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

Sen Su, Xiang Cheng
State Key Laboratory of Networking and Switching Technology
Beijing University of Posts and Telecommunications
{*susen, chengxiang*}*@bupt.edu.cn*

Lixin Gao, Jiangtao Yin
Department of Electrical and Computer Engineering
UMASS Amherst
{*lgao, jyin*}*@ecs.umass.edu*

***Abstract*—Co-clustering is a powerful data mining tool for co-occurrence and dyadic data. As data sets become increasingly large, the scalability of co-clustering becomes more and more important. In this paper, we propose two approaches to parallelize co-clustering with sequential updates in a distributed environment. Based on these two approaches, we present a new distributed framework, Co-ClusterD, that supports efficient implementations of co-clustering algorithms with sequential updates. We design and implement Co-ClusterD, and show its efficiency through two co-clustering 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. Our evaluation shows that co-clustering algorithms implemented in Co-ClusterD can achieve better results and run faster than their traditional concurrent counterparts.**

***Keywords*-Co-Clustering; Concurrent Updates; Sequential Updates; Cloud Computing; Distributed Framework**

I. 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*.

Clustering algorithms with sequential updates intuitively outperform their concurrent counterparts, since they always leverage the most up-to-date cluster information to group input points. A number of existing studies (e.g., [4], [5]) have supported this claim. Despite the potential advantages of sequential updates, parallelizing co-clustering algorithms with sequential updates is challenging and is not directly supported by the existing distributed frameworks (e.g., Hadoop [6] or Spark [7]). 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, co-clustering algorithms with sequential updates cannot be easily performed in a distributed manner.

Toward this end, we propose two approaches to parallelize sequential updates for co-clustering 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*. 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.

Based on these two approaches, we design and implement a distributed framework, *Co-ClusterD*, to support efficient implementations of co-clustering algorithms with sequential updates. Co-ClusterD provides an abstraction for co-clustering algorithms with sequential updates and

allows programmers to specify the sequential update operations via simple APIs. We evaluate Co-ClusterD through two co-clustering algorithms: fast nonnegative matrix trifactorization (FNMTF) [8] and information theoretic co-clustering (ITCC) [9]. Experimenting on a local cluster of machines and the Amazon EC2 cloud, we show that co-clustering algorithms implemented in Co-ClusterD can obtain better results while running faster than their traditional concurrent counterparts.

II. Co-clustering and Update Strategies

A. Definitions and Overview

Co-clustering is also known as bi-clustering, block clustering or direct clustering [10]. 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. Specifically, $\rho : \{u_0, u_1, ..., u_m\} \rightarrow \{p_1, p_2, ..., p_k\}$ and $\gamma : \{v_0, v_1, ..., v_n\} \rightarrow \{q_1, q_2, ..., q_l\}$, where $\rho(u) = p$ means that row u is in row cluster p, and $\gamma(v) = q$ 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*.

Typically, the goal of data co-clustering is to find (ρ, γ) such that the following objective function is minimized.

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

where $C(Z, \tilde{Z})$ is the *approximation error* between the original matrix Z and the approximation matrix $\tilde{Z}$ uniquely determined by (ρ, γ), w_{uv} denotes the pre-specified weight of pair (u, v), d_ϕ is a given distance measure (e.g., Euclidean distance), z_{uv} and $\tilde{z_{uv}}$ are the elements of Z and $\tilde{Z}$ respectively, s_{pq} is the cluster information that gives the statistic on co-cluster (p, q).

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 algorithms can be referred to as *alternate minimization based co-clustering algorithms* [11], which are considered as our main focus in this paper.

Typically, alternate minimization based co-clustering 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.

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

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

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

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

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

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

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

B. Co-Clustering with Sequential Updates

Motivated by the fact that sequential updates can achieve faster convergence and better results than concurrent updates for clustering algorithms, we introduce sequential updates for alternate minimization based co-clustering 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, alternate minimization based co-clustering 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 1** 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.

When performing sequential updates, a row (or column) re-assignment only gives rise to the update of the statistics of co-clusters related to the reassigned row (or column). In addition, if the statistic can be updated incrementally (e.g., the statistic are the mean or summation of the co-cluster), we can update the statistics by subtracting or adding the effects of the reassigned row (or column). Therefore, updating the cluster information frequently does not necessarily introduce much computational overhead.

III. Parallelizing Co-Clustering with Sequential Updates

A. 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 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 ($p < k$ and $p < l$), 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 $S_1^r, S_2^r, ..., S_p^r$. These subsets are distributed to each worker in a one-to-one manner. When worker w_i receives S_i^r, it can perform row clustering with sequential updates for its rows R_i among the subset of row clusters S_i^r. For example, assume that S_i^r is $\{1, 3, 6\}$, w_i will perform row clustering for its rows whose current cluster assignments are in S_i^r, and allow these rows to change their 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.

B. 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 alternate minimization based co-clustering 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 details of the batching points approach are described 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 $R_i^1, R_i^2, ..., R_i^p$. 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.

IV. Co-clusterD Framework

A. Design and Implementation

Since existing distributed frameworks cannot directly support co-clustering algorithms with sequential updates, it calls for a distributed framework that inherently supports sequential updates for co-clustering algorithms. Based on the proposed approaches for parallelizing sequential updates in the previous section, we design and implement Co-ClusterD, a distributed framework for co-clustering algorithms with sequential updates. Co-ClusterD consists of a number of basic workers and a leading worker. Each basic worker performs row and column clustering independently. The leading worker plays a coordination role during the data co-clustering process.

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 the initial cluster assignments. 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. It alternatively performs row and column clusterings 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. To reduce the communication overhead, the input data matrix will not be repartitioned during the data co-clustering stage. In other words, row and column subsets held by each worker will not be shuffled and only the updated cluster assignments and the cluster information will

be synchronized among workers over iterations through the network.

Co-ClusterD is implemented based on iMapreduce [12], which is a distributed framework based on Hadoop and has built-in support for iterative algorithms. In fact, Co-ClusterD is independent of the underlying frameworks. We choose iMapreduce since it can better support the iterative processes of co-clustering algorithms.

B. API

Co-ClusterD allows users without much knowledge on distributed computing to write distributed co-clustering 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*, which 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 right now. 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.

`(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 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 meanings 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` is the slices of S_{cc} given by workers. `rInd` and `cInd` are row-cluster and column-cluster indicators of the input data matrix respectively.

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

Table I
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)

C. 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 [12], 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.

V. EXPERIMENTAL RESULTS

In this section, we evaluate the effectiveness and efficiency of Co-ClusterD in the context of two 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 [13]. Since DisCo does not support co-clustering 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.

A. Experiment Setup

We use real world data sets downloaded from UCI Machine Learning Repository [14] to evaluate the co-clustering algorithms. These data sets are summarized in Table II.

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 4 machines. Each

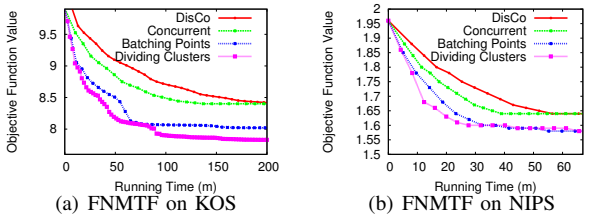
Table II
DESCRIPTION OF DATA SETS

Data sets	samples	features	non-zeros
KOS	3430	6906	467714
NIPS	1500	12419	1900000
ENRON	39861	28102	6400000

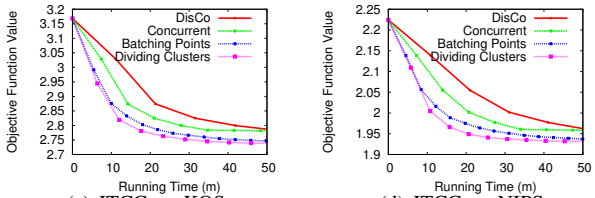
machine has Intel Core 2 Duo E8200 2.66GHz processor, 3GB of RAM, 1TB hard disk, and runs 32-bit Linux Debian 6.0 OS. These 4 machines are connected to a switch with communication bandwidth of 1Gbps. The large-scale cluster consists of 100 High-CPU medium instances on the Amazon EC2 Cloud. Each instance has 1.7GB memory and 5 EC2 compute units.

B. 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.



(a) FNMTF on KOS (b) FNMTF on NIPS



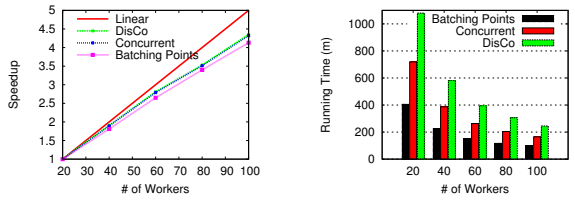
(c) ITCC on KOS (d) ITCC on NIPS

Figure 1. Cost function vs. running time comparisons for co-clustering algorithms with different update strategies

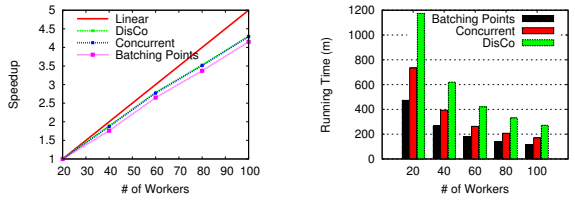
As shown in Figure 1, we can observe that co-clustering algorithms with concurrent updates implemented in Co-ClusterD (denoted by "Concurrent") converge faster than those implemented in DisCo (denoted by "DisCo") although they converge to the same values. This is because Co-ClusterD leverages a persistent 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 fast convergence. In addition, we can also observe that algorithms parallelized by the dividing clusters approach and the batching points approach converge faster and obtain better results than their concurrent counterparts implemented in DisCo and Co-ClusterD.

C. 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.



(a) FNMTF on ENRON (b) FNMTF on ENRON



(c) ITCC on ENRON (d) ITCC on ENRON

Figure 2. Speedup and performance comparisons

As shown in Figure 2(a) and Figure 2(c), algorithms implemented in DisCo (denoted by "DisCo") and Co-ClusterD with concurrent updates (denoted by "Concurrent") obtain the same speedup. This is because both of them perform concurrent updates. We can also observe that their speedups are better than Co-ClusterD using the batching points approach. The reason lies in that concurrent updates performs less cluster information updates than the batching points approach and thus results in less synchronization overhead. However, since the bases of computing speedups are different, a better speedup does not necessarily lead to a shorter running time. As shown in Figure 2(b) and Figure 2(d), co-clustering algorithms parallelized by the batching points approach still converge much faster than their concurrent counterparts implemented in DisCo and Co-ClusterD.

VI. RELATED WORK

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. [15] propose a scheme of implementing k-means with concurrent updates on Microsoft's Windows Azure cloud. Ene et al. [16] design a method of implementing k-center and k-median on Mapreduce. Yin et al. [17] develop a distributed framework called FreEM for parallelizing EM algorithms. However, these studies are different from ours as they are devoted to scaling up one-sided clustering algorithms. Folino et al. [18] propose a parallelized efficient solution to the high-order co-clustering problem (i.e., the problem of simultaneously clustering heterogeneous types of domain) [19]. George et al. [20] design a parallel version of the weighted Bregman co-clustering algorithm [11] and use it to build an efficient real-time collaborative filtering framework. Deodhar et al. [21] develop a parallelized implementation of the simultaneous co-clustering and learning algorithm [22] based on Mapreduce. Papadimitriou et al. [13] propose the distributed co-clustering (DisCo) framework, under which various co-clustering algorithms can be implemented. However, while these studies focus on parallelizing co-clustering with concurrent updates, our work is devoted to parallelizing co-clustering with sequential updates.

VII. Conclusions

In this paper, we propose dividing clusters and batching points approaches to parallelize co-clustering with sequential updates. Based on these two approaches, we design and implement a distributed framework referred to as Co-ClusterD, which supports efficient implementations of co-clustering algorithms with sequential updates. Experimental results show that co-clustering algorithms implemented in Co-ClusterD can obtain better results and run faster than their traditional concurrent counterparts.

Acknowledgment

The work is supported in part by the following funding agencies in China: the National Natural Science Foundation under grant 61170274, the Innovative Research Groups of the National Natural Science Foundation under grant 61121061. This work is also supported in part by the National Science Foundation under grants CNS-1217284 and CCF-1018114.

References

[1] I. Dhillon, "Co-clustering documents and words using bipartite spectral graph partitioning," in *KDD*. ACM, 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 *KDD*. ACM, 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 *SIAM*, vol. 114, 2004.

[4] N. Slonim, N. Friedman, and N. Tishby, "Unsupervised document classification using sequential information maximization," in *SIGIR*. ACM, 2002, pp. 129–136.

[5] I. Dhillon, Y. Guan, and J. Kogan, "Iterative clustering of high dimensional text data augmented by local search," in *ICDM*, 2003, pp. 131 – 138.

[6] "Hadoop," http://http://hadoop.apache.org/.

[7] M. Zaharia, M. Chowdhury, M. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *HotCloud*. USENIX Association, 2010, pp. 10–10.

[8] H. Wang, F. Nie, H. Huang, and F. Makedon, "Fast nonnegative matrix tri-factorization for large-scale data co-clustering," in *IJCAI*, 2011.

[9] I. Dhillon, S. Mallela, and D. Modha, "Information-theoretic co-clustering," in *KDD*. ACM, 2003, pp. 89–98.

[10] J. Hartigan, "Direct clustering of a data matrix," *Journal of the American Statistical Association*, pp. 123–129, 1972.

[11] A. Banerjee, I. Dhillon, J. Ghosh, S. Merugu, and D. Modha, "A generalized maximum entropy approach to bregman co-clustering and matrix approximation," in *KDD*. ACM, 2004, pp. 509–514.

[12] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "iMapreduce: A distributed computing framework for iterative computation," in *IPDPSW*. IEEE, 2011, pp. 1112–1121.

[13] S. Papadimitriou and J. Sun, "Disco: Distributed co-clustering with map-reduce: A case study towards petabyte-scale end-to-end mining," in *ICDM*. IEEE, 2008, pp. 512–521.

[14] "UCI Machine Learning Repository," http://archive.ics.uci.edu/ml/datasets.html.

[15] A. Dave, W. Lu, J. Jackson, and R. Barga, "Cloudclustering: Toward an iterative data processing pattern on the cloud," in *IPDPSW*. IEEE, 2011, pp. 1132–1137.

[16] A. Ene, S. Im, and B. Moseley, "Fast clustering using MapReduce," in *KDD*. ACM, 2011, pp. 681–689.

[17] J. Yin, Y. Zhang, and L. Gao, "Accelerating expectation-maximization algorithms with frequent updates," in *CLUSTER*. IEEE, 2012, pp. 275–283.

[18] F. Folino, G. Greco, A. Guzzo, and L. Pontieri, "Scalable parallel co-clustering over multiple heterogeneous data types," in *HPCS*. IEEE, 2010, pp. 529–535.

[19] G. Greco, A. Guzzo, and L. Pontieri, "Coclustering multiple heterogeneous domains: Linear combinations and agreements," *TKDE*, vol. 22, no. 12, pp. 1649–1663, 2010.

[20] T. George and S. Merugu, "A scalable collaborative filtering framework based on co-clustering," in *ICDM*. IEEE, 2005, pp. 4–pp.

[21] M. Deodhar, C. Jones, and J. Ghosh, "Parallel simultaneous co-clustering and learning with map-reduce," in *GrC*. IEEE, 2010, pp. 149–154.

[22] M. Deodhar and J. Ghosh, "A framework for simultaneous co-clustering and learning from complex data," in *KDD*. ACM, 2007, pp. 250–259.