Co-ClusterD: A Distributed Framework for Data Co-Clustering with Sequential Updates

Xiang Cheng, Sen Su, Lixin Gao, *Fellow, IEEE*, and Jiangtao Yin

Abstract—Co-clustering has emerged to be a powerful data mining tool for two-dimensional co-occurrence and dyadic data. However, co-clustering algorithms often require significant computational resources and have been dismissed as impractical for large data sets. Existing studies have provided strong empirical evidence that expectation-maximization (EM) algorithms (e.g., k-means algorithm) with sequential updates can significantly reduce the computational cost without degrading the resulting solution. Motivated by this observation, we introduce sequential updates for alternate minimization co-clustering (AMCC) algorithms which are variants of EM algorithms, and also show that AMCC algorithms with sequential updates converge. We then propose two approaches to parallelize AMCC algorithms with sequential updates in a distributed environment. Both approaches are proved to maintain the convergence properties of AMCC algorithms. Based on these two approaches, we present a new distributed framework, Co-ClusterD, which supports efficient implementations of AMCC algorithms with sequential updates. We design and implement Co-ClusterD, and show its efficiency through two AMCC algorithms: fast nonnegative matrix tri-factorization (FNMTF) and information theoretic co-clustering (ITCC). We evaluate our framework on both a local cluster of machines and the Amazon EC2 cloud. Empirical results show that AMCC algorithms implemented in Co-ClusterD can achieve a much faster convergence and often obtain better results than their traditional concurrent counterparts.

Index Terms—Co-Clustering, concurrent updates, sequential updates, cloud computing, distributed framework

1 Introduction

Co-clustering is a powerful data mining tool for two-dimensional co-occurrence and dyadic data. It has practical importance in a wide range of applications such as text mining [1], recommendation systems [2], and the analysis of gene expression data [3]. Typically, clustering algorithms leverage an iterative refinement method to group input points into clusters. The cluster assignments are performed based on the current cluster information (e.g., the centroids of clusters in k-means clustering). The resulted cluster assignments can be utilized to further update the cluster information. Such a refinement process is iterated till the cluster assignments become stable. Depending on how frequently the cluster information is updated, clustering algorithms can be broadly categorized into two classes. The first class updates the cluster information after all input points have updated their cluster assignments. We refer to this class of algorithms as clustering algorithms with *concurrent updates*. In contrast, the second class updates the cluster information whenever a point changes its cluster assignment. We refer to this class of algorithms as clustering algorithms with *sequential updates*.

A number of existing studies (e.g., [4], [5], [6], [7], [8]) have provided strong empirical evidence that expectation-maximization (EM) algorithms (e.g., k-means algorithm) with sequential updates can significantly reduce the computational cost without degrading the resulting solution. Motivated by this observation, we introduce sequential updates for *alternate minimization co-clustering* (AMCC) algorithms [9], which are variants of EM algorithms. We show that AMCC algorithms with sequential updates converge. Despite the potential advantages of sequential updates, parallelizing AMCC algorithms with sequential updates is challenging. Specifically, if we let each worker machine update the cluster information sequentially, it might result in inconsistent cluster information across worker machines and thus the convergence properties of co-clustering algorithms cannot be guaranteed; if we synchronize the cluster information whenever a cluster assignment is changed, it will incur large synchronization overhead and thus result in poor performance in a distributed environment. Consequently, AMCC algorithms with sequential updates cannot be easily performed in a distributed manner.

Toward this end, we propose two approaches to parallelize sequential updates for AMCC algorithms. The first approach is referred to as *dividing clusters*. It divides the problem of clustering rows (or columns) into independent tasks and each of which is assigned to a worker. In order to make tasks independent, we randomly divide row (or column) clusters into multiple non-overlapping subsets at the beginning of each iteration, and let each worker perform row (or column) clustering with sequential updates on one of these subsets.

The second approach is referred to as *batching points*. Relaxing the stringent requirement of sequential updates, it parallelizes sequential updates by performing *batch updates*.

- X. Cheng and S. Su are with the State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing, China. E-mail: {chengxiang, susen}@bupt.edu.cn.
- L. Gao and J. Yin are with the Department of Electrical and Computer Engineering, University of Massachusetts Amherst, MA 01003. E-mail: {lgao, jyin}@ecs.umass.edu.

Manuscript received 26 Oct. 2013; revised 10 Apr. 2015; accepted 23 June 2015. Date of publication 30 June 2015; date of current version 3 Nov. 2015. Recommended for acceptance by I. Davidson.
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.2015.2451634

1041-4347 © 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

Instead of updating the cluster information after each change in cluster assignments, batch updates perform a batch of row (or column) cluster assignments, and then update the cluster information. We typically divide rows and columns of the input data matrix into several batches and let all workers perform row (or column) clustering with concurrent updates on each batch.

We formally prove that the dividing clusters and batching points approaches can maintain the convergence properties of AMCC algorithms. Based on these two approaches, we design and implement a distributed framework, *Co-ClusterD*, to support efficient implementations of AMCC algorithms with sequential updates. Co-ClusterD provides an abstraction for AMCC algorithms with sequential updates and allows programmers to specify the sequential update operations via simple APIs. As a result, it eases the process of implementing AMCC algorithms and frees programmer from the detailed mechanisms such as communication and synchronization between workers. We evaluate Co-ClusterD through two AMCC algorithms: fast nonnegative matrix tri-factorization (FNMTF) [10] and information theoretic co-clustering (ITCC) [11]. Experimenting on a local cluster of machines and the Amazon EC2 cloud, we show that AMCC algorithms implemented in Co-ClusterD can run faster and often get better results than their traditional concurrent counterparts. In summary, we make three contributions in this paper.

- We introduce sequential updates for alternate minimization co-clustering (AMCC) algorithms and show that AMCC algorithms with sequential updates converge.
- We propose two approaches (i.e., dividing clusters and batching points) to parallel AMCC algorithms with sequential updates in a distributed environment. We prove that both of these two approaches can maintain the convergence properties of AMCC algorithms. We also provide solutions for setting optimal parameters for these two approaches.
- We design and implement a distributed framework called Co-ClusterD which can support efficient implementations of AMCC algorithms with sequential updates. Empirical results on both a local cluster and the Amazon EC2 cloud show that AMCC algorithms implemented in Co-ClusterD can achieve a much faster convergence and often obtain better results than their traditional concurrent counterparts.

The rest of this paper is organized as follows. Section 2 gives an overview of AMCC and shows how update strategies affect its performance. Section 3 presents our approaches to parallelize AMCC algorithms with sequential updates. In Section 4, we present the design and implementation of our co-clustering framework. Section 5 is devoted to the experimental results. We discuss related work in Section 6 and conclude the paper in Section 7.

2 Alternate Minimization Based Co-Clustering and Sequential Update

In this section, we first describe the co-clustering problem we focus on. Next, we introduce sequential updates for alternate minimization co-clustering algorithms. Then, taking two AMCC algorithms (i.e., fast nonnegative matrix tri-factorization [10] and information theoretic co-clustering [11]) as examples, we show how sequential updates work. At last, we elaborate the performance comparison between concurrent updates and sequential updates through FNMTF and ITCC on a single machine.

$$\mathrm{A}=\begin{pmatrix}3&6&3&6\\9&3&9&3\\3&6&3&6\\9&3&9&3\end{pmatrix}\xrightarrow[\gamma:\{1,2,3,4\}\to\{1,2,1,2\}]{\rho:\{1,2,3,4\}\to\{1,2,1,2\}}\mathrm{B}=\left(\begin{array}{cc|cc}3&3&6&6\\3&3&6&6\\\hline 9&9&3&3\\9&9&3&3\end{array}\right)$$

Fig. 1. According to $\rho(u)$ and $\gamma(v)$, reorder input data matrix A such that the resulting sub-matrices (i.e., co-clusters) in B are correlated.

2.1 Alternate Minimization Based Co-Clustering

Co-clustering is also known as bi-clustering, block clustering or direct clustering [12]. Formally, given a $m \times n$ matrix Z, a *co-clustering* can be defined by two maps ρ and γ, which groups rows and columns of Z into k and l disjoint or hard clusters respectively. In particular, $\rho(u) = p$ $(1 \le p \le k)$ means that row u is in row cluster p, and $\gamma(v) = q$ $(1 \le q \le l)$ indicates that column v is in column cluster q. If we reorder rows and columns of Z and let rows and columns of the same cluster be close to each other, we obtain $k \times l$ correlated sub-matrices. Each sub-matrix is referred to as a *co-cluster*. For example, as shown in Fig. 1, given 4×4 matrix A, after reordering according to $\rho(u)$ and $\gamma(v)$, we get matrix B which contains four correlated sub-matrices (i.e., co-clusters).

The co-clustering problem can be viewed as a lossy data compression problem [9]. Given a specified number of row and column clusters, it attempts to retain as much information as possible about the original data matrix in terms of statistics based on the co-clustering. Let $\tilde{Z}$ be an approximation of the original data matrix Z. The goodness of the underlying co-clustering can be quantified by the expected distortion between Z and $\tilde{Z}$, which is shown as follows.

$$\begin{aligned}E[d_\phi(Z,\tilde{Z})] &= \sum_{u=1}^{m}\sum_{v=1}^{n} w_{uv} d_\phi(z_{uv}, \tilde{z_{uv}}) \\ &= \sum_{p=1}^{k}\sum_{\{u|\rho(u)=p\}}\sum_{q=1}^{l}\sum_{\{v|\gamma(v)=q\}} w_{uv} d_\phi(z_{uv}, s_{pq}),\end{aligned} \tag{1}$$

where d_ϕ is a distance measure and can be any member of the Bregman divergence family [13] (e.g., Euclidean distance), z_{uv} and $\tilde{z_{uv}}$ are the elements of Z and $\tilde{Z}$ respectively, w_{uv} denotes the pre-specified weight of pair (u, v), s_{pq} is the statistic derived from the co-cluster (p, q) of Z, and is used to approximate the element in the co-cluster (p, q) of Z. s_{pq} is considered as *the cluster information of data co-clustering*. The co-clustering problem is then to find (ρ, γ) such that equation (1) is minimized.

To find the optimal (ρ, γ), a broadly applicable approach is to leverage an iterative process, which monotonically decreases the objective function above by intertwining both row and column clustering iterations. Such kind of co-clustering approach is referred to as *alternate minimization*

co-clustering [9], which is considered as our main focus in this paper.

Typically, AMCC algorithms repeat the following four steps till convergence.

Step I: Keep γ fixed, for every row u, find its new row cluster assignment by the following equation:

$$\rho(u) = argmin_p \sum_{q=1}^{l} \sum_{\{v|\gamma(v)=q\}} w_{uv} d_\phi(z_{uv}, s_{pq}). \quad (2)$$

Step II: With respect to (ρ, γ), update the cluster information (i.e., the statistic of each co-cluster) by the following equation:

$$s_{pq} = argmin_{s_{pq}} \sum_{\{u|\rho(u)=p\}} \sum_{\{v|\gamma(v)=q\}} w_{uv} d_\phi(z_{uv}, s_{pq}). \quad (3)$$

Step III: Keep ρ fixed, for every column v, find its new column cluster assignment by the following equation:

$$\gamma(v) = argmin_q \sum_{p=1}^{k} \sum_{\{u|\rho(u)=p\}} w_{uv} d_\phi(z_{uv}, s_{pq}). \quad (4)$$

Step IV: The same as **Step II**.

In the above general algorithm, some implementations might combine Step II and Step IV into one step.

2.2 Sequential Updates

Many existing studies [4], [5], [6], [7], [8] have provided strong empirical evidence that expectation-maximization algorithms (e.g., k-means algorithm) with sequential updates can significantly reduce the computational cost without degrading the resulting solution. Since AMCC algorithms are variants of EM algorithms, we introduce sequential updates for AMCC algorithms. Unlike concurrent updates that perform the cluster information update after all rows (or columns) have updated their cluster assignments, sequential updates perform the cluster information update after each change in cluster assignments.

Specifically, AMCC algorithms with sequential updates repeat the following six steps till convergence.

Step I: Keep γ fixed, pick a row u in some order, find its new row cluster assignment by Eq. (2).

Step II: With respect to (ρ, γ), update the involved statistics of co-clusters by Eq. (3) once u changes its row cluster assignment.

Step III: Repeat **Step I** and **Step II** until all rows have been processed.

Step IV: Keep ρ fixed, pick a column v in some order, find its new column cluster assignment by Eq. (4).

Step V: With respect to (ρ, γ), update the involved statistics of co-clusters by Eq. (3) once v changes its column cluster assignment.

Step VI: Repeat **Step IV** and **Step V** until all columns have been processed.

The following theorem shows that sequential updates maintain the convergence properties of AMCC algorithms.

Theorem 1. *Alternate minimization based co-clustering algorithms with sequential updates monotonically decrease the objective function given by Eq. (1).*

Proof. The overall approximation error $C(Z, \tilde{Z})$ in Eq. (1) can be rewritten as the sum over the approximation errors due to each row and its assignment, i.e., $C(Z, \tilde{Z}) = \sum_{u=1}^{m} C_r(u, \rho(u))$, where $C_r(u, \rho(u))$ is the approximation error due to row u and its assignment $\rho(u)$, and $C_r(u, \rho(u)) = \sum_{q=1}^{l} \sum_{\{v|\gamma(v)=q\}} w_{uv} d_\phi(z_{uv}, s_{pq})$. Since performing row clustering for a given row u by Eq. (2) will not increase $C_r(u, \rho(u))$, $C(Z, \tilde{Z})$ will not be increased in Step1. In addition, $C(Z, \tilde{Z})$ can be also rewritten as the sum over the approximation errors due to the statistic of each co-cluster, i.e., $C(Z, \tilde{Z}) = \sum_{p=1}^{k} \sum_{q=1}^{l} C_s(s_{pq})$, where $C_s(s_{pq})$ is the approximation error due to the statistic of co-cluster (p, q), and $C_s(s_{pq}) = \sum_{\{u|\rho(u)=p\}} \sum_{\{v|\gamma(v)=q\}} w_{uv} d_\phi(z_{uv}, s_{pq})$. Since updating the statistic of each involved co-cluster by Eq. (3) will not increase $C_s(s_{pq})$, $C(Z, \tilde{Z})$ will not be increased in Step 2. Therefore, alternate minimization based co-clustering algorithms with sequential updates monotonically decrease the objective function given by Eq. (1) during the row clustering phase. For the column clustering phase, the proof is similar and is omitted for brevity. □

We observe that, in hard clustering, a row (or column) re-assignment only gives rise to the update of the cluster information related to the reassigned row (or column). Therefore, for AMCC algorithms with sequential updates, after a row (or column) re-assignment occurs, we only need to update the related cluster information instead of recalculating all the cluster information. In this way, we can significantly reduce the overhead of sequential updates. In fact, the ideal of sequential updates can also be adopted by soft clustering algorithms. However, in soft clustering, a row (or column) re-assignment requires the update of all the cluster information, which might cause large computational overhead. Moreover, since the cluster information in AMCC algorithms is the statistics derived from co-clusters, we can incrementally update the related statistics by adding or subtracting the effects of the reassigned row (or column), and further reduce the computational overhead of sequential updates.

In the following section, we will use two concrete examples to show the details of sequential updates and how to incrementally update the cluster information.

2.3 Examples: FNMTF and ITCC

Many AMCC algorithms with different Bregman divergences (e.g., [3], [10], [11], [14]) have been proposed in recent years. However, all of these algorithms are proposed with concurrent updates. To show how sequential updates work and how to incrementally update the cluster information, we take two AMCC algorithms, FNMTF [10] and ITCC [11], as examples. One considers the Euclidean distance as the distance measure, and the other considers the $Kullback-Leibler$ divergence as the distance measure.

2.3.1 FNMTF

FNMTF considers the Euclidean distance as the distance measure between the original matrix and the approximation matrix. Specifically, FNMTF constrains the factor matrices **F** and **G** to be cluster indicator matrices and tries to minimize the following objective function:

$$||Z - FSG^T||^2 \quad s.t. \quad F \in \Psi^{m\times k}, G \in \Psi^{n\times l}, \tag{5}$$

where **Z** is the input data matrix, and matrix **S** is composed of the statistics given on co-clusters determined by **F** and **G**.

FNMTF alternately solves the three variables **F**, **G** and **S** in Eq. (5), and iterates till convergence. In each iteration, it contains the following three steps.

First, fixing **G** and **S**, for each row $z_{u\cdot}$ in **Z**, find its new cluster assignment using the following equation:

$$f_{up} = \begin{cases} 1, & argmin_p ||z_{u\cdot} - \tilde{\eta_{p\cdot}}||^2, \\ 0, & otherwise, \end{cases} \tag{6}$$

where $\tilde{\eta_{p\cdot}}$ is the p-th row of SG^T and serves as the *row-cluster prototype* which is similar to the "centroid" in k-means clustering.

Second, fixing **F** and **S**, for each column $z_{\cdot v}$ in **Z**, find its new cluster assignment using the following equation:

$$g_{vq} = \begin{cases} 1, & argmin_q ||z_{\cdot v} - \tilde{\mu_{\cdot q}}||^2, \\ 0, & otherwise, \end{cases} \tag{7}$$

where $\tilde{\mu_{\cdot q}}$ is the q-th column of FS and serves as the *column-cluster prototype*.

In the last step of this iteration, according to the current co-clustering determined by **F** and **G**, it updates each element of **S** by the following equation:

$$s_{pq} = \frac{\sum_{\{u|\rho(u)=p\}} \sum_{\{v|\gamma(v)=q\}} z_{uv}}{|p| \cdot |q|}, \tag{8}$$

where $|p|$ denotes the number of rows in row cluster p and $|q|$ denotes the number of columns in column cluster q. s_{pq} provides the statistic of co-cluster (p, q), which is the mean of all elements in co-cluster (p, q).

The procedures of FNMTF with concurrent updates are summarized in Algorithm 1.

Algorithm 1. FNMTF with Concurrent Updates

```
1   Initialize G ∈ Ψ^{n×l} and F ∈ Ψ^{m×k} with arbitrary cluster
    indicator matrices;
2   repeat
3       Fixing F and G, update S by (8);
4       Fixing F and S, update G by (7);
5       Fixing G and S, update F by (6);
6   until converges;
```

Unlike FNMTF with concurrent updates, FNMTF with sequential updates requires the most up-to-date **S** to perform row (or column) cluster assignments and thus **S** should be updated immediately once a row (or column) changes its cluster assignment.

Since a row (or column) re-assignment involves only two rows (or columns) of **S** and the elements of **S** are the means of co-clusters, we can update **S** incrementally and thus reduce the overhead of frequent updates of **S**. Algorithm 2 shows the incremental update for **S** after a particular row $z_{u\cdot}$ changes its row cluster assignment from p to $\hat{p}$. The incremental update for **S** after reassigning a column can be done in a similar way and is omitted for brevity.

Algorithm 2. Incremental Update for **S**

1 **for** $q \leftarrow 1$ **to** l **do**

2 $\quad s_{pq} = \frac{s_{pq}\cdot|p|\cdot|q| - \sum_{\{u|\rho(u)=p\}} \sum_{\{v|\gamma(v)=q\}} z_{uv}}{(|p|-1)\cdot|q|}$;

3 $\quad s_{\hat{p}q} = \frac{s_{\hat{p}q}\cdot|\hat{p}|\cdot|q| + \sum_{\{u|\rho(u)=p\}} \sum_{\{v|\gamma(v)=q\}} z_{uv}}{(|\hat{p}|+1)\cdot|q|}$;

4 $|p| = |p| - 1$;

5 $|\hat{p}| = |\hat{p}| + 1$;

The procedures of FNMTF with sequential updates are summarized in Algorithm 3.

Algorithm 3. FNMTF with Sequential Updates

```
1   Initialize G ∈ Ψ^{n×l} and F ∈ Ψ^{m×k} with arbitrary cluster
    indicator matrices;
2   Initialize S by (8);
3   repeat
4       Fixing G;
5       foreach z_{u·} ∈ Z do
6           update F by (6);
7           update S with Algorithm (2);
8       end
9       Fixing F;
10      foreach z_{·v} ∈ Z do
11          update G by (7);
12            update S in an incremental way (similar to
              Algorithm (2));
13      end
14  until converges;
```

2.3.2 ITCC

ITCC considers a non-negative matrix as the joint probability distribution of two discrete random variables and formulates the co-clustering problem as an optimization problem in information theory. Formally, let X and Y be discrete random variables taking values in $\{x_i\}_{i=1}^m$ and $\{y_i\}_{j=1}^n$ respectively. Suppose we want to cluster X into k disjoint clusters $\{\hat{x}_i\}_{i=1}^k$ and Y into l disjoint clusters $\{\hat{y}_j\}_{j=1}^l$ simultaneously. ITCC tries to find a co-clustering, which minimizes the loss in mutual information:

$$I(X;Y) - I(\hat{X};\hat{Y}) = D(p(X,Y)||q(X,Y)), \tag{9}$$

where $I(X;Y)$ is the mutual information between X and Y, $D(\cdot||\cdot)$ denotes the $Kullback - Leibler$ divergence, $p(X,Y)$ denotes the joint probability distribution between X and Y, and $q(X,Y)$ is a distribution of the form

$$q(x,y) = p(\hat{x},\hat{y})p(x|\hat{x})p(y|\hat{y}), \quad x \in \hat{x}, \quad y \in \hat{y}, \tag{10}$$

where $p(\hat{x},\hat{y}) = \sum_{x\in\hat{x}} \sum_{y\in\hat{y}} p(x,y)$, and $p(x|\hat{x}) = \frac{p(x)}{p(\hat{x})}$ for $\hat{x} = C_X(x)$ and 0 otherwise, and similarly for $p(y|\hat{y})$. $p(\hat{x},\hat{y})$

provides the statistic of co-cluster $(\hat{x}, \hat{y})$, which is the sum of all the elements in co-cluster $(\hat{x}, \hat{y})$. $p(X, Y)$ and $q(X, Y)$ are both $m \times n$ matrices, $p(\hat{X}, \hat{Y})$ is a $k \times l$ matrix, $p(X|\hat{X})$ and $p(Y|\hat{Y})$ are $m \times k$ and $n \times l$ column orthogonal matrices respectively.

ITCC starts with an initial co-clustering (C_X, C_Y) and refines it through an iterative row and column clustering process. According to (C_X, C_Y), ITCC computes the following three distributions:

$$q(\hat{X}, \hat{Y}), q(X|\hat{X}), q(Y|\hat{Y}), \tag{11}$$

where $q(\hat{X}, \hat{Y}) = p(\hat{X}, \hat{Y})$, $q(X|\hat{X}) = p(X|\hat{X})$, and $q(Y|\hat{Y}) = p(Y|\hat{Y})$.

For row clustering, fixing C_Y, it tries to find new cluster assignment for each row x as

$$C_X(x) = argmin_{\hat{x}} D(p(Y|x)||q(Y|\hat{x})), \tag{12}$$

where $q(y|\hat{x}) = p(y|\hat{y})p(\hat{y}|\hat{x})$ and serves as the *row-cluster prototype*. Recall that $p(Y|\hat{Y})$ is a $n \times l$ column orthogonal matrix. It implies that given $p(Y|\hat{Y})$ and a row of $p(\hat{X}, \hat{Y})$, we can get the corresponding row-cluster prototype without matrix multiplication. Then, according to the new row cluster assignment, it updates the distributions in (11).

For column clustering, fixing C_X, it tries to find new cluster assignment for each column y as

$$C_Y(y) = argmin_{\hat{y}} D(p(X|y)||q(X|\hat{y})), \tag{13}$$

where $q(x|\hat{y}) = p(x|\hat{x})p(\hat{x}|\hat{y})$ and serves as the *column-cluster prototype*. Then, according to the new column cluster assignment, it updates the distributions in (11).

The procedures of ITCC with concurrent updates are summarized in Algorithm 4.

Algorithm 4. ITCC with Concurrent Updates

1 Initialize C_X and C_Y with arbitrary cluster indicator matrices;
2 Initialize distributions in (11);
3 **repeat**
4 Fixing C_Y, update C_X by Eq. (12);
5 update $q(\hat{X}, \hat{Y})$ and $q(X|\hat{X})$;
6 Fix C_X, update C_Y by Eq. (13);
7 update $q(\hat{X}, \hat{Y})$ and $q(Y|\hat{Y})$;
8 **until** *converges*;

To sequentialize ITCC, we should update the distributions in (11) once a row (or column) of $p(X, Y)$ changes its cluster assignment. Since $q(\hat{x}, \hat{y})$ is the sum of all the elements in the corresponding co-cluster, if a row (or column) changes its cluster assignment, only two rows (or columns) of $q(\hat{X}, \hat{Y})$ need to be updated. Thus, we can update $q(\hat{X}, \hat{Y})$ incrementally. Moreover, for distribution $q(X|\hat{X})$, according to Eq. (10) and Eq. (11), it can be also obtained in an incremental manner. Since $q(X)$ is constant, we only perform incremental update on $q(\hat{X})$ and calculate $q(X|\hat{X})$ whenever we need. Algorithm 5 shows how to incrementally update $q(\hat{X}, \hat{Y})$ and $q(\hat{X})$ after a particular row x changes its column cluster assignment from $\hat{x}$ to $\hat{x^*}$. The incremental update for

TABLE 1
Description of Data Sets

Data sets	samples	features	non-zeros
Coil20	1,440	1,024	967,507
NG20	4,000	1,000	266,152

$q(\hat{X}, \hat{Y})$ and $q(\hat{Y})$ after reassigning a column can be done in a similar way and is omitted for brevity.

Algorithm 5. Incremental Update for $q(\hat{X}, \hat{Y})$ and $q(\hat{X})$

1 **for** $\hat{y} \leftarrow 1$ **to** l **do**
2 $q(\hat{x}, \hat{y}) = q(\hat{x}, \hat{y}) - \sum_{i \in \hat{x}, j \in \hat{y}} x_{ij}$;
3 $q(\hat{x^*}, \hat{y}) = q(\hat{x^*}, \hat{y}) + \sum_{i \in \hat{x}, j \in \hat{y}} x_{ij}$;
4 $q(\hat{x}) = q(\hat{x}) - q(x)$;
5 $q(\hat{x^*}) = q(\hat{x^*}) + q(x)$;

The procedures of ITCC with sequential updates are summarized in Algorithm 6.

Algorithm 6. ITCC with Sequential Updates

1 Initialize C_X and C_X with arbitrary cluster indicator matrices;
2 Initialize the distributions in (11);
3 **repeat**
4 Fix C_Y;
5 **foreach** $x \in X$ **do**
6 update C_X by Eq. (12);
7 update $q(\hat{X}, \hat{Y})$ and $q(X|\hat{X})$ with Algorithm (5);
8 Fix C_X;
9 **foreach** $y \in Y$ **do**
10 update C_Y by Eq. (13);
11 update $q(\hat{X}, \hat{Y})$and $q(Y|\hat{Y})$ in an incremental way (similar to Algorithm (5));
12 **until** *converges*;

2.4 Concurrent versus Sequential

To show the benefits of sequential updates for AMCC algorithms, we implement FNMTF and ITCC with different update strategies in Java. Two real-world data sets Coil20 [15] and 20-Newsgroup [16] are used to evaluate the co-clustering algorithms. For the 20-Newsgroup data set, we selected the top 1,000 words by mutual information. The details of the data sets are summarized in Table 1. The number of row (or column) clusters is set to 20. For the same co-clustering algorithm, we report the minimum objective function value it can obtain and the corresponding running time. The clustering results are also evaluated by two widely used metrics, i.e., clustering accuracy and normalized mutual information (NMI).

The experiments are performed on a single machine, which has Intel Core 2 Duo E8200 2.66 GHz processor, 3 GB of RAM, 1 TB hard disk, and runs 32-bit Linux Debian 6.0 OS. For a fair comparison, all of these algorithms use the same cluster assignments for initialization. Moreover, we run each algorithm 10 times, and use a different cluster initialization for each time. The average results of these algorithms are reported.

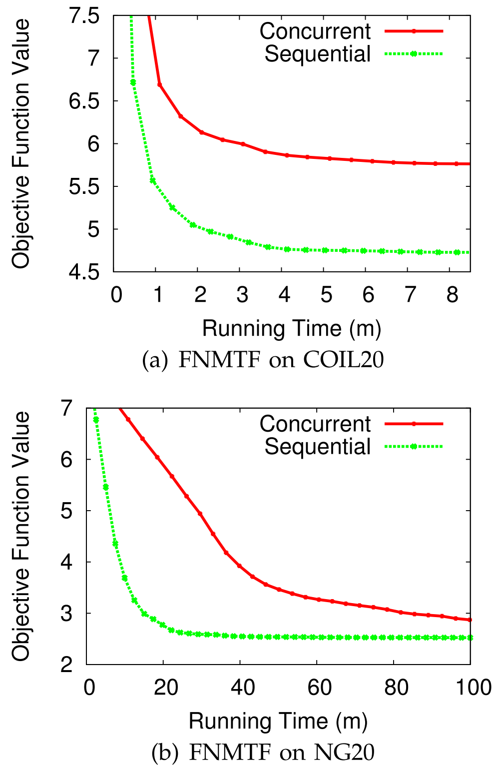


Fig. 2. Compare convergence speed and quality between concurrent updates and sequential updates.

As shown in Figs. 2 and 3, both of these two AMCC algorithms with sequential updates converge faster and get better results than their concurrent counterparts. It is

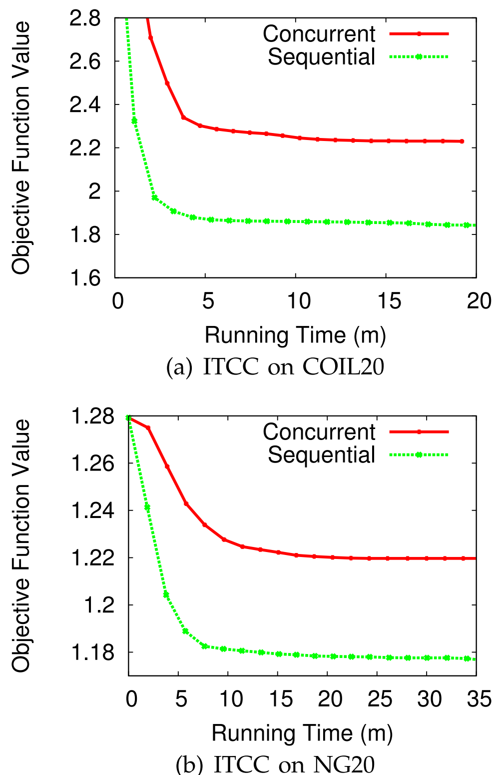


Fig. 3. Compare convergence speed and quality between concurrent updates and sequential updates.

TABLE 2
Clustering Results of FNMTF Measured by Accuracy and NMI

Data sets	Metrics	Concurrent	Sequential
Coil20	Accuracy	0.685	0.721
	NMI	0.573	0.612
NG20	Accuracy	0.475	0.485
	NMI	0.481	0.490

not surprising that the AMCC algorithms with sequential updates also outperform their concurrent counterparts in terms of clustering accuracy and NMI as shown in Tables 2 and 3. Such encouraging results motivate us to provide parallel solutions for AMCC algorithms with sequential updates.

3 Parallelizing Alternate Minimization Co-Clustering Algorithms with Sequential Updates

Parallelizing alternate minimization co-clustering algorithms with sequential updates is challenging. In this section, we propose two approaches to solve this challenge.

3.1 Dividing Clusters Approach

Suppose in a distributed environment which consists of a number of worker machines, each worker independently performs sequential updates during the iterative process. The statistics of co-clusters (S_{cc}) should be updated whenever a row (or column) changes its cluster assignment. However, since the workers run concurrently, it may result in inconsistent S_{cc} across workers. Thus, the convergence properties of co-clustering algorithms cannot be maintained. Therefore, we a propose dividing clusters approach to solve this problem.

The details of the dividing clusters approach are described as follows. Suppose we want to group the input data matrix into k row clusters and l column clusters, the number of workers is p ($p \leq min\{\frac{k}{2}, \frac{l}{2}\}$), and each worker w_i holds a subset of rows R_i and a subset of columns C_i. When performing row clustering, we randomly divide row clusters S^r into p non-overlapping row subsets $S^r_1, S^r_2, \ldots, S^r_p$, and guarantee that each subset contains at least two clusters. These subsets are distributed to each worker in a one-to-one manner. When worker w_i receives S^r_i, it can perform row clustering with sequential updates for its rows R_i among the subset of row clusters S^r_i. For example, assume that S^r_i is $\{1, 3, 6\}$, w_i will perform row clustering for its rows whose current cluster assignments are in S^r_i, and allow these rows to change their

TABLE 3
Clustering Results of ITCC Measured by Accuracy and NMI

Data sets	Metrics	Concurrent	Sequential
Coil20	Accuracy	0.681	0.696
	NMI	0.564	0.574
NG20	Accuracy	0.478	0.483
	NMI	0.489	0.495

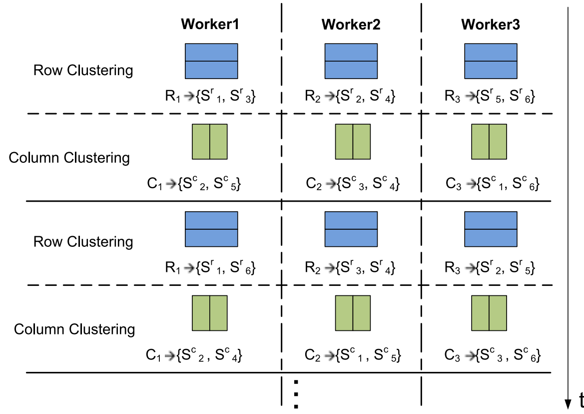


Fig. 4. Flow of dividing clusters approach.

cluster assignments among row clusters 1, 3, and 6. Since w_i updates only a non-overlapping subset of S_{cc}, the sequential updates on worker w_i will never affect the updates on other workers. The subsets of S_{cc} and cluster indicators updated by each worker will be combined and synchronized over iterations. Here we have illustrated how to perform row clustering. Column clustering can be done in a similar way.

An example for the dividing clusters approach is shown in Fig. 4. There are three workers in the distributed environment. Each worker holds a subset of rows R_i and a subset of columns C_i, and performs sequential updates on a non-overlapping subset of row (or column) clusters in each iteration. The mapping between workers and cluster subsets changes over iterations. Row and column clustering iterations are performed alternatively until convergence.

The following lemma and theorem guarantee the convergence properties of AMCC algorithms parallelized by the dividing clusters approach.

Lemma 2. *For alternate minimization based co-clustering algorithms, performing sequential updates for its row (or column) clustering iteration on any subset of row (or column) clusters monotonically decreases the objective function given by Eq. (1).*

Proof. On any given subset of row clusters, performing row clustering for a given row u by Eq. (2) in Step 1 will still not increase the approximation due to u and $\rho(u)$, i.e., $C_r(u, \rho(u))$. Since the overall approximation error $C(Z, \tilde{Z})$ in Eq. (1) is the sum over the approximation errors due to each row and its assignment, $C(Z, \tilde{Z})$ will not be increased. In addition, while keeping the co-clustering (ρ, γ) fixed, performing the cluster information update by Eq. (3) in Step 2 will never increase $C(Z, \tilde{Z})$. Therefore, performing row clustering with sequential updates on any subset of row clusters monotonically decreases the objective function given by Eq. (1). For the column clustering case, the proof is similar and is omitted for brevity. □

Based on Lemma 2, we can easily derive the following theorem.

Theorem 3. *Dividing clusters approach maintains the convergence properties of alternate minimization based co-clustering algorithms.*

Discussion. Since each worker performs row (or column) clustering only on a subset of clusters in each iteration, sequential updates for row (or column) clustering should be executed in a series of iterations before switching to the other side of clustering. Intuitively, to ensure the subset of rows (or columns) of the input data matrix held on each worker has sufficient opportunity to move to any row (or column) cluster, the number of such iterations should not be less than the number of non-overlapping subsets of clusters. We will further discuss the setting for this number in Section 5.3.

3.2 Batching Points Approach

The dividing clusters approach eliminates the dependency on the cluster information for each worker and enables co-clustering with sequential updates in a parallel and distributed manner. However, it assumes that the number of workers is less than the number of clusters. Such assumption might restrict the scalability of this approach. For example, when the number of workers is larger than the number of clusters, this approach cannot utilize the extra workers to perform data clustering. Therefore, by relaxing the stringent constraint of sequential updates, we introduce *batch updates* for AMCC algorithms. The difference between batch and sequential updates is that batch updates perform the cluster information update after a batch of rows (or columns) have updated their cluster assignments, rather than after each change in cluster assignments. The following lemma shows that batch updates maintain the convergence properties of AMCC algorithms.

Lemma 4. *Alternate minimization based co-clustering algorithms with batch updates monotonically decrease the objective function given by Eq. (1).*

Proof. The proof is similar to that of Theorem 1 and is omitted for brevity. □

Obviously, sequential or concurrent updates are the extreme case of batch updates. If the number of rows (or columns) in a batch is one, it is equivalent to sequential updates; if the number of rows (or columns) is the number of input rows (or columns), it is equivalent to concurrent updates. Batch updates provide a useful knob for tuning the update frequency for AMCC algorithms.

Compared to concurrent updates, batch updates update the cluster information more frequently, which implies the advantages of sequential updates could be preserved. In addition, if the number of rows (or columns) in a batch is small and S_{cc} can be updated incrementally, we can still leverage incremental computation to achieve computational efficiency. Specifically, after processing a batch of rows (or columns) concurrently, S_{cc} could be incrementally updated according to the re-assignments in this batch. Therefore, the computational overhead of frequent updates of S_{cc} could be reduced.

We propose a batching points approach to parallel batch updates. The details of this approach are as follows. Suppose each worker w_i holds a subset of rows R_i and a subset of columns C_i. When performing row clustering, we randomly divide R_i into p non-overlapping subsets

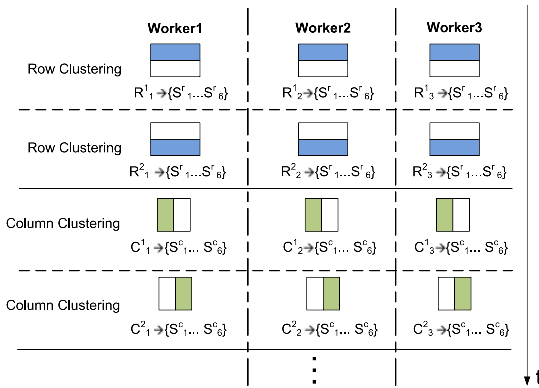


Fig. 5. Flow of batching points approach.

$R_i^1, R_i^2, \ldots, R_i^p$. We refer to R_i^j as a *batch*. Each worker processes only one of its batches with concurrent updates in each iteration. A synchronization process for the cluster information update is initiated at the end of each iteration. Such iteration continues until all batches have been processed, then it switches to column clustering which is performed in a similar way. During the synchronization process, each worker computes the statistics of co-clusters on R_i (or C_i). We refer to such statistics obtained by each worker as one *slice* of S_{cc}. All of the slices will be combined to obtain a new S_{cc} used for the next iteration.

Fig. 5 presents an example for the batching points approach. There are three workers in the distributed environment. Each worker holds a subset of rows R_i and a subset of columns C_i, and further divides R_i and C_i into two batches respectively. For a row clustering iteration, each worker selects one of its row batches to perform co-clustering with concurrent updates. The row clustering iteration continues two times, then the column clustering iteration begins. Such row and column clustering iterations are performed until convergence.

The following theorem guarantees the convergence properties of AMCC algorithms parallelized by the batching points approach, which can be easily derived by Lemma 4.

Theorem 5. *Batching points approach maintains the convergence properties of alternate minimization based co-clustering algorithms.*

Discussion. The potential performance gain of the batching points approach is not free, since performing batch updates requires more synchronization than concurrent updates. In other words, there is a trade-off between communication overhead and computational efficiency in this approach. To find the optimal number of batches num_{batch} ($num_{batch} \geq 1$) that offers the best trade-off, we need to quantify both performance gain and performance overhead for the batching points approach. We define *one-pass processing* as follows. All rows and columns of the input data matrix are processed by a given co-clustering algorithm in one pass.

Specifically, for concurrent updates, one-pass processing consists of a row clustering iteration and a column clustering iteration. For batch updates, it consists of num_{batch} row clustering iterations and num_{batch} column clustering iterations. Suppose t_{bat} and t_{con} are the running time of one-pass processing with batch updates and concurrent updates respectively. $\Delta cost_{bat}$ and $\Delta cost_{con}$ are the corresponding decreases of the objective function respectively. We consider the ratio of $\Delta cost_{bat}$ to $\Delta cost_{con}$ as the performance gain of batch updates, and consider the ratio of t_{bat} to t_{con} as its performance overhead. Typically, we expect the performance gain of batch updates should be less than its performance overhead, and thus num_{batch} should satisfy the following inequality:

$$\frac{t_{bat}}{t_{con}} = \frac{t_{comp} + 2 \cdot num_{batch} \cdot t_{sync}}{t_{comp} + 2 \cdot t_{sync}} \leq \frac{\Delta cost_{bat}}{\Delta cost_{con}}, \tag{14}$$

where t_{comp} is the computational time for one-pass processing, and t_{sync} is the time consumption of synchronization among workers for one iteration. Obviously, both the performance gain and the performance overhead are functions of num_{batch} and they are specific to the data set and the experimental environment. In practice, it is not realistic to try all possible numbers to find the optimal number of batches which can maximize the performance (i.e., maximize the ratio of performance gain to performance overhead). In Section 5.3, we will provide a practical method to determine this number.

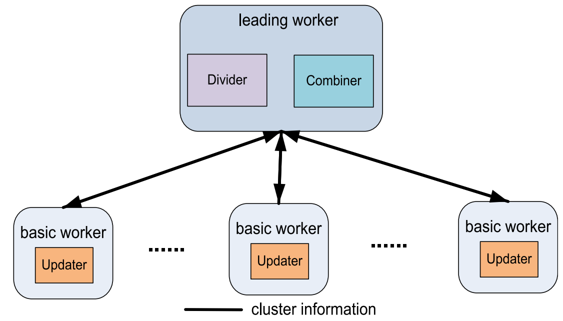


Fig. 6. The architecture of the co-clusterd framework.

4 Co-ClusterD Framework

In this section, we first show the design and implementation of our Co-ClusterD framework for alternate minimization co-clustering algorithms with sequential updates. Then we give the API sets that the framework provides. At last, we describe the fault tolerance and load balance mechanisms adopted by Co-ClusterD.

4.1 Design and Implementation

Based on the proposed approaches for parallelizing sequential updates in the previous section, we design and implement Co-ClusterD, a distributed framework for AMCC algorithms with sequential updates. The architecture of the Co-ClusterD framework is shown in Fig. 6.

Co-ClusterD consists of a number of basic workers and a leading worker. Each basic worker independently performs row and column clustering by its updater component. As shown in Fig. 7, it maintains five key-value pair tables in its memory to store the following variables: a non-overlapping subset of rows of the input data matrix R_i, a non-overlapping subset of columns of the input data matrix C_i, row-cluster indicators of the input data matrix I_r, column-cluster

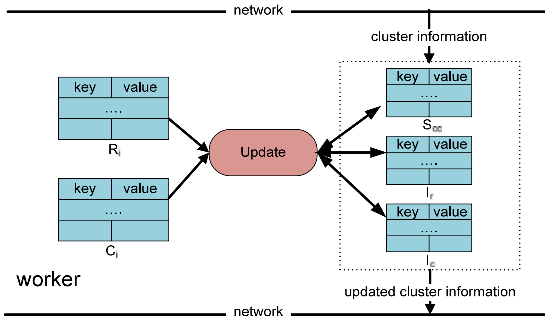


Fig. 7. Basic worker.

indicators of the input data matrix I_c, and the statistics of co-clusters S_{cc}. Specifically, the key field of the table for storing R_i (or C_i) is the row (or column) identifier, and the value field is the corresponding elements; the key field of the table for storing I_r (or I_c) is the row (or column) identifier, and the value field is the cluster identifier the row (or column) belongs to; the key field of the table for storing S_{cc} is the co-cluster identifier, and the value field is the statistic of the co-cluster. R_i and C_i are constant values during the data co-clustering process. At the beginning of each row (or column) clustering iteration, each basic worker receives S_{cc} and I_r (or I_c) from the leading worker, and then performs row (or column) clustering. The updated subsets of S_{cc} used in the dividing clusters approach (or the slices of S_{cc} used in the batching points approach) and cluster indicators are sent to the leading worker by basic workers over each clustering iteration.

In Co-ClusterD, the leading worker plays a coordination role during the data co-clustering process. It uses its divider and combiner components to perform dividing clusters, and combining non-overlapping subsets of S_{cc} or slices of S_{cc}. In addition, it is also responsible for synchronizing cluster assignments and the cluster information among workers. In particular, after the leading work receives the subsets (or slices) of S_{cc} and cluster indicators from all the workers over each clustering iteration, it reconstructs S_{cc} and I_r (or I_c), and then broadcasts them to the basic workers.

For a given co-clustering job, Co-ClusterD proceeds in two stages: cluster information initialization and data co-clustering. In the cluster information initialization stage, assuming there are w workers in the distributed environment, Co-ClusterD first partitions the input data matrix into w row and w column subsets. Next, each worker loads one row subset, one column subset, and *randomly* initializes the cluster assignments for the rows and columns it holds. Then, each worker calculates its slice of S_{cc} and sends it to the leading worker. Finally, the leading worker combines all slices of S_{cc} and thus the initial S_{cc} is obtained. In the data co-clustering stage, Co-ClusterD works on the co-clustering algorithm implemented by users. The algorithm can be easily implemented by overriding a number of APIs provided by Co-ClusterD (see Section 4.2 for details). It alternatively performs row and column clustering until the number of iterations exceeds a user-defined threshold. In particular, for the dividing clusters approach, users can specify the number of iterations repeated for row (or column) clustering before switching to the other side of clustering. For the batching points approach, users can specify the number of row (or column) batches each work holds.

Co-ClusterD is implemented based on iMapreduce [17], which is a distributed framework based on Hadoop and has built-in support for iterative algorithms (see [17] for details). In fact, Co-ClusterD is independent of the underlying distributed frameworks. It can be also implemented on other distributed frameworks (e.g., Spark [18], and MPI). We choose iMapreduce since it can better support the iterative processes of co-clustering algorithms.

Notice that, in the Co-ClusterD framework, row and column subsets held by each worker will not be shuffled and only the updated cluster information S_{cc} and cluster assignments I_r (or I_c) are synchronized among workers over iterations through the network. Since S_{cc} ($k \times l$ matrix, $k \ll m$ and $l \ll n$), I_r ($m \times 1$ matrix) and I_c ($n \times 1$ matrix) are very small, the communication overhead of the Co-ClusterD framework is small.

4.2 API

Co-ClusterD allows users without much knowledge on distributed computing to write distributed AMCC algorithms. Users need to implement only a set of well defined APIs provided by Co-ClusterD. In fact, these APIs are callback functions, which will be automatically invoked by the framework during the data co-clustering process. The descriptions of these APIs are as follows.

1) `cProto genClusterProto(bRow, pointS, Ind)`: Users specify how to generate the cluster prototype. Recall that the cluster prototype plays the role of "centroid" in row (or column) clustering, and it can be constructed by the cluster indicators and the statistics of co-clusters S_{cc}. The parameter `bRow` indicates whether row clustering is performed. If `bRow` is true, `pointS` is one row of S_{cc}, and `Ind` is the column-cluster indicators of the input data matrix. Otherwise, `pointS` is one column of S_{cc}, and `Ind` is the row-cluster indicators of the input data matrix.
2) `double disMeasure(point, cProto)`: Given a `point` and a cluster prototype `cProto`, users specify a measure to quantify the distance between them. `point` denotes a row or a column of the input data matrix.
3) `Scc updateSInc(bRow, point, preCID, curCID, Scc, rInd, cInd)`: Users specify how to incrementally update S_{cc} when a `point` changes its cluster assignment from previous cluster `preCID` to current cluster `curCID`. `rInd` and `cInd` are row-cluster and column-cluster indicators of the input data matrix respectively.
4) `slice updateSliceInc(bRow, point, preCID, curCID, slice, subInd, Ind)`: Users specify how to incrementally update a `slice` of S_{cc} when a `point` changes its cluster assignment from previous cluster `preCID` to current cluster `curCID`. If `bRow` is true, `subInd` is the row-cluster indicators of the subset of rows the worker holds, and `Ind` is the column-cluster indicators of the input data matrix. Otherwise, `subInd` is the column-cluster

TABLE 4
API Sets for Different Approaches

Approach	Initialization	Data Co-clustering
Dividing Clusters	(5), (6)	(1), (2), (3)
Batching Points	(5), (6)	(1), (2), (4), (6)

indicators of the subset of columns the worker holds, and `Ind` is the row-cluster indicators of the input data matrix.

5) `slice buildOneSlice(bRow, subInd, Ind)`: Users specify how to build one slice of S_{cc}. The parameters in this function have the same meaning as the parameters in function (4).
6) `Scc combineSlices(slices, rInd, cInd)`: Users specify how to combine the slices of S_{cc} sent by workers. `slices` are all the slices of S_{cc} sent by workers. `rInd` and `cInd` are row-cluster and column-cluster indicators of the input data matrix respectively.

Notice that we do not add constant variables such as the subset of rows (or columns) to the parameter lists of these APIs. Constant variables are initialized by an initialization process and exposed as global variables.

The API sets used by the dividing clusters approach and the batching points approach are summarized in Table 4.

4.3 Fault Tolerance and Load Balance

In Co-ClusterD, fault tolerance is implemented by a global checkpoint/restore mechanism, which is performed at a user-defined time interval. The cluster information and cluster assignments in the leading worker are dumped to a reliable file system every period of time. If any worker fails (including the leading worker), the computation will roll back to the most recent iteration checkpoint and resume from that iteration. In addition, since Co-ClusterD is based on iMapreduce [17], it also inherits all the salient features of Mapreduce's style fault tolerance.

The synchronous computation model used by our Co-ClusterD framework also makes load balance become an important issue. This is because the running time of each iteration is dependent on the slowest worker. Therefore, if the capacity of each computer in the distributed environment is homogeneous, the workload should be evenly distributed. Otherwise, the workload should be distributed according to the capacity of each worker.

5 Experimental Results

In this section, we evaluate the effectiveness and efficiency of our Co-ClusterD framework in the context of two alternate minimization based co-clustering algorithms ITCC and FNMTF on several real world data sets. We compare Co-ClusterD to a state-of-the-art Hadoop based co-clustering framework DisCo [19]. Since DisCo does not support AMCC algorithms with sequential updates, we implement only concurrent updates in DisCo. However, in Co-ClusterD, both concurrent and sequential updates are implemented. The experiments are performed on both small-scale and large-scale clusters.

TABLE 5
Description of Data Sets

Data sets	samples	features	non-zeros
KOS	3,430	6,906	467,714
NIPS	1,500	12,419	1,900,000
ENRON	39,861	28,102	6,400,000

5.1 Experiment Setup

We use real world data sets downloaded from UCI Machine Learning Repository [16] to evaluate the co-clustering algorithms. These data sets are summarized in Table 5.

Since these data sets do not have class labels, the metrics, such as clustering accuracy and normalized mutual information, are not used. In fact, for a clustering algorithm, the quality of the clustering results obtained by its different update strategies can be also reflected by the minimum objective function value it can obtain. Therefore, in our experiments, we only consider the objective function value as the performance metric to measure the quality of the clustering results. We also report the running time of the clustering algorithms.

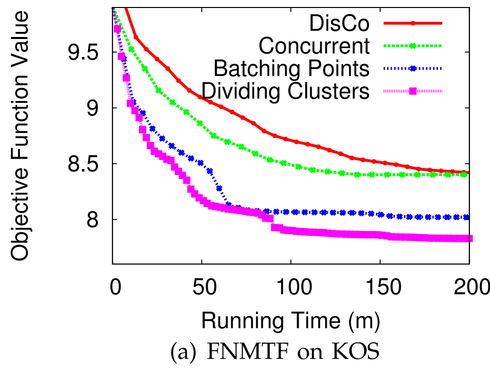
We build a small-scale cluster of local machines and a large-scale cluster on the Amazon EC2 cloud to run experiments. The small-scale cluster consists of four machines. Each machine has Intel Core 2 Duo E8200 2.66 GHz processor, 3 GB of RAM, 1 TB hard disk, and runs 32-bit Linux Debian 6.0 OS. These four machines are connected to a switch with communication bandwidth of 1 Gbps. The large-scale cluster consists of 100 High-CPU medium instances on the Amazon EC2 Cloud. Each instance has 1.7 GB memory and five EC2 compute units.

For a fair comparison, we enforced all the algorithms to be initialized by the same cluster assignments. We run the algorithms 10 times, and each time a different initialization is used. The average results of these algorithms are reported.

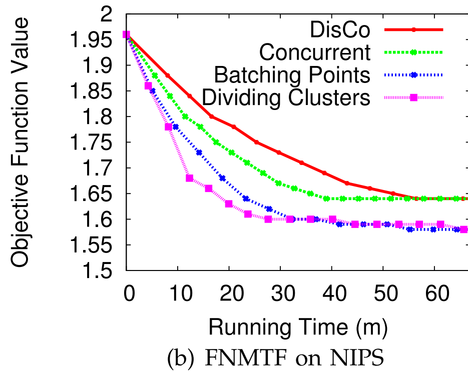
5.2 Small-Scale Experiments

For small-scale experiments, data sets KOS and NIPS are used for evaluation. The number of row (or column) clusters is set to 40. For the dividing clusters approach, each worker is randomly assigned a subset of row (column) clusters with size 10 in each iteration. In addition, the number of iterations repeated for row (or column) clustering is set to 4. For the batching points approach, each worker divides its subset of rows (or columns) into 16 batches. We will discuss how to choose these parameters in the next section.

As shown in Figs. 8 and 9, we can observe that co-clustering algorithms with concurrent updates implemented in Co-ClusterD (denoted by "Concurrent") converge faster than those implemented in DisCo (denoted by "DisCo") although they converge to the same values. This is because Co-ClusterD leverages a persistent job for the iterative process of the co-clustering algorithm rather than using one job for one row (or column) clustering iteration which is adopted by DisCo. Hence, Co-ClusterD reduces the repeated job initialization overhead in each iteration and achieves faster convergence. In addition, we can also observe that algorithms parallelized by the dividing clusters approach and the batching points approach

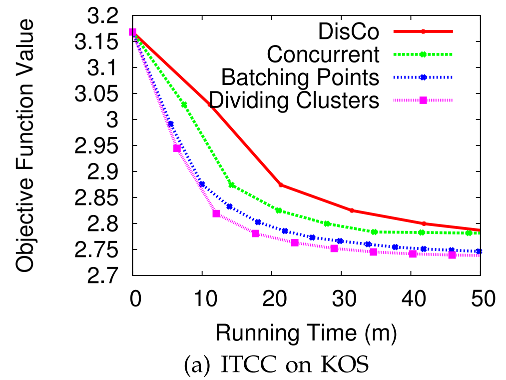


(a) FNMTF on KOS

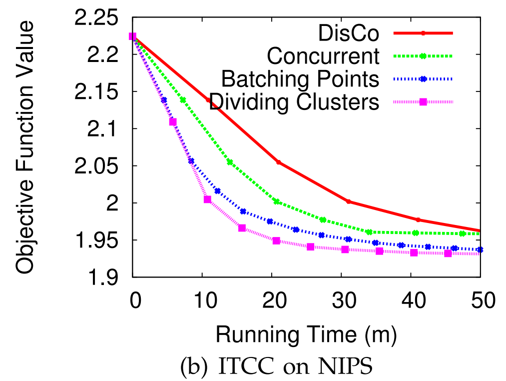


(b) FNMTF on NIPS

Fig. 8. Cost function versus running time comparisons for co-clustering algorithms with different update strategies.



(a) ITCC on KOS



(b) ITCC on NIPS

Fig. 9. Cost function versus running time comparisons for co-clustering algorithms with different update strategies.

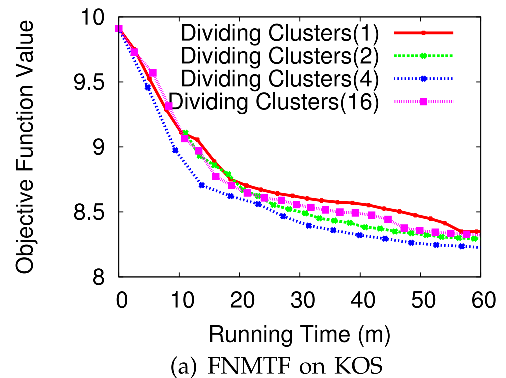
(denoted by "Dividing Clusters" and "Batching Points" respectively) converge much faster and obtain better results than their concurrent counterparts implemented in DisCo and Co-ClusterD.

5.3 Tuning Parameters

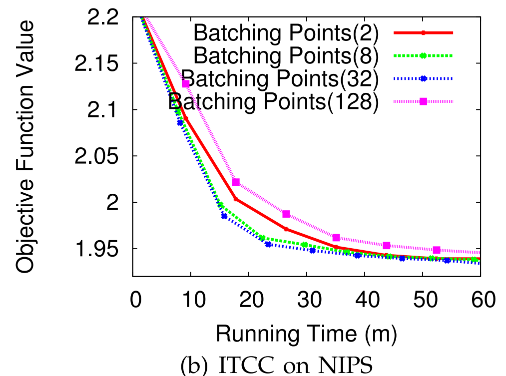
To quantify the effects of the parameters used in Co-ClusterD, we conduct the following two experiments. In the first experiment, we run algorithms using the dividing clusters approach with different number of row (or column) clustering iterations before switching to the other side of clustering. In the second experiment, we run algorithms using the batching points approach with different number of batches. The number of row (or column) clusters is set to 40. All of these experiments are performed on our small-scale local cluster. Notice that we actually did these two experiments on different data sets for different co-clustering algorithms. Due to space limitation, we show only the representative results.

Fig. 10a presents the results of the first experiment, where the number that follows the name of the approach denotes the number of row (or column) iterations it continues to perform. From this figure, we can observe that setting this number to too small or too large values results in slow convergence. According to our empirical results, setting this number to the number of workers (i.e., the number of the non-overlapping subsets of clusters) typically shows good performance for the dividing clusters approach.

Fig. 10b plots the results of the second experiment, where the number that follows the name of the approach denotes the number of row (or column) batches each worker holds. The results shown in this figure indicate that setting this number to too small or too large values results in poor performance. In addition, it also shows that there is an interval between the small number and the large number where the

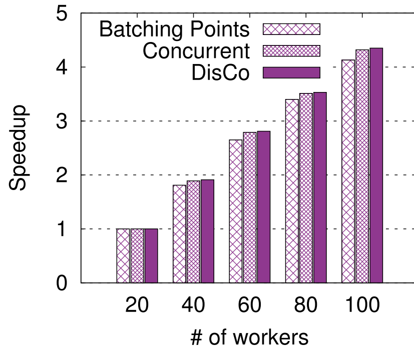


(a) FNMTF on KOS

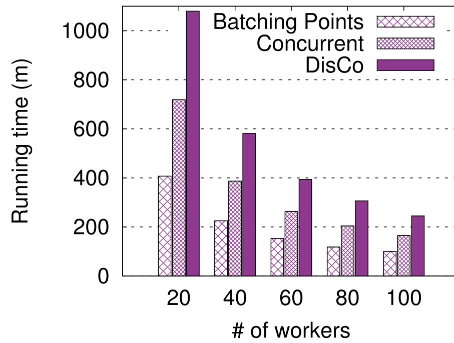


(b) ITCC on NIPS

Fig. 10. Tuning parameters for dividing clusters and batching points approaches.

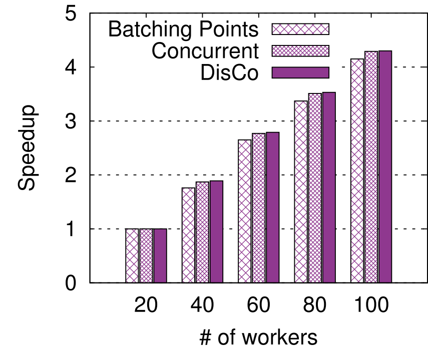


(a) FNMTF on ENRON

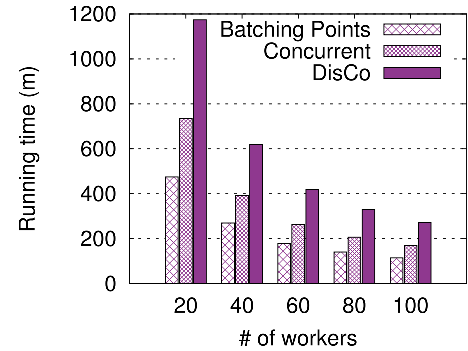


(b) FNMTF on ENRON

Fig. 11. Speedup and running time comparisons.



(a) ITCC on ENRON



(b) ITCC on ENRON

Fig. 12. Speedup and running time comparisons.

performance is not sensitive to the number of batches. For example, the performance of eight batches is similar to the performance of 32 batches. Based on these observations, we can use the following method to find a good candidate for the number of batches. We first give an initial guess for the number of batches, which is relatively small (e.g., 4 or 8), then run the co-clustering algorithm to measure other variables in Eq. (14). When we get the ratio $\frac{\Delta cost_{bat}}{\Delta cost_{con}}$ under the initial guess, we replace this ratio with a higher value (e.g., 1x~2x the obtained ratio) and consider this new ratio as the expected gain. Then we can derive a range for num_{batch} according to the expected gain and Eq. (14). Typically, selecting a number around the middle of this range can achieve good performance.

5.4 Large-Scale Experiments

To validate the scalability of our framework, we also run experiments on the Amazon EC2 cloud. The data set ENRON is used for evaluation. The number of row (or column) clusters is set to 20. Since the scalability of the dividing clusters approach is dependent on the relationship between the number of workers and the number of clusters, only the batching points approach is evaluated. The number of row (or column) batches each worker holds is set to 4.

As shown in Figs. 11a and 12a, the algorithms implemented in DisCo (denoted by "DisCo") and the algorithms implemented in the Co-ClusterD framework obtain similar speedups. The main reason is that the algorithms implemented in the Co-ClusterD framework do not shuffle the row and column subsets held by workers and only synchronize the updated cluster information S_{cc} and cluster assignments I_r (or I_c) over clustering iterations. We can also observe that the speedup obtained by Co-ClusterD with concurrent updates (denoted by "Concurrent") is a little better than Co-ClusterD using the batching points approach (denoted by "Batching Points"). The reason lies in that concurrent updates performs less cluster information updates than the batching points approach and thus results in less synchronization overhead.

In Figs. 11b and 12b, we can observe that, as the number of workers increases, the running time of the algorithms implemented in DisCo and Co-ClusterD is significantly reduced. Notice that, since the bases of computing speedups are different, a better speedup does not necessarily lead to a shorter running time. As shown in Figs. 11b and 12b, co-clustering algorithms parallelized by the batching points approach converge almost two times faster than their concurrent counterparts implemented in Co-ClusterD.

We also evaluate how the co-clustering algorithms implemented in our Co-ClusterD framework scale with increasing input size by adjusting input size to keep the amount of computation per worker fixed with increasing the number of workers. For this experiment, the ideal scaling has constant running time as input size increases with the number of workers. As shown in Figs. 13b and 13b, the achieved scaling is within 20 percent of the ideal number, which is acceptable.

6 Related Work

In this section, we review the work related to data co-clustering, and large scale data clustering.

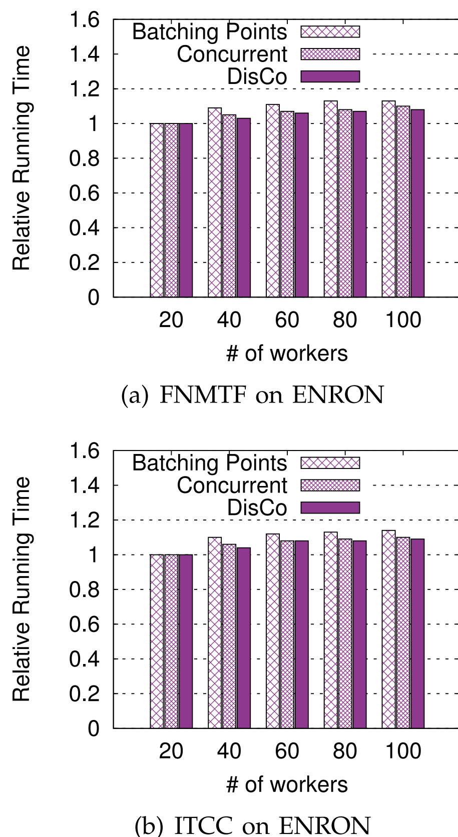


Fig. 13. Scaling input size.

6.1 Data Co-Clustering

There is a large body of work on algorithms for co-clustering. Dhillon [1] formulate co-clustering problem as a graph partition problem and propose a spectral co-clustering algorithm that uses eigenvectors to co-cluster input data matrix. Dhillon et al. [11] explore the relationships between nonnegative matrix factorization (NMF) [20] and k-means clustering, and propose to use nonnegative matrix tri-factorization [21] to co-cluster the input data matrix. Information theoretic co-clustering proposed in [11] is an alternate minimization based co-clustering algorithm, which monotonically optimizes the objective function by intertwining both row and column clustering iterations at all stages. Many variations of this kind of co-clustering algorithms have been proposed by using different optimization criteria such as sum-squared distance [3], and code length [14]. Based on Bregman divergence, a more general framework for this kind of algorithms has proposed in [9]. Our framework can perfectly support these alternate minimization based co-clustering algorithms with different update strategies and they can be easily implemented by overriding a a set of APIs provided by our framework.

6.2 Large Scale Data Clustering

As huge data sets become prevalent, improving the scalability of clustering algorithms has drawn more and more attention. Many scalable clustering algorithms are proposed recently. Dave et al. [22] propose a scheme of implementing k-means with concurrent updates on Microsoft's Windows Azure cloud. Ene et al. [23] design a method of implementing k-center and k-median on Mapreduce. These studies are different from ours as they are devoted to scaling up one-sided clustering algorithms. In order to minimize the I/O cost and the network cost among processing nodes, Cordeiro et al. [24] propose the best of both worlds (BoW) method for data clustering with MapReduce. This method can automatically spot the bottleneck and choose a good strategy. Since the BoW method can treat as a plug-in tool for most of the clustering algorithms, it is orthogonal to our work.

Folino et al. [25] propose a parallelized efficient solution to the high-order co-clustering problem (i.e., the problem of simultaneously clustering heterogeneous types of domain) [26]. George and Merugu [27] design a parallel version of the weighted Bregman co-clustering algorithm [9] and use it to build an efficient real-time collaborative filtering framework. Deodhar et al. [28] develop a parallelized implementation of the simultaneous co-clustering and learning algorithm [29] based on Mapreduce. Papadimitriou and Sun [19] propose the distributed co-clustering (DisCo) framework, under which various co-clustering algorithms can be implemented. Using barycenter heuristic [30], Nisar et al. [31] propose a high performance parallel for data co-clustering. Kwon and Cho [32] parallelize all the co-clustering algorithms in the Bregman co-clustering framework [9] using message passing interface (MPI). Narang et al. [33], [34] present a novel hierarchical design for soft real-time distributed co-clustering based collaborative filtering algorithm. However, while these studies focus on parallelizing co-clustering with concurrent updates, our work is devoted to parallelizing co-clustering with sequential updates.

7 Conclusions

In this paper, we introduce sequential updates for alternate minimization based co-clustering algorithms and propose dividing clusters and batching points approaches to parallelize AMCC algorithms with sequential updates. We prove both of these two approaches maintain the convergence properties of AMCC algorithms. Based on these two approaches, we design and implement a distributed framework referred to as Co-ClusterD, which supports efficient implementations of AMCC algorithms with sequential updates. Empirical results show that AMCC algorithms implemented in Co-ClusterD can achieve a much faster convergence and often obtain better results than their traditional concurrent counterparts.

Acknowledgments

The authors thank the reviewers for their valuable comments which significantly improved this paper. The work was supported in part by the following funding agencies of China: National Natural Science Foundation under grant 61170274, National Basic Research Program (973 Program) under grant 2011CB302506, Fundamental Research Funds for the Central Universities under grant 2014RC1103. This work is also supported in part by the US National Science Foundation under grants CNS-1217284 and CCF-1018114. The preliminary version of this paper has been accepted for publication in the proceedings of the 2013 IEEE International Conference on Data Mining series (ICDM 2013), Dallas, Texas, December 7-11, 2013. Prof. Sen Su is the corresponding author.

REFERENCES

[1] I. Dhillon, "Co-clustering documents and words using bipartite spectral graph partitioning," in *Proc. Knowl. Discovery Data Mining*, 2001, pp. 269–274.

[2] S. Daruru, N. M. Marin, M. Walker, and J. Ghosh, "Pervasive parallelism in data mining: Dataflow solution to co-clustering large and sparse netflix data," in *Proc. 15th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2009, pp. 1115–1124.

[3] H. Cho, I. Dhillon, Y. Guan, and S. Sra,, "Minimum sum-squared residue co-clustering of gene expression data," in *Proc. SIAM Int. Conf. Data Mining*, 2004, vol. 114, pp. 114-125.

[4] R. M. Neal and G. E. Hinton, "A view of the EM algorithm that justifies incremental, sparse, and other variants," *Learning in Graphical Models*, pp. 355–368, 1999.

[5] B. Thiesson, C. Meek, and D. Heckerman, "Accelerating EM for large databases," *Mach. Learn.*, vol. 45, no. 3, pp. 279–299, 2001.

[6] N. Slonim, N. Friedman, and N. Tishby, "Unsupervised document classification using sequential information maximization," in *Proc. 25th Annu. Int. ACM SIGIR Conf. Res. Develop. Inf. Retrieval*, 2002, pp. 129–136.

[7] I. Dhillon, Y. Guan, and J. Kogan, "Iterative clustering of high dimensional text data augmented by local search," in *Proc. IEEE Int. Conf. Data Mining*, 2003, pp. 131–138.

[8] 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.

[9] A. Banerjee, I. Dhillon, J. Ghosh, S. Merugu, and D. Modha, "A generalized maximum entropy approach to Bregman Co-clustering and matrix approximation," in *Proc. 10th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2004, pp. 509–514.

[10] H. Wang, F. Nie, H. Huang, and F. Makedon, "Fast nonnegative matrix tri-factorization for large-scale data co-clustering," in *Proc. 22nd Int. Joint Conf. Artif. Intell.*, 2011, pp. 1553–1558.

[11] I. Dhillon, S. Mallela, and D. Modha, "Information-theoretic Co-clustering," in *Proc. 9th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2003, pp. 89–98.

[12] J. Hartigan, "Direct clustering of a data matrix," *J. Am. Statist. Assoc.*, vol. 67, pp. 123–129, 1972.

[13] A. Banerjee, S. Merugu, I. S. Dhillon, and J. Ghosh, "Clustering with Bregman divergences," in *Proc. SDM*, 2004, pp. 234–245.

[14] D. Chakrabarti, S. Papadimitriou, D. Modha, and C. Faloutsos, "Fully automatic Cross-associations," in *Proc. 10th ACM SIGKDD Int. Conf. Knowl. Discovery data Mining*, 2004, pp. 79–88.

[15] Columbia University Image Library. [Online]. Available: http://www.cs.columbia.edu/CAVE/software/softlib/coil-20.php, 1996.

[16] UCI Machine Learning Repository. [Online]. Available: http://archive.ics.uci.edu/ml/datasets.html, 2013.

[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] M. Zaharia, M. Chowdhury, M. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proc. 2nd USENIX Conf. Hot Topics Cloud Comput.*, 2010, pp. 10–10.

[19] S. Papadimitriou and J. Sun, "Disco: Distributed co-clustering with map-reduce: A case study towards petabyte-scale end-to-end mining," in *Proc. 8th IEEE Int. Conf. Data Mining*, 2008, pp. 512–521.

[20] D. Lee and H. Seung, "Learning the parts of objects by non-negative matrix factorization," *Nature*, vol. 401, no. 6755, pp. 788–791, 1999.

[21] C. Ding, T. Li, W. Peng, and H. Park, "Orthogonal nonnegative matrix t-factorizations for clustering," in *Proc. 12th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2006, pp. 126–135.

[22] A. Dave, W. Lu, J. Jackson, and R. Barga, "Cloudclustering: Toward an iterative data processing pattern on the cloud," in *Proc. IEEE Int. Symp. Parallel Distrib. Process. Workshops Phd Forum*, 2011, pp. 1132–1137.

[23] A. Ene, S. Im, and B. Moseley, "Fast clustering using MapReduce," in *Proc. 17th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2011, pp. 681–689.

[24] R. Cordeiro, C. T. Jr, A. Traina, J. Lopez, U. Kang, and C. Faloutsos, "Clustering very large multi-dimensional datasets with MapReduce," in *Proc. 17th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, Aug. 2011, no. 17, pp. 690–698.

[25] F. Folino, G. Greco, A. Guzzo, and L. Pontieri, "Scalable parallel co-clustering over multiple heterogeneous data types," in *Proc. Int. Conf. High Perform. Comput. Simul.*, 2010, pp. 529–535.

[26] G. Greco, A. Guzzo, and L. Pontieri, "Coclustering multiple heterogeneous domains: Linear combinations and agreements," *IEEE Trans. Knowl. Data Eng.*, vol. 22, no. 12, pp. 1649–1663, Dec. 2010.

[27] T. George and S. Merugu, "A scalable collaborative filtering framework based on co-clustering," in *Proc. 5th IEEE Int. Conf. Data Mining*, 2005, pp. 625–628.

[28] M. Deodhar, C. Jones, and J. Ghosh, "Parallel simultaneous co-clustering and learning with map-reduce," in *Proc. IEEE Int. Conf. Granular Comput.*, 2010, pp. 149–154.

[29] M. Deodhar and J. Ghosh, "A framework for simultaneous co-clustering and learning from complex data," in *Proc. 13th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2007, pp. 250–259.

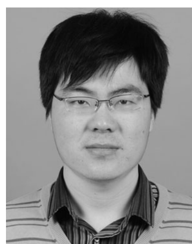
[30] W. Ahmad and A. Khokhar, "Chawk: A highly efficient biclustering algorithm using bigraph crossing minimization," in *Proc. 2nd Int. Workshop Data Mining Bioinf.*, 2007, pp. 1–12.

[31] A. Nisar, W. Ahmad, W. Liao, and A. N. Choudhary, "High performance parallel/distributed biclustering using barycenter heuristic," in *Proc. SIAM Int. Conf. Data Mining*, 2009, pp. 1050–1061.

[32] B. Kwon and H. Cho, "Scalable co-clustering algorithms," in *Proc. ICA3PP*, 2010, pp. 32–43.

[33] A. Narang, A. Srivastava, and N. P. K. Katta, "High performance distributed co-clustering and collaborative filtering," in IBM Res., NY, United States, Tech. Rep. RI11019, 2011, pp. 1–28.

[34] A. Narang, A. Srivastava, and N. P. K. Katta, "High performance offline and online distributed collaborative filtering," in *Proc. IEEE 12th Int. Conf. Data Mining*, 2012, pp. 549–558.



Xiang Cheng received the PhD degree in computer science from the Beijing University of Posts and Telecommunications, China, in 2013. He is currently an assistant professor at the Beijing University of Posts and Telecommunications. His research interests include information retrieval, data mining, and data privacy.

Sen Su received the PhD degree in computer science from the University of Electronic Science and Technology, China, in 1998. He is currently a professor at the Beijing University of Posts and Telecommunications. His research interests include data privacy, cloud computing, and internet services.

Lixin Gao is a professor of electrical and computer engineering at the University of Massachusetts at Amherst, and the PhD degree in computer science from the University of Massachusetts at Amherst in 1996. Her research interests include social networks, and Internet routing, network virtualization, and cloud computing. She is a fellow of the ACM and the IEEE.

Jiangtao Yin received the BE degree from the Beijing Institute of Technology, China, in 2006, the ME degree from the Beijing University of Posts and Telecommunications, China, in 2009, and is currently working toward the PhD degree in the Department of Electrical and Computer Engineering, the University of Massachusetts at Amherst. Since fall 2009, he has been working as a research assistant in the University of Massachusetts at Amherst. His current research interests include distributed systems, big data analytics, and cloud computing.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.