Scalable Distributed Nonnegative Matrix Factorization with Block-Wise Updates

Jiangtao Yin, Lixin Gao, *Fellow, IEEE*, and Zhongfei Zhang

Abstract—Nonnegative Matrix Factorization (NMF) has been applied with great success on a wide range of applications. As NMF is increasingly applied to massive datasets such as web-scale dyadic data, it is desirable to leverage a cluster of machines to store those datasets and to speed up the factorization process. However, it is challenging to efficiently implement NMF in a distributed environment. In this paper, we show that by leveraging a new form of update functions, we can perform local aggregation and fully explore parallelism. Therefore, the new form is much more efficient than the traditional form in distributed implementations. Moreover, under the new form of update functions, we can perform frequent updates and lazy updates, which aim to use the most recently updated data whenever possible and avoid unnecessary computations. As a result, frequent updates and lazy updates are more efficient than their traditional concurrent counterparts. Through a series of experiments on a local cluster as well as the Amazon EC2 cloud, we demonstrate that our implementations with frequent updates or lazy updates are up to two orders of magnitude faster than the existing implementation with the traditional form of update functions.

Index Terms—NMF, block-wise updates, frequent updates, lazy updates, concurrent updates, MapReduce

1 Introduction

Nonnegative matrix factorization (NMF) [2] is a popular dimension reduction and factor analysis method that has attracted a lot of attention recently. It arises from a wide range of applications, including genome data analysis [3], text mining [4], and recommendation systems [5]. NMF factorizes an original matrix into two nonnegative low-rank factor matrices by minimizing a loss function, which measures the discrepancy between the original matrix and the product of the two factor matrices. Due to its wide applications, many algorithms [6], [7], [8], [9], [10], [11], [12] for solving it have been proposed. NMF algorithms typically leverage update functions to iteratively and alternatively refine factor matrices.

Many practitioners nowadays have to deal with NMF on massive datasets. For example, recommendation systems in web services such as Netflix have been dealing with NMF on web-scale dyadic datasets, which involve millions of users, millions of movies, and billions of ratings. For such web-scale matrices, it is desirable to leverage a cluster of machines to speed up the factorization process. MapReduce and its variants [13], [14], [15] has emerged as a popular distributed framework for data intensive computation. It provides a simple programming model where a user can focus on the computation logic without worrying about the complexity of parallel computation. Prior approaches (e.g., [16]) of handling NMF on MapReduce usually pick an NMF algorithm and then focus on implementing matrix operations on MapReduce.

In this paper, we present a new form of factor matrix update functions. This new form operates on blocks of matrices. In order to support the new form, we partition the factor matrices into blocks along the short dimension to maximize the parallelism and split the original matrix into corresponding blocks. The new form allows us to update distinct blocks independently and simultaneously when updating a factor matrix. It also facilitates a distributed implementation. Different blocks of one factor matrix can be updated in parallel, and can be distributed in memories of all machines of a cluster and thus avoid overflowing the memory of one single machine. Storing factor matrices in memory can support random access and local aggregation. As a result, the new form of update functions leads to an efficient MapReduce implementation. We illustrate that the new form works for NMFs with a wide class of loss functions.

Moreover, under the new form of update functions, each time we can just update a subset of its blocks instead of all the blocks when we update a factor matrix. The number of blocks in the subset is adjustable, and the only requirement is that when one factor matrix is updated the other one is fixed. For instance, we can update one block of a factor matrix and then immediately update all the blocks of the other factor matrix. We refer to this kind of updates as *frequent block-wise updates*. Frequent block-wise updates aim to utilize the most recently updated data whenever possible. As a result, frequent block-wise updates are more efficient than their traditional concurrent counterparts, *concurrent block-wise updates*, which update all the blocks of either

- *J. Yin is with Palo Alto Networks, Santa Clara, CA 95054. E-mail: jyin@paloaltonetworks.com.*
- *L. Gao is with the Department of Electrical and Computer Engineering, University of Massachusetts Amherst, Amherst, MA 01003. E-mail: lgao@ecs.umass.edu.*
- *Z. Zhang is with the Department of Computer Science, Watson School, SUNY Binghamton, Binghamton, NY 13903. E-mail: zhongfei@cs.binghamton.edu.*

Manuscript received 8 Jan. 2017; revised 3 Aug. 2017; accepted 11 Dec. 2017. Date of publication 0 . 0000; date of current version 0 . 0000. (Corresponding author: Jiangtao Yin.) Recommended for acceptance by J. Ye. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TKDE.2017.2785326

factor matrix alternately. Furthermore, we can apply even more flexible update scheduling for frequent block-wise updates. Instead of updating blocks in a cyclic manner, we can select a number of important blocks to update in each iteration for further improving performance. This approach is referred to as *lazy block-wise updates*. Additionally, we prove that both frequent block-wise updates and lazy block-wise updates maintain the convergence property.

We present implementations of block-wise updates for two classical NMFs: one uses the square of Euclidean distance as the loss function, and the other uses the generalized KL-divergence. We implement concurrent block-wise updates on MapReduce, and implement concurrent, frequent, and lazy block-wise updates on an extended version of MapReduce, iMapReduce [17], which supports iterative computations more efficiently. We evaluate these implementations on a local cluster as well as the Amazon EC2 cloud. With both synthetic and real-world datasets, the evaluation results show that our MapReduce implementation for concurrent block-wise updates is 19x - 107x faster than the existing MapReduce implementation [16] with the traditional form of update functions, and our iMapReduce implementation further achieves up to 3x speedup over our MapReduce implementation. Furthermore, the iMapReduce implementations with frequent block-wise and lazy block-wise updates can be much faster than that with concurrent block-wise updates. Accordingly, our iMapReduce implementation with frequent block-wise updates (or lazy block-wise updates) is up to two orders of magnitude faster than the existing MapReduce implementation.

The rest of this paper is organized as follows. Section 2 briefly reviews the background of NMF. Section 3 introduces block-wise updates. Concurrent block-wise updates, frequent block-wise updates, and lazy block-wise updates, are presented in Section 4. Section 5 provides our efficient implementations of distributed block-wise updates. Section 6 presents the evaluation results. Section 7 surveys related work, and this paper is concluded in Section 8.

2 Background

NMF aims to factorize an original matrix A into two nonnegative low-rank factor matrices W and H. Matrix A's elements must be nonnegative by assumption. The achieved factorization has the property of $A \simeq WH$. A loss function is leveraged to measure the discrepancy between A and WH. More formally:

Given $A \in \mathbb{R}_+^{m \times n}$ and a positive integer $k \ll \min\{m, n\}$, find $W \in \mathbb{R}_+^{m \times k}$ and $H \in \mathbb{R}_+^{k \times n}$, such that a loss function $L(A, W, H)$ is minimized.

The loss function $L(A, W, H)$ is typically not convex in both W and H together. Hence, it is unrealistic to have an approach of finding the global minimum. Fortunately, there are many techniques for finding local minima.

A general approach is to adopt the block coordinate descent rules [7]:

- Initialize W, H with nonnegative W^0, H^0, $t \leftarrow 0$.
- Repeat until a convergence criterion is satisfied:
  Find H^{t+1}: $L(A, W^t, H^{t+1}) \leq L(A, W^t, H^t)$;
  Find W^{t+1}: $L(A, W^{t+1}, H^{t+1}) \leq L(A, W^t, H^{t+1})$.

TABLE 1
Decomposable Loss Functions

Square of Euclidean distance	$\sum_{(i,j)}(A_{ij} - [WH]_{ij})^2$
KL-divergence	$\sum_{(i,j)} A_{ij} \log \frac{A_{ij}}{[WH]_{ij}}$
Generalized KL-divergence (I-divergence)	$\sum_{(i,j)}(A_{ij} \log \frac{A_{ij}}{[WH]_{ij}} - A_{ij} + [WH]_{ij})$
Itakura-Saito distance	$\sum_{(i,j)}(\frac{A_{ij}}{[WH]_{ij}} - \log \frac{A_{ij}}{[WH]_{ij}} - 1)$

When the matrix loss function is the square of the Euclidean distance, i.e.,

$$L(A, W, H) = ||A - WH||_F^2, \tag{1}$$

where $||\cdot||_F$ is the Frobenius norm, one of the most well-known algorithms for implementing the above rules is Lee and Seung's multiplicative update approach [6]. It updates W and H as follows

$$H = H * \frac{W^T A}{W^T W H}, \quad W = W * \frac{A H^T}{W H H^T}, \tag{2}$$

where the symbol "$*$" and the symbol "-" (or equivalently "/") are used to denote the element-wise matrix multiplication and division, respectively.

3 Distributed NMF

In this section, we present how to apply the block coordinate descent rules to NMF in a distributed environment.

3.1 Decomposition

A loss function is usually decomposable [18]. That is, it can be represented as the sum of losses for all the elements in the matrix. For example, the well adopted loss function, the square of the Euclidean distance, is decomposable. We list several popular decomposable loss functions in Table 1. To achieve better sparsity in W and H, regularization terms have been proposed to add into loss functions [19]. For example, the square of the Euclidean distance with an L1-norm regularization on W and H can achieve a more sparse solution

$$L(A, W, H) = ||A - WH||_F^2 + \alpha \sum_{(i,r)} W_{ir} + \beta \sum_{(r,j)} H_{rj}, \tag{3}$$

where $\alpha > 0$ and $\beta > 0$ are regularization parameters which trade off the original loss function with the regularizer. Another common loss function with the regularization term [5] is as follows:

$$L(A, W, H) = ||A - WH||_F^2 + \lambda(||W||_F^2 + ||H||_F^2), \tag{4}$$

where λ is the regularization parameter. One can also replace $(||W||_F^2 + ||H||_F^2)$ with $\sum_{i,j}(||W_i||^2 + ||H_j||^2)$ (where $||\cdot||$ denotes the L2-norm) to obtain another loss function. The regularization term itself is usually decomposable as well. Therefore, the final loss function is decomposable. We focus on NMF with decomposable loss functions.

Distributed NMF needs to partition the matrices W, H, and A across compute nodes. To this end, we leverage a well-adopted scheme in gradient descent algorithms [20], [21], which partitions W and H into blocks along the short

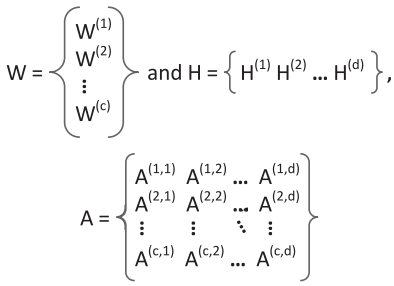


Fig. 1. Block-wise partition scheme for distributed NMF.

dimension to maximize the parallelism and splits the original matrix A into corresponding blocks. We use symbol $W^{(I)}$ to denote the Ith block of W, $H^{(J)}$ to denote the Jth block of H, and $A^{(I,J)}$ to denote the corresponding block of A (i.e., the (I,J)th block). Under this partition scheme, $A^{(I,J)}$ is only related to $W^{(I)}$ and $H^{(J)}$ when computing the loss function and is independent of other blocks of W and H, in terms of loss value (computed by the loss function). We refer to the partition scheme as *block-wise partition*. The view of the block-wise partition scheme is shown in Fig. 1. Previous work on distributed NMF [16] also proposes to partition W and H along the short dimension. The key difference between this partition scheme and the block-wise partition scheme is that the former splits W and H into row and column vectors, respectively, while the latter splits W and H into blocks. Since one block of W and one block of H can contain a set of row and column vectors, respectively, the block-wise partition scheme can be considered as a more general scheme. Moreover, the block-wise partition scheme splits A into blocks as well.

Due to its decomposability, loss function $L(A,W,H)$ can be expressed as

$$L(A,W,H)=\sum_I\sum_J L(A^{(I,J)},W^{(I)},H^{(J)}). \tag{5}$$

Let

$$F_I=\sum_J L(A^{(I,J)},W^{(I)},H^{(J)}), \tag{6}$$

$$G_J=\sum_I L(A^{(I,J)},W^{(I)},H^{(J)}), \tag{7}$$

then we have

$$L(A,W,H)=\sum_I F_I=\sum_J G_J. \tag{8}$$

F_I and G_J can be seen as local loss functions. The overall loss function L is a sum of local loss functions. By fixing H, F_I is independent of each other. Therefore, F_I can be minimized independently and simultaneously by fixing H. Similarly, G_J can be minimized independently and simultaneously by fixing W.

3.2 Block-Wise Updates

The block-wise partition allows us to update its blocks independently when updating a factor matrix (by fixing the other factor matrix). In other words, each block can be treated as one update unit. We refer to this kind of updates as *block-wise updates*. In the following, we illustrate how to update one block of W (by minimizing F_I) and that of H (by minimizing G_J). We take the square of the Euclidean distance and the generalized KL-divergence as examples. Nevertheless, the techniques derived in this section can be applied to any other decomposable loss function.

3.2.1 Square of Euclidean Distance

Here we first show how to update one block of H (i.e., $H^{(J)}$) when the square of the Euclidean distance is leveraged as the loss function. We refer to this type of NMF as *SED-NMF*. When W is fixed, minimizing G_J can be expressed as follows:

$$\min_{H^{(J)}} G_J=\min_{H^{(J)}}\sum_I ||A^{(I,J)}-W^{(I)}H^{(J)}||_F^2. \tag{9}$$

We here leverage gradient descent to update $H^{(J)}$

$$H_{uv}^{(J)}=H_{uv}^{(J)}-\eta_{uv}\frac{\partial G_J}{\partial H_{uv}^{(J)}}, \tag{10}$$

where $H_{uv}^{(J)}$ denotes the element at the uth row and the vth column of $H^{(J)}$, and η_{uv} is an individual step size for the corresponding gradient element, and

$$\frac{\partial G_J}{\partial H_{uv}^{(J)}}=\left[\sum_I((W^{(I)})^TW^{(I)}H^{(J)}-(W^{(I)})^TA^{(I,J)})\right]_{uv}. \tag{11}$$

If all step sizes are set to some sufficiently small positive number, the update should reduce G_J. However, if the number is too small, the decreasing speed can be very slow. To obtain a good speed, we derive step sizes by following Lee and Seung's approach

$$\eta_{uv}=\frac{H_{uv}^{(J)}}{[\sum_I(W^{(I)})^TW^{(I)}H^{(J)}]_{uv}}. \tag{12}$$

Then, we have

$$H_{uv}^{(J)}=H_{uv}^{(J)}\frac{[\sum_I(W^{(I)})^TA^{(I,J)}]_{uv}}{[\sum_I(W^{(I)})^TW^{(I)}H^{(J)}]_{uv}}. \tag{13}$$

Similarly, we can derive the update formula for $W^{(I)}$ as follows:

$$W_{uv}^{(I)}=W_{uv}^{(I)}\frac{[\sum_J A^{(I,J)}(H^{(J)})^T]_{uv}}{[\sum_J W^{(I)}H^{(J)}(H^{(J)})^T]_{uv}}. \tag{14}$$

We have derived the update formulae with the gradient descent method. It is important to note that we can also utilize other techniques, such as the active set method [8] and the block principal pivoting method [22], to derive the update formulae. Furthermore, we can even use different methods for different blocks at the same time. For example, we can use the gradient descent method to update half of blocks of H and use the active set method for the other half.

3.2.2 Generalized KL-Divergence

Now we derive the update for one block of H when the generalized KL-divergence is used as the loss function. We refer to this type of NMF as *KLD-NMF*. When W is fixed, minimizing G_J can be expressed as follows:

$$\min_{H^{(J)}} G_J = \min_{H^{(J)}} \sum_I \sum_{i\in I, j\in J} \left(A_{ij} \log \frac{A_{ij}}{[WH]_{ij}} - A_{ij} + [WH]_{ij} \right). \quad (15)$$

We also leverage gradient descent to update $H^{(J)}$

$$H^{(J)}_{uv} = H^{(J)}_{uv} - \eta_{uv} \frac{\partial G_J}{\partial H^{(J)}_{uv}}, \quad (16)$$

where $\frac{\partial G_J}{\partial H^{(J)}_{uv}} = \sum_I \sum_{i\in I} [W_{iu} - W_{iu} \frac{A_{iv}}{[WH]_{iv}}]$. Again we derive step sizes by following Lee and Seung's approach: $\eta_{uv} = \frac{H^{(J)}_{uv}}{\sum_I \sum_{i\in I} W_{iu}}$.

Then, we have

$$H^{(J)} = H^{(J)} * \frac{\sum_I (W^{(I)})^T \frac{A^{(I,J)}}{W^{(I)}H^{(J)}}}{\sum_I (W^{(I)})^T E^{(I,J)}}, \quad (17)$$

where $E^{(I,J)}$ is a $a \times b$ matrix with all the elements being 1 (a is the number of rows in $W^{(I)}$ and b is the number of columns in $H^{(J)}$).

Similarly, we can derive the update formula for $W^{(I)}$

$$W^{(I)} = W^{(I)} * \frac{\sum_J \frac{A^{(I,J)}}{W^{(I)}H^{(J)}} (H^{(J)})^T}{\sum_J E^{(I,J)} (H^{(J)})^T}. \quad (18)$$

4 Update Approaches

Block-wise updates can handle each block of one factor matrix independently. This flexibility allows us to have different ways to update blocks. We can simultaneously update all the blocks of one factor matrix and then update all the blocks of the other factor matrix. Also, we can update a subset of blocks of one factor matrix and then update a subset of blocks of the other one, and the number of blocks in the subset is adjustable. Furthermore, we do not necessary update blocks of a factor matrix in a cyclic manner. Actually, we can select a number of blocks to update in each iteration, and the selection is based on a block's importance.

4.1 Concurrent Block-Wise Updates

With block-wise updates, one basic way of fulfilling the block coordinate descent rules is to alternatively update all the blocks of H and all the blocks of W. Since this way updates all the blocks of one factor matrix concurrently, we refer to it as *concurrent block-wise updates*.

From the matrix operation perspective, we can show concurrent block-wise updates derived in the previous section are equivalent to the multiplicative update approach. Take SED-NMF for example. We can show that updates in Eqs. (13) and (14) are equivalent to those in Eq. (2). Without loss of generality, we assume that the $H^{(J)}$ is one block of H from the J_0th column to the J_bth column. Let Y be one block of W^TWH from the J_0th column to the J_bth column, then we have $Y = \sum_I (W^{(I)})^T W^{(I)} H^{(J)}$, since $W^TW = \sum_I (W^{(I)})^T W^{(I)}$. Assuming that X is one block of W^TA from the J_0th column to the J_bth column, we can show that $X = \sum_I (W^{(I)})^T A^{(I,J)}$. Hence, for both concurrent block-wise updates and the multiplicative update approach, the formula for updating $H^{(J)}$ is equivalent to $H^{(J)} = H^{(J)} * \frac{X}{Y}$. That is, Eq. (13) is equivalent to the formula for updating H in Eq. (2). Similarly, we can show Eq. (14) is equivalent to the formula for updating W.

4.2 Frequent Block-Wise Updates

Since all the blocks of one factor matrix can be updated independently when the other matrix is fixed, another (more general) way of fulfilling block coordinate descent rules is to update a subset of blocks of H, and then update a subset of blocks of W. Since this way updates a factor matrix more frequently, we refer to it as *frequent block-wise updates*. Frequent block-wise updates aim to utilize the most recently updated data whenever possible, and thus can potentially accelerate convergence.

More formally, frequent block-wise updates start with some initial guess of W and H, and then seek to minimize the loss function by iteratively applying the following two steps:

Step I: Fix W, update a subset of blocks of H.
Step II: Fix H, update a subset of blocks of W.

In both steps, the subset's size is a parameter, and we rotate the subset on all the blocks to guarantee that each block has an equal chance to be updated. The subset's size controls the update frequency. In an extreme case, if we always set the subset to include all the blocks, frequent updates degrade to concurrent updates.

Frequent block-wise updates provide a high flexibility to update factor matrices. For simplicity, we update a subset of blocks of one factor matrix and then update all the blocks of the other one in each iteration. Here, we assume that we update a subset of blocks of W and then update all the blocks of H. Intuitively, updating H frequently might incur a large additional overhead. Fortunately, we will show (in Section 4.4) that the formula for updating H can be incrementally computed. That is, the cost of updating H grows linearly with the number of W blocks that have been updated in the last iteration.

4.3 Lazy Block-Wise Updates

We can apply even more flexible update scheduling for frequent block-wise updates. Instead of updating blocks of a factor matrix in a cyclic manner, we can select a number of important blocks to update in each iteration. Since this approach updates a block only when the block is determined to be important, we refer to it as *lazy block-wise updates*. The high level idea behind it is that not all the blocks are of equal importance for each iteration. For example, if elements of the original matrix A are not evenly distributed (i.e., some parts of A have more non-zero values while other parts have much less non-zero values), the blocks corresponding to the dense part of A may need to be updated more often. Selective updates have shown promising results in several iterative algorithms [23], [24], [25].

Similar to frequent block-wise updates, lazy block-wise updates also start with some initial guess of W and H, and then seek to minimize the loss function by iteratively applying the following two steps:

Step I: Fix W, update a number of selected blocks of H.
Step II: Fix H, update a number of selected blocks of W.

The selection of blocks to update s based on a block's importance. For example, we can use the L1-norm of the delta

block, which is the change of a block before and after update to measure its importance. That is, $||\Delta W^{(I)}||_1 = ||W^{(I)new} - W^{(I)}||_1$ is used to measure the importance of block $W^{(I)}$. For simplicity, we only perform lazy block-wise updates on W. That is, each iteration we update a number of selected blocks of W and then update all the blocks of H. The idea of lazy block-wise updates applies to H as well.

4.4 Incremental Computation

We here illustrate how to incrementally update H when a subset of blocks of W has been updated. In order to update H, we need to compute certain global statistics over all the blocks of W. This is because one block of H is related to all the blocks of W when calculating the loss function. For example, when calculating G_J (defined in Eq. (7)), a particular block of H (i.e., $H^{(J)}$) and all the blocks of W are involved. The global statistics over all the blocks of W can be expressed as a summation of local statistics over each individual block of W. If a block does not change, the corresponding local statistics does not change as well. As a result, if caching the local statistics for all the blocks, we do not need to recompute them for unchanged blocks. In this way, unnecessary operations can be avoided. Furthermore, we can also cache the global statistics. Then, we can refresh it by accumulating the old value and the changes on local statistics. Next, we introduce incremental computation for SED-NMF and KLD-NMF, respectively, through identifying global statistics and local statistics.

4.4.1 Incremental Computation for SED-NMF

For SED-NMF, in order to incrementally update H when a subset of W blocks are updated, we introduce a few auxiliary matrices. Let $X^J = \sum_I (W^{(I)})^T A^{(I,J)}$, $X_I^J = (W^{(I)})^T A^{(I,J)}$, $S = \sum_I (W^{(I)})^T W^{(I)}$, and $S_I = (W^{(I)})^T W^{(I)}$. Among them, X^J and S can be considered as global statistics, and X_I^J and S_I can be seen as local statistics. Then, $H_{uv}^{(J)}$ can be updated by

$$H_{uv}^{(J)} = H_{uv}^{(J)} \frac{X_{uv}^J}{[SH^{(J)}]_{uv}}. \tag{19}$$

We next show how to incrementally calculate X_J and S by saving their values from last iteration. When a subset of $W^{(I)}$ ($I \in C$) have been updated, the new value of X_J and S can be computed as follows:

$$X_J = X_J + \sum_{I \in C} \Big[(W^{(I)new})^T A^{(I,J)} - X_I^J \Big]; \tag{20}$$

$$S = S + \sum_{I \in C} \Big[(W^{(I)new})^T W^{(I)new} - S_I \Big]. \tag{21}$$

From Eqs. (19), (20), and (21), we can see that the cost of incrementally updating $H^{(J)}$ depends on the number of W blocks that have been updated rather than the total number of blocks that W has.

4.4.2 Incremental Computation for KLD-NMF

For KLD-NMF, we also introduce a few auxiliary matrices to incrementally update H when a subset of W blocks are updated. Let $X^J = \sum_I [(W^{(I)})^T \frac{A^{(I,J)}}{W^{(I)}H^{(J)}}]$, $X_I^J = (W^{(I)})^T \frac{A^{(I,J)}}{W^{(I)}H^{(J)}}$, $S = \sum_I [(W^{(I)})^T E^{(I,J)}]$ (S is a vector), and $S_I = (W^{(I)})^T E^{(I,J)}$. Again, X^J and S can be considered as global statistics, and X_I^J and S_I can be seen as local statistics. Then, $H_{uv}^{(J)}$ can be updated by $H_{uv}^{(J)} = H_{uv}^{(J)} \frac{X_{uv}^J}{S_u}$.

We next show how to incrementally calculate X_J and S by saving their values from last iteration. When a subset of $W^{(I)}$ ($I \in C$) have been updated, the new value of X_J and S can be computed as follows: $X_J = X_J + \sum_{I \in C} [(W^{(I)new})^T \frac{A^{(I,J)}}{W^{(I)new}H^{(J)}} - X_I^J]$; $S = S + \sum_{I \in C} [(W^{(I)new})^T E^{(I,J)} - S_I]$.

Again, we can observe that the cost of incrementally updating $H^{(J)}$ depends on the number of W blocks that have been updated rather than the total number of W blocks.

4.5 Convergence of Proposed Approaches

We here prove that both frequent block-wise updates and lazy block-wise updates maintain the convergence property. We use SED-NMF as an instance. The proof for KLD-NMF can be derived similarly. For SED-NMF, we first prove that G_J and F_I are nonincreasing under formulae Eqs. (13) and (14), respectively (as stated in the following two thorems). We then show the overall loss function L is nonincreasing when either frequent block-wise updates or lazy block-wise updates are applied.

Theorem 4.1. *G_J is nonincreasing under formula Eq. (13). G_J is constant if and only if $H^{(J)}$ is at a stationary point of G_J.*

Theorem 4.2. *F_I is nonincreasing under formula Eq. (14). F_I is constant if and only if $W^{(I)}$ is at a stationary point of F_I.*

We leverage the concept of auxiliary functions [6] these two theorems.

Definition 1. *$U(s, s^t)$ is an auxiliary function for $F(s)$, if $U(s, s^t) \geq F(s)$ and $U(s, s) = F(s)$.*

The auxiliary function turns out to be useful because of the following lemma.

Lemma 4.3. *If U is an auxiliary function, then F is non-increasing under the updating rule $s^{t+1} = \arg\min_s U(s, s^t)$.*

Proof.

$$F(s^{t+1}) \leq U(s^{t+1}, s^t) \leq U(s^t, s^t) = F(s). \tag{22}$$

□

We can see that the sequence formed by the iterative application of Lemma 4.3 leads to a monotonic decrease in the objective function value $F(s)$. Therefore, for an algorithm that iteratively updates s in order to minimize $F(s)$, we can prove its convergence by constructing an appropriate auxiliary function.

We now construct auxiliary functions for $\sum_I ||A^{(I,J)} - W^{(I)}H^{(J)}||_F^2$.

Lemma 4.4. *If $V(h^t)$ is the diagonal matrix*

$$V_a(h^t) = \frac{\delta_{ab}[\sum_I (W^{(I)})^T W^{(I)} h^t]_a}{h_a^t}, \tag{23}$$

then

$$\begin{aligned} U(h, h^t) = F(h^t) + (h - h^t)\nabla F(h^t) \\ + (h - h^t)^T V(h^t)(h - h^t) \end{aligned} \tag{24}$$

is an auxiliary function for

$$F(h) = \sum_I ||A_i^{(I,J)} - W^{(I)}h)||^2, \tag{25}$$

where h can be seen as one column of $H^{(J)}$.

Proof. $U(h,h) = F(h)$ is obvious. Next, we show $U(h,h^t) \geq F(h)$. We have

$$\begin{aligned} F(h) &= F(h + h^t - h^t) \\ &= F(h^t) + (h - h^t)\nabla F(h^t) \\ &\quad + (h - h^t)^T \sum_I [(W^{(I)})^T W^{(I)}](h - h^t). \end{aligned} \tag{26}$$

Then,

$$\begin{aligned} U(h,h^t) - F(h) = (h - h^t)^T \Big\{ & V_a(h^t) \\ & - \sum_I [(W^{(I)})^T W^{(I)}] \Big\} (h - h^t). \end{aligned} \tag{27}$$

The proof of $U(h,h^t) \geq F(h)$ can be completed by showing $V_a(h^t) - \sum_I [(W^{(I)})^T W^{(I)}]$ is a positive semidefinite matrix. □

For the auxiliary function in Eq. (24), we have

$$h^{t+1} = \underset{h}{\operatorname{argmin}}\, U(h,h^t) = h^t - V(h^t)^{-1}\nabla F(h^t). \tag{28}$$

According to Lemmas 4.3 and 4.4, this update will not increase $F(h)$, and $G_J = \sum_i F(h)$. The update can be rewritten as

$$h_a^{t+1} = h_a^t \frac{[\sum_I (W^{(I)})^T A^{(I,J)}]_a}{[\sum_I (W^{(I)})^T W^{(I)} h^t]_a}, \tag{29}$$

which is another equivalent form of Eq. (13). Therefore, we complete the proof of Theorem 4.1. Similarly, we can finish the proof of Theorem 4.2.

Given Theorems 4.1 and 4.2, we have the following theorem.

Theorem 4.5. *L is nonincreasing when either frequent block-wise updates or lazy block-wise updates are applied. L is constant if and only if W and H are at a stationary point of L.*

Proof. As illustrated in Eq. (8), the overall loss function L is the sum of local loss functions, G_J or F_I. G_J is nonincreasing when $H^{(J)}$ is updated for any J. F_I is nonincreasing as well when $W^{(I)}$ is updated for any I. Therefore, either frequent block-wise updates or lazy block-wise updates will not increase L when W (or H) is updated, no matter how many blocks of W (or H) are selected for updating in each iteration. Additionally, if and only if all the blocks of W (or H) are at a stationary point of L, L does not decrease. □

5 Distributed Implementations

Block-wise updates also facilitate a distributed implementation. Different blocks of one factor matrix can be updated in parallel, and can be distributed in memories of all the machines and thus avoid overflowing the memory of one single machine. Storing factor matrices in memory supports random access and local aggregation, which are highly useful when updating them. MapReduce [13] and its extensions (e.g, [17]) have emerged as distributed frameworks for data intensive computation. In this section, we illustrate the efficient implementation of concurrent block-wise updates on MapReduce. Also, we show how to implement frequent block-wise updates and lazy block-wise updates on an extended version of MapReduce, iMapReduce [17], which supports iterative computations more efficiently. To ground our discussion, we begin with an overview of the state-of-the-art work that implements the traditional form of update functions on MapReduce.

5.1 Traditional Updates on MapReduce

The previous effort by Liu et al. [16] is a piece of state-of-the-art work of implementing the traditional form of update functions on MapReduce. For performing matrix multiplication (with two large matrices), it needs to join a row (or column) of one matrix with each column (or row) of the other one with two MapReduce jobs. As a result, a huge amount of intermediate data have to be generated and shuffled. The intermediate data explosion is a huge issue in terms of performance.

In order to elaborate the intermediate data explosion issue of implementing the traditional form of update functions, we take SED-NMF as an instance. To implement the update for H (as shown in Eq. (2)) on MapReduce, the previous work [16] needs five jobs: two jobs for computing W^TA, two jobs for computing W^TWH, and one job for the final update. Among them, the two jobs for computing W^TA are the bottleneck. The first job generates the intermediate data $< j, A_{ij}W_{i\cdot}^T >$ for any i and $j \in \mathbb{O}^i$, where $\mathbb{O}^i$ denotes the set of nonzero elements on the ith row of A. The second job takes the intermediate data as its input. The intermediate data take $O(\rho mnk)$ space (where ρ is the sparsity of A), which can be huge considering m and n are at the order of hundreds of thousands or even millions. Consequently, dumping and loading the intermediate data dominate the time of updating H. Similar conclusion can be reached for updating W.

5.2 Concurrent Block-Wise Updates on MapReduce

Block-wise updates enable efficient distributed implementations. With block-wise updates, the basic computation units in the update functions (e.g., Eqs. (13) and (14)) are blocks of factor matrices and blocks of the original matrix. The size of a block is adjustable. As a result, when performing an essential matrix operation, which involves two blocks of matrices (e.g., $(W^{(I)})^T$ and $A^{(I,J)}$), we can assume that at least the smaller block can be held in the memory of a single worker. Since W and H are low-rank factor matrices, they usually are much smaller than A, and thus the assumption that one of their blocks can be held in the memory of one single worker is reasonable. The result matrix of an essential matrix operation (e.g., $(W^{(I)})^T A^{(I,J)}$) is usually relatively small and can be held in the memory of one single worker as well. Storing a matrix (or a block of a matrix) in memory efficiently supports random and repeated access, which is commonly needed in a matrix operation such as multiplication. Maintaining the result matrix in memory supports local aggregation. Therefore, one worker can complete an

essential matrix operation locally and efficiently. Note that the other (larger) matrix (e.g., one block of A) is still in disk so as to scale to large NMF problems.

Accordingly, the MapReduce programming model fits block-wise updates well. An essential matrix operation with two blocks can be realized in one mapper, and the aggregation of the results of essential matrix operations can be realized in reducers. To realize matrix multiplication with two blocks of matrices in one mapper, we exploit the fact that a mapper can cache data in memory before processing input key-value pairs and that a mapper can maintain state across the processing of multiple input key-value pairs and defer emission of intermediate key-value pairs until all the input pairs have been processed. We next illustrate efficient implementations of concurrent block-wise updates for SED-NMF and KLD-NMF, respectively.

5.2.1 MapReduce Implementation for SED-NMF

Inspired by the previous work [16], which decomposes the update formula of SED-NMF for H into three components, we consider the update formula for $H^{(J)}$ (Eq. (13)) into three parts as well: $X^{(J)} = \sum_I (W^{(I)})^T A^{(I,J)}$, $Y^{(J)} = \sum_I (W^{(I)})^T W^{(I)} H^{(J)}$, and $H^{(J)} = H^{(J)} * \frac{X^{(J)}}{Y^{(J)}}$. However, we have much more efficient implementation for each part than the previous work, as demonstrated in the following.

We have one job to compute $X^{(J)} = \sum_I X^{(I,J)} = \sum_I (W^{(I)})^T A^{(I,J)}$. Let $X^{(I,J)}_{:j}$ represent the jth column of $X^{(I,J)}$, then

$$X^{(I,J)}_{:j} = \sum_{i=1}^{a} A^{(I,J)}_{i,j} (W^{(I)}_{i:})^T, \tag{30}$$

where a is the number of rows of $A^{(I,J)}$, and $W^{(I)}_{i:}$ is the ith row of $W^{(I)}$. When holding $W^{(I)}$ in memory, a mapper can leverage Eq. (30) to compute $X^{(I,J)}$ via continuously reading elements of $A^{(I,J)}$. $X^{(I,J)}$ (which is usually small) stays in memory for local aggregation. After computing $X^{(I,J)}$ in the mapper, the aggregation $X^{(J)} = \sum_I X^{(I,J)}$ can be computed in a reducer. Different reducers compute $X^{(J)}$ for different J.

Two jobs are used to compute $Y^{(J)} = \sum_I (W^{(I)})^T W^{(I)} H^{(J)}$. We first compute $S = \sum_I (W^{(I)})^T W^{(I)}$ and then calculate $Y^{(J)} = SH^{(J)}$. $(W^{(I)})^T W^{(I)}$ (a $k \times k$ matrix) can be performed in one mapper as follows:

$$(W^{(I)})^T W^{(I)} = \sum_{i=1}^{a} (W^{(I)}_{i:})^T W^{(I)}_{i:}. \tag{31}$$

Then, all mappers send $(W^{(I)})^T W^{(I)}$ to one particular reducer for a global summation. After computing $S = \sum_I (W^{(I)})^T W^{(I)}$, calculating $Y^{(J)} = SH^{(J)}$ can be done in a job with the map phase only, by $Y^{(J)}_{:j} = SH^{(J)}_{:j}$.

Last, we have one job (with the map phase only) to compute $H^{(J)} \leftarrow H^{(J)} * \frac{X^{(J)}}{Y^{(J)}}$. In summary, the MapReduce operations for updating H are as follows.

- Job-I Map: Load $W^{(I)}$ in memory, calculate $X^{(I,J)}$ using Eq. (30) (take $A^{(I,J)}$ as input), and emit $< I, X^{(I,J)} >$.
- Job-I Reduce: Take $< I, X^{(I,J)} >$, and emit $< J, X^{(J)} >$.
- Job-II Map: Load $W^{(I)}$ in memory, calculate $(W^{(I)})^T W^{(I)}$ using Eq. (31), and emit $< I, (W^{(I)})^T W^{(I)} >$.
- Job-II Reduce: Take $< I, (W^{(I)})^T W^{(I)} >$, and emit $< 0, S >$.
- Job-III Map: Load S in memory. Emit tuples $< j, Y^{(J)}_{:j} >$.
- Job-IV Map: Read $< j, H^{(J)}_{:j} >$, $< j, X^{(J)}_{:j} >$, and $< j, Y^{(J)}_{:j} >$. Emit tuples in the form of $< j, H^{(J)new}_{:j} >$, where $H^{(J)new}_{:j} = H^{(J)}_{:j} * \frac{X^{(J)}_{:j}}{Y^{(J)}_{:j}}$.

In the previous implementation, we try to minimize data shuffling by utilizing local aggregation. However, in each iteration it still needs four MapReduce jobs to update H. In addition, intermediate data (e.g., $X^{(J)}$) need to be dumped into disk and be reloaded in later jobs. We next illustrate how to minimize the number of jobs and the amount of intermediate data to be reloaded.

Job-II can be kept (as Job-1), since it only produces a small ($k \times k$) matrix and reloading its output does not take much time. Job-I, Job-III, and Job-IV can be integrated into one job so as to avoid dumping and reloading $X^{(J)}$ and $Y^{(J)}$. The integrated job has the same map phase with Job-I. In the reduce phase, besides computing $X^{(J)}_{:j}$, it also computes $Y^{(J)}_{:j}$ and finally calculates $H^{(J)new}_{:j} = H^{(J)}_{:j} * [X^{(J)}_{:j} / Y^{(J)}_{:j}]$. The overview of our optimized implementation is presented in Fig. 2, and the MapReduce operations in the integrated job (Job-2) are described as follows.

- Job-2 Map: Load $W^{(I)}$ in memory, calculate $X^{(I,J)}$ using Eq. (30) (take $A^{(I,J)}$ as input), and emit $< I, X^{(I,J)} >$.
- Job-2 Reduce: Take $< I, X^{(I,J)} >$, and first calculate $X^{(J)}_{:j}$. Load S in memory. Then, read $H^{(J)}_{:j}$ and compute $Y^{(J)}_{:j}$. Last, calculate $H^{(J)new}_{:j}$.

In the above, we describe the MapReduce operations used to complete the update of H for one iteration. Updating W can be performed in the same fashion. We next provide a sketch of its design and omit the description of the operations.

The formula for updating W (Eq. (14)) can also be treated as three parts: $U^{(I)} = \sum_J A^{(I,J)} (H^{(J)})^T$, $V^{(I)} = \sum_J W^{(I)} H^{(J)} (H^{(J)})^T$, and $W^{(I)} = W^{(I)} * \frac{U^{(I)}}{V^{(I)}}$. Let $U^{(I,J)} = A^{(I,J)} (H^{(J)})^T$, then

$$U^{(I,J)}_{i:} = \sum_{j=1}^{a} A^{(I,J)}_{i,j} \left(H^{(J)}_{:j}\right)^T. \tag{32}$$

To efficiently compute $V^{(I)}$, we compute $H^{(J)}(H^{(J)})^T$ first in the following way

$$H^{(J)}(H^{(J)})^T = \sum_{j=1}^{b} H^{(J)}_{:j} \left(H^{(J)}_{:j}\right)^T. \tag{33}$$

5.2.2 MapReduce Implementation for KLD-NMF

For KLD-NMF, we also decompose the update formula for $H^{(J)}$ (Eq. (17)) into three parts: $X^{(J)} = \sum_I [(W^{(I)})^T \frac{A^{(I,J)}}{W^{(I)} H^{(J)}}]$, $Y^{(J)} = \sum_I [(W^{(I)})^T E^{(I,J)}]$, and $H^{(J)} = H^{(J)} * \frac{X^{(J)}}{Y^{(J)}}$.

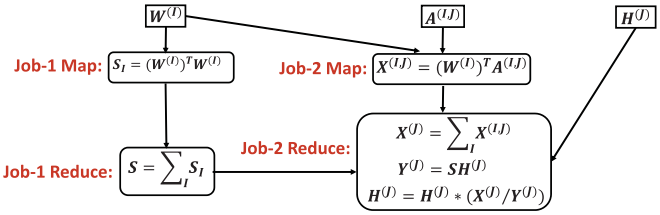


Fig. 2. Overview of the optimized implementation for updating $H^{(J)}$ of SED-NMF on MapReduce.

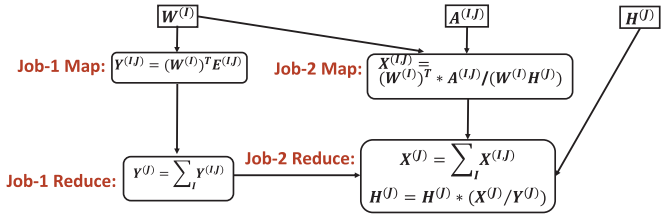


Fig. 3. Overview of the implementation for updating $H^{(J)}$ of KLD-NMF on MapReduce.

We use one job to compute $X^{(J)} = \sum_I [(W^{(I)})^T \frac{A^{(I,J)}}{W^{(I)}H^{(J)}}]$. Let $X^{(I,J)}_{\cdot j}$ represent the jth column of $X^{(I,J)}$, then

$$X^{(I,J)}_{\cdot j} = \sum_{i=1}^{a} (W^{(I)}_{i\cdot})^T \frac{A^{(I,J)}_{i,j}}{W^{(I)}_{i\cdot} H^{(J)}_{\cdot j}}. \quad (34)$$

When holding $W^{(I)}$ and $H^{(J)}$ in memory, a mapper can leverage Eq. (34) to compute $X^{(I,J)}$ via continuously reading elements of $A^{(I,J)}$. $X^{(I,J)}$ stays in memory for local aggregation. After computing $X^{(I,J)}$ in the mapper, the aggregation $X^{(J)} = \sum_I X^{(I,J)}$ can be computed in a reducer.

One job is used to compute $Y^{(J)} = \sum_I [(W^{(I)})^T E^{(I,J)}]$. Let $Y^{(I,J)} = (W^{(I)})^T E^{(I,J)}$. Computing $Y^{(I,J)}$ seems time-consuming because it multiplies two dense matrices. But since all elements of $E^{(I,J)}$ is 1, all the columns of $Y^{(I,J)}$ are the same. Therefore, we actually only need to calculate one column of $Y^{(I,J)}$. For example,

$$Y^{(I,J)}_{\cdot j} = \sum_{i=1}^{a} (W^{(I)}_{i\cdot})^T. \quad (35)$$

After computing $Y^{(I,J)}$ in the mapper, the aggregation $Y^{(J)} = \sum_I Y^{(I,J)}$ can be computed in a reducer.

The formula for updating W (Eq. (18)) can also be treated as three parts: $U^{(I)} = \sum_J \frac{A^{(I,J)}}{W^{(I)}H^{(J)}}(H^{(J)})^T$, $V^{(I)} = \sum_J E^{(I,J)}(H^{(J)})^T$, and $W^{(I)} = W^{(I)} * \frac{U^{(I)}}{V^{(I)}}$. Let $U^{(I,J)} = \frac{A^{(I,J)}}{W^{(I)}H^{(J)}}(H^{(J)})^T$, then

$$U^{(I,J)}_{i\cdot} = \sum_{j=1}^{b} \frac{A^{(I,J)}_{i,j}}{W^{(I)}_{i\cdot} H^{(J)}_{\cdot j}} (H^{(J)}_{\cdot j})^T. \quad (36)$$

Let $V^{(I,J)} = E^{(I,J)}(H^{(J)})^T$, and it can be calculated in the following way

$$V^{(I,J)}_{i\cdot} = \sum_{j=1}^{b} (H^{(J)}_{\cdot j})^T. \quad (37)$$

Then $V^{(I)}$ can be computed through $V^{(I)} = \sum_J V^{(I,J)}$.

After decomposing the update formulae for KLD-NMF, the MapReduce operations can be easily derived by following the way of achieving operations for SED-NMF. The overview of these operations is shown in Fig. 3, while the details are omitted.

5.2.3 Analysis

The intermediate data and the memory usage of implementing concurrent block-wise updates on MapReduce are analyzed here. Assume that W has c blocks and H has d blocks. Take SED-NMF as an example. Similar conclusions can be obtained for KLD-NMF. We first analyze the intermediate data. For updating H, the main intermediate data it generates are $X^{(I,J)}$ (for any I and J, cd copies in total), which take $O(k\frac{n}{d})$ space. Therefore, the main intermediate data take $O(knc)$ space in total. Similarly, we can show that the main intermediate data of updating W take $O(kmd)$ space in total. We can control the values of c and d and typically have $c \ll m$ and $d \ll n$. Therefore, the implementation of concurrent updates doest not suffer from the intermediate data explosion issue, and is much more efficient than the implementation of the traditional form of updates.

We then analyze the memory usage. For updating H, the main memory usage happens in the map phase. A mapper at most needs to cache $W^{(I)}$ and $X^{(I,J)}$ in memory, which take $O(k\frac{m}{c} + k\frac{n}{d})$ space. Similarly, we can show that for updating W a mapper at most needs $O(k\frac{m}{c} + k\frac{n}{d})$ memory space. We know k is typically small (since NMF is a low-rank approximation). Therefore, for m and n even at the order of millions, a commodity server does not have the memory overflow problem.

5.3 Frequent Block-Wise Updates on iMapReduce

Although frequent updates have potential to speed up NMF, parallelizing frequent updates in a distributed environment is challenging. Computations such as global summations need to be done in a centralized way. When processing the distributed blocks of factor matrices, the system has to synchronize the global summations frequently. Synchronizing the global resources in a distributed environment may result in considerable overhead, especially on MapReduce. MapReduce starts a new job for each computation errand. Each job needs to be initialized and load its input data, even when the data are from a previous job. Frequent updates bring more jobs. As a result, the initialization overhead and the cost of repeatedly loading data may vanish the benefit of frequent updates.

We here propose implementations for frequent block-wise updates and lazy block-wise updates on iMapReduce [17]. iMapReduce uses persistent mappers and reducers to avoid job initialization overhead. Each mapper is paired with one reducer. One pair of mapper and reducer can be seen as one logical worker. Data shuffling between mappers and reducers is the same with that of MapReduce. In addition, a reducer of iMapReduce can redirect its output to its paired mapper. Since mappers and reducers are persistent, data can be maintained in memory across different iterations, and thus can avoid repeatedly loading data. As a result, iMapReduce decreases the overhead of frequent block-wise updates.

We implement frequent block-wise updates on iMapReduce in the following way. H is evenly split into r blocks, and W is evenly partitioned into $p * r$ blocks, where r is the number of workers and p is a parameter used to control update frequency. Each worker handles p blocks of W and one block of H. In each iteration a worker updates its H block and one selected W block. That is, there are r blocks of W in total to be updated in each iteration. Each worker rotates the selected W block on all its W blocks. The setting of p plays an important role on frequent block-wise updates. Setting p too large may incur considerable overhead for synchronization. Setting it too small may degrade the effect of the frequent updates. In an extreme case, we can set $p = 1$, then frequent block-wise updates degrade to concurrent block-wise updates. We can derive the optimal p theoretically. Let $F(p)$ represent the total times of the row of W being updated in one worker to reach the convergence point. Each W block (in one worker) has $m/(p * r)$ rows. Then, since one W block is updated, one H block is updated, and thus $F(p) * p * r/m$ is the total times of the H block being updated in one worker. We use T_1 to denote the time of updating one row of W and T_2 to denote the time of updating the H block. Then, finding the optimal p becomes the problem of finding the p that minimize $\{F(p) * T_1 + F(p) * p * r * T_2/m\}$. We evaluate $F(p)$ through experiments and find it roughly has such a relationship with p that $F(p) = a/p + b$. Then, we can derive the optimal p (i.e., p^*): $p^* = \sqrt{a * m * T_1/(b * r * T_2)}$. Therefore, one way to set p is to measure a, b, T_1, and T_2 (m and r are known). In Section 6.4, we will also provide a practical way of setting p.

The only difference between the implementation of frequent block-wise updates and that of lazy block-wise updates is the way of selecting the W block for updating in each iteration. The former selects a block in a round-robin way, while the latter makes selection based on the a block's importance (e.g., L1-norm of the change). Therefore, the implementation of lazy block-wise updates is omitted due to space limitations.

5.3.1 iMapReduce Implementation for KLD-NMF

We here show how to implement frequent updates for KLD-NMF on iMapReduce. Map-1x represents different stages of a mapper, and Reduce-1x represents different stages of a reducer.

- Map-1a: Load a subset (i.e., p) of W blocks (e.g., $(W^{(B)new})$) in memory (first iteration only) or receive one updated W block from last iteration. For all loaded or received blocks, compute S_l via $S_l = (W^{(B)new})^T E^{(B,J)}$ (first iteration) or $S_l = S_l + ((W^{(B)new})^T E^{(B,J)} - (W^{(B)})^T E^{(B,J)})$, and replace $W^{(B)}$ with $W^{(B)new}$. Broadcast $< d, S_l >$ to all reducers, where d is the corresponding reducer ID.
- Reduce-1a: Take $< d, S_l >$, compute $S = \sum_l S_l$, and store S in memory.
- Map-1b: For each loaded or received W block in the previous phase (e.g., $(W^{(B)new})$), read $A^{(B,J)}$ and $H^{(J)}$, and then emit tuples in the form of $< B, X^{(B,J)} >$ where $X^{(B,J)}$ is calculated using Eq. (34) (first iteration) or in the form of $< B, \Delta X^{(B,J)} >$ where $\Delta X^{(B,J)} = (W^{(B)new})^T \frac{A^{(B,J)}}{W^{(B)new} H^{(J)}} - X^{(B,J)}$.
- Reduce-1b: Take $< B, X^{(B,J)} >$ and calculate $X^{(J)} = \sum_B X^{(B,J)}$ (first iteration) or take $< B, \Delta X^{(B,J)} >$ and calculate $X^{(J)} = X^{(J)} + \sum_B \Delta X^{(B,J)}$. Then, calculate $H^{(J)new}$ by $(H^{(J)new} = H^{(J)} * \frac{X^{(J)}}{S})$, store it in memory, and pass one copy to Map-1c in the form of $< J, H^{(J)new} >$.
- Map-1c: Receive (just updated) $H^{(J)}$ from Reduce-1b. Broadcast $< J, E^{(I,J)} (H^{(J)})^T >$ to all reducers.
- Reduce-1c: Take $< J, E^{(I,J)} (H^{(J)})^T >$, compute $Z = \sum_J E^{(I,J)} (H^{(J)})^T$, and store Z in memory.
- Map-1d: For a W block that is selected in current iteration (e.g., $(W^{(B)})$), read $A^{(B,J)}$ and $H^{(J)}$, and then emit tuples in the form of $< J, U^{(B,J)} >$, where $U^{(B,J)}$ is calculated using Eq. (36).
- Reduce-1d: Take $< J, U^{(B,J)} >$, and calculate $U^{(B)} = \sum_J U^{(B,J)}$. Then, calculate $W^{(B)new} = W^{(B)} * \frac{U^{(B)}}{Z}$, store it in memory, and pass one copy to Map-1a.

The iMapReduce Implementation for SED-NMF is omitted due to space limitations. We can show that our implementation of frequent block-wise updates takes only $O(km + kn)$ aggregate memory of the cluster for either SED-NMF or KLD-NMF. Since k is typically small, even a small cluster of commodity servers can handle the NMF problem with m and n at the order of millions without memory overflow.

6 Evaluation

In this section, we evaluate the effectiveness and efficiency of block-wise updates on both synthetic and real-word datasets. For MapReduce, we use its open source implementation, Hadoop. Experiments are performed on both small-scale and large-scale clusters.

6.1 Experiment Setup

We build both a small-scale cluster of local machines and a large-scale cluster on the Amazon EC2 cloud. The local cluster consists of 4 machines, and each one has dual-core 2.66 GHz CPU, 4 GB of RAM, 1TB hard disk. The Amazon cluster consists of 100 medium instances.

Both synthetic and real-word datasets are used in our experiments. We use two real-world datasets. One is a document-term matrix, NYTimes, from UCI Machine Learning Repository [26]. The other one is a user-movie matrix from the Netflix prize [27]. We also generate several matrices with different choices of m (the number of rows) and n (the number of columns). The sparsity is set to 0.1 unless otherwise specified, and each element is a random integer number uniformly selected from range 1 to 5. The datasets are summarized in Table 2.

Unless otherwise specified, we use rank $k = 10$, and use $p = 8$ for frequent block-wise updates (which means each worker updates $\frac{1}{8}$ of its W blocks in each iteration).

6.2 Comparison with Existing Work

The first set of experiments focuses on demonstrating the advantage of our (optimized) implementation of concurrent block-wise updates on MapReduce. We compare it with a

TABLE 2
Dataset Summary

Dataset	# of rows	# of columns	# of nonzero elements
Netflix	480,189	17,770	$100M$
NYTimes	300,000	102,660	$70M$
Syn-m-n	m	n	$0.1 * m * n$

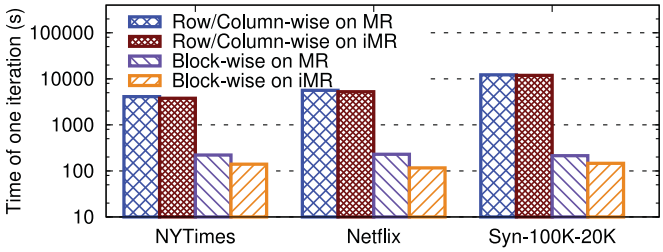


Fig. 4. Time taken in one iteration for SED-NMF on the local cluster. The y-axis is in log scale.

piece of state-of-the-art work of implementing the traditional form of update functions, which is discussed in Section 4.1. The implementation of concurrent block-wise updates on iMapReduce is added into the comparison to show iMapReduce's superiority over MapReduce. For a comprehensive comparison, the iMapReduce implementation of the traditional form is taken into consideration as well. As described in Section 4, concurrent block-wise updates are equivalent to the multiplicative update, and thus they need the same number of iterations to reach the convergence point. Therefore, we leverage the time taken in a single iteration to directly compare their performance.

Fig. 4 shows the time taken in one iteration of all the four implementations for SED-NMF on both synthetic and real-word datasets. Note that the y-axis is in log scale. Our implementation on MapReduce (denoted by "Block-wise on MR") is 19x - 57x faster than the existing approach (denoted by "Row/Column-wise on MR"). Moreover, for the block-wise updates, the implementation on iMapReduce (denoted by "Block-wise on iMR") is up to 2x faster than that on MapReduce, since iMapReduce can eliminate the job initialization overhead and the cost of repeatedly dumping/loading factor matrices (note that the original matrix still needs to be loaded from the file system at each iteration). For the traditional form of update functions, the improvement by iMapReduce is quite limited. The reasons are twofold. One is that its implementation does not store factor matrices in memory, and thus there is no benefit of eliminating the cost of repeatedly dumping/loading factor matrices. The other is that compared to the long running time of a job, the job initialization overhead is almost

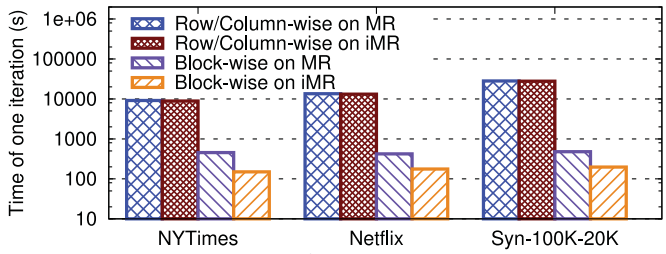


Fig. 5. Time taken in one iteration for KLD-NMF on the local cluster. The y-axis is in log scale.

ignorable, and thus eliminating the job initialization overhead does not make a huge difference.

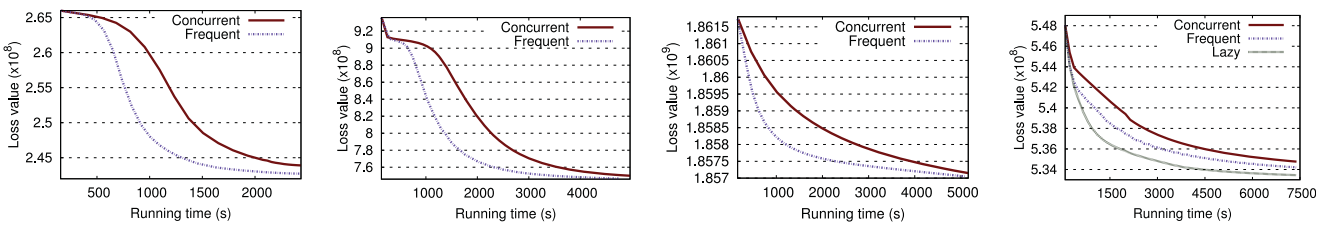
Fig. 5 shows the time taken in one iteration of all the four implementations for KLD-NMF. Similar to SED-NMF, our implementation on MapReduce is 20x - 59x faster than the existing approach. Furthermore, for the block-wise updates, the implementation on iMapReduce is up to 3x faster than that on MapReduce; while for the traditional form of update functions, the improvement by iMapReduce is ignorable.

6.3 Effect of Frequent Updates and Lazy Updates

Frequent block-wise updates leverage more up-to-date H to update W than concurrent block-wise updates, since they update H more frequently. Therefore, they have the potential to reach the convergence criterion with less workload. To evaluate their effect, we compare frequent block-wise updates with concurrent block-wise updates when both implemented on iMapReduce.

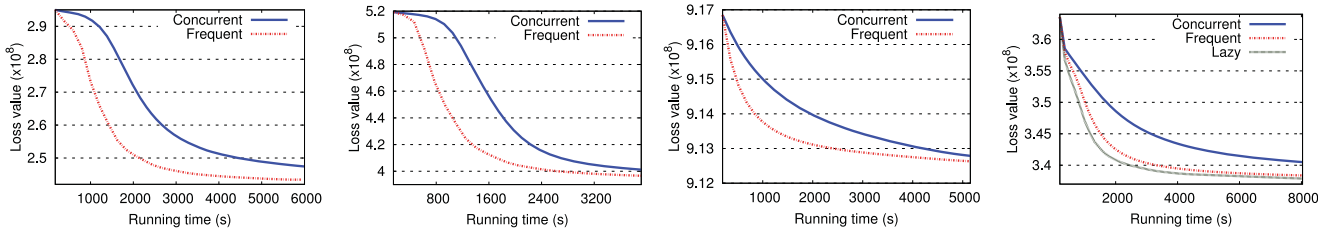
We find that lazy block-wise updates usually work well when matrix A is unbalanced. That is, some of its blocks have more nonzero element, while others have less nonzero elements. To illustrate the performance, we generate a synthetic matrix, Syn-100k-20k-Spa. During the generation, sparsity is set to 0.1 for 20 percent of the rows and set to 0.01 for the rest of rows. Lazy block-wise updates are implemented on iMapReduce as well.

All update approaches start with the same initial values when compared on the same dataset. Fig. 6 plots the performance comparison for SED-NMF. We can see that frequent block-wise updates ("Frequent") converge faster than concurrent block-wise updates ("Concurrent") on all the four datasets. In other words, if we use a predefined loss value as the convergence criterion, frequent block-wise updates would have much shorter running time. Lazy block-wise updates ("Lazy") can converge even faster than frequent block-wise updates. Similar phenomena are observed for KLD-NMF, as shown in Fig. 7. Since lazy block-wise updates usually only work well for unbalanced matrices, we focus on frequent block-wise updates in the rest.



(a) NYTimes (b) Netflix (c) Syn-100K-20K (d) Syn-100K-20K-Spa

Fig. 6. Convergence speed of SED-NMF on the local cluster.



(a) NYTimes (b) Netflix (c) Syn-100K-20K (d) Syn-100K-20K-Spa

Fig. 7. Convergence speed of KLD-NMF on the local cluster.

6.4 Tuning Update Frequency

As stated in Section 5.3, the update frequency can make a huge impact on the performance of frequent block-wise updates. The best setting of p is a square root of several metrics, which means it is not sensitive to those metrics. In experiments, we indeed find that a quite large range of p can allow frequent block-wise updates to have better performance than their concurrent counterparts, and the best setting of p stays in the range from 4 to 16. That is also why we set $p = 8$ by default. For example, Fig. 8 shows the convergence speed with different settings on dataset Netflix for SED-NMF. Another interesting finding is that if a setting is better during the first few iterations, it will continue to be better. Hence, another practical way of obtaining a good setting of p is to test several candidate settings, each for a few iterations, and then choose the best one. Similar trends are observed for KLD-NMF and are omitted here.

6.5 Different Data Sizes

We then measure how block-wise updates scale with increasing size of the original matrix A. We generate synthetic datasets of different sizes by fixing the number of ($100k$) rows and increasing the number of columns. Fig. 9 shows the time taken in one iteration of the block-wise updates and the traditional row/column-wise updates as dataset sizes vary. The time of either implementation increases as the number of columns increases, and the time of the latter increases much faster. When the number of columns is $100k$, our implementation of block-wise updates is 90x faster than the implementation of the traditional updates (compared to 57x speedup when the number of columns is $20k$).

We next compare the running time of concurrent block-wise updates with that of frequent block-wise updates. We use the loss value when concurrent block-wise updates run for 25 iterations as the convergence point. Then the time used to reach this convergence point is measured as the running time. This criterion also applies to later comparisons. As presented in Fig. 10, the running time of either type of updates increases sub-linearly with the size of the dataset. Moreover, frequent block-wise updates are up to 2.7x faster than concurrent block-wise updates. The results for KLD-NMF have similar trends and are omitted here.

6.6 Different Settings of Rank

We also measure how block-wise updates scale with different settings of the rank. Due to space limitation, we only present the results for SED-NMF. Fig. 11 shows the time taken in one iteration of the block-wise updates and the traditional row/column-wise updates on dataset Syn-100K-20K as k varies from 10 to 50. It can be seen that the time of either implementation increases as k increases, and the time

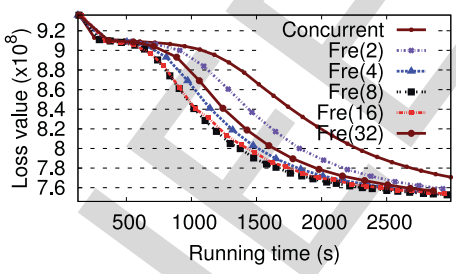


Fig. 8. Convergence speed versus update frequency. The numbers associated with "Fre" represent settings of p.

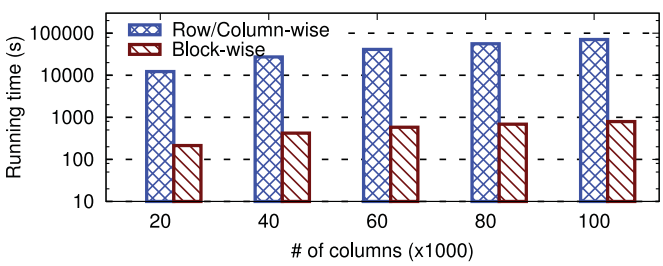


Fig. 9. Comparing Row/Column-wise updates with block-wise updates through varying dataset size.

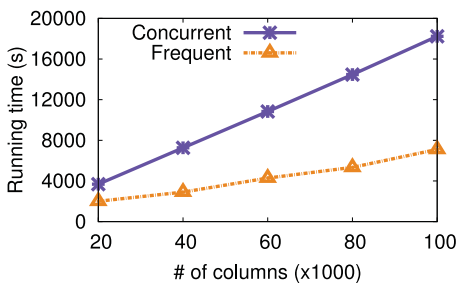


Fig. 10. Comparing concurrent block-wise updates with frequent block-wise updates through varying dataset size.

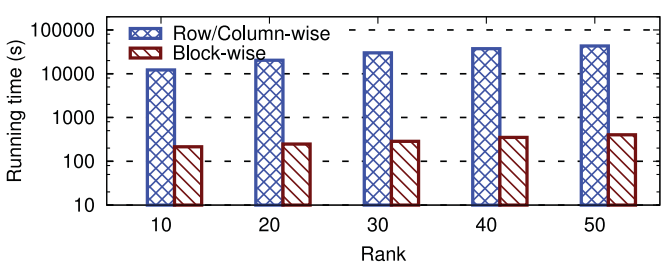


Fig. 11. Time taken in one iteration versus different settings of rank on the local cluster for SED-NMF on MapReduce.

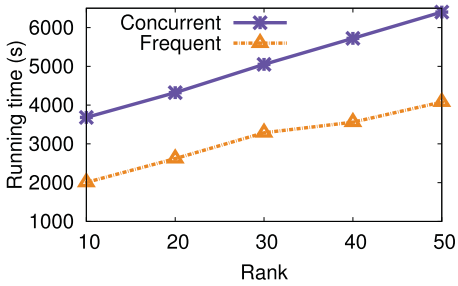


Fig. 12. Running time versus sdifferent settings of rank on the local cluster for SED-NMF on iMapReduce.

of the latter increases much faster. When $k = 50$, our implementation of block-wise updates is 107x faster than the implementation of the traditional updates (compared to 57x speedup when $k = 10$).

We then compare the running time of concurrent block-wise updates with that of frequent block-wise updates as k varies. As plotted in Fig. 12, the running time of either type of updates increases sub-linearly with k. Furthermore, the running time of concurrent block-wise updates increases faster.

6.7 Scaling Performance

To validate the scalability of our implementations, we evaluate them on the Amazon EC2 cloud. The results of SED-NMF are reported. We use dataset Syn-1M-20K, which has 1 million rows, 20 thousand columns, and 2 billion nonzero elements.

As described in Section 4.1, concurrent block-wise updates are equivalent to row/column-wise updates from the matrix operation perspective, and thus they need the same number of iterations to reach the convergence point. Therefore, we leverage the time taken in a single iteration to directly compare their performance. Fig. 13 plots the time taken in a single iteration when all four implementations running on 100 nodes (i.e., instances). Our implementation on MapReduce is 23x faster than that of the existing approach. For block-wise updates, the implementation on iMapReduce is 1.5x faster than that on MapReduce. Fig. 14 shows the time taken in one iteration of the block-wise updates and the traditional row/column-wise updates as the number of nodes being used increases from 20 to 100. The time of either implementation decreases as the number of nodes increases.

Fig. 15 compares the running time of concurrent block-wise updates with that of frequent block-wise updates as the number of nodes increases. We can see that the running time of either frequent block-wise updates or concurrent

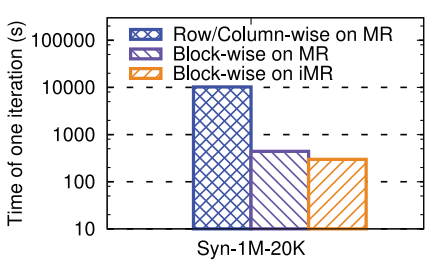


Fig. 13. Time taken in one iteration for KLD-NMF on Amazon EC2 cloud. The y-axis is in log scale.

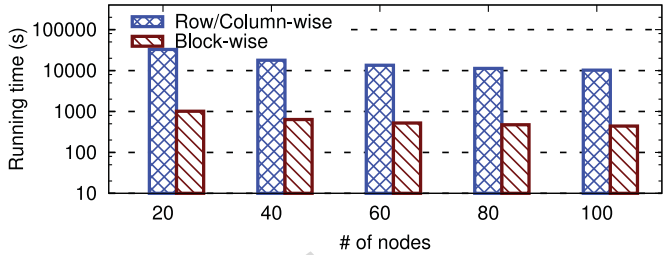


Fig. 14. Scaling performance of MapReduce implementations on Amazon EC2 cloud. The y-axis is in log scale.

block-wise updates decreases smoothly as the number of nodes increases. In addition, frequent block-wise updates outperform concurrent block-wise updates with any number of nodes in the cluster.

7 Related Work

Matrix factorization has been applied very widely [3], [4], [12], [27], [28], [29]. Due to its popularity and increasingly larger datasets, many approaches for paralleling it have been proposed. Zhou et al. [30] and Schelter et al. [31] show how to distribute the alternating least squares algorithm for matrix factorization. Both approaches require that each worker has a copy of one factor matrix when the one is updated. This requirement limits its scalability. For large matrix factorization problems, it is important that factor matrices can be distributed. Several efforts handle matrix factorization using distributed gradient descent methods, which can distribute factor matrix updates across a cluster of machines [5], [20], [21], [32], [33]. These approaches mainly focus on in-memory implementation, in which both the original matrix and factor matrices are in the aggregate memory of the cluster, and use the forms of update functions differently from the form we present. Additionally, our approach puts the original matrix on disk so as to scale to large NMF problems using commodity servers. A closely related work is from Liu et al. [16]. They propose a scheme of implementing the multiplicative update approach on MapReduce. Their scheme is based on the traditional form of update functions and thus has the intermediate data explosion issue.

It has been shown that frequent updates can accelerate Expectation Maximization (EM) algorithms [24], [34], [35], [36], [37]. Somewhat surprisingly, there has been no attempt to apply this approach to NMF, even though there is equivalence between certain variations of NMF and some EM algorithms like K-means [38]. Our work demonstrates that frequent updates can also accelerate NMF.

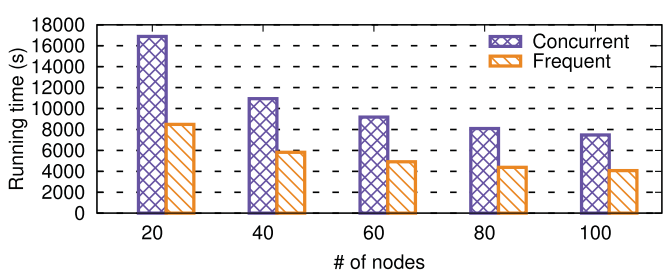


Fig. 15. Scaling performance of iMapReduce implementations on Amazon EC2 cloud.

8 Conclusion

In this paper, we find that by leveraging a new form of factor matrix update functions, block-wise updates, we can perform local aggregation and thus have an efficient MapReduce implementation for NMF. Moreover, we propose frequent block-wise updates and lazy block-wise updates, which aim to use the most recently updated data whenever possible and avoid unnecessary computations. As a result, frequent block-wise updates and lazy block-wise updates can further improve the performance, compared with concurrent block-wise updates. We implement concurrent block-wise updates on MapReduce and implement all block-wise updates on iMapReduce for two classical NMFs: one uses the square of Euclidean distance as the loss function, and the other uses the generalized KL-divergence. With both synthetic and real-world datasets, the evaluation results show that our iMapReduce implementation with frequent block-wise updates is up to two orders of magnitude faster than the existing MapReduce implementation with the traditional form of update functions.

Acknowledgments

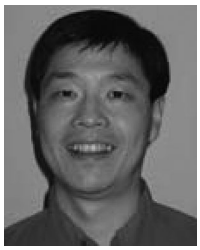
This work is partially supported by US National Science Foundation grants CNS-1217284, CCF-1018114, and CCF-1017828. Part of this work has been published in the *Proceedings of ECML/PKDD'14: 2014 European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases* [1].

References

[1] J. Yin, L. Gao, and Z. M. Zhang, "Scalable nonnegative matrix factorization with block-wise updates," in *Proc. Joint Eur. Conf. Mach. Learn. Knowl. Discovery Databases*, 2014, pp. 337–352.
[2] D. D. Lee and H. S. Seung, "Learning the parts of objects by non-negative matrix factorization," *Nature*, vol. 401, pp. 788–791, 1999.
[3] J.-P. Brunet, P. Tamayo, T. R. Golub, and J. P. Mesirov, "Metagenes and molecular pattern discovery using matrix factorization," *Proc. Nat. Academy Sci.* United States America, vol. 101, pp. 4164–4169, 2004.
[4] V. P. Pauca, F. Shahnaz, M. W. Berry, and R. J. Plemmons, "Text mining using non-negative matrix factorizations," in *Proc. SIAM Int. Conf. Data Mining*, 2004, pp. 452–456.
[5] H.-F. Yu, C.-J. Hsieh, S. Si, and I. Dhillon, "Scalable coordinate descent approaches to parallel matrix factorization for recommender systems," in *Proc. IEEE 12th Int. Conf. Data Mining*, 2012, pp. 765–774.
[6] D. D. Lee and H. S. Seung, "Algorithms for non-negative matrix factorization," in *Proc. Conf. Neural Inf. Process. Syst.*, 2000, pp. 556–562.
[7] C.-J. Lin, "Projected gradient methods for nonnegative matrix factorization," *Neural Comput.*, vol. 19, pp. 2756–2779, 2007.
[8] H. Kim and H. Park, "Nonnegative matrix factorization based on alternating nonnegativity constrained least squares and active set method," *SIAM J. Matrix Anal. Appl.*, vol. 30, pp. 713–730, 2008.
[9] Y. Wang and Y. Zhang, "Nonnegative matrix factorization: A comprehensive review," *IEEE Trans. Knowl. Data Eng.*, vol. 25, no. 6, pp. 1336–1353, Jun. 2013.
[10] S. Yang, Z. Yi, M. Ye, and X. He, "Convergence analysis of graph regularized non-negative matrix factorization," *IEEE Trans. Knowl. Data Eng.*, vol. 26, no. 6, pp. 2151–2165, Sep. 2014.
[11] M. Shiga and H. Mamitsuka, "Non-negative matrix factorization with auxiliary information on overlapping groups," *IEEE Trans. Knowl. Data Eng.*, vol. 27, no. 6, pp. 1615–1628, Jun. 2015.
[12] X. Zhao, et al., "Scalable linear visual feature learning via online parallel nonnegative matrix factorization," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 27, no. 12, pp. 2628–2642, Dec. 2016.
[13] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proc. 6th Conf. Symp. Opearting Syst. Des. Implementation*, 2004, pp. 107–113.
[14] J. Yin, Y. Liao, M. Baldi, L. Gao, and A. Nucci, "GOM-Hadoop: A distributed framework for efficient analytics on ordered datasets," *J. Parallel Distrib. Comput.*, vol. 83, pp. 58–69, 2015.
[15] Z. Wang, Y. Gu, Y. Bao, G. Yu, and J. X. Yu, "Hybrid pulling/pushing for i/o-efficient distributed and iterative graph computing," in *Proc. Int. Conf. Manage. Data*, 2016, pp. 479–494.
[16] C. Liu, H.-C. Yang, J. Fan, L.-W. He, and Y.-M. Wang, "Distributed nonnegative matrix factorization for web-scale dyadic data analysis on mapreduce," in *Proc. 19th Int. Conf. World Wide Web*, 2010, pp. 681–690.
[17] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "iMapReduce: A distributed computing framework for iterative computation," in *Proc. IEEE Int. Symp. Parallel Distrib. Process. Workshops PhD Forum*, 2011, pp. 1112–1121.
[18] A. P. Singh and G. J. Gordon, "A unified view of matrix factorization models," in *Proc. Eur. Conf. Mach. Learn. Knowl. Discovery Databases*, 2008, pp. 358–373.
[19] P. O. Hoyer, "Non-negative matrix factorization with sparseness constraints," *J. Mach. Learn. Res.*, vol. 5, pp. 1457–1469, 2004.
[20] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis, "Large-scale matrix factorization with distributed stochastic gradient descent," in *Proc. 17th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2011, pp. 69–77.
[21] C. Teflioudi, F. Makari, and R. Gemulla, "Distributed matrix completion," in *Proc. 12th Int. Conf. Data Mining*, 2012, pp. 655–664.
[22] J. Kim and H. Park, "Fast nonnegative matrix factorization: An active-set-like method and comparisons," *SIAM J. Sci. Comput.*, vol. 33, pp. 3261–3281, 2011.
[23] J. Yin and L. Gao, "Scalable distributed belief propagation with prioritized block updates," in *Proc. 23rd ACM Int. Conf. Inf. Knowl. Manage.*, 2014, pp. 1209–1218.
[24] B. Thiesson, C. Meek, and D. Heckerman, "Accelerating EM for large databases," *Mach. Learn.*, vol. 45, pp. 279–299, 2001.
[25] J. Yin and L. Gao, "Asynchronous distributed incremental computation on evolving graphs," in *Proc. Joint Eur. Conf. Mach. Learn. Knowl. Discovery Databases*, 2016, pp. 722–738.
[26] UCI Machine Learning Repository, (2017). [Online]. Available: http://archive.ics.uci.edu/ml
[27] Y. Koren, R. Bell, and C. Volinsky, "Matrix factorization techniques for recommender systems," *Comput.*, vol. 42, pp. 30–37, 2009.
[28] F. Wang, T. Li, X. Wang, S. Zhu, and C. Ding, "Community discovery using nonnegative matrix factorization," *Data Mining Knowl. Discovery*, vol. 22, pp. 493–521, May 2011.
[29] A. K. Menon and C. Elkan, "Link prediction via matrix factorization," in *Proc. Eur. Conf. Mach. Learn. Knowl. Discovery Databases*, 20111, pp. 437–452.
[30] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan, "Large-scale parallel collaborative filtering for the netflix prize," in *Proc. 4th Int. Conf. Algorithmic Aspects Inf. Manage.*, 2008, pp. 337–348.
[31] S. Schelter, C. Boden, M. Schenck, A. Alexandrov, and V. Markl, "Distributed matrix factorization with mapreduce using a series of broadcast-joins," in *Proc. 7th ACM Conf. Recommender Syst.*, 2013, pp. 281–284.
[32] B. Li, S. Tata, and Y. Sismanis, "Sparkler: Supporting large-scale matrix factorization," in *Proc. 16th Int. Conf. Extending Database Technol.*, 2013, pp. 625–636.
[33] H. Yun, H.-F. Yu, C.-J. Hsieh, S. V. N. Vishwanathan, and I. Dhillon, "Nomad: Non-locking, stochastic multi-machine algorithm for asynchronous and decentralized matrix completion," *Proc. VLDB Endow.*, vol. 7, pp. 975–986, 2014.
[34] R. Neal and G. E. Hinton, "A view of the EM algorithm that justifies incremental, sparse, and other variants," *Learn. Graph. Models*, vol. 89, pp. 355–368, 1998.
[35] J. Yin, Y. Zhang, and L. Gao, "Accelerating expectation-maximization algorithms with frequent updates," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2012, pp. 275–283.
[36] X. Cheng, S. Su, L. Gao, and J. Yin, "Co-ClusterD: A distributed framework for data co-clustering with sequential updates," *IEEE Trans. Knowl. Data Eng.*, vol. 27, no. 12, pp. 3231–3244, Dec. 2015.
[37] J. Yin, Y. Zhang, and L. Gao, "Accelerating distributed expectationmaximization algorithms with frequent updates," *J. Parallel Distrib. Comput.*, vol. 111, pp. 65–75, 2018.
[38] C. Ding, T. Li, and M. I. Jordan, "Convex and semi-nonnegative matrix factorizations," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 32, no. 1, pp. 45–55, Jan. 2010.

Jiangtao Yin received the PhD degree in electrical and computer engineering from the University of Massachusetts at Amherst, in 2016. He is currently working at Palo Alto Networks. His current research interests include cloud computing, large-scale data processing, stream processing, and distributed systems.

Lixin Gao received the PhD degree in computer science from the University of Massachusetts at Amherst, in 1996. She is a professor of electrical and computer engineering with the University of Massachusetts at Amherst. Her research interests include social networks, and Internet routing, network virtualization, and cloud computing. Between May 1999 and January 2000, she was a visiting researcher at AT&T Research Labs and DIMACS. She was an Alfred P. Sloan fellow between 2003-2005 and received an NSF CAREER Award in 1999. She won the best paper award from IEEE INFOCOM 2010, and the test-of-time award in ACM SIGMETRICS 2010. Her paper in ACM Cloud Computing 2011 was honored with "Paper of Distinction". She received the Chancellor's Award for Outstanding Accomplishment in Research and Creative Activity in 2010. She is a fellow of the ACM and the IEEE.



Zhongfei Zhang received the BS (with honors) degree in electronics engineering and the MS degree in information sciences, both from Zhejiang University, China, and the PhD degree in computer science from the University of Massachusetts at Amherst. He is a professor in the Computer Science Department at the State University of New York (SUNY) at Binghamton, New York, where he directs the Multimedia Research Laboratory in the department. His research interests include knowledge discovery from multimedia data and relational data, multimedia information indexing and retrieval, and computer vision and pattern recognition.