Scalable Nonnegative Matrix Factorization with Block-wise Updates

Jiangtao Yin[1], Lixin Gao[1], and Zhongfei (Mark) Zhang[2]

[1] University of Massachusetts Amherst, Amherst, MA 01003, USA
[2] Binghamton University, Binghamton, NY 13902, USA
{jyin,lgao}@ecs.umass.edu, zhongfei@cs.binghamton.edu

Abstract. Nonnegative Matrix Factorization (NMF) has been applied with great success to many applications. As NMF is applied to massive datasets such as web-scale dyadic data, it is desirable to leverage a cluster of machines to speed up the factorization. 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. Moreover, under the new form of update functions, we can perform frequent updates, which aim to use the most recently updated data whenever possible. As a result, frequent 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 implementation with frequent updates is up to two orders of magnitude faster than the existing implementation with the traditional form of update functions.

1 Introduction

Nonnegative matrix factorization (NMF) [8] 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 [15], recommendation systems [7], and social network analysis [13, 20]. NMF factorizes an original matrix into two low-rank factor matrices by minimizing a loss function that measures the discrepancy between the original matrix and the product of the two factor matrices. NMF algorithms typically use update functions to iteratively and alternately refine factor matrices.

Many practitioners 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. MapReduce [4] 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., [12]) of handling NMF on MapReduce usually select an existing NMF algorithm and then focus on implementing matrix operations.

T. Calders et al. (Eds.): ECML PKDD 2014, Part III, LNCS 8726, pp. 337–352, 2014.

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 and split the original matrix into corresponding blocks. The new form of update functions 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. Additionally, the blocks can be distributed in memories of all the machines in a cluster, thus avoiding overflowing the memory of one single machine. Storing factor matrices in memory allows random access and local aggregation. As a result, the new form of update functions leads to an efficient MapReduce implementation.

Moreover, under the new form of update functions, we can update only a subset of its blocks when we update a factor matrix, and the number of blocks in the subset can be adjusted. The only requirement is that when one factor matrix is being updated, the other one has to be fixed. For instance, we can update one block of a factor matrix and then immediately update all 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 blocks of either factor matrix alternately. Additionally, frequent block-wise updates maintain the convergence property in theory.

We implement concurrent block-wise updates on MapReduce and implement both concurrent and frequent block-wise updates on an extended version of MapReduce, iMapReduce [25], 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 - 57x faster than the existing MapReduce implementation [12] (with the traditional form of update functions) and that our iMapReduce implementation further achieves up to 2x speedup over our MapReduce implementation. Furthermore, the iMapReduce implementation with frequent block-wise updates is up to 2.7x faster than that with concurrent block-wise updates. Accordingly, our iMapReduce implementation with frequent block-wise updates is up to two orders of magnitude faster than the existing MapReduce implementation.

2 Background

NMF aims to factorize an original matrix A into two low-rank factor matrices W and H. Matrix A's elements must be nonnegative by assumption. The achieved factorization has the property of $A \simeq WH$, and the factor matrices W and H are also nonnegative. A loss function is used to measure the discrepancy between A and WH. The NMF problem can be formulated as follows.

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 $L(A, WH)$ is minimized.

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

A general approach is to adopt the block coordinate descent rules [11]:

- 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}: $L(A, W^{t+1} H^{t+1}) \leq L(A, W^t H^{t+1})$.

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

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

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 [9]. It updates W and H as follows:

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

3 Distributed NMF

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

3.1 Decomposition

The loss function is usually decomposable [17]. That is, it can be represented as the sum of losses for each element in the matrix. For example, the widely adopted loss function, the square of the Euclidean distance, is decomposable. We list several popular decomposable loss functions in Table 1. We focus on NMF with decomposable loss functions in this paper.

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 I-divergence	$\sum_{(i,j)} (A_{ij} \log \frac{A_{ij}}{[WH]_{ij}} - (A_{ij} - [WH]_{ij}))$
Itakura-Saito distance	$\sum_{(i,j)} (\frac{A_{ij}}{[WH]_{ij}} - \log \frac{A_{ij}}{[WH]_{ij}} - 1)$

Distributed NMF needs to partition the matrices W, H, and A across computing nodes. To this end, we leverage a popular scheme in gradient descent algorithms [6,18] that partitions W and H into blocks along the short dimension to fully explore parallelism and splits the original matrix A into corresponding blocks. We use symbol $W^{(I)}$ to denote the I^{th} block of W, $H^{(J)}$ to denote the J^{th}

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 $H^{(J)}$, and it is independent of other blocks of W and H, in terms of the 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 Figure 1.

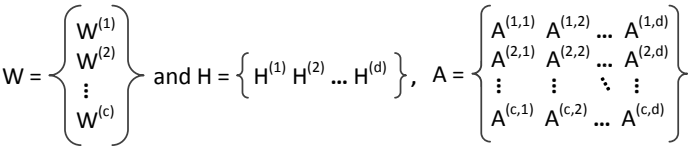


Fig. 1. The block-wise partition scheme for distributed NMF

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

$$L(A, WH) = \sum_I \sum_J L(A^{(I,J)}, W^{(I)}H^{(J)}). \tag{3}$$

Let $F_I = \sum_J L(A^{(I,J)}, W^{(I)}H^{(J)})$ and $G_J = \sum_I L(A^{(I,J)}, W^{(I)}H^{(J)})$, then

$$L(A, WH) = \sum_I F_I = \sum_J G_J. \tag{4}$$

F_I and G_J can be considered as local loss functions. The overall loss function $L(A, WH)$ is the sum of the local loss functions. By fixing H, F_I is independent of one another. 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

In this paper, we use the square of the Euclidean distance as an example of decomposable loss functions. Nevertheless, the techniques derived in this section can be applied to any decomposable loss function.

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).

Here we first show how to update one block of H (i.e., $H^{(J)}$). When W is fixed, minimizing G_J can be expressed as follows:

$$\min_{H^{(J)}} G_J = \min_{H^{(J)}} \sum_I ||A^{(I,J)} - W^{(I)}H^{(J)}||_F^2. \tag{5}$$

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

$$H^{(J)}_{\alpha\mu} = H^{(J)}_{\alpha\mu} - \eta_{\alpha\mu}[\frac{\partial G_J}{\partial H^{(J)}}]_{\alpha\mu}, \tag{6}$$

where $H^{(J)}_{\alpha\mu}$ denotes the element at the α^{th} row and the μ^{th} column of $H^{(J)}$, $\eta_{\alpha\mu}$ is an individual step size for the corresponding gradient element, and

$$\frac{\partial G_J}{\partial H^{(J)}} = \sum_I [(W^{(I)})^T W^{(I)} H^{(J)} - (W^{(I)})^T A^{(I,J)}]. \tag{7}$$

If all step sizes are set to a sufficiently small positive number, the update should reduce G_J. However, if the number is too small, the decrease speed can be very slow. To obtain a good speed and to guarantee convergence, we derive step sizes by following Lee and Seung's multiplicative update approach [9]:

$$\eta_{\alpha\mu} = \frac{H^{(J)}_{\alpha\mu}}{[\sum_I (W^{(I)})^T W^{(I)} H^{(J)}]_{\alpha\mu}}. \tag{8}$$

Then, substituting Eq. (7) and Eq. (8) into Eq. (6), we have:

$$H^{(J)}_{\alpha\mu} = H^{(J)}_{\alpha\mu} * \frac{[\sum_I (W^{(I)})^T A^{(I,J)}]_{\alpha\mu}}{[\sum_I (W^{(I)})^T W^{(I)} H^{(J)}]_{\alpha\mu}}. \tag{9}$$

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

$$W^{(I)}_{\alpha\mu} = W^{(I)}_{\alpha\mu} * \frac{[\sum_J A^{(I,J)} (H^{(J)})^T]_{\alpha\mu}}{[\sum_J W^{(I)} H^{(J)} (H^{(J)})^T]_{\alpha\mu}}. \tag{10}$$

Block-wise updates can update each block of one factor matrix independently. This flexibility allows us to have different ways of updating the 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, where the number of blocks in the subsets can be adjusted. Additionally, block-wise updates also facilitate a distributed implementation. Different blocks of one factor matrix can be updated in parallel. Furthermore, the blocks can be distributed in memories of all the machines in a cluster, thus avoiding overflowing the memory of one single machine (when there are large factor matrices). Storing factor matrices in memory allows random access and local aggregation, which are highly useful for updating them.

3.3 Concurrent Block-wise Updates

With block-wise updates, a straightforward way of fulfilling the block coordinate descent rules is to update all blocks of H and then update all blocks of W. Since this approach updates all blocks of H (or W) concurrently, we refer to it as *concurrent block-wise updates*.

From matrix operation perspective, we can show that concurrent block-wise updates (using Eq. (9) and Eq. (10)) are equivalent to the multiplicative update approach shown in Eq. (2). Without loss of generality, we assume that $H^{(J)}$ is a block of H from the J_0^{th} column to the J_b^{th} column. Let Y be a block of W^TWH from the J_0^{th} column to the J_b^{th} 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 a block of W^TA from the J_0^{th} column to the J_b^{th} column, we can show that $X = \sum_I (W^{(I)})^T A^{(I,J)}$. As a result, for both the concurrent block-wise updates and the multiplicative update approach, the formula for updating $H^{(J)}$ is equivalent to $H^{(J)} = H^{(J)} * \frac{X}{Y}$. Therefore, Eq. (9) is equivalent to the formula for updating H in Eq. (2). Similarly, we can show that Eq. (10) is equivalent to the formula for updating W in Eq. (2).

3.4 Frequent Block-wise Updates

Since all 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 W and then update a subset of blocks of H. Since this approach updates the factor matrices more frequently (compared to concurrent block-wise updates), 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 an 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 using Eq. (9).

Step II: Fix H, update a subset of blocks of W using Eq. (10).

In both steps, the size of the subset is a parameter, and we rotate the subset among all the blocks to guarantee that each block has an equal chance to be updated. The size of the subset controls the update frequency. For example, if we always set the subset to include all the blocks, frequent block-wise updates degrade to concurrent block-wise updates.

Frequent block-wise updates maintain the convergence property. Using techniques similar to that used in [9], we can prove that G_J and F_I are nonincreasing under formulae Eq. (9) and Eq. (10), respectively. Then, it is straightforward to prove that L is nonincreasing when frequent block-wise updates are applied and that L is constant if and only if W and H are at a stationary point of L.

Frequent block-wise updates provide a high flexibility in updating factor matrices. For simplicity, we update a subset of blocks of one factor matrix and then update all 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 additional overhead. However, we find 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 previous iteration. As a result, performing frequent updates on H does not necessarily introduce a large additional cost.

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 $S^{(I)} = (W^{(I)})^T W^{(I)}$. Then, $H^{(J)}_{\alpha\mu}$ can be updated by

$$H^{(J)}_{\alpha\mu} = H^{(J)}_{\alpha\mu} * \frac{X^J_{\alpha\mu}}{[SH^{(J)}]_{\alpha\mu}}. \tag{11}$$

We next show how to incrementally calculate $X^{(J)}$ and S by saving their values from the previous 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:

$$X^{(J)} = X^{(J)} + \sum_{I \in C} [(W^{(I)new})^T A^{(I,J)} - X^{(I,J)}]; \tag{12}$$

$$S = S + \sum_{I \in C} [(W^{(I)new})^T W^{(I)new} - S^{(I)}]. \tag{13}$$

From Eq. (11), Eq. (12), and Eq. (13), 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 Implementation on Distributed Frameworks

MapReduce [4] and its extensions (e.g, [21, 22, 25]) have emerged as distributed frameworks for data intensive computation. MapReduce expresses a computation task as a series of jobs. Each job typically has one map task (mapper) and one reduce task (reducer). In this section, we illustrate the implementation of concurrent block-wise updates on MapReduce. Also, we show how to implement frequent block-wise updates on an extended version of MapReduce, iMapReduce [25], which supports iterative computations more efficiently.

Block-wise updates enable efficient distributed implementation. With block-wise updates, the basic computation units in update functions (Eq. (9) and Eq. (10)) are blocks of factor matrices and of the original matrix. The size of a block can be adjusted. As a result, when performing essential matrix operations that involve 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 machine. Since W and H are low-rank factor matrices, they are usually much smaller than A, and thus the assumption that one of their blocks can be held in the memory of a single machine 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 a single machine 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, each single machine can complete an essential matrix operation locally and efficiently. Note

that the other (larger) block (e.g., a 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. In contrast, the previous work [12], which implements the traditional form of update functions on MapReduce, has a poor performance. For example, to perform matrix multiplication (with two large matrices), a row (or column) of one matrix needs to join with each column (or row) of the other one. Since neither of these two large matrices can be held in memory, a huge amount of intermediate data has to be generated and shuffled.

4.1 Concurrent Block-wise Updates on MapReduce

We first show an efficient MapReduce implementation for concurrent block-wise updates. To realize matrix multiplication with two blocks of matrices in one mapper, we exploit the fact that a mapper can load 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 input pairs have been processed.

The update formula for $H^{(J)}$ (Eq. (9)) can be split into three parts: $X^{(J)} = \sum_I (W^{(I)})^T A^{(I,J)}$, $Y^{(J)} = \sum_I (W^{(I)})^T W^{(I)} H^{(J)}$, and $H^{(J)} = H^{(J)} * \frac{X^{(J)}}{Y^{(J)}}$.

Computing $X^{(J)}$ can be done in one MapReduce job. The mapper calculates $(W^{(I)})^T A^{(I,J)}$, and the reducer performs summation. Let $X^{(I,J)} = (W^{(I)})^T A^{(I,J)}$. When holding $W^{(I)}$ in memory, a mapper can compute $X^{(I,J)}$ via continuously reading elements of $A^{(I,J)}$ from disk: $X^{(I,J)}_{\cdot j} = \sum_{i=1} A^{(I,J)}_{i,j} (W^{(I)}_{i\cdot})^T$, where $X^{(I,J)}_{\cdot j}$ is the j^{th} column of $X^{(I,J)}$, and $W^{(I)}_{i\cdot}$ is the i^{th} row of $W^{(I)}$. $X^{(I,J)}$ (which is usually small) stays in memory for local aggregation. Then, the aggregation $X^{(J)} = \sum_I X^{(I,J)}$ can be computed in a reducer. The operations of this job (Job-I) are illustrated as follows.

- **Map:** Load $W^{(I)}$ in memory first, then calculate $X^{(I,J)} = (W^{(I)})^T A^{(I,J)}$, and last emit $< J, X^{(I,J)} >$.
- **Reduce:** Take $< J, X^{(I,J)} >$ (for any I) and emit $< J, X^{(J)} >$, where $X^{(J)} = \sum_I X^{(I,J)}$.

Computing $Y^{(J)} = \sum_I (W^{(I)})^T W^{(I)} H^{(J)}$ naturally needs two MapReduce jobs. One job (Job-II) is used to compute $S = \sum_I (W^{(I)})^T W^{(I)}$, and the other one is used to calculate $Y^{(J)} = SH^{(J)}$. Let $S^{(I)} = (W^{(I)})^T W^{(I)}$. Calculating $S^{(I)}$ (a small $k \times k$ matrix) can be performed in one mapper. Then, all the mappers send $(W^{(I)})^T W^{(I)}$ to one particular reducer for a global summation. The MapReduce operations are stated as follows.

- **Map:** Load $W^{(I)}$ in memory first, then calculate $S^{(I)} = (W^{(I)})^T W^{(I)}$, and last emit $< 0, S^{(I)} >$ (sending to reducer 0).
- **Reduce:** Take $< 0, S^{(I)} >$ and emit $< 0, S >$, where $S = \sum_I S^{(I)}$.

After computing $S = \sum_I (W^{(I)})^T W^{(I)}$, calculating $Y^{(J)} = SH^{(J)}$ can be done in a MapReduce job (Job-III) with the map phase only, as follows.

– **Map**: Load S in memory. Emit tuples $< J, Y^{(J)} >$, where $Y^{(J)} = SH^{(J)}$.

Lastly, one MapReduce job (with the map phase only) can compute $H^{(J)} \leftarrow H^{(J)} * \frac{X^{(J)}}{Y^{(J)}}$. The operations of this job (Job-IV) are described as follows.

– **Map:** Read $< J, H^{(J)} >$, $< J, X^{(J)} >$, and $< J, Y^{(J)} >$ (column by column). Emit tuple $< J, H^{(J)new} >$, where $H^{(J)new} = H^{(J)} * \frac{X^{(J)}}{Y^{(J)}}$.

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., $X^{(J)}$) needs to be dumped into disk and be reloaded in latter jobs.

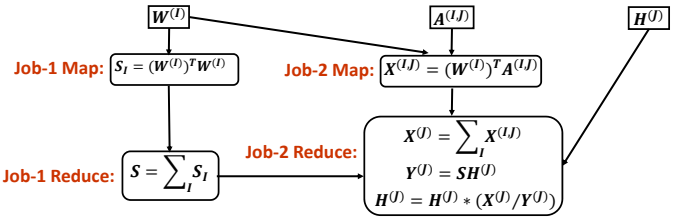


Fig. 2. Overview of the optimized implementation for updating $H^{(J)}$ on MapReduce

To avoid dumping and reloading intermediate data, such as $X^{(J)}$ and $Y^{(J)}$, and to minimize the number of jobs, we integrate Job-I, Job-III, and Job-IV into one job (Job-2). The integrated job has the same map phase as Job-I. However, in the reduce phase, besides computing $X^{(J)}$, it also computes $Y^{(J)}$ and finally calculates $H^{(J)new} = H^{(J)} * [X^{(J)}/Y^{(J)}]$. Job-II can be kept (as Job-1) for the simplicity of implementation since it only produces a small ($k \times k$) matrix and reloading its output does not take much time. The overview of our optimized implementation is presented in Figure 2, and the MapReduce operations in the integrated job (Job-2) are described as follows (the operations in Job-1 are skipped since they are the same with those in Job-II).

– **Map:** Load $W^{(I)}$ in memory first, then calculate $X^{(I,J)} = (W^{(I)})^T A^{(I,J)}$, and last emit $< I, X^{(I,J)} >$.

– **Reduce:** Take $< I, X^{(I,J)} >$, and first calculate $X^{(J)}$. Load S in memory. Then, read $H^{(J)}$ so as to compute $Y^{(J)}$. Last, calculate $H^{(J)new}$.

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. The formula for W (Eq. (10)) can be also 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 $W^{(I)} = W^{(I)} * \frac{U^{(I)}}{V^{(I)}}$. Due to space limitations, we omit the description of its MapReduce operations.

4.2 Frequent Block-wise Updates on iMapReduce

Although frequent block-wise updates have potential to speed up NMF, parallelizing frequent block-wise updates in a distributed environment is challenging.

Computations such as global summations need to be done in a centralized way. Synchronizing the global resources in a distributed environment may result in a considerable overhead, especially on MapReduce. MapReduce starts a new job for each computation errand. Each job needs to be initialized and to load its input data, even when the data is from the previous job. Frequent updates introduce more jobs. Consequently, the initialization overhead and the cost of repeatedly loading data may cause the benefit of frequent updates to vanish.

In this subsection, we propose an implementation of frequent block-wise updates on iMapReduce [25]. iMapReduce uses persistent mappers and reducers to avoid job initialization overhead. Each mapper is paired with one reducer. A 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, state data can be maintained in memory across different iterations. Accordingly, iMapReduce decreases the overhead of a job. As a result, it provides frequent block-wise updates with an opportunity to achieve a good performance.

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 among 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. Nevertheless, we will show in our experiments (Section 5.4) that quite a wide range of p can enable frequent block-wise updates to have better performance than concurrent block-wise updates. Note that like the MapReduce implementation, our iMapReduce implementation still reads A from disk every time rather than holds it in memory so as to scale to large NMF problems. The operations in our iMapReduce implementation are illustrated as follows (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 the previous iteration. For all the loaded or received blocks, compute S_l via $S_l = \sum_B (W^{(B)new})^T W^{(B)new}$ (first iteration) or $S_l = S_l + ((W^{(B)new})^T W^{(B)new} - (W^{(B)})^T W^{(B)})$, and replace $W^{(B)}$ with $W^{(B)new}$. Broadcast $< 0, S_l >$ to all the reducers.
- **Reduce-1a:** Take $< 0, S_l >$, compute $S = \sum_l S_l$, and store S in memory.
- **Map-1b:** For each loaded or received W block in the previous stage (e.g., $(W^{(B)new})$), emit tuple $< J, X^{(B,J)} >$ where $X^J = (W^{(B)new})^T A^{(B,J)}$ (first iteration) or $< J, \Delta X^{(B,J)} >$ where $\Delta X^{(B,J)} = (W^{(B)new})^T A^{(B,J)} - X^{(B,J)}$.

- **Reduce-1b:** Take $< J, X^{(B,J)} >$ and calculate $X^{(J)} = \sum_B X^{(B,J)}$ (first iteration) or take $< J, \Delta X^{(B,J)} >$ and calculate $X^{(J)} = X^{(J)} + \sum_B \Delta X^{(B,J)}$. Then, load $H^{(J)}$ into memory (first iteration) and compute $Y^{(J)} = SH^{(J)}$. Last, calculate $H^{(J)new}$ by $(H^{(J)new} = H^{(J)} * \frac{X^{(J)}}{Y^{(J)}})$, 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 $< J, H^{(J)}(H^{(J)})^T >$ to all the reducers.
- **Reduce-1c:** Take $< J, H^{(J)}(H^{(J)})^T >$, compute $Z = \sum_J H^{(J)}(H^{(J)})^T$, and store Z in memory.
- **Map-1d**: For a W block that is selected in the current iteration (e.g., $(W^{(B)})$), emit tuples in the form of $< (B, U^{(B,J)} >$, where $U^{(B,J)} = A^{(B,J)}(H^{(J)})^T$.
- **Reduce-1d**: Take $< B, U^{(B,J)} >$ and calculate $U^{(B)} = \sum_J U^{(B,J)}$. Then, compute $V^{(B)} = W^{(B)}Z$. Last, calculate $W^{(B)new} = W^{(B)} * \frac{U^{(B)}}{V^{(B)}}$, store it in memory, and pass one copy to Map-1a.

5 Evaluation

In this section, we evaluate the efficiency of block-wise updates. To show the performance improvement, we use the existing implementation [12] as a reference point. For MapReduce, we leverage its open source version, Hadoop [1].

5.1 Experiment Setup

We build both a local cluster and a large-scale cluster on Amazon EC2. The local cluster consists of 4 machines, and each one has a dual-core 2.66GHz CPU, 4GB of RAM, 1TB of disk. The Amazon cluster consists of 100 medium instances, and each instance has one core, 3.7GB of RAM, and 400GB of disk.

Table 2. Summary of datasets

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	$0.1 * m * n$

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 [2]. The other one is a user-movie matrix from the Netflix prize [7]. We also generate several matrices with different choices of m and n. The sparsity is set to 0.1, and each element is a random integer uniformly selected from the range $[1, 5]$. 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 1/8 of its local W blocks in each iteration).

5.2 Comparison with Existing Work

The first set of experiments focus on the advantage of our (optimized) implementation of concurrent block-wise updates on MapReduce. We compare it with the existing work of implementing the multiplicative update approach on MapReduce [12], which uses the traditional form of update functions. We also include the implementation of concurrent block-wise updates on iMapReduce in the comparison (by setting $p = 1$) to show iMapReduce's superiority over MapReduce. As described in Section 3.4, concurrent block-wise updates are equivalent to the multiplicative update approach, and thus we focus on the time taken in a single iteration to directly compare the performance. Figure 3 shows the time taken in one iteration for all the three implementations. 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 MapReduce implementation (denoted by "Row/Column-wise on MR"). Moreover, the implementation on iMapReduce (denoted by "Block-wise on iMR") is up to 2x faster than that on MapReduce.

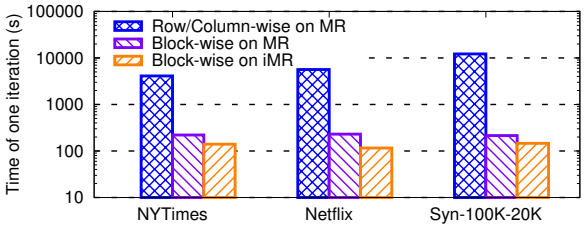


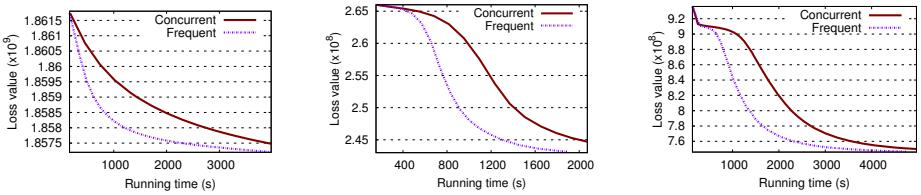
Fig. 3. Time taken in one iteration for different implementations on the local cluster

5.3 Effect of Frequent Block-wise updates

To evaluate the effect of frequent block-wise updates, we compare frequent block-wise updates with concurrent block-wise updates when both are implemented on iMapReduce. Both update approaches start with the same initial values when compared on the same dataset. Figure 4 plots the performance comparison. We can see that frequent block-wise updates ("Frequent") converge much faster than concurrent block-wise updates ("Concurrent") on all the three 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.

5.4 Tuning Update Frequency

As stated in Section 4.2, the update frequency affects the performance of frequent block-wise updates. In the experiments, we find that quite a wide range of p can allow frequent block-wise updates to have a better performance than their



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

Fig. 4. Convergence speed on the local cluster

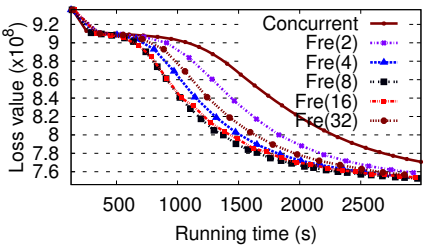


Fig. 5. Convergence speed vs. update frequency on dataset Netflix. The numbers associated with "Fre" represent different settings of p.

concurrent counterparts, and the best setting of p stays in the range from 4 to 32. This is also why we set $p = 8$ by default. For example, Figure 5 shows the convergence speed with different settings on dataset Netflix. Another interesting finding is that if a setting is better during a first few iterations, it will continue to be better. Hence, another 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.

5.5 Different Data Sizes

We also measure how block-wise updates scale with the increasing size of matrix A. We generate synthetic datasets of different sizes by fixing the number of $(100k)$ rows and increasing the number of columns. We use the loss value when concurrent block-wise updates run for 25 iterations as the convergence point. The time used to reach this convergence point is measured as the running time. This criterion also applies to the latter comparisons. As presented in Figure 6, the running times of both types of updates increase linearly with the size of the dataset. Moreover, frequent block-wise updates are up to 2.7x faster than concurrent block-wise updates.

To summarize, our iMapReduce implementation with frequent block-wise updates ("Frequent") is up to two orders of magnitude faster than the existing MapReduce implementation ("Row/Column-wise on MR"). Take dataset Syn-100K-20K for example. Our MapReduce implementation with concurrent

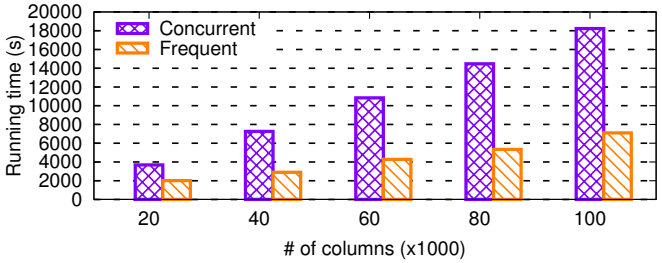
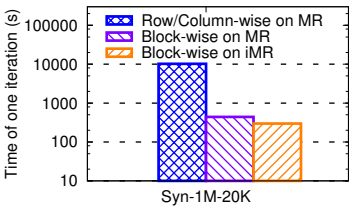


Fig. 6. Running time vs. dataset size on the local cluster

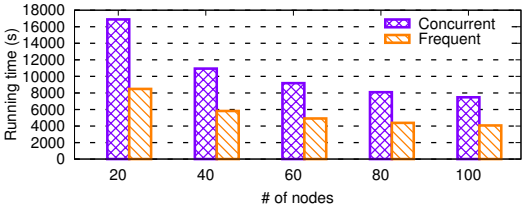
block-wise updates ("Block-wise on MR") is 57x faster than the existing MapReduce implementation (as shown in Figure 3). The iMapReduce implementation ("Block-wise on iMR") achieves 1.5x speedup over the MapReduce implementation. Furthermore, on iMapReduce, frequent block-wise updates are 1.8x faster than concurrent block-wise updates. In total, our iMapReduce implementation with frequent block-wise updates is 154x faster than the existing MapReduce implementation for dataset Syn-100K-20K.

5.6 Scaling Performance

To validate the scalability of our implementations, we evaluate them on the Amazon EC2 cloud. We use dataset Syn-1M-20K, which has 1 million rows, 20 thousand columns, and 2 billion nonzero elements. Figure 7a plots the time taken in a single iteration when all the three implementations are running on 100 nodes. Our implementation on MapReduce is 23x faster than the existing implementation. Moreover, the implementation on iMapReduce is 1.5x faster than that on MapReduce. Figure 7b shows the performance as the number of nodes being used increases from 20 to 100. We can see that the running times of both frequent block-wise updates and concurrent block-wise updates decrease 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.



(a) 100 nodes (y-axis in log scale)



(b) Varying number of nodes

Fig. 7. Performance on the Amazon EC2 cloud

6 Related Work

Matrix factorization has been applied very widely [3,7,13,15,20]. Due to its popularity and increasingly larger datasets, many approaches for parallelizing it have been proposed. Zhou et al. [26] and Schelter et al. [16] show how to distribute the alternating least squares algorithm. Both approaches require each computing node to have a copy of one factor matrix when the other factor matrix is updated. This requirement limits their scalability. For large matrix factorization problems, it is important that factor matrices can be distributed. Several works handle matrix factorization using distributed gradient descent methods [6,10,18,24]. These approaches mainly focus on in-memory settings, in which both the original matrix and factor matrices are held in memory, and the forms of update functions used are different from the form we present. Additionally, our approach allows the original matrix to be in disk so as to scale to large NMF problems. A closely related work is from Liu et al. [12]. They propose a scheme of implementing the multiplicative update approach on MapReduce. Their scheme is based on the traditional form of update functions, which results in a poor performance.

It has been shown that frequent updates can accelerate expectation maximization (EM) algorithms [14,19,23]. Somewhat surprisingly, there has been no attempt to apply this method to NMF, even though there is equivalence between certain variations of NMF and some particular EM algorithms like K-means [5]. Our work demonstrates that frequent updates can also accelerate NMF.

7 Conclusion

In this paper, we find that by leveraging a new form of 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, which aim to use the most recently updated data whenever possible. As a result, frequent block-wise updates can further improve the performance, comparing with concurrent block-wise updates. We implement frequent block-wise updates on iMapReduce, an extended version of MapReduce. The evaluation results show that our iMapReduce implementation is up to two orders of magnitude faster than the existing MapReduce implementation.

Acknowledgments. We thank anonymous reviewers for their insightful comments. This work is partially supported by NSF grants CNS-1217284, CCF-1018114, and CCF-1017828. Any opinions, findings, conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the sponsor.

References

1. Apache Hadoop, `http://hadoop.apache.org/`
2. UCI Machine Learning Repository, `http://archive.ics.uci.edu/ml`
3. Brunet, J.-P., Tamayo, P., Golub, T.R., Mesirov, J.P.: Metagenes and molecular pattern discovery using matrix factorization. PNAS 101(12), 4164–4169 (2004)

4. Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large clusters. In: OSDI 2004, pp. 107–113 (2004)
5. Ding, C., Li, T., Jordan, M.I.: Convex and semi-nonnegative matrix factorizations. TPAMI 32(1), 45–55 (2010)
6. Gemulla, R., Nijkamp, E., Haas, P.J., Sismanis, Y.: Large-scale matrix factorization with distributed stochastic gradient descent. In: KDD 2011, pp. 69–77 (2011)
7. Koren, Y., Bell, R., Volinsky, C.: Matrix factorization techniques for recommender systems. Computer 42(8), 30–37 (2009)
8. Lee, D.D., Seung, H.S.: Learning the parts of objects by non-negative matrix factorization. Nature 401, 788–791 (1999)
9. Lee, D.D., Seung, H.S.: Algorithms for non-negative matrix factorization. In: NIPS, pp. 556–562. MIT Press (2000)
10. Li, B., Tata, S., Sismanis, Y.: Sparkler: Supporting large-scale matrix factorization. In: EDBT 2013, pp. 625–636 (2013)
11. Lin, C.-J.: Projected gradient methods for nonnegative matrix factorization. Neural Comput. 19(10), 2756–2779 (2007)
12. Liu, C., Yang, H.-C., Fan, J., He, L.-W., Wang, Y.-M.: Distributed nonnegative matrix factorization for web-scale dyadic data analysis on mapreduce. In: WWW 2010, pp. 681–690 (2010)
13. Menon, A.K., Elkan, C.: Link prediction via matrix factorization. In: Gunopulos, D., Hofmann, T., Malerba, D., Vazirgiannis, M. (eds.) ECML PKDD 2011, Part II. LNCS, vol. 6912, pp. 437–452. Springer, Heidelberg (2011)
14. Neal, R., Hinton, G.E.: A view of the EM algorithm that justifies incremental, sparse, and other variants. In: Learning in Graphical Models, pp. 355–368 (1998)
15. Pauca, V.P., Shahnaz, F., Berry, M.W., Plemmons, R.J.: Text mining using non-negative matrix factorizations. In: SDM, pp. 452–456 (2004)
16. Schelter, S., Boden, C., Schenck, M., Alexandrov, A., Markl, V.: Distributed matrix factorization with mapreduce using a series of broadcast-joins. In: RecSys 2013, pp. 281–284 (2013)
17. Singh, A.P., Gordon, G.J.: A unified view of matrix factorization models. In: Daelemans, W., Goethals, B., Morik, K. (eds.) ECML PKDD 2008, Part II. LNCS (LNAI), vol. 5212, pp. 358–373. Springer, Heidelberg (2008)
18. Teflioudi, C., Makari, F., Gemulla, R.: Distributed matrix completion. In: ICDM 2012, pp. 655–664 (2012)
19. Thiesson, B., Meek, C., Heckerman, D.: Accelerating EM for large databases. Mach. Learn. 45(3), 279–299 (2001)
20. Wang, F., Li, T., Wang, X., Zhu, S., Ding, C.: Community discovery using non-negative matrix factorization. Data Min. Knowl. Discov. (May 2011)
21. Yin, J., Liao, Y., Baldi, M., Gao, L., Nucci, A.: Efficient analytics on ordered datasets using mapreduce. In: HPDC 2013, pp. 125–126 (2013)
22. Yin, J., Liao, Y., Baldi, M., Gao, L., Nucci, A.: A scalable distributed framework for efficient analytics on ordered datasets. In: UCC 2013, pp. 131–138 (2013)
23. Yin, J., Zhang, Y., Gao, L.: Accelerating expectation-maximization algorithms with frequent updates. In: CLUSTER 2012, pp. 275–283 (2012)
24. Yu, H.-F., Hsieh, C.-J., Si, S., Dhillon, I.: Scalable coordinate descent approaches to parallel matrix factorization for recommender systems. In: ICDM 2012 (2012)
25. Zhang, Y., Gao, Q., Gao, L., Wang, C.: iMapReduce: A distributed computing framework for iterative computation. In: DataCloud 2011, pp. 1112–1121 (2011)
26. Zhou, Y., Wilkinson, D., Schreiber, R., Pan, R.: Large-scale parallel collaborative filtering for the netflix prize. In: Fleischer, R., Xu, J. (eds.) AAIM 2008. LNCS, vol. 5034, pp. 337–348. Springer, Heidelberg (2008)