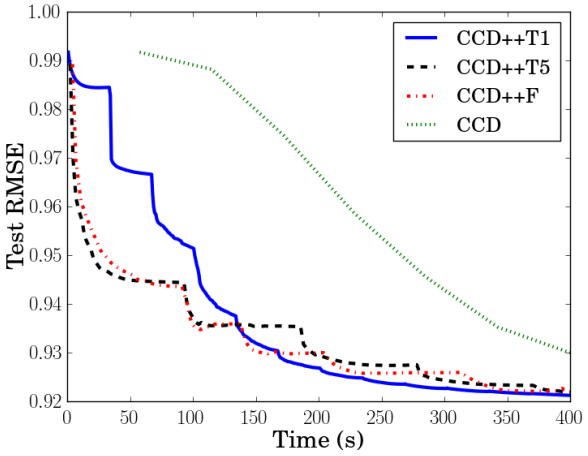


Figure 2. Comparison between CCD and CCD++ on netflix dataset. Clearly, CCD++, the feature-wise update approach, has faster convergence than CCD, the item/user-wise update approach.

u_i^* decreases the function by

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

where both terms are available when updating u_i. Thus function value reduction can be obtained without extra effort.

We close this section by a comparison between CCD and CCD++ in Figure 2. Here we compare four settings with the netflix dataset on a machine with enough memory: one setting is the item/user-wise CCD, and three others are CCD++ with fixed $T = 1$ (CCD++T1), fixed $T = 5$ (CCD++T5), and adaptive T based on the function value reduction (CCD++F, $\epsilon = 10^{-3}$ is used), respectively. From the figure, we clearly observe that the feature-wise update approach CCD++, even when $T = 1$, converges faster than CCD, which confirms the observation for NMF in [19]. We also observe that larger T improves convergence of 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). We also notice that the technique to adaptively control T slightly shortens such periods.

IV. 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 parallelization: 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.

A. 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 (16). In CCD++, the approximate solution to the subproblem is obtained by updating $\boldsymbol{u}$ and $\boldsymbol{v}$ alternately. If we fix $\boldsymbol{v}$, each variable in $\boldsymbol{u}$ can be independently updated by (5) and (8). Therefore, we are able to divide the task of updating $\boldsymbol{u}$ into several independent subtasks that 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. Each core r then

$$\textit{updates } u_i \textit{ by } (5) \textit{ and } (8), \ \forall i \in S_r. \tag{20}$$

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

$$\textit{updates } v_j \textit{ by } (11) \textit{ and } (10), \ \forall j \in G_r. \tag{21}$$

As all cores in a machine share 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:

$$R_{ij} \leftarrow R_{ij} + \bar{w}_{ti}\bar{h}_{tj} - u_i v_j, \ \forall (i,j) \in \Omega_{S_r}, \tag{22}$$

$$(\bar{\boldsymbol{w}}_t^r, \bar{\boldsymbol{h}}_t^r) \leftarrow (\boldsymbol{u}^r, \boldsymbol{v}^r), \tag{23}$$

where $\Omega_{S_r} = \bigcup_{i \in S_r} \{(i,j) : j \in \Omega_i\}$. We summarize the parallel CCD++ in Algorithm 3.

Algorithm 3 Parallel CCD++ in multi-core systems

Input: Initial $R = A$, $W = 0$, H, λ, and k
  for $iter = 1, 2, \ldots,$ **do**
    for $t = 1, 2, \ldots, k$ **do**
      Initialize $\boldsymbol{u} = \mathbf{0}$ and $\boldsymbol{v} = \mathbf{0}$.
      for $inneriter = 1, 2, \ldots, T$ **do**
        Parallel: core r updates $\boldsymbol{u}^r$ using (20).
        Parallel: core r updates $\boldsymbol{v}^r$ using (21).
        Parallel: core r updates R using (22).
        Parallel: core r updates $\bar{\boldsymbol{w}}_t^r$ and $\bar{\boldsymbol{h}}_t^r$ using (23).
      end for
    end for
  end for

Algorithm 4 Parallel CCD++ in distributed systems

Input: Initial $R = A$, $W = 0$, H, λ, and k
  for $iter = 1, 2, \ldots$ **do**
    for $t = 1, 2, \ldots, k$ **do**
      Broadcast: machine r broadcasts $\bar{\boldsymbol{w}}_t^r$ and $\bar{\boldsymbol{h}}_t^r$.
      for $inneriter = 1, 2, \ldots T$ **do**
        Parallel: machine r updates $\boldsymbol{u}^r$ using (20).
        Broadcast: machine r broadcasts $\boldsymbol{u}^r$.
        Parallel: machine r updates $\boldsymbol{v}^r$ using (21).
        Broadcast: machine r broadcasts $\boldsymbol{v}^r$.
      end for
      Parallel: machine r updates R using (24).
      Parallel: machine r updates $\bar{\boldsymbol{w}}_t^r$, $\bar{\boldsymbol{h}}_t^r$ using (23).
    end for
  end for

B. 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 (16) 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.

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 in a distributed system such that we can conduct all updates in CCD++ 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, and $R_{ij}\ \forall (i,j) \in \bar{\Omega}_{G_r}$ 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 accessed from machine r:

$$\Omega^r = \Omega_{S_r} \cup \bar{\Omega}_{G_r}.$$

Thus, we only store $R_{ij}\ \forall (i,j) \in \Omega^r$ 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 subproblem (16). Unlike the shared-memory environment, we need to broadcast $\bar{\boldsymbol{w}}_t$ and $\bar{\boldsymbol{h}}_t$ in the distributed version of parallel CCD++ such that each machine has a complete local copy of the latest $\bar{\boldsymbol{w}}_t$ and $\bar{\boldsymbol{h}}_t$ for updating $\boldsymbol{u}^r$. Similarly, machine r needs to broadcast the updated $\boldsymbol{u}^r$ to others before updating $\boldsymbol{v}^r$.

After each machine obtains $((\boldsymbol{u}^*)^r, (\boldsymbol{v}^*)^r)$ by T alternating iterations, the residual R can also be updated without extra communication as $(\bar{\boldsymbol{w}}_t^r, \bar{\boldsymbol{h}}_t^r)$ is also available in each machine r. Therefore machine r can update $R_{ij}\ \forall (i,j) \in \Omega^r$ by

$$R_{ij} \leftarrow R_{ij} + \bar{w}_{ti}\bar{h}_{tj} - u_i^* v_j^*\ \forall (i,j) \in \Omega^r, \qquad (24)$$

and update $(\bar{\boldsymbol{w}}_t, \bar{\boldsymbol{h}}_t)$ by (23). Our parallel CCD++ method in a distributed system is described in Algorithm 4.

C. Scalability Analysis of Other Methods

As mentioned in Section II-A, 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

Table I
THE STATISTICS AND PARAMETERS FOR EACH DATASET

dataset	movielens10m	netflix	yahoo-music	synthetic
m	71,567	2,649,429	1,000,990	1,000,000
n	65,133	17,770	624,961	1,000,000
$\|\Omega\|$	9,301,274	99,072,112	252,800,275	1,999,822,277
$\|\Omega^{\text{Test}}\|$	698,780	1,408,395	4,003,960	21,736,041
k	40	40	100	10
λ	0.1	0.05	1	0.001

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 restricted ordering. Similar to our approach, DSGD stores W, H, and A in a distributed manner such that each machine only needs to store $O((n+m)k/p)$ variables and $O(|\Omega|/p)$ rating entries. Thus both DSGD and CCD++ can handle recommender systems with very large W and H.

V. 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 one adaptive technique based on the function value reduction. We implement parallel ALS with the Intel Math Kernel Library.[8] Based on the observation in Section II, we choose DSGD among variants of parallel SGD for its faster and more stable convergence. Each algorithm is implemented in C++ to make a fair comparison. Similar to [1], all of our implementations use the weighted λ regularization.[9]

Datasets. We consider three public datasets for the experiment: 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 dataset from a 1M by 1M matrix with rank 10. We first build the ground truth W and H with each variable drawn from a zero-mean Guassian distribution with a small variance. We then randomly sample about 2 billion entries from WH^T with a small noise as our training set and sample about 20 million entries without noise as the test set. See Table I for more information about the statistics and parameters used (k and λ) for each dataset.

A. 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[10] 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 three 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 is a parallel algorithm when we increase the number of cores. 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. We also include HogWild in Figure 4. We 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.

B. 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. We use a MPI[11] cluster as our distributed platform. Each computing node in the cluster is a Intel Xeon 5355 2.66 GHz CPU machine with 16 GB memory and communicates by Fast Ethernet (100 Mbps). For a fair comparison, we implement a distributed version with MPI in C++ for each method. The reason we do not use Hadoop is that almost all operations in Hadoop need to access data and variables from disks, which is slow and not suited for iterative methods. It is reported in [20] that ALS implemented with MPI is 40 to 60 times faster than the Hadoop implementation in the Mahout project.

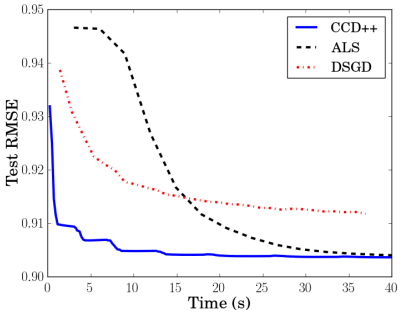
Results on yahoo-music. First we show the comparison on yahoo-music dataset, which is the largest real-world dataset we used. Figure 5 shows the result with 4 computing nodes – we can make similar observations as in Figure 3. In a distributed environment, CCD++ still outperforms ALS and DSGD. However, one may find that distributed CCD++ with 4 nodes is slower than CCD++ on a multi-core machine. This is a result of the high communication cost introduced in the distributed system. Thus, when data can fit in the memory of a single machine, the multi-core setting would be a better choice for parallelization.

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

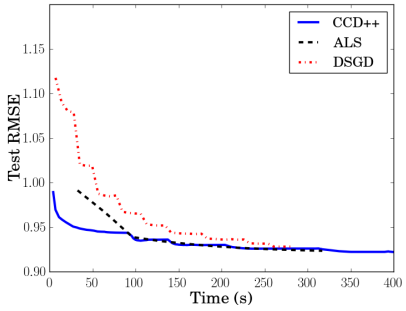
[9] $\lambda\left(\sum_i |\Omega_i|\|\boldsymbol{w}_i\|^2 + \sum_j |\bar{\Omega}_j|\|\boldsymbol{h}_j\|^2\right)$ is used to replace the regularization term in (1).

[10] http://openmp.org/

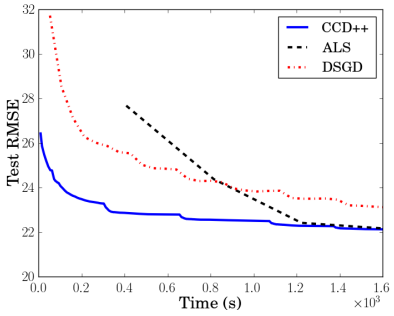
[11] http://www.mcs.anl.gov/research/projects/mpi/



(a) movielens10m: Time versus RMSE.



(b) netflix: Time versus RMSE.



(c) yahoo-music: Time versus RMSE.

Figure 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.

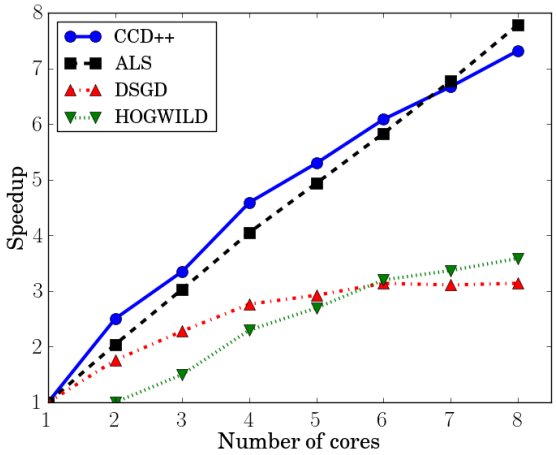


Figure 4. Speedup comparison between three algorithms in a shared-memory multi-core environment. All of them have nearly linear speedup. CCD++ and ALS have better performance than DSGD and HogWild because of better locality.

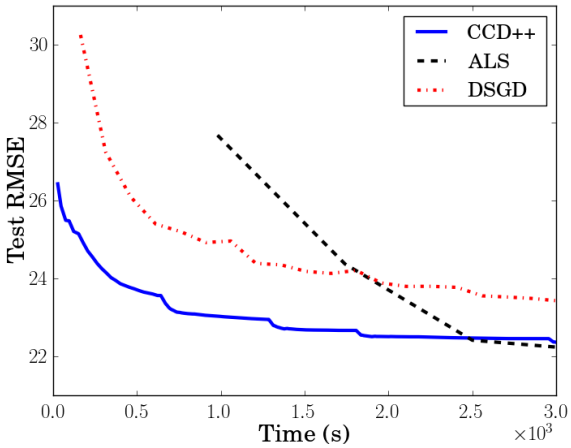
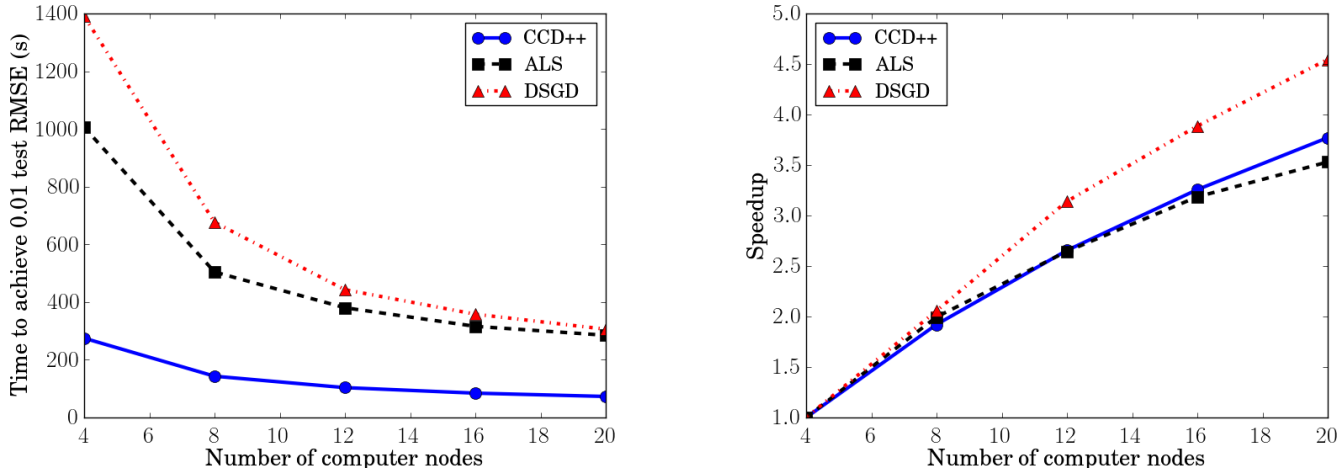


Figure 5. Comparison among CCD++, ALS, and DSGD with the yahoo-music dataset on a MPI distributed system with 4 computing nodes.

Results on synthetic data. When data is large enough, we believe that the benefit of distributed environments would be obvious. In our implementation, which uses the compressed sparse row format, the memory need for synthetic data (see Table I for details) is more than 48 GB. Given the fact that each node only has 16 GB memory, it requires at least 4 computing nodes to load this data. To conduct scalability comparison, each method is run with varying numbers of computing nodes, ranging from 4 to 20. Here we calculate the time taken by each method to achieve 0.01 test RMSE. The result is shown in Figure 6a. We can see clearly that our method CCD++ outperforms both DSGD and ALS. Specifically, CCD++ is 4 times faster than both DSGD and ALS. We also show the speedup in Figure 6b. Note that since the data cannot be loaded in memory of a single machine, the speedup using p machines is T_p/T_4, where T_p is the time taken on p machines. All three methods have similar speedup, although the speedup of DSGD is slightly better than the other two because of its lower communication cost per iteration. Note 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 in distributed systems.

VI. 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



(a) Number of computation nodes versus training time.

(b) Number of computation nodes versus speedup.

Figure 6. Comparison among CCD++, ALS and DSGD on the synthetic dataset (2 billion ratings) in 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 V-B, speedup is T_p/T_4, where T_p is the time taken on p machines.

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, on a large-scale synthetic dataset (2 billion ratings), CCD++ is 4 times faster than both ALS and DSGD on a distributed system with 20 machines.

VII. Acknowledgments

This research was supported by NSF grants CCF-0916309, CCF-1117055, and DOD Army grant W911NF-10-1-0529.

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," *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, pp. 41–75, October 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 *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] 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, pp. 716–727, 2012.