The Thirty-Second AAAI Conference
on Artificial Intelligence (AAAI-18)

DID: Distributed Incremental Block Coordinate Descent forNonnegative Matrix Factorization

Tianxiang Gao, Chris Chu
Department of Electrical and Computer Engineering,
Iowa State University, Ames, IA, 50011, USA
{gaotx, cnchu}@iastate.edu

Abstract

Nonnegative matrix factorization (NMF) has attracted much attention in the last decade as a dimension reduction method in many applications. Due to the explosion in the size of data, naturally the samples are collected and stored distributively in local computational nodes. Thus, there is a growing need to develop algorithms in a distributed memory architecture. We propose a novel distributed algorithm, called *distributed incremental block coordinate descent* (DID), to solve the problem. By adapting the block coordinate descent framework, closed-form update rules are obtained in DID. Moreover, DID performs updates incrementally based on the most recently updated residual matrix. As a result, only one communication step per iteration is required. The correctness, efficiency, and scalability of the proposed algorithm are verified in a series of numerical experiments.

1 Introduction

Nonnegative matrix factorization (NMF) (Lee and Seung 1999) extracts the latent factors in a low dimensional subspace. The popularity of NMF is due to its ability to learn *parts*-based representation by the use of nonnegative constraints. Numerous successes have been found in document clustering (Xu and Gong 2004; Lu, Hong, and Wang 2017a), computer vision (Lee and Seung 1999), signal processing (Gao, Olofsson, and Lu 2016; Lu, Hong, and Wang 2017b), etc.

Suppose a collection of N samples with M nonnegative measurements is denoted in matrix form $X \in \mathbf{R}_+^{M\times N}$, where each column is a sample. The purpose of NMF is to approximate X by a product of two nonnegative matrices $B \in \mathbf{R}_+^{M\times K}$ and $C \in \mathbf{R}_+^{K\times N}$ with a desired low dimension K, where $K \ll \min\{M, N\}$. The columns of matrix B can be considered as a *basis* in the low dimension subspace, while the columns of matrix C are the *coordinates*. NMF can be formulated as an optimization problem in (1):

$$\underset{B,C}{\text{minimize}} \qquad f(B,C) = \frac{1}{2}\left\|X - BC\right\|_F^2 \tag{1a}$$

$$\text{subject to} \qquad B, C \geq 0, \tag{1b}$$

where "≥ 0" means element-wise nonnegative, and $\|\cdot\|_F$ is the Frobenius norm. The problem (1) is nonconvex with respect to variables B and C. Finding the global minimum is NP-hard (Vavasis 2009). Thus, a practical algorithm usually converges to a local minimum.

Copyright © 2018, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

Many algorithms have been proposed to solve NMF such as *multiplicative updates* (MU) (Lee and Seung 2001), *hierarchical alternating least square* (HALS) (Cichocki, Zdunek, and Amari 2007; Li and Zhang 2009), *alternating direction multiplier method* (ADMM) (Zhang 2010), and *alternating nonnegative least square* (ANLS) (Kim and Park 2011). Amongst those algorithms, ANLS has the largest reduction of objective value per iteration since it exactly solves *nonnegative least square* (NNLS) subproblems using a *block principal pivoting* (BPP) method (Kim and Park 2011). Unfortunately, the computation of each iteration is costly. The algorithm HALS, on the other hand, solves subproblems inexactly with cheaper computation and has achieved faster convergence in terms of time (Kim and Park 2011; Gillis and Glineur 2012). Instead of iteratively solving the subproblems, ADMM obtains closed-form solutions by using auxiliary variables. A drawback of ADMM is that it is sensitive to the choice of the tuning parameters, even to the point where poor parameter selection can lead to algorithm divergence (Sun and Fevotte 2014).

Most of the proposed algorithms are intended for centralized implementation, assuming that the whole data matrix can be loaded into the RAM of a single computer node. In the era of massive data sets, however, this assumption is often not satisfied, since the number of samples is too large to be stored in a single node. As a result, there is a growing need to develop algorithms in distributed system. Thus, in this paper, we assume the number of samples is so large that the data matrix is collected and stored distributively. Such applications can be found in e-commerce (*e.g.*, Amazon), digital content streaming (*e.g.*, Netflix) (Koren, Bell, and Volinsky 2009) and technology (*e.g.*, Facebook, Google) (Tan, Cao, and Fong 2016), where they have hundreds of millions of users.

Many distributed algorithms have been published recently. The distributed MU (Liu et al. 2010; Yin, Gao, and Zhang 2014) has been proposed as the first distributed algorithm to solve NMF. However, MU suffers from slow and ill convergence in some cases (Lin 2007). Kannan, Ballard, and Park (Kannan, Ballard, and Park 2016) proposed high performance ANLS (HPC-ANLS) using 2D-grid partition

of a data matrix such that each node only stores a submatrix of the data matrix. Nevertheless, six communication steps per iteration are required to obtain intermediate variables so as to solve the subproblems. Thus, the communication overhead is significant. Moreover, the computation is costly as they use ANLS framework. The most recent work is distributed HALS (D-HALS) (Zdunek and Fonal 2017). However, they assume the factors B and C can be stored in the shared memory of the computer nodes, which may not be the case if N is large. Boyd et al. (Boyd et al. 2011) suggested that ADMM has the potential to solve NMF distributively. Du et al. (Du et al. 2014) demonstrated this idea in an algorithm called Maxios. Similar to HPC-ANLS, the communication overhead is expensive, since every latent factor or auxiliary variable has to be gathered and broadcasted over all computational nodes. As a result, eight communication steps per iteration are necessary. In addition, Maxios only works for sparse matrices since they assume the whole data matrix is stored in every computer node.

In this paper, we propose a distributed algorithm based on block coordinate descent framework. The main contributions of this paper are listed below.

- We propose a novel distributed algorithm, called *distributed incremental block coordinate descent* (DID). By splitting the columns of the data matrix, DID is capable of updating the coordinate matrix C in parallel. Leveraging the most recent residual matrix, the basis matrix B is updated distributively and incrementally. Thus, only one communication step is needed in each iteration.
- A scalable and easy implementation of DID is derived using *Message Passing Interface* (MPI). Our implementation does not require a master processor to synchronize.
- Experimental results showcase the correctness, efficiency, and scalability of our novel method.

The paper is organized as follows. In Section 2, the previous works are briefly reviewed. Section 3 introduces a distributed ADMM for comparison purpose. The novel algorithm DID is detailed in Section 4. In Section 5, the algorithms are evaluated and compared. Finally, the conclusions are drawn in Section 6.

2 Previous Works

In this section we briefly introduce three standard algorithms to solve NMF problem (1), *i.e.*, ANLS, HALS, and ADMM, and discuss the parallelism of their distributed versions.

Notations. Given a nonnegative matrix $X \in \mathbf{R}_+^{M\times N}$ with M rows and N columns, we use $x_i^r \in \mathbf{R}_+^{1\times N}$ to denote its i-th row, $x_j \in \mathbf{R}_+^{M\times 1}$ to denote the j-th column, and $x_{ij} \in \mathbf{R}_+$ to denote the entry in the i-th row and j-th column. In addition, we use $x_i^{rT} \in \mathbf{R}_+^{N\times 1}$ and $x_j^T \in \mathbf{R}_+^{1\times M}$ to denote the transpose of i-th row and j-th column, respectively.

2.1 ANLS

The optimization problem (1) is biconvex, *i.e.*, if either factor is fixed, updating another is in fact reduced to a *nonnegative least square* (NNLS) problem. Thus, ANLS (Kim and Park 2011) minimizes the NNLS subproblems with respect to B and C, alternately. The procedure is given by

$$C := \operatorname{argmin}_{C\geq 0} \|X - BC\|_F^2 \tag{2a}$$

$$B := \operatorname{argmin}_{B\geq 0} \|X - BC\|_F^2 . \tag{2b}$$

The optimal solution of a NNLS subproblem can be achieved using BPP method.

A naive distributed ANLS is to parallel C-minimization step in a column-by-column manner and B-minimization step in a row-by-row manner. HPC-ANLS (Kannan, Ballard, and Park 2016) divides the matrix X into 2D-grid blocks, the matrix B into P_r row blocks, and the matrix C into P_c column blocks so that the memory requirement of each node is $\mathcal{O}(\frac{MN}{P_rP_c} + \frac{MK}{P_r} + \frac{NK}{P_c})$, where P_r is the number of rows processor and P_c is the number of columns processor such that $P = P_cP_r$ is the total number of processors. To really perform updates, the intermediate variables CC^T, XC^T, B^TB, and B^TX are computed and broadcasted using totally six communication steps. Each of them has a cost of $\log P\cdot(\alpha+\beta\cdot NK)$, where α is *latency*, and β is *inverse bandwidth* in a *distributed memory network* model (Chan et al. 2007). The analysis is summarized in Table 1.

2.2 HALS

Since the optimal solution to the subproblem is not required when updating one factor, a comparable method, called HALS, which achieves an approximate solution is proposed by (Cichocki, Zdunek, and Amari 2007). The algorithm HALS successively updates each column of B and row of C with an optimal solution in a closed form.

The objective function in (1) can be expressed with respect to the k-th column of B and k-th row of C as follows

$$\left\|X - BC\right\|_F^2 = \left\|X - \textstyle\sum_{i=1}^K b_i c_i^r\right\| = \left\|X - \textstyle\sum_{i\neq k} b_i c_i^r - b_k c_k^r\right\|_F^2,$$

Let $A \triangleq X - \sum_{i\neq k} b_i c_i^r$ and fix all the variables except b_k or c_k^r. We have subproblems in b_k and c_k^r

$$\min_{b_k\geq 0}\ \|A - b_k c_k^r\|_F^2 , \tag{3a}$$

$$\min_{c_k^r\geq 0}\ \|A - b_k c_k^r\|_F^2 \tag{3b}$$

By setting the derivative with respect to b_k or c_k^r to zero and projecting the result to the nonnegative region, the optimal solution of b_k and c_k^r can be easily written in a closed form

$$b_k := \left[(c_k^r c_k^{rT})^{-1}(Ac_k^{rT})\right]_+ \tag{4a}$$

$$c_k^r := \left[(b_k^T b_k)^{-1}(A^T b_k)\right]_+ \tag{4b}$$

where $[z]_+$ is $\max\{0, z\}$. Therefore, we have K inner-loop iterations to update every pair of b_k and c_k^r. With cheaper computational cost, HALS was confirmed to have faster convergence in terms of time.

Zdunek and Fonal in 2017 proposed a distributed version of HALS, called DHALS. They also divide the data matrix X into 2D-grid blocks. Comparing with HPC-ANLS, the resulting algorithm DHALS only requires two communication steps. However, they assume matrices B and C can be loaded into the shared memory of a single node. Therefore, DHALS is not applicable in our scenario where we assume N is so big that even the latent factors are stored distributively. See the detailed analysis in Table 1.

2.3 ADMM

The algorithm ADMM (Zhang 2010) solves the NMF problem by alternately optimizing the Lagrangian function with respect to different variables. Specifically, the NMF (1) is reformulated as

$$\begin{aligned} \underset{B,C,W,H}{\text{minimize}} \quad & \frac{1}{2}\left\|X - WH\right\|_F^2 && \text{(5a)}\\ \text{subject to} \quad & B = W, C = H && \text{(5b)}\\ & B, C \geq 0, && \text{(5c)} \end{aligned}$$

where $W \in \mathbf{R}^{M\times K}$ and $H \in \mathbf{R}^{K\times N}$ are *auxiliary variables* without *nonnegative* constraints. The *augmented Lagrangian function* is given by

$$\mathcal{L}(B,C,W,H;\Phi,\Psi)_\rho = \frac{1}{2}\left\|X - WH\right\|_F^2 + \langle\Phi, B - W\rangle + \langle\Psi, C - H\rangle + \frac{\rho}{2}\left\|B - W\right\|_F^2 + \frac{\rho}{2}\left\|C - H\right\|_F^2 \quad (6)$$

where $\Phi \in \mathbf{R}^{M\times K}$ and $\Psi \in \mathbf{R}^{K\times N}$ are *Lagrangian multipliers*, $\langle\cdot,\cdot\rangle$ is the matrix inner product, and $\rho > 0$ is the penalty parameter for equality constraints. By minimizing $\mathcal{L}$ with respect to W, H, B, C, Φ, and Ψ one at a time while fixing the rest, we obtain the update rules as follows

$$\begin{aligned} W &:= (XH^T + \Phi + \rho B)(HH^T + \rho I_K)^{-1} && \text{(7a)}\\ H &:= (W^TW + \rho I_K)^{-1}(W^TX + \Psi + \rho C) && \text{(7b)}\\ B &:= [W - \Phi/\rho]_+ && \text{(7c)}\\ C &:= [H - \Psi/\rho]_+ && \text{(7d)}\\ \Phi &:= \Phi + \rho(B - W) && \text{(7e)}\\ \Psi &:= \Psi + \rho(C - H) && \text{(7f)} \end{aligned}$$

where $I_K \in \mathbf{R}^{K\times K}$ is the identity matrix. The auxiliary variables W and H facilitate the minimization steps for B and C. When ρ is small, however, the update rules for W and H result in unstable convergence (Sun and Fevotte 2014). When ρ is large, ADMM suffers from a slow convergence. Hence, the selection of ρ is significant in practice.

Analogous to HPC-ANLS, the update of W and B can be parallelized in a column-by-column manner, while the update of H and C in a row-by-row manner. Thus, Maxios (Du et al. 2014) divides matrix W and B in column blocks, and matrix H and C in row blocks. However, the communication overhead is expensive since one factor update depends on the others. Thus, once a factor is updated, it has to be broadcasted to all other computational nodes. As a consequence, Maxios requires theoretically eight communication steps per iteration and only works for sparse matrices. Table 1 summarizes the analysis.

3 Distributed ADMM

This section derives a *distributed ADMM* (DADMM) for comparison purpose. DADMM is inspired by another centralized version in (Boyd et al. 2011; Hajinezhad et al. 2016), where the update rules can be easily carried out in parallel, and is stable when ρ is small.

As the objective function in (1) is *separable* in columns, we divide matrices X and C into column blocks of P parts

$$\frac{1}{2}\left\|X - BC\right\|_F^2 = \sum_{i=1}^{P}\frac{1}{2}\left\|X_i - BC_i\right\|_2^2, \quad (8)$$

where $X_i \in \mathbf{R}_+^{M\times N_i}$ and $C_i \in \mathbf{R}_+^{K\times N_i}$ are column blocks of X and C such that $\sum_{i=1}^{P} N_i = N$. Using a set of auxiliary variables $Y_i \in \mathbf{R}^{M\times N_i}$, the NMF (1) can be reformulated as

$$\begin{aligned} \underset{Y_i,B,C}{\text{minimize}} \quad & \sum_{i=1}^{P}\frac{1}{2}\left\|X_i - Y_i\right\|_F^2 && \text{(9a)}\\ \text{subject to} \quad & Y_i = BC_i, \quad \text{for } i = 1,2,\cdots,P && \text{(9b)}\\ & B, C \geq 0. && \text{(9c)} \end{aligned}$$

The associated augmented Lagrangian function is given by

$$\mathcal{L}(Y_i,B,C;\Lambda_i)_\rho = \sum_{i=1}^{P}\frac{1}{2}\left\|X_i - Y_i\right\|_F^2 + \sum_{i=1}^{P}\langle\Lambda_i, Y_i - BC_i\rangle + \sum_{i=1}^{P}\frac{\rho}{2}\left\|Y_i - BC_i\right\|_F^2, \quad (10)$$

where $\Lambda_i \in \mathbf{R}^{M\times K}$ are the Lagrangian multipliers. The resulting ADMM is

$$Y_i := \underset{Y_i}{\operatorname{argmin}}\, \frac{1}{2}\left\|X_i - Y_i\right\|_2^2 + \frac{\rho}{2}\left\|\Lambda_i/\rho + Y_i - BC_i\right\|_F^2 \quad \text{(11a)}$$

$$C_i := \underset{C_i \geq 0}{\operatorname{argmin}}\, \left\|\Lambda_i/\rho + Y_i - BC_i\right\|_2^2 \quad \text{(11b)}$$

$$B := \underset{B \geq 0}{\operatorname{argmin}}\, \left\|\Lambda/\rho + Y - BC\right\|_F^2 \quad \text{(11c)}$$

$$\Lambda_i := \underset{\Lambda_i}{\operatorname{argmax}}\langle\Lambda_i, Y_i - BC_i\rangle \quad \text{(11d)}$$

where $\Lambda \triangleq [\Lambda_1 \;\; \Lambda_2 \cdots \Lambda_P]$ and $Y \triangleq [Y_1 \;\; Y_2 \cdots Y_P]$. Clearly, the Y_i update has a closed-form solution by taking the derivate and setting it to zero, *i.e.*,

$$Y_i := \frac{1}{1+\rho}(X_i - \Lambda_i + \rho BC_i) \quad (12)$$

Moreover, the updates for Y_i, C_i, and Λ_i can be carried out in *parallel*. Meanwhile, B needs a central processor to update since the step (11c) requires the whole matrices Y, C, and Λ. If we use the solver BPP, however, we do not really need to gather those matrices, because the solver BPP in fact does not explicitly need Y, C, and Λ. Instead, it requires two intermediate variables $W \triangleq CC^T$ and $H \triangleq (\Lambda/\rho + Y)C^T$, which can be computed as follows:

$$W \triangleq CC^T = \sum_{i=1}^{P} C_iC_i^T, \quad \text{(13a)}$$

$$H \triangleq (\Lambda/\rho + Y)C^T = \sum_{i=1}^{P}(\Lambda_i/\rho + Y_i)C_i^T. \quad \text{(13b)}$$

Algorithm	Runtime	Memory per processor	Communication time	Communication volume
HPC-ANLS	BPP	$\mathcal{O}(MN/(P_cP_r)+MK/P_r+NK/P_c)$	$3(\alpha+\beta NK)\log P_r+3(\alpha+\beta MK)\log P_c$	$\mathcal{O}(MKP_c+NKP_r)$
D-HALS	$\mathcal{O}(MNK(1/P_c+1/P_r))$	$\mathcal{O}(MN/(P_cP_r)+MK+NK)$	$(\alpha+\beta NK)\log P_r+(\alpha+\beta MK)\log P_c$	$\mathcal{O}(MKP_c+NKP_r)$
Maxios	$\mathcal{O}(K^3+MNK/P)$	$\mathcal{O}(MN)$	$4(2\alpha+\beta(N+M)K)\log P$	$\mathcal{O}((M+N)KP)$
DADMM	BPP	$\mathcal{O}(MN/P+MK)$	$(\alpha+\beta MK)\log P$	$\mathcal{O}(MKP)$
DBCD	$\mathcal{O}(MNK/P)$	$\mathcal{O}(MN/P+MK)$	$K(\alpha+\beta MK)\log P$	$\mathcal{O}(MKP)$
DID	$\mathcal{O}(MNK/P)$	$\mathcal{O}(MN/P+MK)$	$(\alpha+\beta MK)\log P$	$\mathcal{O}(MKP)$

Table 1: Analysis of distributed algorithms per iteration on runtime, memory storage, and communication time and volume.

It is no doubt that those intermediate variables can be calculated distributively. Let $U_i = \Lambda_i/\rho$, which is called *scaled dual variable*. Using the scaled dual variable, we can express DADMM in a more efficient and compact way. A simple MPI implementation of algorithm DADMM on each computational node is summarized in Algorithm 1.

Algorithm 1: DADMM for each computational node

Input: X_i, C_i, B
Initialize P processors, along with Y_i, B, C_i, X_i
repeat
1. $U_i := U_i + (Y_i - BC_i)$
2. $Y_i := \frac{1}{1+\rho}(X_i - \rho U_i + \rho BC_i)$
3. $C_i := \operatorname{argmin}_{C_i \geq 0} \|U_i + Y_i - BC_i\|_2^2$
4. $(W, H) := Allreduce(C_iC_i^T, (U_i+Y_i)C_i^T)$
5. $B := \text{BPP}(W, H)$

until *stopping criteria satisfied*;

At line 4 in Algorithm 1, theoretically we need a master processor to *gather* $C_iC_i^T$ and $(U_i{+}Y_i)C_i^T$ from every local processor and then *broadcast* the updated value of CC^T and $(U+Y)C^T$ back. As a result, the master processor needs a storage of $\mathcal{O}(MKP)$. However, we use a collaborative operation called *Allreduce* (Chan et al. 2007). Leveraging it, the master processor is discarded and the storage of each processor is reduced to $\mathcal{O}(MK)$.

4 Distributed Incremental Block Coordinate Descent

The popularity of ADMM is due to its ability of carrying out subproblems in parallel such as DADMM in Algorithm 1. However, the computation of ADMM is costly since it generally involves introducing new auxiliary variables and updating dual variables. The computational cost is even more expensive as it is required to find optimal solutions of subproblems to ensure convergence. In this section, we will propose another distributed algorithm that adapts block coordinate descent framework and achieves approximate solutions at each iteration. Moreover, leveraging the current residual matrix facilitates the update for matrix B so that columns of B can be updated incrementally.

4.1 Distributed Block Coordinate Descent

We firstly introduce a naive parallel and distributed algorithm, which is inspired by HALS, called *distributed block coordinate descent* (DBCD). Since the objective function in (1) is separable, the matrix X is partitioned by columns, then each processor is able to update columns of C in parallel, and prepare messages concurrently to update matrix B.

Analogous to DADMM, the objective function in (1) can be expanded as follows

$$\left\|X - BC\right\|_F^2 = \sum_{j=1}^N \left\|x_j - Bc_j\right\|^2 = \sum_{j=1}^N \left\|x_j - \sum_{k=1}^K b_k c_{kj}\right\|^2$$

By coordinate descent framework, we only consider one element at a time. To update c_{ij}, we fix the rest of variables as constant, then the objective function becomes

$$\sum_{j=1}^{N} \left\|x_j - \sum_{k\neq i} b_k c_{kj} - b_i c_{ij}\right\|^2. \tag{14}$$

Taking the partial derivative of the objective function (14) with respect to c_{ij} and setting it to zero, we have

$$b_i^T \left(b_i c_{ij} - \left(x_j - \sum_{k\neq i} b_k c_{kj}\right)\right) = 0. \tag{15}$$

The optimal solution of c_{ij} can be easily derived in a closed form as follows

$$c_{ij} := \left[\frac{b_i^T(x_j - \sum_{k\neq i} b_k c_{kj})}{b_i^T b_i}\right]_+ \tag{16a}$$

$$= \left[\frac{b_i^T(x_j - Bc_j + b_i c_{ij})}{b_i^T b_i}\right]_+ \tag{16b}$$

$$= \left[c_{ij} + \frac{b_i^T(x_j - Bc_j)}{b_i^T b_i}\right]_+ \tag{16c}$$

Based on the equation (16c), the j-th column of C is required so as to update c_{ij}. Thus, updating a column c_j has to be sequential. However, the update can be executed in parallel for all j's. Therefore, the columns of matrix C can be updated independently, while each component in a column c_j is optimized in sequence.

The complexity of updating each c_{ij} is $\mathcal{O}(MK)$. Thus, the entire complexity of updating matrix C is $\mathcal{O}(MNK^2/P)$. This complexity can be reduced by bringing $x_j - Bc_j$ outside the loop and redefining as $e_j \triangleq x_j - Bc_j$. The improved update rule is

$$e_j := e_j + b_i c_{ij} \tag{17a}$$

$$c_{ij} := \left[\frac{b_i^T e_j}{b_i^T b_i}\right]_+ \tag{17b}$$

$$e_j := e_j - b_i c_{ij} \tag{17c}$$

By doing so, the complexity is reduced to $\mathcal{O}(MNK/P)$.

The analogous derivation can be carried out to update the i-th column of matrix B, *i.e.*, b_i. By taking partial derivative

of the objective function (14) with respect to b_i and setting it to zero, we have equation

$$\sum_{j=1}^{N}\left(b_i c_{ij} - \left(x_j - \sum\nolimits_{k\neq i} b_k c_{kj}\right)\right) c_{ij} = 0 \tag{18}$$

Solving this linear equation gives us a closed-form to the optimal solution of b_i

$$b_i := \left[\frac{\sum_{j=1}^{N}(x_j - Bc_j + b_i c_{ij})c_{ij}}{\sum_{j=1}^{N} c_{ij}^2}\right]_+ \tag{19a}$$

$$= \left[b_i + \frac{\sum_{j=1}^{N}(x_j - Bc_j)c_{ij}}{\sum_{j=1}^{N} c_{ij}^2}\right]_+ \tag{19b}$$

$$= \left[b_i + \frac{(X - BC)c_i^{rT}}{c_i^r c_i^{rT}}\right]_+ \tag{19c}$$

Unfortunately, there is no way to update b_i in parallel since the equation (19c) involves the whole matrices X and C. That is the reason why sequential algorithms can be easily implemented in the shared memory but cannot directly be applied in distributed memory. Thus, other works (Kannan, Ballard, and Park 2016; Zdunek and Fonal 2017; Du et al. 2014) either use *gather* operations to collect messages from local processors or assume small size of the latent factors.

By analyzing the equation (19a), we discover the potential parallelism. We define a vector y_j and a scaler z_j as follows

$$y_j \triangleq (x_j - Bc_j + b_i c_{ij})c_{ij} = (e_j + b_i c_{ij})c_{ij} \tag{20a}$$

$$z_j \triangleq c_{ij}^2 \tag{20b}$$

The vector y_j and scaler z_j can be computed in parallel. After receiving messages including y_j's and z_j's from other processors, a master processor updates the column b_i as a scaled summation of y_j with scaler $z \triangleq \sum_{j=1}^{N} z_j$, that is,

$$b_i := [y/z]_+ \tag{21}$$

where $y \triangleq \sum_{j=1}^{N} y_j$. Thus, the update for matrix B can be executed in parallel but indirectly. The complexity of updating b_i is $\mathcal{O}(MN/P)$ as we reserve error vector e_j and concurrently compute y_j and z_j. The complexity of updating entire matrix B is $\mathcal{O}(MNK/P)$.

By partitioning the data matrix X by columns, the update for matrix C can be carried out in parallel. In addition, we identify vectors y_j's and scalars z_j's to update matrix B, and their computation can be executed concurrently among computational nodes. A MPI implementation of this algorithm for each processor is summarized in Algorithm 2.

Algorithm 2: DBCD for each computational node

Input: x_j, c_j, B
repeat
  `// Update` C
  $e_j := x_j - Bc_j$
  for *all* $i \in \{1, 2, \cdots, K\}$ **do**
    Update c_{ij} using equations (17)
  end
  `// Update` B
  for *all* $i \in \{1, 2, \cdots, K\}$ **do**
    $e_j = e_j + b_i c_{ij}$
    $y_j = e_j c_{ij}$
    $z_j = c_{ij}^2$
    $(y, z) = Allreduce(y_j, z_j)$
    $b_i := [y/z]_+$
    $e_j = e_j - b_i c_{ij}$
  end
until *stopping criteria satisfied*;

4.2 Incremental Update for b_i

The complexity of algorithm DBCD is $\mathcal{O}(MNK/P)$ per iteration, which is perfectly parallelizing a sequential block coordinate descent algorithm. However, the performance of DBCD could be deficient due to the delay in network. In principle, DBCD sends totally KP messages to a master processor per iteration, which is even more if we implement DBCD using *Allreduce*. Any delay of a message could cause a diminished performance. In contrast, the algorithm DID has a novel way to update matrix B incrementally using only a single message from each processor per iteration.

To successfully update matrix B, the bottleneck is to iteratively compute y_j and z_j for associated b_i since once b_i is updated, the y_j and z_j have to be recomputed due to the change occurred in matrix B from equation (19b). Nevertheless, we discovered this change can be represented as several arithmetic operations. Thus, we in fact do not need to communicate every time in order to update each b_i.

Suppose that after t-th iteration, the i-th column of matrix B is given, *i.e.*, b_i^t, and want to update it to b_i^{t+1}. Let $E = X - BC$, which is the most current residual matrix after t-th iteration. From equation (19c), we have

$$b_i^{t+1} := \left[b_i^t + \frac{Ec_i^{rT}}{c_i^r c_i^{rT}}\right]_+ \tag{22}$$

Once we update b_i^t to b_i^{t+1}, we need to update b_i in matrix B so as to get new E to update the next column of B, *i.e.*, b_{i+1}. However, we do not really need to recalculate E. Instead, we can update the value by

$$E := E + b_i^t c_i^r - b_i^{t+1} c_i^r \tag{23}$$

We define and compute a variable δb_i as

$$\delta b_i \triangleq b_i^{t+1} - b_i^t. \tag{24}$$

Using the vector δb_i, we have a compact form to update E

$$E := E - \delta b_i c_i^r \tag{25}$$

The updated E is substituted into the update rule of b_{i+1} in equation (22), and using b_{i+1}^t we obtain

$$b_{i+1}^{t+1} := \left[b_{i+1}^t + \frac{(E - \delta b_i c_i^r)c_{i+1}^{rT}}{c_{i+1}^r c_{i+1}^{rT}}\right]_+ \tag{26a}$$

$$= \left[b_{i+1}^t + \frac{Ec_{i+1}^{rT}}{c_{i+1}^r c_{i+1}^{rT}} - \frac{c_i^r c_{i+1}^{rT}}{c_{i+1}^r c_{i+1}^{rT}}\delta b_i\right]_+ \tag{26b}$$

In the equation (26b), the first two terms is the same as general update rule for matrix B in DBCD, where Ec_{i+1} can be computed distributively in each computational node. On the other hand, the last term allows us to update the column b_{i+1} still in a closed form but without any communication step. Therefore, the update for matrix B can be carried out incrementally and the general update rule is given by

$$b_i^{t+1} := \left[b_i^t + \frac{E c_i^{rT}}{c_i^r c_i^{rT}} - \frac{\sum_{k<i}(c_i^r c_k^{rT})\delta b_k}{c_i^r c_i^{rT}} \right]_+ \tag{27}$$

Comparing to the messages used in DBCD, *i.e.*, (y_j, z_j), we need to compute the coefficients for the extra term, that is, $c_i^r c_k^{rT}$ for all $k < i$. Thus, a message communicated among processors contains two parts: the weighted current residual matrix W_j, and a lower triangular matrix V_j maintaining the inner product of matrix C. The matrices W_j and V_j are defined as below

$$W_j \triangleq \begin{bmatrix} | & | & \cdots & | \\ e_j c_{1j} & e_j c_{2j} & \cdots & e_j c_{Kj} \\ | & | & \cdots & | \end{bmatrix} \tag{28}$$

$$V_j \triangleq \begin{bmatrix} c_{1j}^2 & 0 & 0 & \cdots & 0 \\ c_{2j}c_{1j} & c_{2j}^2 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & 0 \\ c_{Kj}c_{1j} & c_{Kj}c_{2j} & c_{Kj}c_{3j} & \cdots & c_{Kj}^2 \end{bmatrix} \tag{29}$$

Using variables W_j and V_j, the update rule to columns of matrix B becomes

$$b_i := \left[b_i + w_i/v_{ii} - \sum_{k<i}(v_{ik}/v_{ii})\delta b_k \right]_+ \tag{30}$$

where w_i is the i-th column of matrix W, v_{ij} is the i-th component of j-th column of matrix V, and matrices W and V are the summations of matrices W_j and V_j, respectively, *i.e.*, $W \triangleq \sum_{j=1}^N W_j$ and $V \triangleq \sum_{j=1}^N V_j$.

For each processor, they store a column of X, a column of C, and the matrix B. They execute the same algorithm and a MPI implementation of this incremental algorithm for each computational node is summarized in Algorithm 3. Clearly, the entire computation is unchanged and the volume of message stays the same as DBCD, but the number of communication is reduced to once per iteration.

Algorithm 3: DID for each computational node

```
Input: x_j, c_j, B
repeat
    // Update C
    e_j := x_j − B c_j
    for all i ∈ {1, 2, ··· , K} do
        Update c_ij using equations (17)
    end
    Compute W_j and V_j from equations (28) and
     (29).
    (W, V) := Allreduce(W_j, V_j)
    // Update B
    for all i ∈ {1, 2, ··· , K} do
        b_i^{t+1} := [b_i^t + w_i/v_ii − Σ_{k<i} (v_ik/v_ii) δb_k]_+
        δb_i := b_i^{t+1} − b_i^t
    end
until stopping criteria satisfied;
```

5 Experiments

We conduct a series of numerical experiments to compare the proposed algorithm DID with HALS, ALS, ADMM, BCD, DBCD, DADMM, and HPC-ANLS. The algorithm BCD is the sequential version of DBCD. Due to the ill convergence of ADMM and Maxios in (Zhang 2010; Du et al. 2014), we derive DADMM in Section 3 and set $\rho = 1$ as default. Since we assume M and K are much smaller than N, HPC-ANLS only has column partition of the matrix X, *i.e.*, $P_c = P$ and $P_r = 1$.

We use a cluster[1] that consists of 48 SuperMicro servers each with 16 cores, 64 GB of memory, GigE and QDR (40Gbit) InfiniBand interconnects. The algorithms are implemented in C code. The linear algebra operations use GNU Scientific Library (GSL) v2.4[2] (Gough 2009). The Message Passing Interface (MPI) implementation OpenMPI v2.1.0[3] (Gabriel et al. 2004) is used for communication. Note that we do not use multi-cores in each server. Instead, we use *a single core per node* as we want to achieve consistent communication overhead between cores.

Synthetic datasets are generated with number of samples $N = 10^5, 10^6, 10^7$ and 10^8. Due to the storage limits of the computer system we use, we set the dimension $M = 5$ and low rank $K = 3$, and utilize $P = 16$ number of computational nodes in the cluster. The random numbers in the synthetic datasets are generated by the Matlab command `rand` that are uniformly distributed in the interval $[0, 1]$.

We also perform experimental comparisons on four real-world datasets. The MNIST dataset[4] of handwritten digits has 70,000 samples of 28x28 image. The 20News dataset[5] is a collection of 18,821 documents across 20 different newsgroups with totally 8,165 keywords. The UMist dataset[6] contains 575 images of 20 people with the size of 112x92. The YaleB dataset[7] includes 2,414 images of 38 individuals with the size of 32x32. The MNIST and 20News datasets are sparse, while UMist and YaleB are dense.

The algorithms HALS, (D)BCD, and DID could fail if $\|b_i\|$ or $\|c_i^r\|$ is close to zero. This could appear if B or C is badly scaled. That means the entries of $E = X - BC$ are strictly negative. We avoid this issue by using well scaled initial points for the synthetic datasets and K-means method to generate the initial values for the real datasets. All the algorithms are provided with the same initial values.

When an iterative algorithm is executed in practice, a

[1] http://www.hpc.iastate.edu/

[2] http://www.gnu.org/software/gsl/

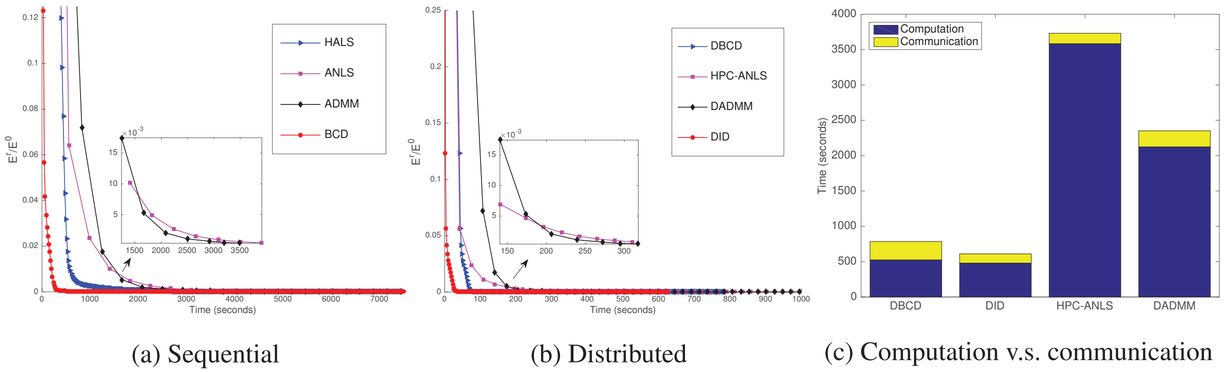
[3] https://www.open-mpi.org/

[4] http://yann.lecun.com/exdb/mnist/

[5] http://qwone.com/~jason/20Newsgroups/

[6] https://cs.nyu.edu/~roweis/data.html

[7] http://www.cad.zju.edu.cn/home/dengcai/Data/FaceData.html



(a) Sequential

(b) Distributed

(c) Computation v.s. communication

Figure 1: Convergence behaviors of different algorithms with respect to time consumption of communication and computation on the dataset with $N = 10^8$ samples.

	Number of iterations								Time (seconds)							
N	HALS	ANLS	ADMM	BCD	HPC-ANLS	DADMM	DBCD	**DID**	HALS	ANLS	ADMM	BCD	HPC-ANLS	DADMM	DBCD	**DID**
10^5	1281	**141**	170	549	**141**	170	549	549	16.88	56.59	45.88	10.61	4.31	3.46	1.42	**1.17**
10^6	225	238	**115**	396	238	**115**	396	396	36.86	630.43	476.83	95.24	50.47	37.06	14.04	**8.61**
10^7	**596**	1120	1191	654	1120	1191	654	654	587.47	29234.61	31798.51	909.76	2372.47	2563.47	126.01	**106.60**
10^8	339	163	**97**	302	163	**97**	302	302	3779.11	43197.12	27590.16	8808.92	10172.55	5742.37	785.57	**610.09**
MNIST	495	**197**	199	492	**197**	**199**	492	492	705.32	395.61	610.65	942.68	**31.84**	46.17	170.65	133.50
20News	302	**169**	**169**	231	**169**	**169**	231	231	2550.02	745.28	714.61	2681.49	**131.12**	172.69	651.52	559.70
UMist	677	1001	953	**622**	1001	953	**622**	**622**	314.72	657.14	836.76	422.11	492.72	471.01	92.49	**82.34**
YaleB	1001	352	**224**	765	352	**224**	765	765	223.58	201.22	149.35	236.13	50.69	40.61	44.08	**36.45**

Table 2: Performance comparison for algorithms on synthetic and real datasets with $P = 16$ number of computing nodes.

stopping criteria is required. In our experiments, the stopping criteria is met if the following condition is satisfied

$$\left\|E^t\right\|_F^2 \leq \epsilon \left\|E^0\right\|_F^2, \tag{31}$$

where E^t is the residual matrix after t-th iteration. Throughout the experiments, we set $\epsilon = 10^{-6}$ as default. In addition, we combine the stopping criterion with *a limit on time* of 24 hours and *a maximum iteration* of 1000 for real datasets. The experimental results are summarized in the Table 2.

Correctness In principle, the algorithms HALS, (D)BCD, and DID have the same update rules for the latent factors B and C. The difference is the update order. The algorithm DID has the exact same number of iterations as BCD and DBCD, which demonstrates the correctness of DID.

Efficiency As presented in Table 2, DID always converges faster than the other algorithms in term of time. HALS and BCD usually use a similar number of iterations to reach the stopping criteria. ANLS and ADMM use much fewer iterations to converge. Thanks to auxiliary variables, ADMM usually converges faster than ANLS. Figure 1(a) shows that comparing with HALS, BCD actually reduces the objective value a lot at the beginning but takes longer to finally converge. Such phenomenon can also be observed in the comparison between ANLS and ADMM. In Figure 1(b), DID is faster than DBCD. The reason is shown in Figure 1(c) that DID involves much less communication overhead than DBCD. Based on the result in Table 2, DID is about 10-15% faster than DBCD by incrementally updating matrix B. (HPC-)ANLS works better in MNIST and 20News datasets because these datasets are very sparse.

Scalability As presented in Table 2, the runtime of DID scales linearly as the number of samples increases, which is much better than the others. It can usually speed up a factor of at least 10 to BCD using 16 nodes. (D)ADMM is also linearly scalable, which is slightly better than (HPC-)ANLS. Due to the costly computation, (D)ADMM is not preferred to solve NMF problems.

6 Conclusion

In this paper, we proposed a novel distributed algorithm DID to solve NMF in a distributed memory architecture. Assume the number of samples N to be huge, DID divides the matrices X and C into column blocks so that updating the matrix C is perfectly distributed. Using the variables δb, the matrix B can be updated distributively and incrementally. As a result, only a single communication step per iteration is required. The algorithm is implemented in C code with OpenMPI. The numerical experiments demonstrated that DID has faster convergence than the other algorithms. As the update only requires basic matrix operations, DID achieves linear scalability, which is observed in the experimental results. In the future work, DID will be applied to the cases where updating matrix B is also carried out in parallel. Using the techniques introduced by (Hsieh and Dhillon 2011) and (Gillis and Glineur 2012), DID has the possibility to be accelerated. How to better treat sparse datasets is also a potential research direction.

References

Boyd, S.; Parikh, N.; Chu, E.; Peleato, B.; and Eckstein, J. 2011. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends in Machine Learning* 1–122.

Chan, E.; Heimlich, M.; Purkayastha, A.; and Van De Geijn, R. 2007. Collective communication: theory, practice, and experience. *Concurrency and Computation: Practice and Experience* 1749–1783.

Cichocki, A.; Zdunek, R.; and Amari, S.-i. 2007. Hierarchical als algorithms for nonnegative matrix and 3d tensor factorization. In *International Conference on Independent Component Analysis and Signal Separation*, 169–176. Springer.

Du, S. S.; Liu, Y.; Chen, B.; and Li, L. 2014. Maxios: Large scale nonnegative matrix factorization for collaborative filtering. In *Proceedings of the NIPS 2014 Workshop on Distributed Matrix Computations*.

Gabriel, E.; Fagg, G. E.; Bosilca, G.; Angskun, T.; Dongarra, J. J.; Squyres, J. M.; Sahay, V.; Kambadur, P.; Barrett, B.; Lumsdaine, A.; Castain, R. H.; Daniel, D. J.; Graham, R. L.; and Woodall, T. S. 2004. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, 97–104.

Gao, T.; Olofsson, S.; and Lu, S. 2016. Minimum-volume-regularized weighted symmetric nonnegative matrix factorization for clustering. In *2016 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*, 247–251. IEEE.

Gillis, N., and Glineur, F. 2012. Accelerated multiplicative updates and hierarchical als algorithms for nonnegative matrix factorization. *Neural computation* 1085–1105.

Gough, B. 2009. *GNU scientific library reference manual*. Network Theory Ltd.

Hajinezhad, D.; Chang, T.-H.; Wang, X.; Shi, Q.; and Hong, M. 2016. Nonnegative matrix factorization using admm: Algorithm and convergence analysis. In *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 4742–4746. IEEE.

Hsieh, C.-J., and Dhillon, I. S. 2011. Fast coordinate descent methods with variable selection for non-negative matrix factorization. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, 1064–1072. ACM.

Kannan, R.; Ballard, G.; and Park, H. 2016. A high-performance parallel algorithm for nonnegative matrix factorization. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 9. ACM.

Kim, J., and Park, H. 2011. Fast nonnegative matrix factorization: An active-set-like method and comparisons. *SIAM Journal on Scientific Computing* 3261–3281.

Koren, Y.; Bell, R.; and Volinsky, C. 2009. Matrix factorization techniques for recommender systems. *Computer*.

Lee, D. D., and Seung, H. S. 1999. Learning the parts of objects by non-negative matrix factorization. *Nature* 788–791.

Lee, D. D., and Seung, H. S. 2001. Algorithms for non-negative matrix factorization. In *Advances in neural information processing systems*, 556–562.

Li, L., and Zhang, Y.-J. 2009. Fastnmf: highly efficient monotonic fixed-point nonnegative matrix factorization algorithm with good applicability. *Journal of Electronic Imaging* 033004–033004.

Lin, C.-J. 2007. On the convergence of multiplicative update algorithms for nonnegative matrix factorization. *IEEE Transactions on Neural Networks* 1589–1596.

Liu, C.; Yang, H.-c.; Fan, J.; He, L.-W.; and Wang, Y.-M. 2010. Distributed nonnegative matrix factorization for web-scale dyadic data analysis on mapreduce. In *Proceedings of the 19th international conference on World wide web*, 681–690. ACM.

Lu, S.; Hong, M.; and Wang, Z. 2017a. A Stochastic Nonconvex Splitting Method for Symmetric Nonnegative Matrix Factorization. In Singh, A., and Zhu, J., eds., *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*, volume 54 of *Proceedings of Machine Learning Research*, 812–821. Fort Lauderdale, FL, USA: PMLR.

Lu, S.; Hong, M.; and Wang, Z. 2017b. A nonconvex splitting method for symmetric nonnegative matrix factorization: Convergence analysis and optimality. *IEEE Transactions on Signal Processing*.

Sun, D. L., and Fevotte, C. 2014. Alternating direction method of multipliers for non-negative matrix factorization with the beta-divergence. In *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*, 6201–6205. IEEE.

Tan, W.; Cao, L.; and Fong, L. 2016. Faster and cheaper: Parallelizing large-scale matrix factorization on gpus. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, 219–230. ACM.

Vavasis, S. A. 2009. On the complexity of nonnegative matrix factorization. *SIAM Journal on Optimization* 1364–1377.

Xu, W., and Gong, Y. 2004. Document clustering by concept factorization. In *Proceedings of the 27th annual international ACM SIGIR conference on Research and development in information retrieval*, 202–209. ACM.

Yin, J.; Gao, L.; and Zhang, Z. M. 2014. Scalable nonnegative matrix factorization with block-wise updates. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, 337–352. Springer.

Zdunek, R., and Fonal, K. 2017. Distributed nonnegative matrix factorization with hals algorithm on mapreduce. In *International Conference on Algorithms and Architectures for Parallel Processing*, 211–222. Springer.

Zhang, Y. 2010. An alternating direction algorithm for nonnegative matrix factorization. *preprint*.