2012 IEEE 12th International Conference on Data Mining

High Performance Offline & Online Distributed Collaborative Filtering

Ankur Narang, Abhinav Srivastava
IBM India Research Laboratory
New Delhi, India
(annarang,abhin122)@in.ibm.com

Naga Praveen Kumar Katta
Princeton University
New Jersey, USA
nkatta@cs.princeton.edu

***Abstract*—Big data analytics is a hot research area both in academia and industry. It envisages processing massive amounts of data at high rates to generate new insights leading to positive impact (for both users and providers) of industries such as E-commerce, Telecom, Finance, Life Sciences and so forth. We consider collaborative filtering (CF) and Clustering algorithms that are key fundamental analytics kernels that help in achieving these aims. High throughput CF and co-clustering on highly sparse and massive datasets, along with a high prediction accuracy, is a computationally challenging problem. In this paper, we present a novel hierarchical design for soft real-time (less than 1 minute.) distributed co-clustering based collaborative filtering algorithm. We study both the online and offline variants of this algorithm. Theoretical analysis of the time complexity of our algorithm proves the efficacy of our approach. Further, we present the impact of load balancing based optimizations on multi-core cluster architectures. Using the Netflix dataset($900M$ training ratings with replication) as well as the Yahoo KDD Cup($2.3B$ training ratings with replication) datasets , we demonstrate the performance and scalability of our algorithm on a large multi-core cluster architecture. In offline mode, our distributed algorithm demonstrates around $4\times$ better performance (on Blue Gene/P) as compared to the best prior work, along with high accuracy. In online mode, we demonstrated around $3\times$ better performance compared to baseline MPI implementation. To the best of our knowledge, our algorithm provides the best known online and offline performance and scalability results with high accuracy on multi-core cluster architectures.**

I. Introduction

Collaborative filtering (CF) is a subfield of machine learning that aims at creating algorithms to predict user preferences based on past user behavior in purchasing or rating of items [1], [2]. Here, the input is a set of known item preferences per user, typically in the form of a user-item ratings matrix. This user-item ratings matrix is typically sparse. The collaborative filtering problem is to find the unknown preferences of a user for a specific item, i.e. an unknown entry in the ratings matrix, using the underlying collaborative behavior of the user-item preferences. Collaborative filtering based recommender systems are very important in e-commerce applications. They help people more easily find items that they would like to purchase [3]. This enhances the user experience which typically leads to improvements in sales and revenue. Further, scientific disciplines such as Computational Biology and Personalized medicine (risk stratification) stand to gain immensely from CF [4], [5]. CF systems are also increasingly important in dealing with information overload since they can lead users to information that others like them have found useful. With massive amounts of data (terabytes to petabytes) and high data rates in Telecom (around 6B Call Data Records per day), Finance and other industries, there is a strong need to deliver soft real-time training for CF as it will lead to further enhance the quality of experience of customers along with increase in revenue for the provider.

Typical approaches for CF include matrix factorization based techniques, correlation based techniques , co-clustering based techniques, and concept decomposition based techniques [6]. Matrix factorization [7] and correlation [8] based techniques are computationally expensive hence cannot deliver soft real-time CF. Further, in matrix factorization based approaches, updates to the input ratings matrix leads to non-local changes which leads to higher computational cost for online CF. Concept Decomposition based technique [6] perform spherical k-means followed by least-squares based approximation of the original matrix. This work presents only sequential performance (13.5 minutes) for training of the full Netflix dataset which is far from being considered soft real-time. Co-clustering based techniques [9], [10] have better scalability but have not been optimized to deliver high throughput on massive data sets. Daruru et al. in [10] presented dataflow parallelism based co-clustering implementation which did not scale beyond 8 cores due to cache miss and in-memory lookup overheads. CF over highly sparse data sets leads to lower compute utilization due to load imbalance. For large scale distributed environment (256 nodes and beyond), load imbalance can dominate the overall performance and the communication cost becomes worse with increasing size of the cluster, leading to performance degradation. Thus, high computational demand and low parallel efficiency due to cache misses and load imbalance are the key challenges that need to be addressed to achieve high throughput distributed collaborative filtering on highly sparse massive data sets.

In order to optimize the parallel performance, achieve high parallel efficiency and give soft-real time ($\tilde{1}$min) guarantees on massive datasets, we designed a novel *hierarchical* approach for distributed co-clustering. Specifically, this paper makes the following key contributions:

- We present a novel distributed hierarchical approach for both offline and online co-clustering based collaborative filtering. Performance optimizations including load balancing and computation communication overlap have been incorporated for high throughput and soft real-time (less than 1 min.) performance over highly sparse massive data-sets.
- Analytical parallel time complexity analysis, establishes theoretically that our hierarchical design leads to performance gain of order $O(\log(\pi))$ (where π is number of row and column partitions of the input matrix) over the best prior approach [11].
- We demonstrate soft real-time performance for both offline and online mode collaborative filtering using the Netflix Prize and Yahoo KDD Cup datasets on a 4096-

1550-4786/12 $26.00 © 2012 IEEE
DOI 10.1109/ICDM.2012.128

node multi-core cluster architecture (Blue Gene/P [1]). For offline mode, we achieved a training time (using I-divergence and C6, Section III) of around $9.38s$ with the full Netflix dataset and prediction time of $2.8s$ on $1.4M$ ratings with RMSE (Root Mean Square Error) of 0.87 ± 0.02. This is around $4\times$ better than the best prior distributed algorithm [11] for the same dataset. For $900M$ ratings using Netflix dataset and $2.3B$ ratings using Yahoo KDD Cup dataset, we show soft real-time performance along with high accuracy. For online mode, our algorithm delivers around $3\times$ better performance compared to baseline MPI implementation.

II. Related Work

Co-clustering and collaborative filtering (CF) are fundamental data-mining kernels used in many application domains such as Information Retrieval [12], Telecom [13], Financial markets, Life Sciences [4]. Hassan et al. in [5] evaluate the context of a specific clinical challenge, i.e., risk stratification following acute coronary syndrome (ACS). On over 4,500 patients, this research shows that CF outperforms traditional classification methods such as logistic regression (LR) and support vector machines (SVMs) for predicting both sudden cardiac death and recurrent myocardial infarction within one year of the index event.

Banerjee et al. in [14] consider multiway-clustering of a single tensor or a group of tensors over heterogeneous relational data, using Bregman (Bregman divergence models a broad family of information loss functions that includes squared Euclidean distance, KL-divergence, I-divergence) co-clustering based alternate minimization algorithm and shows its advantages in the domains of social networks, e-commerce using movie recommendation data as well as newsgroup articles. We optimize the Bregman co-clustering algorithm [15] (based on alternate minimization) for distributed systems. Our novel hierarchical approach will also improve the distributed performance of the *multi-way clustering* algorithm over *heterogeneous relational tensor data*.

Typical CF techniques are based on correlation criteria [8] and matrix factorization [7]. The correlation-based techniques use similarity measures such as Pearson correlation and cosine similarity to determine a neighborhood of like-minded users for each user and then predict the user's rating for a product as a weighted average of ratings of the neighbors. Correlation-based techniques are computationally very expensive as the correlation between every pair of users needs to be computed during the training phase. Further, they have much reduced coverage since they cannot detect item synonymy. The matrix factorization approaches include Singular Value Decomposition (SVD [16]) and Non-Negative Matrix Factorization (NNMF) based [7] filtering techniques. They predict the unknown ratings based on a low rank approximation of the original ratings matrix. The missing values in the original matrix are filled using average values of the rows or columns. However, the training component of these techniques is computationally intensive, which makes them impractical to have frequent re-training. Incremental versions of SVD based on folding-in and exact rank-1 updates [17] partially alleviate this problem. But, since the effects of small updates are not localized, the update operations are not very efficient.

George et al in [9] studies a special case of the weighted Bregman co-clustering algorithm. The co-clustering problem is formulated as a matrix approximation problem with non-uniform weights on the input matrix elements. As in the case of SVD and NNMF, the co-clustering algorithm also optimizes the approximation error of a low parameter reconstruction of the ratings matrix. However, unlike SVD and NNMF, the effects of changes in the ratings matrix are localized which makes it possible to have efficient incremental updates. This work presents parallel algorithm design based on co-clustering. It compares the performance of the algorithm against matrix factorization and correlation based approaches on the MovieLens [2] and BookCrossing dataset [18] (269392 explicit rating(1-10) from 47034 users on 133438 books). We consider soft real-time CF framework using hierarchical parallel co-clustering optimized for multi-core clusters using pipelined parallelism and computation communication overlap. We deliver scalable performance over much large data on multicore clusters.

Daruru et al. in [10] use a dataflow parallelism based framework to study performance vs. accuracy trade-offs of co-clustering based CF. However, it doesn't consider re-training time for incremental input changes. Further, the parallel implementation does not scale well beyond 8 cores due to cache miss and in-memory lookup overheads. We demonstrate parallel scalable performance on 4096 nodes of Blue Gene/P and $7\times$ to $10\times$ better training time and better prediction time. Further, while none of the prior work aims at massive scale performance, we provide theoretical and empirical analysis to demonstrate this scale of performance of our distributed algorithm. Hsu et al.in [19] study IO scalable co-clustering by mapping a significant fraction of computations performed by the Bregman co-clustering algorithm to an on-line analytical processing (OLAP) engine. Kwon et al. in [20] study the scalability of basic MPI based implementation of co-clustering. We deliver more than one order of magnitude higher performance compared to this work, by performing communication and load balancing optimizations along with novel hierarchical design for multi-core clusters.

Ampazis in [6] presents results of collaborative filtering using *Concept decomposition* based approach. It has been empirically established by Dhillon et al. in [21] that the approximation power (when measured using the Frobenius norm) of concept decompositions is comparable to the best possible approximations by truncated SVDs [22]. However,this paper presents the results of a sequential concept decomposition based algorithm that takes 13.5mins. training time for the full Netflix data, which is very high when looking at soft real-time performance. Narang et al. in [23] presents a parallel CF algorithm using concept decomposition on 32-code SMP architecture. It achieves $64s$ total training time for Netflix data. Using multi-core clusters, we deliver around two order of magnitude improvement in training time compared to the sequential concept decomposition technique [6] and around one of magnitude improvement compared to the parallel concept decomposition technique [23]. Narang et al. [11] presents a *flat* distributed co-clustering algorithm where all the processors in the system participate in each iteration of the co-clustering algorithm, and both OpenMP and MPI (*hybrid* approach) are used to exploit both intra-node and inter-node parallelism available in Blue Gene/P. Using the Netflix dataset (100M ratings), it demonstrates the performance

[1] www.ibm.com/bluegene

[2] http://www.grouplens.org/data/. 100K ratings(1-5) 943 users, 1682 movies

and scalability of the algorithm on 1024-node Blue Gene/P system: with training time of around 6s on the full Netflix dataset (with Euclidean-divergence).In this paper, we present a hierarchical approach for distributed co-clustering along with load balancing optimizations leading to around average $4\times$ improvement in performance as compared to [11] on Yahoo KDD Cup and Netflix datasets. Theoretical analysis of our hierarchical algorithm firmly establishes the performance gain of $O(\log(\pi))$ (where, π is the number of partitions of rows and columns of the input matrix) as compared to the *flat* algorithm in [11]. Further, we present detailed performance comparison with much larger data as compared to [11].

III. Background and Notation

In this paper, we deal with partitional co-clustering where all the rows and columns are partitioned into disjoint row and column clusters respectively. We consider a general framework for addressing this problem that considerably expands the scope and applicability of the co-clustering methodology. As part of this generalization, we view partitional co-clustering as a lossy data compression problem [15] where, given a specified number of rows and column clusters, one attempts to retain as much information as possible about the original data matrix in terms of statistics based on the co-clustering [24]. The main idea is that a reconstruction based on co-clustering should result in the same set of user-specified statistics as the original matrix.

Let k and l be the number of row and column clusters respectively then a $k*l$ partitional co-clustering is defined as a pair of functions:
$\rho : 1,\ldots,m \longmapsto 1,...,k$; and, $\gamma : 1,\ldots,n \longmapsto 1,...,l$. Let $\hat{U}$ and $\hat{V}$ be random variables that take values in $1,...,k$ and $1,...,l$ such that $\hat{U} = \rho(U)$ and $\hat{V} = \gamma(V)$. Let, $\hat{Z} = [\hat{z}_{uv}] \in S^{m\times n}$ be an approximation of the data matrix Z such that $\hat{Z}$ *depends only upon* a given co-clustering (ρ, γ) and certain summary statistics derived from co-clustering. Let $\hat{Z}$ be a (U,V)-measurable random variable that takes values in this approximate matrix $\hat{Z}$ following w, i.e., $p(\hat{Z}(U,V) = \hat{z}_{uv}) = w_{uv}$. Then, the goodness of the underlying co-clustering can be measured in terms of the expected distortion between Z and $\hat{Z}$, that is,

$$E[d_\phi(Z,\hat{Z})] = \sum_{u=1}^{m}\sum_{v=1}^{n} w_{uv} d_\phi(z_{uv},\hat{z}_{uv}) = d_{\Phi_w}(Z,\hat{Z}) \quad (1)$$

where $\Phi_w : S^{m\times n} \longmapsto \mathbb{R}$ is a is a separable convex function induced on the matrices such that the Bregman divergence ($d_\Phi()$) between any pair of matrices is the weighted sum of the element-wise Bregman divergences corresponding to the convex function ϕ. From the matrix approximation viewpoint, the above quantity is simply the weighted element-wise distortion between the given matrix Z and the approximation $\hat{Z}$. The co-clustering problem is then to find (ρ, γ) such that (1) is minimized.

Now we consider two important convex functions that satisfy the Bregman divergence criteria and are hence studied in this paper.

- **I-Divergence :** Given $z \in \mathbb{R}_+$, let $\phi(z) = zlogz - z$. For $z1, z2 \in \mathbb{R}$, $d_\phi(z_1,z_2) = z_1 log(z_1/z_2) - (z_1 - z_2)$.
- **Squared Euclidean distance :** Given $z \in \mathbb{R}$, let $\phi(z) = z^2$. For $z1, z2 \in \mathbb{R}$, $d_\phi(z_1,z_2) = (z_1 - z_2)^2$.

Given a co-clustering (ρ, γ), Banerjee et al. [15] discuss six co-clustering bases where each co-clustering basis preserves certain summary statistics on the original matrix. It also proves that the possible co-clustering bases ($C1 \ldots C6$) form a hierarchical order in the number of cluster summary statistics they preserve. The co-clustering basis $C6$ preserves all the summaries preserved by the other co-clustering bases and hence is considered the most general among the bases. In this paper we discuss the partitioning co-cluster algorithms for the basis $C6$. For co-clustering basis $C6$ and Euclidean-divergence objective, the matrix approximation is given by: $\hat{A}_{ij} = A^{COC}_{gh} + (A^{CC}_{ih} - A^{RC}_{gj})$, where, $A^{RC}_{gj} = \frac{S^{RC}_{gj}}{W^{RC}_{gj}} = \frac{\sum_{i'|\rho(i')=g} A_{i'j}}{\sum_{i'|\rho(i')=g} W_{i'j}}$; $A^{CC}_{ih} = \frac{S^{CC}_{ih}}{W^{CC}_{ih}} = \frac{\sum_{j'|\gamma(j')=h} A_{ij'}}{\sum_{j'|\gamma(j')=h} W_{ij'}}$ and $A^{COC}_{gh} = \frac{S^{COC}_{gh}}{W^{COC}_{gh}} = \frac{\sum_{i'|\rho(i')=g}\sum_{j'|\gamma(j')=h} A_{i'j'}}{\sum_{i'|\rho(i')=g}\sum_{j'|\gamma(j')=h} W_{i'j'}}$.

The sequential update algorithm for the basis $C6$ is as shown in Algorithm 1 where the approximation matrix $\hat{A}$ for various co-clustering bases can be obtained from [15]. For Euclidean divergence, Step 2b. and 2c. of Algorithm 1 use $d_\phi(A_{ij},\hat{A}_{ij}) = (A_{ij} - \hat{A}_{ij})^2$. For I-divergence, Step 2b. and 2c. of Algorithm 1 use $d_\phi(A_{ij},\hat{A}_{ij}) = A_{ij} * \log(\hat{A}_{ij}/A_{ij}) - A_{ij} + \hat{A}_{ij}$

Algorithm 1 Sequential Static Training via Co-Clustering

Input: Ratings Matrix A, Non-zeros matrix W, No. of row clusters l, No. of column clusters k.
Output: Locally optimal co-clustering (ρ,γ) and averages A^{COC},A^{RC},A^{CC},A^R and A^C.
Method:
1. Randomly initialize (ρ,γ)
while *RMSE value is converging* **do**
  2a. Compute averages A^{COC},A^{RC}_{gj},A^{CC}_{ih},A^R and A^C where $1 \le g \le k$ and $1 \le h \le l$.
  2b. Update row cluster assignments
  $\rho(i) = \underset{1\le g\le k}{\textbf{argmin}} \sum_{j=1}^{n} W_{ij} d_\phi(A_{ij},\hat{A}_{ij})$, $1 \le i \le m$
  2c. Update column cluster assignments
  $\gamma(i) = \underset{1\le h\le l}{\textbf{argmin}} \sum_{i=1}^{m} W_{ij} d_\phi(A_{ij},\hat{A}_{ij})$, $1 \le j \le n$
end

A. Distributed Flat Coclustering Algorithm

In the above sequential algorithm (Algorithm 1), we notice two important steps - a) Calculating the matrix averages, and, b) updating the row and column cluster assignments. Further, given the matrix averages, row and column cluster updates can be done independently, and row updates themselves can be done in parallel.The flat distributed algorithm [11] leverages this inherent data parallelism. As one can see, this algorithm needs three MPI collectives calls: 1) To communicate Row/Column memberships, 2) To communicate Row/Column cluster averages and 3) To communicate cocluster averages. Since, the input ratings matrix is uniformly partitioned across all available processors, the algorithm can support very large matrices and hence has strong memory scalability. However, as the number of processor increases the collectives across all the

processors can become a bottleneck to the strong scalability for performance. In this paper we consider a hierarchical algorithm which reduces both communication and computation cost on multi-core cluster architecture while maintaining similar accuracy.

IV. Hierarchical Coclustering Algorithm

In this section, we present the detailed algorithmic design of our novel hierarchical co-clustering algorithm. The original input *(users*items)* ratings matrix is divided into certain number of row and column partitions. Each partition is assigned to a set of nodes in the cluster architecture. The hierarchical algorithm runs from bottom to top along a computation tree (Fig. 2) as follows. First, flat parallel co-clustering is run in each partition independently. The number of row and column clusters chosen is smaller compared to that specified in the input. Then, for each partition, the row and column clusters generated are merged with the adjacent partition. This gives the next level row and column clusters. At this higher level, flat parallel co-clustering is then run independently in each partition. Then again, the resulting row and column clusters at this level are merged to generate the next higher level row and column clusters. This forms a *computation tree* (Fig. 2) of execution. The alternate flat co-clustering and row/column cluster merge continue up the computation tree until the full matrix is obtained as a single partition (at the highest level in the tree) and finally flat parallel co-clustering is run here with the number of row and column clusters as specified in the input.

This hierarchical design helps in improving the overall time of the co-clustering algorithm without loss in accuracy of CF. At the lower levels of the computation tree, faster co-clustering iterations with smaller number of row and column clusters take place. This reduces the computation time. Moreover, MPI collectives like MPI_Allreduce and MPI_Allgather are usually costly in nature when used over large number of nodes in the system (as in the flat algorithm [11]). However, in the hierarchical algorithm, these collectives occur in smaller subsets of nodes (smaller communication topologies) and hence the communication cost is reduced. Thus, the hierarchical design results in lower computation as well as communication time. Further, row and column clusters at one level, after merge, result in good quality seed clusters for co-clustering at the next level. So, in the same number of iterations as a pure flat co-clustering algorithm [11], one can converge to similar high quality co-clustering for the hierarchical algorithm. Hence, the hierarchical algorithm provides a better trade-off point for speed vs accuracy as compared to the flat algorithm.

Fig. 1 and Fig. 2 illustrate the hierarchical algorithm in detail. Here, the input ratings matrix A is partitioned into $4 * 4 = 16$ partitions ($\pi_r = 4$, $\pi_c = 4$). At level 0 (leaf level of the computation tree, Fig. 2), first each partition, $\Pi^0_{i,j}$ ($1 \leq i \leq 4$, $1 \leq j \leq 4$), performs a certain number of flat co-clustering iterations on its corresponding sub-matrix, $A^0_{i,j}$, independently and in parallel using the G_0 processors allocated to it. Each partition generates, $k/4$ row clusters and $l/4$ column clusters. Then, pairs of adjacent partitions (for instance partition $\Pi^0_{1,1}$ and partition $\Pi^0_{2,1}$), merge their row and column clusters respectively, to generate $k/2$ row clusters and $l/4$ column clusters at level 1. Since, the underlying sub-matrices of the adjacent partitions are concatenated along the rows, this step is called as *row folding* step (See Fig. 1 and Step 3 in Algorithm 2). Then, at level 1, each partition, $\Pi^1_{i,j}$ (with $1 \leq i \leq 2$ and $1 \leq j \leq 4$), independently runs flat co-clustering iterations on the sub-matrix, $A^0_{i,j}$, with $k/2$ row clusters and $l/4$ column clusters. The updated row and column clusters of adjacent partitions are merged to generate k row clusters and $l/4$ column clusters at the next level 2 (another row fold step). These two row fold steps for the corresponding sub-matrices are illustrated in Fig. 1. These are followed by two *column fold* steps. At level 2, each partition, $\Pi^2_{i,j}$ ($i == 1$, $1 \leq j \leq 4$) independently runs flat co-clustering iterations on the sub-matrix, $A^2_{i,j}$, to update the k row clusters and $l/4$ column clusters. Then, each pair of adjacent partitions merges the row and column clusters to generate k new row clusters and $l/2$ column clusters. These, form the seed row and column clusters for level 3. After, the flat co-clustering iterations at level 3, the k row clusters and $l/2$ column clusters of the two partitions at this level, are merged to generate k row clusters and l column clusters at level 4. These clusters are then refined by final set of flat co-clustering iterations. This gives us the full matrix with k row and l column clusters. For exact details refer Algorithm 2.

While merging row and column clusters of one level to generate row and column clusters of the next level, one needs ensure low merge compute and communication time while at the same time generating good quality starting seed clusters for the next level. In order to achieve this, we use maximum weight bi-partite matching across two sets of clusters. During row folds, the number of row clusters simply doubles hence, no merge is required. While, the number of column clusters remains the same at the next level. Hence, using the number of overlapping columns as the weight of the edge connecting two column clusters, we perform maximum weight bi-partite matching algorithm to quickly merge the column clusters. This merging operation requires an additional MPI_Allreduce operation to communicate the cluster memberships from one partition to the other.

The row and column merging (folding) usually happens alternatively to reduce the bias towards row or column clusters. However, to minimize data transfer volume for mitigating load imbalance, one might choose a particular sequence of row or column folds/merge. We leave a detailed study of this effect to future work.

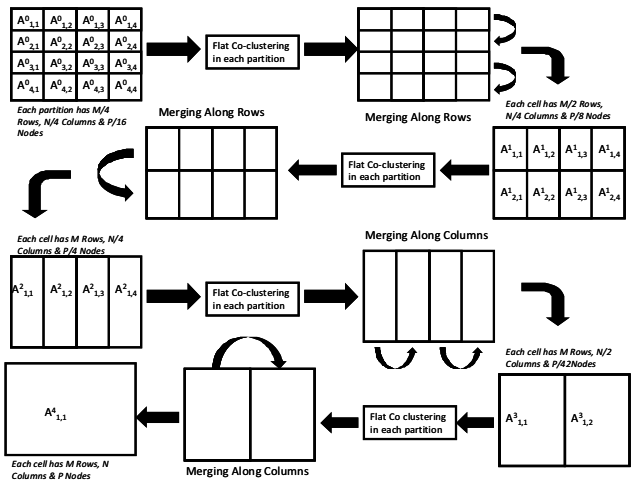


Fig. 1. Hierarchical Co-clustering: Matrix Row/Column Folding

Algorithm 2 Distributed Hierarchical Co-Clustering

Input: Ratings Matrix A, Non-zeros matrix W, No. of row clusters l, No. of column clusters k.
Output: Locally optimal co-clustering (ρ,γ) and averages A^{COC},A^{RC},A^{CC},A^{R} and A^{C}.
Method:
Let A be divided into π_r row partitions and π_c column partitions. Then the hierarchical algorithm proceeds with $\log(\pi_r)$ row folds first and then with $\log(\pi_c)$ column folds. Initialize $x = 0$ and $y = 0$.
while $(x{+}{+}) < (\log(\pi_r))$ **do**

1. In the current iteration, each partition $\Pi^x_{i,j}$ reads only the $\frac{m2^x}{\pi_r} \times \frac{n}{\pi_c}$ submatrix $A^x_{i,j}$ of A where $0 \leq i < \frac{\pi_r}{2^x}$ and $0 \leq j < \pi_c$.
2. Each partition $\Pi^x_{i,j}$ iteratively calculates a $(k.2^x/\pi_r, l/\pi_c)$ locally optimum coclustering ($\rho^x_{i,j}$,$\gamma^x_{i,j}$) for the submatrix $A^x_{i,j}$
3. ***Fold along rows:*** Partition $\Pi^x_{2i,j}$ merges with partition $\Pi^x_{2i+1,j}$ in the following manner to form $\Pi^{x+1}_{i,j}$.
1a. $\rho^x_{2i,j}$ and $\rho^x_{2i+1,j}$ together form $k.2^{x+1}/\pi_r$ new row clusters $\rho^{x+1}_{i,j}$
1b. $\gamma^x_{2i,j}$ and $\gamma^x_{2i+1,j}$ merge using maximum bi-partite matching to form l/π_c new column clusters $\gamma^{x+1}_{i,j}$

end
while $(y{+}{+}) < (\log(\pi_c))$ **do**

1. In the current iteration, each partition $\Pi^y_{i,j}$ reads only the $m \times \frac{n.2^y}{\pi_c}$ submatrix $A^y_{i,j}$ of A where $i = 0$ and $0 \leq j < \frac{\pi_c}{2^y}$.
2. Each partition $\Pi^y_{i,j}$ iteratively calculates a $(k, l.2^y/\pi_c)$ locally optimum coclustering ($\rho^y_{i,j}$,$\gamma^y_{i,j}$) for the submatrix $A^y_{i,j}$
3. ***Fold along columns:*** Partition $\Pi^y_{i,2j}$ merges with partition $\Pi^y_{i,2j+1}$ in the following manner to form $\Pi^{y+1}_{i,j}$.
1a. $\rho^y_{i,2j}$ and $\rho^y_{i,2j+1}$ merge using maximum weight bi-partite matching form k new row clusters $\rho^{y+1}_{i,j}$
1b. $\gamma^y_{i,2j}$ and $\gamma^y_{i,2j+1}$ together form $l.2^{y+1}/\pi_c$ new column clusters $\gamma^{y+1}_{i,j}$

end

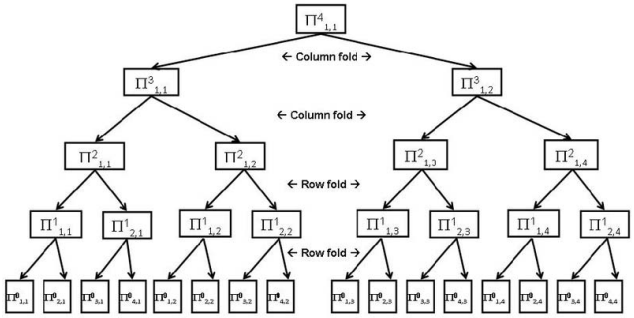


Fig. 2. Hierarchical Co-clustering - Computation Tree

TABLE I
NOTATION

Symbol	Definition
P_0	Total number of nodes for computation
c	Number of threads (cores) per node
(m, n)	Number of rows and columns in the input matrix
s	Sparsity factor of the matrix
(k, l)	Number of row and column clusters
(m/k)	Average number of rows per row cluster
n/l	Average number of columns per column cluster
B_0	Interconnect Bandwidth for AllReduce/Allgather
S_0	Setup cost for AllReduce/Allgather

V. Time Complexity Analysis

In this section, we establish theoretically, the performance and scalability advantage of our optimized distributed hierarchical algorithm. We look at performance of the hierarchical algorithm as compared to the flat algorithm. Refer notation given in Table V.

The time complexity analysis of the flat distributed co-clustering algorithm using MPI + OpenMP (hybrid) is given in detail in [11]. This approach is similar to the MPI only flat algorithm, but additionally exploits communication and compute overlap using multi-core cluster architectures. Thus, the overall time complexity for the flat hybrid distributed co-clustering algorithm, per iteration, is given by:

$$T_h(m, n, P_0, k, l) = mn/P_0 * c + 2 * (mn/B_0) * log(P_0) + S_0 + 3 * mns * (k + l)/(P_0 * c)) \quad (2)$$

A. Analysis of Hierarchical Algorithm

For the parallel hierarchical co-clustering algorithm, we consider 2 way merge at each level, i.e. a binary tree (for sake of simplicity) with Z levels of computation. In case of the binary tree, the base level, l_0, has 2^Z partitions each of size G_0 nodes (processors) such that $G_0 = P_0/(mn)$. At each level, l_z, $z \in [0..Z-1]$, the k' row clusters and l' column clusters from two previous level partitions are merged to form a new initial set of k'' row clusters and l'' column clusters for the next level partition. In the hierarchical computation, first the row to row cluster assignment and the column to column cluster assignment iterations are performed at a level, l_z. The time required for these iterations depends upon the size of the sub-matrix handled by each partition at that level, the number of clusters k' and l', as well as the number of nodes in the partition at that level. The total number of levels in the binary tree of hierarchical computations is given by:

$$Z = \log(\pi_r) + \log(\pi_c) \quad (3)$$

We consider separately, the cost for iterations at each level and the merge overhead to go from one level to the next. For sake of simplicity, we assume that all row-folds (with levels referred to as x, $x \in [0..\log(\pi_r) - 1]$) happen before the col-folds (with levels referred to as y, $y \in [0..\log(\pi_c) - 1]$).

The cost of iterations during the row-fold at each level,

$$T_{(flat)}(m.2^x/\pi_r, n/\pi_c, G_0.2^x, k_x, l/\pi_c), \tag{4}$$

where, $k_x = k.2^x/\pi_r$ (flat hybrid equation). Similarly, the cost of iterations during col-fold at each level, referred to here as,

$$T_{(flat)}(m, n.2^y/\pi_c, P_0.2^y/\pi_c, k, l_y), \tag{5}$$

where $P_0 = G_0 * \pi_r * \pi_c and l_y = l.2^y/\pi_c$. For merge compute and communication cost, let us consider row-fold based merge between two partitions of level, x, to create a new partition at level, $x+1$, and its initial row and column clusters. Here, communication takes place between the nodes of the two partitions at level x to share the row cluster and column cluster mapping. The time for this is given by : $O(S_0 + 2(k_x + l\pi_c) * \log(2^x.G_0)/B_0)$. Then the nodes perform maximum weight bipartite matching between row clusters of the two partitions and also between column clusters of the two partitions. Since, this matching effort is equally distributed across the nodes (and cores within the nodes) of the two partitions, this compute time is given by: $O((k_x + l/\pi_c)/(G_0 * c))$. Once, the merge happens, the assignment of rows to the row clusters and columns to the column clusters is done by each node. The time for this is given by : $O((1/2G_0.c) * ((m.2^x/\pi_r) + n/\pi_c))$. Let $\alpha = \log(\pi_r)$ and $\beta = \log(\pi_c)$. The merge time for row based merge between two partitions at level,x, $0 \leq x \leq (\log(\pi_r) - 1)$, is given by (assuming compute time dominates):

$$\begin{aligned} T_{(r\_merge)}(m.2^x/\pi_r, n/\pi_c, G_0.2^x) = \frac{(k_x + l/\pi_c)}{(G_0 * c)} \\ + (\frac{1}{2G_0.c} * (\frac{m.2^x}{\pi_r} + \frac{n}{\pi_c})) \end{aligned} \tag{6}$$

Similarly, the merge time for column based merge between two partitions at level,$\alpha + y$, $0 \leq y \leq (\beta - 1)$, is given by (assuming compute time dominates):

$$\begin{aligned} T_{(c\_merge)}(m, n.2^y/\pi_c, G_0.\pi_r.2^y) = \frac{(k + l_y)}{(G_0 * \pi_r.2^y.c)} + \\ ((1/2G_0.\pi_r, 2^y.c) * (m + (n.2^y)/\pi_c)) \end{aligned} \tag{7}$$

The total time in the hierarchical computation is given by the time for all row folds T_{row_fold} plus the time for all column folds T_{col_fold}.

$$\begin{aligned} T_{(hier)} &= T_{row\_fold} + T_{col\_fold} \\ T_{row\_fold} &= O(\sum_{x=0}^{\log(\pi_r - 1)} (\frac{(k/\pi_r.2^x + l/\pi_c).m/\pi_r.n/\pi_c.2^x.s}{\frac{P_0.2^x}{\pi_r.\pi_c}})) \\ T_{col\_fold} &= O(\sum_{x=0}^{\log(\pi_c - 1)} (k + \frac{l.2^x}{\pi_c}).\frac{m.n.s}{P_0}) \end{aligned} \tag{8}$$

The total time over all row folds is given by:

$$T_{row\_fold} = O((\frac{k.(\pi_r - 1)}{\pi_r} + \frac{l.\log(\pi_r)}{\pi_c}).\frac{m.n.s}{P_0}) \tag{9}$$

Similarly using T_{col_fold} can be written as:

$$T_{col\_fold} = O((k.\log(\pi_c) + \frac{l.(\pi_c - 1)}{\pi_c}).\frac{m.n.s}{P_0}) \tag{10}$$

Substituting the expression for T_{row_fold}, T_{col_fold} from equation (9), and simplifying equation (10), and assuming the communication cost is low, we get:

$$\begin{aligned} T_{hier} = O(((k.\log(\pi_c) + \frac{l.(\pi_c - 1)}{\pi_c}) \\ + (\frac{k.(\pi_r - 1)}{\pi_r} + \frac{l.\log(\pi_r)}{\pi_c})).\frac{m.n.s}{P_0}) \end{aligned} \tag{11}$$

Now combining results from (11) & (3) and making k=l=C, π_r=π_c=π we get T_{hier} as:

$$T_{hier} = O(\frac{(2.(1 - \frac{C}{\pi}) + C.(\frac{\log(\pi)}{\pi} + 1)).\frac{m.n.s}{P_0}}{2.\log(\pi)}) \tag{12}$$

Hence by doing similar replacement in T_{flat} as above we get :

$$\begin{aligned} T_{flat} &= O(\frac{C.m.n.s}{P_0}) \\ \frac{T_{hier}}{T_{flat}} &= O(\frac{1}{\log(\pi)}) \end{aligned} \tag{13}$$

Equation (13) demonstrates that the distributed hierarchical algorithm performs better than the distributed flat algorithm. In real experiments, the compute and communication merge overheads lead to lesser gain. One can use the above performance model (equation (11)) to compute the optimal values of π_r, π_c and Z. We skip this analysis for brevity.

VI. Load Balancing Optimization

Since the input matrix is highly sparse, one needs to perform load-balancing to achieve the maximum parallel efficiency on large scale parallel systems. We model the load balancing problem as an Integer Linear Program (ILP) for both flat and hierarchical distributed algorithms. This can be used for both static load balancing as well as dynamic load balancing in case of online co-clustering / collaborative filtering algorithms. In the distributed flat algorithm, we need to ensure that each processor has equal compute load based on the rows and columns assigned to that processor. Formally, this problem is related to the *k-partition* problem that is known to be NP-hard.

However, approximation algorithms can be used obtain a good load balanced data distribution for the flat distributed CF algorithm. We employed greedy row and column movement heuristic to ensure good balancing for the flat algorithm. The flat load balancing algorithm works in iterations. In each iteration the total row and column load on each processor, CL_p is computed and using all-reduce this information is obtained at each processor. Then, a matching is computed between processors with heavy loads and processors with light load. After this, the processor with high load sends a certain number of heavy rows and columns to its *matched* processor with low load. The selection of rows and columns to send is made to ensure that these two matched processors end up with similar load after their communication. These iterations are repeated till the overall load imbalance in the system is below a certain threshold.

In the hierarchical algorithm one needs to ensure load balance across the partitions at each level of the computation hierarchy. Performing this *forward-looking* load balancing for all levels in the beginning (at the leaf level) itself will ensure high parallel efficiency at all levels of execution. This can be viewed as a *multi-level* k-partitioning problem. At each level,

the problem is similar (with a small difference) to the flat case, i.e. *k-partition* problem). This *multi-level* k-partition problem is NP-hard since it a generalization of the *k-partition* problem. Further, our problem has additional constraints which makes it computationally challenging. We use similar heuristic as for the flat algorithm at levels close to the leaf of the tree since it at these levels that the load balance leads to severe impact on performance.

A. Online Hierarchical Co-clustering Algorithm

Algorithm 3 presents the hierarchical online distributed co-clustering algorithm. In the online algorithm, the row and column clusters at various levels are updated on the hierarchical computation tree, as a collection of row/column updates in the input matrix come along. The clusters at the top most level and the corresponding averages A_{gh}^{COC},A_{gj}^{RC},A_{ih}^{CC} at that level are hence maintained which can be used for the purpose of prediction later. The key challenge in the online hierarchical algorithm, is to calculate the changed co-clustering for the partition $\Pi_{y+1}^{i,j}$ at level $y+1$, using the updates at the level y in an efficient manner along the hierarchical computation tree. In order to acheive this, the changes in clusters at level y are propagated to level $y+1$ while utilizing the history of the previous clustering at level $y+1$. This reduces the number of cluster assignment iterations to reach an optimal co-clustering at level $y+1$.

The propagation of changes from level y to level $y+1$ is done in the following manner (refer Algorithm 3. Let us assume we have the optimal local clustering for the partitions at level y. We need to propagate the information in this changed assignment of row clusters (say K_y) to the old row clusters at level $y+1$, K_{y+1}. Now, we take each row r at level y that is affected by the updates to the ratings in that partition and hence changed to a cluster R_y in K_y. Now, the assignment of r in K_{y+1} is determined as follows. Find that cluster $R_{y+1} \in K_{y+1}$ such that R^y has a maximal match with R_{y+1} (in terms of number of rows assigned to them), more than any other row cluster in K_{y+1}. Now at level $y+1$, the row r is assigned to this row cluster R_{y+1} initially. In this fashion, one round of initial row assignment updates is done for all the changed rows in the partition $\Pi_{y+1}^{i,j}$ by propagating the changes from the previous level while using the history at this level. Then, a few rounds of flat row cluster assignment update iterations are run to reduce the error in the divergence function chosen. For reducing the run time further, these iterations at lower levels of the hierarchical tree (closer to the leaves) can be skipped, since the number of changes within a partition at a lower level maybe so less that just a reassignment of clusters by propagating the cluster updates is enough to maintain the clusters and ensure good accuracy. In this case, we will significantly reduce the algorithm execution time while not loosing much on the accuracy of cluster assignments at lower levels. Further, successive chunks (collections) of updates can proceed in parallel by carefully, allocating and multiplexing the cores in each node to process a chunk (collection) of updates. Thus, at any point of time, multiple updates can proceed in parallel up (from leaf to the root) along the hierarchical computation tree in a *pipelined* fashion. This pipelined parallelism leads to soft real-time online CF performance by enabling higher utilization of the underlying compute nodes in the system.

VII. Results & Analysis

The hybrid flat and hierarchical distributed algorithms were both implemented using MPI and OpenMP. The *Netflix Prize* dataset ($100M$ training ratings and $1.5M$ validation ratings on a scale of 1..5, over $480K$ users and $17K$ movies), and, *Yahoo KDD Cup* ($252M$ training ratings and $4M$ validation ratings on a scale of 1..100, over $1M$ users and $624K$ songs) datasets were used to evaluate and compare the performance and scalability of these distributed algorithms.

Platform: The experiments were performed on the Blue gene/P (MPP) architecture. Each node in Blue Gene/P is a quad-core chip with frequency of 850 MHz having 2 GB of DRAM, 32 KB of L1 instruction and data caches per core, 2KB prefetch buffer (L2 cache) and 8MB of L3 cache. Blue Gene/P has 3D torus interconnect with 3.4 Gbps bandwidth in each of the six directions per node along with separate collective and global barrier networks. MPI was used across the nodes for communication while within each node OpenMP was used to parallelize the computation and communication amongst the four cores.

For all the experiments, we obtained RMSE in the range 0.87 ± 0.02 on the Netflix validation data and RMSE in the range 26 ± 4 on the Yahoo KDD Cup data. The sequential implementation 1 and the flat distributed algorithm [11] obtains similar accuracy for both datasets. Below, k refers to the number of row clusters ($k = 16$ for Netflix, $k = 20$ for Yahoo KDD Cup) generated while l refers to the number of column clusters ($l = 16$ for Netflix, $l = 20$ for Yahoo KDD Cup) generated. For all the experiments we used the *C6* constraints (refer section III).

A. Scalability Analysis

We present the strong, weak and data scalability analysis including the *training phase* and the *prediction phase* for I-divergence with Netflix dataset and for Euclidean divergence with Yahoo KDD Cup dataset.

1) Strong Scalability: Fig. 3(a) compares the strong scalability curves of the hierarchical algorithm and the flat algorithm. The hierarchical algorithm with load balancing (*Hier-lb*) has better performance of around $2\times$ ($77s$ vs $144s$ at 64 nodes) to $4\times$ ($9.38s$ vs $38.2s$ at 4096 nodes) over the flat algorithm. This gap increases with increasing number of nodes as the hierarchical algorithm has better load balance across the nodes along with lower communication time, while achieving the same accuracy as flat (0.87 ± 0.02 RMSE). This is a very desirable property for massive scale analytics and comes from the novel hierarchical design of our algorithm. This also demonstrates soft real-time training ($9.38s$) performance for the full Netflix dataset even with the computationally expensive I-divergence objective. In the hierarchical algorithm, as the number of nodes increases by $64\times$, from 64 to 4096, the time decreases by $8.2\times$ (from $77s$ to $9.38s$). The prediction time was $0.7s$ for $1.4M$ ratings. This gives an average prediction time of $0.5\mu s$ per rating using $4K$ nodes. Fig. 4(a) illustrates the performance gain of the hierarchical algorithm over the flat algorithm for Euclidean-divergence with the Yahoo KDD Cup dataset. The hierarchical algorithm consistently performs better than the flat by around $4\times$ ($61s$ vs $267s$ at 64 nodes and $11.85s$ vs $51s$ at 4096 nodes). This also demonstrates soft real-time training performance ($13.28s$) for the full Yahoo KDD Cup data. Because of the fundamental advantage of lesser overall compute requirement and lesser load imbalance and communication cost (while giving the same accuracy 26 ± 4

Algorithm 3 Hierarchical Online Distributed Coclustering update algorithm

Input: Original Matrix A, Updated Ratings Matrix U, Previous Hierarchical Co-clusters ($\rho_y^{i,j}$,$\gamma_y^{i,j}$) for each partition $\Pi_y^{i,j}$ ($1 \leq y \leq Y_0$, $1 \leq i \leq m_1$, $1 \leq j \leq n_1$) and averages A_{gh}^{COC},A_{gj}^{RC},A_{ih}^{CC} at level Y_0

Output: Updated optimal co-clustering ($\rho_y^{i,j}$,$\gamma_y^{i,j}$) and averages A_{gh}^{COC},A_{gj}^{RC},A_{ih}^{CC} at level Y_0.

Method:

1. Before starting the iterations, update the $m \times n$ matrix A with changes from U. 2. Following the hierarchical structure, at level y, Partition $\Pi_y^{i,j}$ ($1 \leq i \leq m_1$, $1 \leq j \leq n_1$), as shown in the figure gets $\frac{m}{m_1}$ rows (i.e, a $\frac{m}{m_1} \times n$ submatrix A_i^R) and $\frac{n}{n_1}$ columns (i.e, a $m \times \frac{n}{n_1}$ submatrix A_j^C) where $m_1 = 2^{\lfloor \frac{y}{2} \rfloor}$ and $n_1 = 2^{\lceil \frac{y}{2} \rceil}$.
Each node p in this partition gets $\frac{m}{m_1 * G_0}$ rows and $\frac{n}{G_0 * n_1}$ columns where G_0 is the number of nodes in the partition. At any level y, to each partition $\Pi_y^{i,j}$ only the $\frac{m}{m_1} \times \frac{n}{n_1}$ submatrix $A_{i,j}$ is visible/read by the nodes in the partition. This matrix is coclustered into $\frac{k}{m_1}$ row clusters and $\frac{l}{n_1}$ column clusters.

while $(y++) \leq Y_0$ **do**

3. Update the locally optimum coclustering ($\rho_y^{i,j}$,$\gamma_y^{i,j}$) for the updated submatrix $A_{i,j}$ at each partition $\Pi_y^{i,j}$

4. **If** y **is odd**,

Fold along rows: Partition $\Pi_y^{2i,j}$ merges with partition $\Pi_y^{2i+1,j}$ in the following manner to form $\Pi_{y+1}^{i,j}$.

1a. Each row r that updated its cluster in $\rho_y^{2i,j}$ or $\rho_y^{2i+1,j}$ at level y does a maximal match of that cluster with one of the old clusters $\rho_{y+1}^{i,j}$ at level $y+1$ and joins it, eventually leading to k_1 changed Row clusters $\rho_{y+1}^{i,j}$.

1b. Each column c that updates its cluster in $\gamma_y^{2i,j}$ or $\gamma_y^{2i+1,j}$ at level y does a maximal match of that cluster with one of the old clusters $\gamma_{y+1}^{i,j}$ at level $y+1$ and joins it, eventually leading to l_1 changed Column clusters $\gamma_{y+1}^{i,j}$.

1c. Some flat row and column update iterations are run if $y < H$ before proceeding to the next level and making $m_1 = \frac{m_1}{2}$

else If y **is even**,

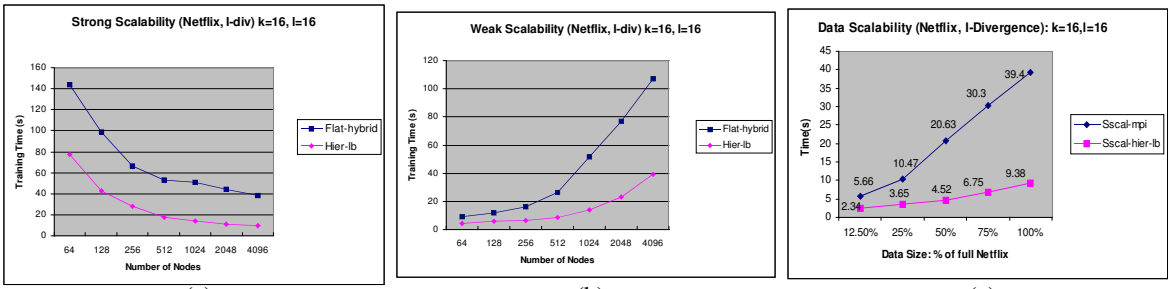
Fold along Columns: Partition $\Pi_y^{i,2j}$ merges with partition $\Pi_y^{i,2j+1}$ in the following manner to form $\Pi_{y+1}^{i,j}$.

1a. Each row r that updated its cluster in $\rho_y^{i,2j}$ or $\rho_y^{i,2j+1}$ at level y does a maximal match of that cluster with one of the old clusters $\rho_{y+1}^{i,j}$ at level $y+1$ and joins it, eventually leading to k_1 changed Row clusters $\rho_{y+1}^{i,j}$.

1b. Each column c that updates its cluster in $\gamma_y^{i,2j}$ or $\gamma_y^{i,2j+1}$ at level y does a maximal match of that cluster with one of the old clusters $\gamma_{y+1}^{i,j}$ at level $y+1$ and joins it, eventually leading to l_1 changed Column clusters $\gamma_{y+1}^{i,j}$.

1c. Some flat row and column update iterations are run $y < H$ before proceeding to the next level and making $n_1 = \frac{1}{2}$

end

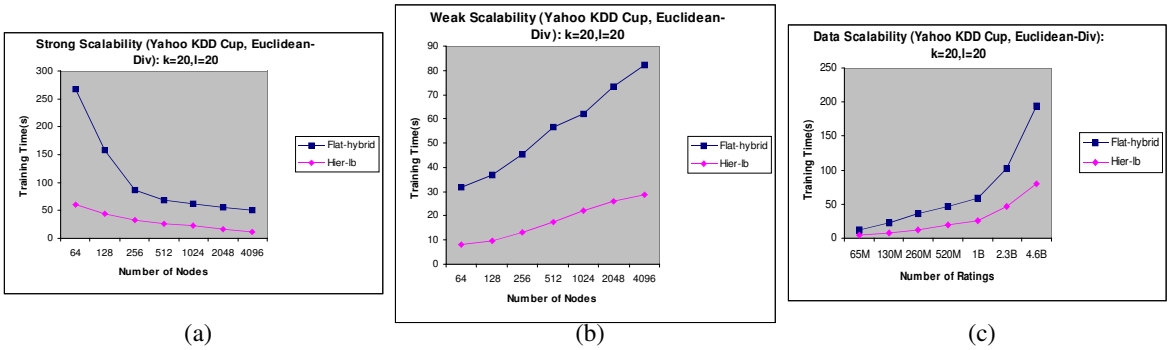


(a) (b) (c)

Fig. 3. Netflix (I-div/C6): (a) Strong Scalability. (b) Weak Scalability. (c) Data Scalability

RMSE) as compared to the flat algorithm, the hierarchical algorithm achieves better performance and hence is ideally suited for massive scale analytics. The prediction time was $3.2s$ for $4M$ ratings. This gives an average prediction time of $0.8\mu s$ per rating. The parallel efficiency here is lower than the Netflix data since the Yahoo data has much higher sparsity and hence load imbalance, but it can be further improved by fine tuning the load balance further as well as optimizing the merge phase in the hierarchical algorithm.

2) Weak Scalability: Fig. 3(b) compares the weak scalability curves for hierarchical algorithm and the flat algorithm, using I-divergence based co-clustering with C6 constraints. As the number of nodes (P_0) increases from 64 to 4096 and the training data increases from 6.25% to 400% (400M ratings) of the full Netflix dataset (with $k = 16$, $l = 16$), the total training time for the hierarchical algorithm increases by around $8.7\times$ ($4.5s$ to $39s$), while that for the flat algorithm increases by $11.87\times$ ($9s$ to $107s$), thus demonstrating better weak scalability. Further, the hierarchical algorithm performs consistently better compared to the flat algorithm, around $2\times$ ($4.5s$ vs $9s$) with 64 nodes and $2.7\times$ ($39s$ vs $107s$) at 4096 nodes. Fig. $4(b)$ demonstrates the weak scalability of the hierarchical algorithm for Euclidean divergence with Yahoo KDD Cup dataset: with $64\times$ increase in the data ($16.25M$ to $1B$ ratings) and number of nodes (64 to 4096), the training time only increases by $3.5\times$ ($8.15s$ to 28.5). Further, the hybrid algorithm performs consistently better than the flat algorithm, $3.9\times$ ($8.15s$ vs $32s$) at 64 nodes and $2.9\times$ ($28.5s$



(a) (b) (c)

Fig. 4. Yahoo-KDD (Euclidean/C6): (a) Strong Scalability. (b) Weak Scalability. (c) Data Scalability

vs $82.4s$) at 4096 nodes.

3) Data Scalability: Fig. 3(c) compares the data scalability curves of the hierarchical algorithm and the flat algorithm. As the training data increases from $13M$ to $900M$ (using replication of Netflix dataset), while $P_0 = 4096$, the training time for the hierarchical algorithm increases by $34\times$ ($1.87s$ to $64s$) which is much lesser than that of the flat algorithm increases by $48\times$. This demonstrates better than linear data scalability of the hierarchical algorithm and better data scalability over the flat algorithm. Further, the hierarchical performs consistently better than the flat algorithm, $2.24\times$ at $13M$ ratings ($1.87s$ vs $4.2s$) and $3.2\times$ at $900M$ ratings ($64s$ vs $203s$). Moreover, this gap increases with increasing input size of the data, that makes the hierarchical algorithm attractive for massive scale data. Fig. 4(c) compares the data scalability curves for the hierarchical and the flat algorithm on the Yahoo KDD Cup dataset (with $P_0 = 4096$, Euclidean divergence/C6). The hierarchical algorithm demonstrates better than linear data scalability ($21\times$ increase in time with $64\times$ increase in data from $65M$ ratings to $4.6B$ ratings). It performs better than the flat algorithm by $3.15\times$ at $65M$ ratings ($3.8s$ vs $12s$) and $2.4\times$ at $4.6B$ ratings ($80s$ vs $194s$). On $1B$ as well as for $2.3B$ ratings, the hierarchical algorithm achieves soft real-time performance, $25s$ and $47s$ respectively.

B. Detailed Scalability Comparison

In this section we present detailed comparison of the gains obtained by the hierarchical algorithm and load balancing. Fig. $5(a)$ presents the curves for strong scalability for the flat hybrid algorithm, the hybrid flat load balanced algorithm and the hierarchical load balanced algorithm. The hybrid flat load balanced algorithm performs around $2\times$ better than the flat hybrid algorithm and this gap increases with increasing number of nodes. This is because at $P_0 = 64$, the flat hybrid algorithm is able to utilize the 4 threads per node efficiently, while also being able to effectively overlap computation with communication. However, at higher values of P_0, the load imbalance problem dominates its overall throughput. Hence, its performance degrades w.r.t the load balanced flat algorithm by $2\times$ at $P_0 = 1024$, and its speedup is only $2.9\times$ over $16\times$ increase in the number of nodes. The hybrid flat load balanced algorithm eliminates this problem by making sure that each node roughly processes the same number of entries. Hence, the hybrid flat load balanced algorithm, achieves $3.6\times$ speedup over $16\times$ increase in the number of nodes. The hierarchical algorithm further demonstrates an additional $2\times$ performance over the flat load balanced algorithm at $P_0 = 1024$ and an improvement in speedup to $7.2\times$ with $16\times$ increase in the number of nodes.

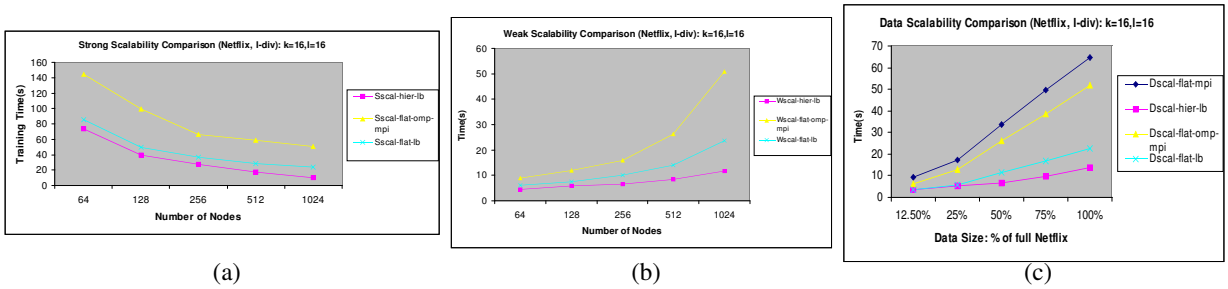
Fig. $5(b)$ presents the comparison curves for weak scalability. Here, the hybrid flat algorithm incurs $5.6\times$ increase in time with $16\times$ increase in data and number of nodes, and the flat load balanced algorithm incurs $3.95\times$ increase in time; while the hierarchical algorithm incurs only $2.6\times$ increase in time. This can be attributed to better efficiency in the hierarchical algorithm as compared to the flat algorithm even with load balance. Further, the hierarchical algorithm has consistently superior performance over the hybrid flat load balanced algorithm by around $2\times$ (at 1024 nodes); while the flat load balanced algorithm has around $2\times$ performance over the flat hybrid algorithm owing to its better work distribution amongst the nodes. Fig. $5(c)$ presents the comparison curves for data scalability. The hybrid flat load balanced algorithm achieves gain to $1.8\times$ at $P_0 = 64$ and $2.15\times$ at $P_0 = 1024$ over the flat hybrid algorithm. Further, the hybrid flat load balanced algorithm improves the overall data scalability over the flat hybrid algorithm ($6.6\times$ increase in time overall vs $14.6\times$ for flat hybrid). The hierarchical algorithm further improves the data scalability by achieving only $3.2\times$ overall increase in time with $16\times$ increase in data size.

C. Online Algorithm: Performance Analysis

Fig. $6(a)$ illustrates the performance gain of the hierarchical online algorithm over the baseline MPI online algorithm for Euclidean-divergence with 4% change in data. The hierarchical online algorithm performs better than the baseline by around $3.64\times$ to $7\times$. This also demonstrates soft real-time online training performance ($2.81s$) of our algorithm. The parallel efficiency can be further improved here, by having better load balance. Further, the hierarchical algorithm involves node to node communication during the merge phase. This leads to increase in the communication time, leading to decrease in parallel efficiency.

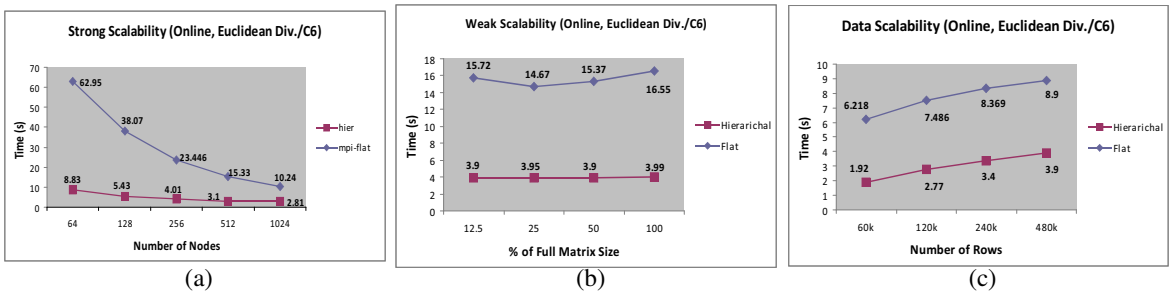
Fig. $6(b)$ illustrates the weak scalability of the online hierarchical algorithm for Euclidean divergence with 4% incremental change in the Netflix dataset. Here, with $16\times$ increase in the data (matrix size) and number of nodes, the training time remains pretty much the same. Further, the hierarchical online algorithm performs consistently better than the online baseline algorithm by around $4\times$.

Fig. $6(c)$ compares the data scalability curves for the online hierarchical and the online baseline algorithm with $P_0 = 1024$, Euclidean divergence/C6 and 4% incremental change in the Netflix dataset. The online hierarchical algorithm demonstrates



(a) (b) (c)

Fig. 5. Detailed Comparison(Netflix): (a) Strong Scalability. (b) Weak Scalability. (c) Data Scalability



(a) (b) (c)

Fig. 6. Netflix Online(Euclidean/C6): (a) Strong Scalability. (b) Weak Scalability. (c) Data Scalability

linear data scalability ($2\times$ time increase with $8\times$ increase in data (number of rows)) and performs better than the baseline algorithm by around $2.3\times$ to $3.24\times$.

VIII. Conclusions & Future Work

Real-time co-clustering and collaborative filtering with high prediction accuracy are computationally challenging problems. We have presented a novel hierarchical approach for both offline and online distributed co-clustering and collaborative filtering along with theoretical analysis of parallel time complexity. Soft real-time performance and superior scalability of our algorithm has been demonstrated experimentally using Netlfix and Yahoo KDD Cup datasets on multi-core cluster architecture. Our hierarchical algorithm outperforms all known prior results for collaborative filtering while maintaining high accuracy. In future, we intend to investigate theoretical analysis of convergence for this algorithm.

References

[1] P. Resnick and H. R. Varian, "Recommender systems - introduction to special section," *Comm. ACM*, vol. 40, no. 3, pp. 56–58, 1997.

[2] B. M. Sarwar, G. Karypis, J. A. Konstan, and J. Riedl, "Analysis of recommendation algorithms for e-commerce," in *ACM Conference on Electronic Commerce*, 2000, pp. 158–167.

[3] J. B. Schafer, J. A. Konstan, and J. Riedi, "Recommender systems in e-commerce," in *ACM Conference on Electronic Commerce*, 1999, pp. 158–166.

[4] R. G. Pensa and J.-F. Boulicaut, "Constrained co-clustering of gene expression data," in *SDM*, 2008, pp. 25–36.

[5] S. Hassan and Z. Syed, "From netflix to heart attacks: collaborative filtering in medical datasets," in *International Health Informatics Symposium (IHI)*, 2010, pp. 128–134.

[6] N. Ampazis, "Collaborative filtering via concept decomposition on the netflix dataset," in *ECAI*, 2008, pp. 143–175.

[7] N. Srebro and T. Jaakkola, "Weighted low rank approximation," in *Twentieth International Conference on Machine Learning*, 2003, pp. 720–728.

[8] J. S. Breese, D. Heckerman, and C. Kadie, "Empirical analysis of predictive algorithms for collaborative filtering," in *Fourteenth International Conference on Uncertainty in Artificial Intelligence*, 1998, pp. 43–52.

[9] T. George and S. Merugu, "A scalable collaborative filtering framework based on co-clustering," in *Fifth International Conference on Data Mining*, 2005, pp. 625–628.

[10] 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 *15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2009, pp. 1115–1124.

[11] A. Narang, A. Srivastava, and P. Katta, "Distributed scalable collaborative filtering algorithm," in *EuroPar 2011*, France, 2011.

[12] K. Kummamuru, A. Dhawale, and R. Krishnapuram, "Fuzzy co-clustering of documents and keywords," in *IEEE International Conference on Fuzzy Systems*, 2003.

[13] A. de Spindler, M. C. Norrie, M. Grossniklaus, and B. Signer, "Spatio-temporal proximity as a basis for collaborative filtering in mobile environments," in *UMICS*, 2006.

[14] A. Banerjee, S. Basu, and S. Merugu, "Multi-way clustering on relation graphs," in *SDM*, 2007.

[15] A. Banerjee, I. Dhillon, J. Ghosh, S. Merugu, and D. S. Modha, "A generalized maximum entropy approach to bregman co-clustering and matrix approximation," *Journal of Machine Learning Research*, vol. 8, no. 1, pp. 1919 – 1986, Aug. 2007.

[16] B. Sarwar, G. Karypis, J. Konstan, and J. Reidl, "Application of dimensionality reduction in recommender systems: a case study," in *WebKDD Workshop*, 2000.

[17] M. Brand, "Fast online svd revisions for lightweight recommender systems," in *SIAM International Conference on Data Mining*, 2003, pp. 37–48.

[18] C. N. Ziegler, S. M. McNee, J. A. Konstan, and G. Lausen, "Improving recommendation lists through topic diversification," in *Fourteenth International World Wide Web Conference*, 2005.

[19] K.-W. Hsu, A. Banerjee, and J. Srivastava, "I/o scalable bregman co-clustering," in *Proceedings of the 12th Pacific-Asia conference on Advances in knowledge discovery and data mining*, 2008.

[20] B. Kwon and H. Cho, "Scalable co-clustering algorithms," *Algorithms and Architectures for Parallel Processing, Lecture Notes in Computer Science*, vol. 6081, pp. 32–43, 2010.

[21] I. S. Dhillon and D. S. Modha, "Concept decompositions for large sparse text data using clustering," in *Machine Learning*, 1999, pp. 143–175.

[22] G. H. Golub and C. F. V. Loan, *Matrix computations*. Baltimore, MD, USA: The Johns Hopkins University Press, 1996.

[23] A. Narang, R. Gupta, V. Garg, and A. Joshi, "Highly scalable parallel collaborative filtering algorithm," in *IEEE International Conference on High Performance Computing*, Goa, India, 2010.

[24] I. Dhillon, S. Mallela, and D. Modha, "Information-theoretic co-clustering," in *Proceedings of the 9th International Conference on Knowledge Discovery and Data Mining*, 2003, pp. 89–98.