RI 11019 November, 2011 Computer Science

IBM Research Report

High Performance Distributed Co-clustering and Collaborative Filtering

Ankur Narang, Abhinav Srivastava

IBM Research Division
IBM India Research Lab
4, Block C, Institutional Area, Vasant Kunj
New Delhi - 110070. India.

Naga Praveen Kumar Katta
Department of Computer Science
Princeton University
35 Olden Street,Princeton
New Jersey - 08540. USA.

IBM Research Division
Almaden - Austin - Beijing - Delhi - Haifa - T.J. Watson - Tokyo - Zurich

High Performance Distributed Co-clustering and Collaborative Filtering

Ankur Narang, Abhinav Srivastava
{annarang, abhin122}@in.ibm.com
IBM India Research Laboratory, New Delhi, INDIA

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

Abstract

Petascale Analytics is a hot research area both in academia and industry. It envisages processing massive amounts of data at extremely high rates and generating new scientific insights along with 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 analytic kernels that help in achieving these aims. Real-time CF and co-clustering on highly sparse massive datasets, while achieving a high prediction accuracy, is a computationally challenging problem. In this report, we present the novel designs for soft real-time (less than 1 *min.*) distributed co-clustering based Collaborative Filtering algorithm. Our distributed algorithms have been optimized for multi-core cluster architectures. Theoretical analysis of the time complexity of our algorithms prove the efficacy of our approach. Using Netflix [4] ($900M$ training ratings with replication) as well as the Yahoo KDD Cup [1] ($4.6B$ training ratings with replication) datasets , we demonstrate the performance and scalability of our flat and hierarchial algorithm on a 1024 & 4096-node multi-core cluster architecture respectively. Our hybrid distributed flat algorithm performs $2.1\times$ better than the previous flat algorithms on 1024 nodes of BlueGene/P while our batch mode hierarchical distributed algorithm (implemented using OpenMP with MPI) demonstrates around $4\times$ better performance (on Blue Gene/P) as compared to the our hybrid distributed flat algorithm [best prior (flat MPI+OMP based algorithm) work [21]], along with high accuracy (26 ± 4 RMSE for Yahoo KDD Cup data and 87 ± 0.02 for Netflix data). For online execution, we demonstrate around $3\times$ gain vs the online baseline MPI algorithm. To the best of our knowledge, these are the best known performance results for collaborative filtering, at high prediction accuracy, for multi-core cluster architectures.

1 Introduction

Petascale Analytics is a hot research area both in academia and industry. It aims at processing massive amounts (petabytes) of data at extremely high rates and generating new scientific insights in areas such as Theoretical Physics, Astronomy and Life Sciences; along with positive impact (for both users and providers) of industries such as E-commerce, Telecom, Finance, Life Sciences and so forth. Specifically, Collaborative filtering (CF) and Clustering algorithms are key fundamental kernels that help in achieving these aims. Their wide applicability in multitude of application domains has made it imperative for them to be considered for distributed optimizations at Petascale levels.

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 [23], [26]. 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 very 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 find more easily, items that they would like to purchase [27]. This enhances the users' experience which typically lead

[1] http://kddcup.yahoo.com/

to improvements in sales and revenue. Further, scientific disciplines such as Computational Biology and Personalized medicine (risk stratification) stand to gain immensely from CF [22], [14]. 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 for large Telco providers), Finance and other industries, there is a strong need to deliver soft real-time training for CF as it will lead to further increase in customer experience and revenue generation. Hence, soft real-time CF (with less than 1 *min.*) based recommender systems are very useful.

Typical approaches for CF include matrix factorization based techniques, correlation based techniques , co-clustering based techniques, and concept decomposition based techniques [1]. Matrix factorization [28] and correlation [6] 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. Co-clustering based techniques [12], [8] have better scalability but have not been optimized to deliver high throughput on massive data sets. Daruru et.al [8] 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 / cluster 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, low parallel efficiency (due to cache misses and low compute utilization) and communication overheads 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 *hydrid flat* approach for distributed co-clustering. We optimized our distributed algorithm using pipelined parallelism, compute communication overlap and communication optimizations (including topology mapping, steiner node for communication time reduction) for massively parallel multi-core cluster architectures such as Blue Gene/P [2]. In order to maintain high parallel efficiency, our algorithm makes compute vs. communication trade-offs at various phases of the algorithm. Analytical parallel time complexity analysis proves the scalability provided by our performance optimizations as compared to the naive MPI based approach that has been used in all prior implementations. We evaluated our parallel CF algorithm on the prestigious Netflix Prize data set [4].

To further optimize the parallel efficiency, we have designed a novel *hierarchical* algorithm for co-clustering. The hierarchical design of the algorithm helps to reduce the computation and communication performed in the algorithm; while maintaining nearly the same quality of output (RMSE). The hierarchical approach in clustering also provide opportunity for parameter free clustering [17]. Analytical parallel time complexity analysis proves the scalability provided of our algorithm as compared to the flat OpenMP+MPI based approach that is the best available in prior implementations. We evaluated our parallel CF algorithm on the following real datasets: (a) Prestigious Netflix Prize data set [4] (Training ratings: $100M$, Validation ratings: $1.5M$),and (b) Yahoo KDD Cup dataset (Track 1 - Training ratings: $252M$, Validation ratings: $4M$) [3]. Using replication of these datasets, we have evaluated our algorithm on $900M$ ratings of the Netflix data and $4.6B$ ratings from the Yahoo KDD Cup dataset.

Specifically, this paper makes the following key contributions:

- We present the design of a novel distributed co-clustering based Collaborative Filtering algorithm for soft real-time (less than 10 *sec.*) performance over highly sparse massive data sets on multi-core cluster architectures. Our algorithm involves performance optimizations such as pipelined parallelism, computation communication overlap and communication optimizations (including topology mapping and steiner nodes for communication cost reduction)

- Analytical parallel time complexity analysis, theoretically establishes the improvement in performance and scalability using our algorithm.

- We demonstrate soft real-time distributed CF using the Netflix Prize dataset on a 1024-node Blue Gene/P system. We achieved a training time of around $6s$ with the full Netflix dataset and prediction time of $2.5s$ on $1.4M$ ratings with RMSE (Root Mean Square Error) of 0.87 ± 0.02. Our algorithm also demonstrates high scalability for large number of nodes on MPP architectures.

[2] www.research.ibm.com/bluegene

[3] http://kddcup.yahoo.com/datasets.php

- We present the design of a novel distributed hierarchical co-clustering algorithm for soft real-time CF over highly sparse massive data sets on multi-core cluster architectures. Further, a novel load balancing approach has been formulated for the hierarchical algorithm.

- Analytical parallel time complexity analysis, establishes theoretically that our hierarchical design leads to improvement in performance ($O(\log(\pi))$ better where, π is the number of partitions of rows and columns of the input matrix) and scalability as compared to the flat MPI+OpenMP based algorithm [21].

- We demonstrate soft real-time parallel CF on the Netflix Prize and Yahoo KDD Cup datasets using a 4096-node multi-core cluster architecture (Blue Gene/P [4]). We achieved a training time (using I-divergence and C6, Section 3) 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 [21] for the same dataset. To the best of our knowledge, this is the highest known parallel performance at such high accuracy. Further, we have shown soft real-time performance for over $900M$ ratings from the Netflix dataset (with high accuracy) and $4.6B$ ratings from the Yahoo KDD Cup dataset (with high accuracy of 26 ± 4 RMSE). Our algorithm demonstrates strong, weak and data scalability for large number of nodes on multi-core cluster architectures.

2 Related Work

Co-clustering and Collaborative Filtering (CF) are fundamental data-mining kernels used in many application domains such as Information Retrieval [18], Telecom [9], Financial markets, Life Sciences [22]. [14] evaluates 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. [2] considers 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 [3] (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 [6] and matrix factorization [28]. 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 [25]) and Non-Negative Matrix Factorization (NNMF) based [28] 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 [5] partially alleviate this problem. But, since the effects of small updates are not localized, the update operations are not very efficient.

[12] 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. [12] 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 [5] and BookCrossing dataset [29] (269392 explicit rating(1-10) from 47034 users on 133438 books).

[4] www.ibm.com/bluegene

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

[8] uses a dataflow parallelism based framework (in Java) 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 1024 nodes of Blue Gene/P and $7\times$ to $10\times$ better training time and better prediction time along with high prediction accuracy (0.87 ± 0.02 RMSE). 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. [24] uses jointly derived neighbourhood interpolation weight instead of "global effect" for partitioning datasets into k clusters. Though the RMSE presented by their algorithm is better than Netlfix prize winners, they were only able to achieve 0.89 RMSE while our hierarchical algorithm was able to perform at RMSE of 0.87 for same Netflix datasets. [15] studies 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. [19] studies 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.

[1] presents results of collaborative filtering using *Concept decomposition* based approach. It has been empirically established [10] that the approximation power (when measured using the Frobenius norm) of concept decompositions is comparable to the best possible approximations by truncated SVDs [13]. However, [1] 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. [20] 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 [1] and around one of magnitude improvement compared to the parallel concept decomposition technique [20]. Our design presents a *flat* distributed co-clustering algorithm where all the processors in the system participate in one 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. It also presents theoretical parallel time complexity analysis of this *flat* and *hybrid* algorithm. Using Netflix dataset (100M ratings), it demonstrates the performance and scalability of the algorithm on 1024-node Blue Gene/P system: with training time of around 6s on the full Netflix dataset and prediction time of 2.5s on 1.4M ratings (1.78μs per rating prediction). We also bring up a novel hierarchical approach for distributed co-clustering along with load balancing optimizations leading to around $2\times$ improvement in performance as compared to *flat* algorithm [21]. 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 [21]. Further, we present detailed performance comparisons with much larger data, around $4.6B$ Yahoo KDD Cup ratings (as compared to $252B$ in [21]) and on 4096 Blue Gene/P system (as compared to 1024 in [21]).

3 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 [3] 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 [16]. 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. There are two key components in formulating a co-clustering problem: (i) choosing a set of critical co-clustering-based statistics of the original data matrix that need to be preserved, and (ii) selecting an appropriate measure to quantify the information loss or the discrepancy between the original data matrix and the compressed representation provided by the co-clustering. For example, in the work of Cheng [7], the row and column averages of each co-cluster are preserved and the discrepancy between the original and the compressed representation is measured in terms of the sum of element-wise squared deviation. In contrast, information-theoretic co-clustering [16], which is applicable to data matrices representing joint probability distributions, preserves a different set of summary statistics, that is, the row and column averages and the co-cluster averages. Further, the quality of the compressed representation is measured in terms of the sum of element-wise I-divergence.

In this section, we explain the Bregmann divergence based partitional co-clustering algorithm [3] and the matrix approximation strategy.

We start by defining Bregman divergences (Bregman, 1967; Censor and Zenios, 1998), which form a large class of well-behaved loss functions with a number of desirable properties.

Definition Let ϕ be a real-valued convex function of Legendre type (Rockafellar, 1970; Banerjee et al, 2005b) defined on the convex set $S \equiv dom(\phi)(\subseteq \mathbb{R}^d)$. The Bregman divergence $d_\phi : S \times ri(S) \longmapsto \mathbb{R}_+$ is defined as

$$d_\phi(z_1, z_2) = \phi(z_1) - \phi(z_2) - \langle z_1 - z_2, \nabla\phi(z_2)\rangle$$

where $\nabla\phi$ is the gradient of ϕ.

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,

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

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 $z1, z2 \in \mathbb{R}$, $d_\phi(z_1, z_2) = (z_1 - z_2)^2$.

3.1 Coclustering bases

Given a co-clustering (ρ, γ), Modha et al. discuss six co-clustering bases where each co-clustering basis preserves certain summary statistics on the original matrix. More precisely their definition of a co-clustering basis is of the form:

Definition A co-clustering basis C is a set of elements of Γ_2, that is, an element of the power set 2^{Γ_2}, which satisfies the following two conditions:
(a) There exist $\eta_1, \eta_2 \in C$ (with η_1 possible the same as η_2) such that $\hat{U} \in \eta_1$ and $\hat{V} \in \eta_2$
(b) There do not exist $\eta_1, \eta_2 \in C$, $\eta_1 \neq \eta_2$ such that η_2 is a sub-σ-algebra of η_1.

where

$$\Gamma_2 = \left\{\{U_\phi, V_\phi\}, \left\{U_\phi, \hat{V}\right\}, \{U_\phi, V\}, \left\{\hat{U}, V_\phi\right\}, \left\{\hat{U}, \hat{V}\right\}\right\} \cup \left\{\left\{\hat{U}, V\right\}, \{U, V_\phi\}, \left\{U, \hat{V}\right\}, \{U, V\}\right\}$$

Γ_2 determines the set of all summary statistics that one may be interested in preserving. A particular choice of an element of Γ_2 such as $\left\{\hat{U}, \hat{V}\right\}$, leads to an approximation scheme where the reconstruction matrix preserves all the

corresponding co-cluster means. In the above definition for a coclustering basis, condition (a) ensures that the approximation depends on the co-clustering while condition (b) ensures that for any pair η_1, η_2, the conditional expectation $E[Z|\eta_2]$ cannot be obtained from $E[Z|\eta_1]$. Then they show that there are only six possible co-clustering bases, each of which leads to a distinct matrix approximation scheme which are as follows: $C_1 = \{\{\hat{U}\}, \{\hat{V}\}\}$,
$C_2 = \{\{\hat{U}, \hat{V}\}\}$,
$C_3 = \{\{U\}, \{\hat{U}, \hat{V}\}\}$,
$C_4 = \{\{V\}, \{\hat{U}, \hat{V}\}\}$,
$C_5 = \{\{U\}, \{V\}, \{\hat{U}, \hat{V}\}\}$,
$C_6 = \{U, \hat{V}\}, \{\hat{U}, V\}\}$
Now given the choice of a particular basis (i.e, the statistics to be reserved by the approximation matrix), we need to decide on the "best" reconstruction $\hat{Z}$ for a given co-clustering (ρ, γ). Then the general co-clustering problem will effectively reduce to one of finding an optimal co-clustering $(\rho*, \gamma*)$ whose reconstruction has the lowest approximation error with respect to the original Z. It also proves that the possible co-clustering bases $(C1 \dots C6)$ 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 [3]. 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 $d_\phi(A_{ij}, \hat{A}_{ij}) = A_{ij} * \log(\hat{A}_{ij}/A_{ij}) - A_{ij} + \hat{A}_{ij}$

4 Distributed Flat Coclustering Algorithm

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.
  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

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 following distributed algorithm 2 leverages this inherent data parallelism. In algorithm 2 each node of the cluster gets equal number of rows and columns. (i.e, the corresponding submatrices are stored in sparse-row, sparse column format in the node's memory). In step 3., the row and column averages of all the rows/columns are available with every node before hand. In Step 5., all the nodes exchange their local row/column assignments with all the other nodes. And then each node calculates the k Row-cluster and l column cluster averages by itself. Then all the nodes communicate the $k \times l$ global cocluster averages to each other. As one can see, this algorithm needs three

Algorithm 2 Distributed (MPI) Static Training via Co-Clustering (C_6 Basis)

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. Node p gets m_p rows (i.e, a $m_p \times n$ submatrix A^{rp}) and n_p columns (i.e, a $m \times n_p$ submatrix A^{cp}) where $m_p = \frac{m}{p}$ and $n_p = \frac{n}{p}$.
2. Randomly initialize (ρ^p,γ^p)
3. Gather all the row and column sums/weights S_i^R, S_j^C, W_i^R, W_j^C $\forall i, j$ from the other nodes using MPI_Allgather.
4. Each node calculates **all** row and column averages $A_i^R = \frac{S_i^R}{W_i^R}$ and A_j^C.
while *RMSE value is converging,* ***Each node*** *does the following* **do**
5. Gather the **global** Row-cluster and column-cluster memberships (ρ,γ) by concatenating (ρ^p,γ^p) using MPI_Allgather.
6a. Calculate the contribution of the local rows and columns to the Row-cluster and Col-cluster sums/weights i.e, S_{gj}^{RC}, W_{gj}^{RC}, S_{ih}^{CC} and W_{ih}^{CC}.
6b. Do an All_reduce on above contributions and get the **global** Row-Cluster and Column-Cluster Averages A_{gj}^{RC},A_{ih}^{CC}.
7. Calculate the contribution of the local rows and columns to the co-cluster sums/weights i.e, S_p^{COC} and W_p^{COC}.
8. Do an All_reduce on above contributions and get the **global** co-cluster sums/weights S^{COC}, W^{COC}. Subsequently calculate A^{COC} using them.
2a. Compute averages A^{COC},A^{RC},A^{CC},A^{R} and A^{C}.
9a. Update all the local row cluster assignments ρ^p
$\rho^p(i) = \underset{1 \le g \le k}{\textbf{\textit{argmin}}} \sum_{j=1}^{n} w_{ij} d_\phi(A_{ij}, \hat{A}_{ij}),\ i \in A^{rp}$
9b. Update all the local column cluster assignments γ^p
$\gamma^p(i) = \underset{1 \le h \le l}{\textbf{\textit{argmin}}} \sum_{i=1}^{m} w_{ij} d_\phi(A_{ij}, \hat{A}_{ij}),\ j \in A^{cp}$
end

MPI communication 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. Hence, we have designed performance optimizations to reduce these communication time bottlenecks on multi-core cluster architectures.

4.1 Time Complexity Analysis for Flat Algorithm

In this section, we establish theoretically, the performance and scalability advantage of our optimized distributed flat algorithm. Refer notation given in Table 1. The flat algorithm described in section 1 has I iterations. In each iteration the rows are assigned to row clusters and columns are assigned to column clusters. Each iteration has multiple phases. In the first phase, all nodes communicate using all-reduce operation to aggregate row to row-cluster mapping information. This communication time is given by: $O(S_0 + (m/B_0) * log(P_0))$.
In the second phase, one core in each node is involved in all-reduce communication with the all other nodes to aggregate column to column-cluster mapping information. This communication time is given by: $O(S_0 + (n/B_0) * log(P_0))$. Simultaneoulsy, the remaining three cores in each node compute the row-cluster average for all the row-clusters. This compute time is given by: $O(m/(P_c - 1))$. Due to the compute-communication overlap in the second phase, the time for the second phase is given by: $\max[O(S_0 + (n/B_0) * log(P_0)), O(m/(P_c - 1)]$.
In the third phase, each node computes the column cluster average for all column clusters and also computes its contribution to all co-cluster averages. The time for this phase is given by: $O((n/P_c) + mns/(P_0 * P_c))$. In the fourth phase, one core per node performs all-reduce operation with all other nodes, to aggregate and determine the co-cluster average over all co-clusters. This communication time is given by: $O(S_0 + (kl/B_0) * log(P_0))$. Simultaneously, three cores per node compute the partial values required for making the decision about the assignment of the rows (that node contains) to the row-clusters and the assignment of the columns (that node contains) to the column clusters. The com-

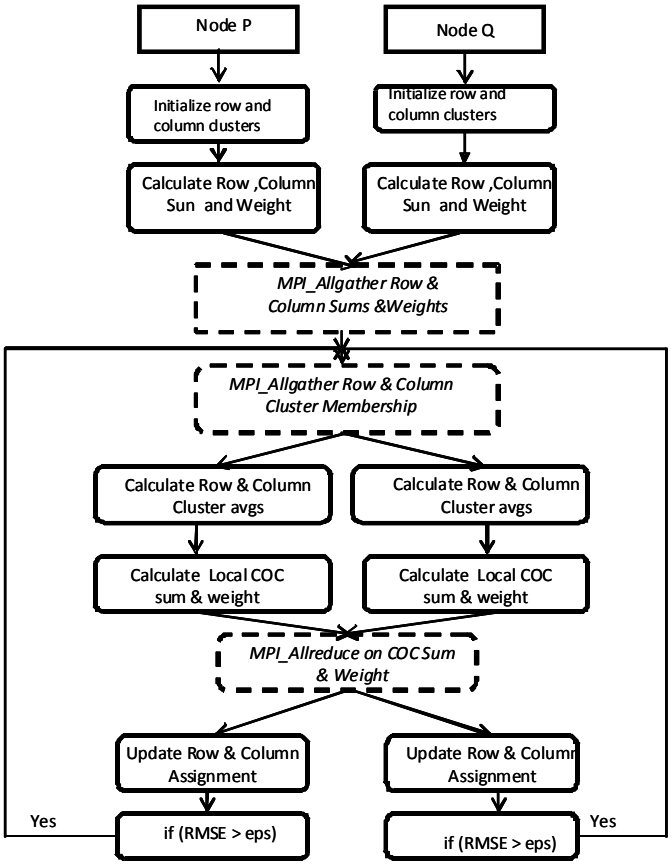


Figure 1: Flat Coclustering - MPI

putation cost varies depending on the divergence function - Euclidean, I-divergence or general Bregmann divergence. For each row assignment, any of the divergence functions involve computations over all populated columns, ns, per row and checking assignment to all possible k row clusters. Same applies for column cluster assignments. Hence, the compute time is given by: $O((mns/(P_0 \cdot (P_c - 1))) * (k + l)))$. Due to compute communication overlap, the time in the fourth phase is given by: $\max[(S_0 + (kl/B_0) * log(P_0)), ((mns/(P_0 \cdot (P_c - 1))) * (k + l))]$. In the fifth and the final phase per iteration, all four cores within a node compute the final row cluster assignments and the column cluster assignments for the rows and columns in that node. The time required is given by: $O((mns/(P_0 \cdot P_c)) * (k + l))$. The overall time for the parallel flat co-clustering algorithm , with I iterations, is given by:

$$\begin{aligned} T(m,n,P_0) = O(I * ((S_0 + (n/B_0) * log(P_0)) + \\ \max[O(S_0 + (n/B_0) * log(P_0)), O(m/(P_c - 1)] + \\ (n/P_c) + mns/(P_0 * P_c) + \\ \max[(S_0 + (kl/B_0) * log(P_0)), \\ ((mns/(P_0 \cdot (P_c - 1))) * (k + l))] + \\ ((mns/(P_0 \cdot P_c)) * (k + l)) \end{aligned} \tag{2}$$

Based on whether the compute cost or the communication cost dominates per iteration, we consider the followin two cases. In the first case, when the compute cost dominates, the parallel flat co-cluster time complexity is given by:

$$\begin{aligned} T(m,n,P_0) = O(I * ((S_0 + (n/B_0) * log(P_0)) + \\ \frac{(m+n)}{P_c} + \frac{mns(k+l)}{P_0 P_c})) \end{aligned} \tag{3}$$

Table 1: Notation

Symbol	Definition
P_0	Total number of nodes for computation
P_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
π_r	Number of partitions of the rows
π_c	Number of partitions of the columns
G_0	Number of nodes per partition
B_0	Interconnect Bandwidth for AllReduce/Allgather
S_0	Setup cost for AllReduce/Allgather

In the second case, when the communication cost dominates per iteration, then the parallel flat algorithm time complexity is given by:

$$T(m,n,P_0) = O(I * (S_0 + \frac{(m+n+kl)\log(P_0)}{B_0} + \frac{n}{P_c} + \frac{mns(k+l)}{P_0 P_c})) \tag{4}$$

4.2 Optimized Distributed Co-clustering Algorithm

For multi-core cluster architectures, one can utilize the available intra-node parallelism along with inter-node parallelism to get highly scalable distributed co-clustering algorithm as given in algorithm 3. Let, c be the number of cores (threads) per node in the distributed architecture, referred to as $T_1 \dots T_c$. These cores (threads) per node can be used to obtain computation communication overlap as well pipelining across the iterations in the distributed algorithm. This can significantly reduce the communication bottlenecks of the algorithm. Algorithm 3 presents the distributed algorithm with these performance optimizations. The *while loop* executes iterations until the RMSE value converges to within a given error bound. Within each iteration the following steps (Step5..Step10) get executed. In **Step 5.**, threads $(T_2 \dots T_c)$ compute the partial contribution to row-cluster averages, A^{RC}_{gj}; while simultaneously, thread T_1, performs *MPI_Allgather* to get the column-cluster membership (γ). Thus, (intra-iteration) computation communication overlap is achieved which leads to improved performance. Similarly, computation communication overlap is achieved in the following steps. In **Step 6.**, threads $(T_2 \dots T_c)$ compute the partial contribution to column-cluster averages, A^{CC}_{ih}; while simultaneously, thread T_1, performs *MPI_Allreduce* to compute the row-cluster averages A^{RC}_{gj}. In **Step 7.**, threads $(T_2 \dots T_c)$ compute the partial contribution to co-cluster averages, A^{COC}_{gh}; while simultaneously, thread T_1, performs *MPI_Allreduce* to compute the column-cluster averages A^{CC}_{ih}. In **Step 8.**, threads $(T_2 \dots T_c)$ compute the partial $\hat{A}_{ij}$ values using A^{CC}_{ih} and A^{RC}_{gj}; while simultaneously, thread T_1, performs *MPI_Allreduce* to compute the co-cluster averages A^{COC}_{gh}. In **Step 9.**, all threads $(T_1 \dots T_c)$ in a node, compute final row-cluster memberships for all the rows that are owned by that node. In **Step 10.**, threads $(T_2 \dots T_c)$ compute final column-cluster memberships while simultaneously, thread T_1, performs *MPI_Allgather* to get the row-cluster memberships from all other nodes.

4.3 Time Complexity Analysis for Flat Hybrid Algorithm

The distributed algorithm described in section 3 takes a certain number of iterations, say I. In each iteration, rows are assigned to row clusters and columns to column clusters. Each iteration has multiple steps. In *Step 5.*, the thread T_1 of all nodes communicate using all-gather operation to aggregate column to column-cluster mapping information. This communication time is given by: $O(S_0 + (n/B_0) * log(P_0))$. Simultaneously, threads $T_2 \dots T_c$ of each node compute partial contributions of each node towards A^{RC}_{gj}. This computation time is $O(mn/(P_0.c))$. The overall time for *Step 5.* is given by $\max(O(S_0 + (n/B_0) * log(P_0)), mn/(P_0.c))$. Assuming, that compute time dominates, the time complexity for *Step 5.* can be approximated by $O(mn/(P_0 * c))$.

In *Step 6.*, the thread T_1 of all nodes communicate using all-reduce operation to compute the row-cluster averages A^{RC}_{gj}. This communication time is given by: $O(S_0 + (mn/B_0) * log(P_0))$. Simultaneously, threads $T_2 \dots T_c$ of

Algorithm 3 Distributed hybrid Co-Clustering algorithm

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: (Each node now has 4 threads - $\{T_0 \dots T_3\}$**)**
1. $T_0 \dots T_3$ of node p each get $m_{p'}$ rows (i.e, a $m_{p'} \times n$ submatrix $A^{rp'}$) and $n_4 p$ columns (i.e, a $m \times np'$ submatrix $A^{cp'}$) where $m'_p = \frac{m}{p'}$ and $n'_p = \frac{n}{p'}$ and $p' \times 4$.
2. each T_i : Randomly iniatalize (ρ_i^p,γ_i^p)
3. T_0 : Gather all the row and column sums/weights S_i^R, S_j^C, W_i^R, W_j^C $\forall i, j$ from the other nodes using MPI_Allgather.
4. $T_0 \dots T_3$: Calculate **all** row and column averages $A_i^R = \frac{S_i^R}{W_i^R}$ and A_j^C.
(Please note that Step 3 and Step 4 can be pipelined)
5. T_0: Gather the **global** Row-cluster membership (ρ) by concatenating (ρ^p) using MPI_Allgather.
while *RMSE value is converging,* ***Each thread*** *in a node does the following* **do**
6. T_1, T_2, T_3 : Calculate the global Row-Cluster Averages A^{RC}
T_0 : Gather the **global** Column-cluster membership (γ) by concatenating (γ^p) using MPI_Allgather.

7. $T_0 \dots T_3$: Calculate the global Column-Cluster Averages A^{CC}
8. $T_0 \dots T_3$: Calculate the contribution of the local rows and columns to the Cocluster sums/weights i.e, S_p^{COC} and W_p^{COC}.

9. T_0:Do an All_reduce on above contributions and get the **global** CoCluster sums/weights S^{COC}, W^{COC} and caluclate A^{COC}.
$\{T_1, T_2, T_3\}$: Partially compute $\hat{A}^R(i, j, g)$, $\hat{A}^C(i, j, h)$ the local row cluster and column cluster assignment steps for each choice of assignment g,h

10. $T_0 \dots T_3$: Update all the local row cluster assignments ρ^p by first updating $\hat{A}^R(i, j, g)$ with the cocluster averages to generate $\hat{A}_{ij}$
$\rho^p(i) = \underset{1 \le g \le k}{\textbf{\textit{argmin}}} \sum_{j=1}^{n} w_{ij} d_\phi(A_{ij}, \hat{A}_{ij}),\ i \in A^{rp}$

11. T_0 : Gather the **global** Row-cluster membership (ρ) by concatenating (ρ^p) using MPI_Allgather.
$T_1 \dots T_3$: Update all the local column cluster assignments γ^p by first updating $\hat{A}^C(i, j, h)$ with the cocluster averages to generate $\hat{A}_{ij}$

$\gamma^p(i) = \underset{1 \le h \le l}{\textbf{\textit{argmin}}} \sum_{i=1}^{m} w_{ij} d_\phi(A_{ij}, \hat{A}_{ij}),\ j \in A^{cp}$
end

each node compute partial contributions of each node towards A_{ih}^{CC}. This computation time is $O(mn/(P_0.c))$. Thus, the overall time for *Step 6.* is given by $\max(O(S_0 + (mn/B_0) * log(P_0)), mn/(P_0.c))$. Assuming, that the communication time dominates, the time complexity for *Step 6.* can be approximated by $O(S_0 + (mn/B_0) * log(P_0))$. Similarly, the time complexity for *Step 7.* can be approximated by $O(S_0 + (mn/B_0) * log(P_0))$. In *Step 7.*, the thread T_1 of all nodes communicate using all-reduce operation to compute the column-cluster averages A_{ih}^{CC}. This communication time is given by: $O(S_0 + (mn/B_0) * log(P_0))$. Simultaneously, threads $T_2 \dots T_c$ of each node compute partial contributions of each node towards co-cluster averages A_{gh}^{COC}. This computation time is $O(mns/(P_0.c))$. Thus, the overall time for *Step 7.* is given by $\max(O(S_0 + (mn/B_0) * log(P_0)), mns/(P_0.c))$. Assuming that the communication time dominates, the time complexity for *Step 7.* can be approximated by $O(S_0 + (mn/B_0) * log(P_0))$.

In *Step 8.*, the thread T_1 of all nodes communicate using all-reduce operation to compute the co-cluster averages A_{gh}^{COC}. This communication time is given by: $O(S_0 + (kl/B_0) * log(P_0))$. Simultaneously, threads $T_2 \dots T_c$ of each node compute partial values for assignment of each row (and column) to k possible row-clusters (and l possible column-clusters). This computation time is $O(mns * (k + l)/(P_0.c))$. Thus, the overall time for *Step 8.* is given by $\max(O(S_0 + (kl/B_0) * log(P_0)), mns * (k + l)/(P_0.c))$. Assuming that the compute time dominates, the time

complexity for *Step 8.* can be approximated by $O(mns * (k+l)/(P_0.c))$. In a similar fashion, the compute time for *Step 9.* is $O(mns * (k+l)/(P_0 * c))$. In *Step 9.*, all threads, $T_1 \dots T_c$ at a node p, compute the row-cluster membership for each row that the node p owns. This computation cost varies depending on the divergence function - Euclidean, I-divergence or general Bregmann divergence. For each row assignment, any of the divergence functions involve computations over all populated columns, ns, per row and checking assignment to all possible k row clusters. Hence, the compute time is given by: $O(mns * (k+l)/(P_0 * c))$. Assuming that the compute time dominates *Step 10.*, its time complexity can be approximated by $O(mns * (k+l)/(P_0 * c))$. In *Step 10.*, the thread T_1 of all nodes communicate using all-gather operation to get row-cluster membership for all rows. This communication time is given by: $O(S_0 + (m/B_0) * log(P_0))$. Simultaneously, threads $T_2 \dots T_c$ of each node compute the column membership for the column it owns.. This computation cost varies depending on the divergence function - Euclidean, I-divergence or general Bregmann divergence. For each column assignment, any of the divergence functions involve computations over all populated rows, ms, per column and checking assignment to all possible l column clusters. Hence, the compute time is given by: $O(mns * (k+l)/(P_0 * c))$. Thus, the overall time for *Step 10.* is given by $\max(O(S_0 + (m/B_0) * log(P_0)), mns * (k+l)/(P_0.c))$. Assuming that the compute time dominates the time complexity for *Step 10.* can be approximated by $O(mns * (k+l)/(P_0 * c))$.

. Thus, the overall time complexity for the flat hybrid distributed co-clustering algorithm, per iteration, is given by:

$$\begin{aligned} T_h(m,n,P_0,k,l) = mn/P_0 * c + 2 * (mn/B_0) * log(P_0) \\ + S_0 + 3 * mns * (k+l)/(P_0 * c)) \end{aligned} \tag{5}$$

4.3.1 Optimum Thread Distribution

In a general case, one can optimize the communication by providing more than one thread for communication. We study this general communication optimization technique in this section and determine the optimum number of threads to achieve best performance.

Let r be the number of threads (cores) that are devoted to computation per step, while the remaining $(c-r)$ threads (cores), perform communication per step. When, multiple threads are used for communication, we assume that it takes x steps to complete one communication task across all nodes. In this case, the time complexity of the hybrid distributed co-clustering algorithm is given by:

$$\begin{aligned} T_h(m,n,r,P_0) = O((mn/P_0 * r) + (S_0 + (mn/B_0) * log(P_0)) * (2x/(c-r)) + \\ 3mns * (k+l)/(P_0 * r)) \end{aligned} \tag{6}$$

Differentiating the above expression for $T_h(m,n,r,P_0)$ with respect to r, and setting it to zero, we can determine the optimum number of threads to be used for computation per node. We get the following quadratic equation to determine the optimum r:

$$2xP_0 * (S_0 + mn/B_0 * \log(P_0)) * r^2 = (mn + 3mns * (k+l)) * (c^2 + r^2 - 2cr) \tag{7}$$

Solving, the optimum value of r is given by:

$$\begin{aligned} r^* = \frac{\sqrt{(}4c^2Z^2 + 8c^2xP_0Y) - 2cZ}{2 * (2xP_0Y - Z)}, \text{where,} \\ Y = S_0 + mn/B_0 * \log(P_0), \text{and, } Z = mn + 3mns(k+l) \end{aligned} \tag{8}$$

4.4 Flat Load Balancing Algorithm

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.

We use the following notation. Let, $S(r_i)$ denote the number of populated entries in the row, $r_i, i \in [1..M]$, and $S(c_j)$ denote the number of populated entries in the column, $c_j, j \in [1..N]$. For the distributed flat algorithm, let,

variable, $x_{i,p}$ denote that the row, r_i is assigned to processor/node p. Similarly, variable, $y_{j,p}$, denote that the column, c_j is assigned to processor/node p. Further, the compute load on a processor, CL_p is given by:

$$CL_p = G * [\sum_i S(r_i) * x_{i,p} + \sum_j S(c_j) * y_{j,p}] \tag{9}$$

In the above equation, G is a proportionality constant. The load imbalance across two processors is given by $\Delta CL_{p,p'} = |CL_p - CL_{p'}|$. The load imbalance across all processors in the system, is given by, $\max_{(p,p')} \Delta CL_{p,p'}$. The load balancing ILP problem for the flat distributed algorithm is as follows:

Objective: $minimize\, Y$
Constraints:

$$\begin{aligned} \forall p, p' \in P, p \neq p' : & Y \geq \Delta CL_{p,p'} \\ \forall i : & \sum_p x_{i,p} \geq 1 \\ \forall j : & \sum_p y_{j,p} \geq 1 \\ \forall i, p : & x_{i,p} \in [0,1] \\ \forall j, p : & y_{j,p} \in [0,1] \end{aligned} \tag{10}$$

The above ILP is NP-hard by reduction from the 3-partition [11] problem. 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.

5 Hierarchical Coclustering Algorithm

5.1 Batch Mode

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. 3) 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. 3) 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, Algorithm 2). 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 as 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

Algorithm 4 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 $\frac{m/2^x}{\pi_r} \times \frac{n}{\pi_c}$ 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

co-clustering algorithm (Algorithm 2), 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.

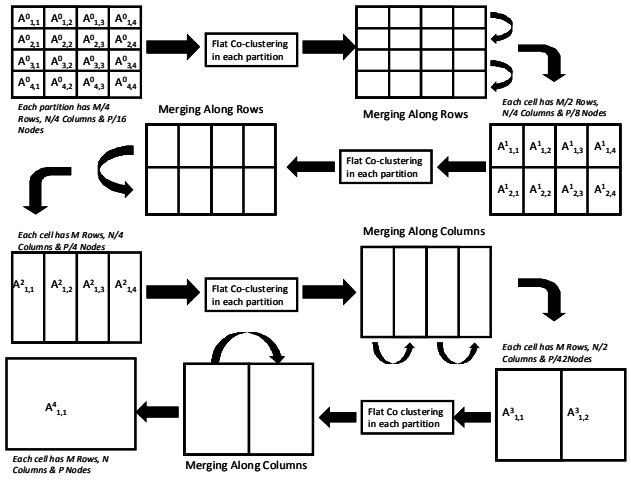


Figure 2: Hierarchical Co-clustering: Matrix Row/Column Folding

Fig. 2 and Fig. 3 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. 3), 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,

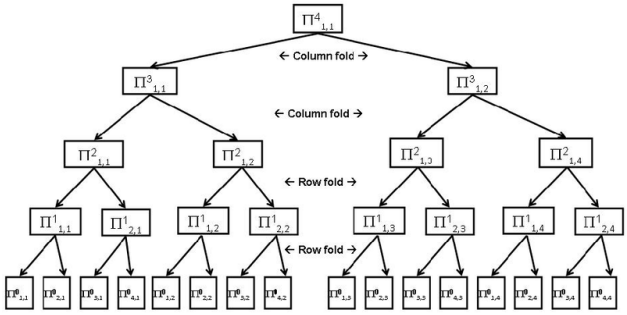


Figure 3: Hierarchical Co-clustering - Computation Tree

$A^0_{i,j}$, 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. 2 and Step 3 in Algorithm 4). 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. 2. 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 4.

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.

5.2 Online Mode

Algorithm 5 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^{COC}_{gh},A^{RC}_{gj},A^{CC}_{ih} 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 $\Pi^{i,j}_{y+1}$ 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 5. 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

Algorithm 5 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

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 (refer section 4.2) 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.

5.3 Time Complexity Analysis for 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:

$$Z = \log(\pi_r) + \log(\pi_c) \tag{11}$$

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

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

$$\begin{aligned} T_{(r\_merge)}(m.2^x/\pi_r, n/\pi_c, G_0.2^x) = \frac{(k_x + l/\pi_c)}{(G_0 * c)} \\ + (\frac{1}{2G_0.c} * (\frac{m.2^x}{\pi_r} + \frac{n}{\pi_c})) \end{aligned} \tag{12}$$

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):

$$\begin{aligned} T_{(c\_merge)}(m, n.2^y/\pi_c, G_0.\pi_r.2^y) = \frac{(k + l_y)}{(G_0 * \pi_r.2^y.c)} + \\ ((1/2G_0.\pi_r, 2^y.c) * (m + (n.2^y)/\pi_c)) \end{aligned} \tag{13}$$

The total number of iterations in the parallel hierarchical algorithm is same as the flat algorithm, ie. I. However, in case of the hierarchical algorithm, the I iterations are distributed across the $Z = \log(\pi_r) + \log(\pi_c)$ levels. As the levels increase from 0 to $Z-1$, the number of iterations per level decrease by a factor of I/Z. 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}.

$$\begin{aligned} T_{(hier)} &= T_{row\_fold} + T_{col\_fold} \\ T_{row\_fold} &= O(\sum_{x=0}^{\log(\pi_r - 1)} (\frac{(k/\pi_r.2^x + l/\pi_c).m/\pi_r.n/\pi_c.2^x.s}{\frac{P_0.2^x}{\pi_r.\pi_c}})) \\ T_{col\_fold} &= O(\sum_{x=0}^{\log(\pi_c - 1)} (k + \frac{l.2^x}{\pi_c}).\frac{m.n.s}{P_0}) \end{aligned} \tag{14}$$

The total time over all row folds is given by:

$$T_{row\_fold} = O((\frac{k.(\pi_r - 1)}{\pi_r} + \frac{l.\log(\pi_r)}{\pi_c}).\frac{m.n.s}{P_0}) \tag{15}$$

Similarly using T_{col_fold} can be written as:

$$\begin{aligned} T_{col\_fold} &= O(\sum_{x=0}^{\log(\pi_c - 1)} (k + \frac{l.2^x}{\pi_c}).\frac{m.n.s}{P_0}) \\ T_{col\_fold} &= O((k.\log(\pi_c) + \frac{l.(\pi_c - 1)}{\pi_c}).\frac{m.n.s}{P_0}) \end{aligned} \tag{16}$$

Substituting the expression for T_{row_fold}, T_{col_fold} from equation (16), and simplifying equation (14), and assuming the communication cost is low, we get:

$$\begin{aligned} T_{hier} = O(((k.\log(\pi_c) + \frac{l.(\pi_c - 1)}{\pi_c}) \\ + (\frac{k.(\pi_r - 1)}{\pi_r} + \frac{l.\log(\pi_r)}{\pi_c})).\frac{m.n.s}{P_0}) \end{aligned} \tag{17}$$

Now combining results from (17) & (11) and making k=l=C, π_r=π_c=π we get T_{hier} as:

$$\begin{aligned} T_{hier} &= O(\frac{(2.(1 - \frac{C}{\pi}) + C.(\frac{\log(\pi)}{\pi} + 1)).\frac{m.n.s}{P_0}}{2.\log(\pi)}) \\ T_{hier} &= O(\frac{C.m.n.s}{P_0.\log(\pi)}) \end{aligned} \tag{18}$$

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

$$\begin{aligned} T_{flat} &= O(\frac{C.m.n.s}{P_0}) \\ \frac{T_{hier}}{T_{flat}} &= O(\frac{1}{\log(\pi)}) \end{aligned} \tag{19}$$

Equation (19) 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 (17)) to compute the optimal values of π_r, π_c and Z. We skip this analysis for brevity.

5.4 Hierarchical Load Balancing Algorithm

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). The *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 provide an overview of this multi-level inter-partition load balance problem below. Strictly speaking one also needs to consider intra-partition load balance for all partitions. One can use flat load balancing type heuristics to ensure this. We skip this for brevity. Let, the partitions, at the leaf level (l_0), in the hierarchical algorithm be denoted by, $\pi^0_{g,h}$, where g denotes the row-index for the partition and h denotes the column index for partition. Let, variable, $x_{i,g}$ denote that row, r_i is assigned to partition, $\pi^0_{g,h}, h \in [1..H]$, and variable, $y_{j,h}$ denote that column, y_j is assigned to partition, $\pi^0_{g,h}, g \in [1..G]$. Let, $S_{gh}(r_i)$, denote the number of populated entries in the part of the row, r_i, that belong to the partition, $\pi^0_{g,h}$. Similarly, let $S_{gh}(c_j)$, denote the number

of populated entries in the part of the column, c_j that belong to the partition, $\pi^0_{g,h}$. The compute load on the partition, $\pi^0_{g,h}$, is given as follows:

$$CL^0(\pi^0_{g,h}) = G^0 * [\sum_i S_{gh}(r_i) * x_{i,g} + \sum_j S_{gh}(c_j) * y_{j,h}] \tag{20}$$

The inter-partition load imbalance across two partitions at leaf level, l_0, is given by:

$$\Delta CL^0(\pi^0_{g,h}, \pi^0_{g',h'}) = |CL^0(\pi^0_{g,h}) - CL^0(\pi^0_{g',h'})| \tag{21}$$

The inter-partition load imbalance across all partitions in the system at level, l_0, is given by : $\max_{(\pi^0,\pi'^0)} \Delta CL^0(\pi^0_{g,h}, \pi^0_{g',h'})$. In a similar fashion one can define the load imbalance at higher levels of the hierarchy. We assume that it is known apriori which partitions are merged to form the next level partitions. In general, the decision of whether to perform row-based merge or column-based merge at a level can also be treated as an optimization problem, but we consider this aspect later. For level, l_1, let the partitions be denoted by $\pi^1_{u,v}$. Let variable, $x_{i,u} = 1$ if row r_i is assigned to partition $\pi^1_{u,v}v \in [1..V]$, and 0 otherwise. Similarly, let variable, $y_{j,v} = 1$, if column, c_j is assigned to partition $\pi^1_{u,v}u \in [1..U]$,and 0 otherwise. The compute load on the partition, $\pi^1_{u,v}$, can be defined as follows:

$$CL^1(\pi^1_{u,v}) = G^1 * [\sum_i S_{uv}(r_i) * x_{i,u} + \sum_j S_{uv}(c_j) * y_{j,v}] \tag{22}$$

Similar to equation (21), one can define the load imbalance at level l_1.

$$\Delta CL^1(\pi^1_{u,v}, \pi^1_{u',v'}) = \left|CL^1(\pi^1_{u,v}) - CL^1(\pi^1_{u',v'})\right| \tag{23}$$

The inter-partition load imbalance across all partitions in the system at level, l_1, is given by, $\max_{(\pi^1,\pi'^1)} \Delta CL^1(\pi^1_{u,v}, \pi^1_{u',v'})$. The load balancing ILP problem for the hierarchical distributed algorithm (considering only levels l_0 and l_1 and assuming row-based merge), is as follows:
Objective: $minimize\ (Y_0 * \gamma_0 + Y_1 * \gamma_1)$
Constraints: Feasibility Constraints :

$$\begin{aligned} &\forall i : \sum_g x_{i,g} \geq 1 \text{ and} \\ &\forall j : \sum_h y_{j,h} \geq 1 \end{aligned} \tag{24}$$

Maxima Constraints :

$$\begin{aligned} &\forall \pi^0, \pi'^0 \text{at level } l_0, \pi^0 \neq \pi'^0 : Y_0 \geq \Delta CL^0_{\pi^0,\pi'^0} \\ &\forall \pi^1, \pi'^1 \text{at level } l_1, \pi^1 \neq \pi'^1 : Y_1 \geq \Delta CL^1_{\pi^1,\pi'^1} \end{aligned} \tag{25}$$

Inter-level Constraints :

$$\forall i, u : x_{i,u} = x_{i,g} + x_{i,g'} \tag{26}$$

(when $\pi_{u,v}$ is obtained from the merge of $\pi_{g,h}$ and $\pi_{g',h}$, $h = v$)

$$\forall j, h, v \text{ and } h = v : y_{j,h} = y_{j,v} \tag{27}$$

Variable Range Constraints :

$$\begin{aligned} &\forall i, g : x_{i,g} \in [0, 1] \text{ and} \\ &\forall j, h : y_{j,h} \in [0, 1] \end{aligned} \tag{28}$$

Here, γ_0 and γ_1 are factors that determine how much weightage to be given for load imbalance at level l_0 and level l_1 respectively. In a similar fashion one can write the ILP for hierarchical load balancing problem across all levels of the hierarchy involving row-based merge. For column based merge, the constraints relating variables of successive levels will denote relations amongst $y_{j,*}$ variables and $x_{i,*}$ variables will remain equal. We skip the full ILP formulation for sake of brevity and clarity.

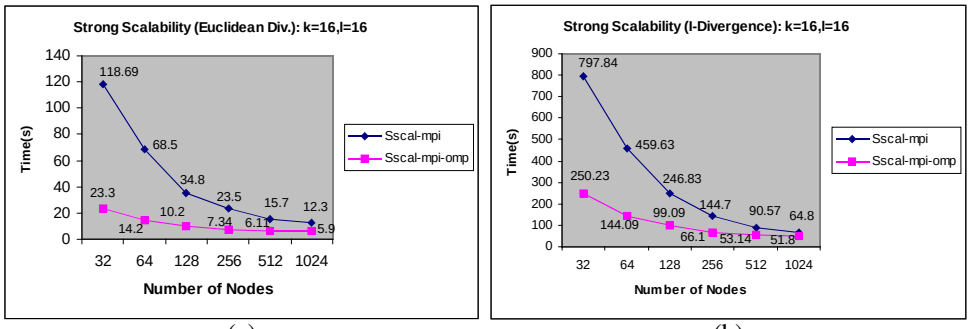
6 Results & Analysis

6.1 Flat Hybrid Algorithm results

The hybrid distributed algorithm was implemented using MPI and OpenMP, while the base distributed algorithm was implemented using only MPI. The Netflix Prize dataset was used to evaluate and compare the performance and scalability of these distributed co-clustering algorithms. 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 and 32 KB of L1 cache per core. Blue Gene/P has the following major interconnects:(a) 3D-Torus interconnect which provides 3.4 Gbps per link on each of the 12 links per node (total 5.1 GBps per node), and, (b) Collective Network that provides 6.8 Gbps per link. 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 data. Below, k refers to the number of row clusters generated while l refers to the number of column clusters generated. Netflix data was used for evaluation of the distributed algorithms. For Netflix, the number of rows, m, is around $480K$; the number of columns, n, is $17,770$, and the sparsity factor, s is around 85. We present the strong, weak and data scalability analysis of the training phase for both Euclidean divergence and I-divergence based co-clustering.

6.1.1 Strong Scalability

For strong scalability, we used the full Netflix data for each experiment, while increasing the number of nodes, from 64 to 1024. Fig. $4(a)$ illustrates that the hybrid algorithm (for Euclidean divergence) has consistently better performance over the base algorithm: $5.1\times$ better than the base when $P_0 = 32$ and $2.1\times$ better at $P_0 = 1024$. Here, the hybrid algorithm has more than $(c = 4)\times$ better performance than the base algorithm due to reduction in load-imbalance as explained in Section 4.3. In the hybrid algorithm, as the number of nodes increases from 32 to 1024, the compute time decreases by $26\times$ while the communication time remains almost the same, this leads to overall $4\times$ decrease in total training time with $32\times$ increase in the number of nodes(P_0). Fig. $4(b)$ illustrates the performance gain of the hybrid algorithm over the base algorithm for I-divergence. Here, the performance gain of hybrid vs base decreases from $3.2\times$ for $P_0 = 32$ nodes to $1.25\times$ for $P_0 = 1024$ nodes. By using more efficient load balancing techniques, the performance of the hybrid (MPI+OpenMP) algorithm can be improved further. Moreover, by using the optimum number of cores for communication using the formula specified in the Section 4.3.1, one can get better overall performance. Further, for I-divergence, the gain for the hybrid algorithm from the decrease in inter-node load-imbalance is offset by the loss from intra-node load-imbalance amongst the threads. Hence, in case of I-divergence the gain of the hybrid algorithm over the base algorithm is not as large as in the Euclidean divergence.



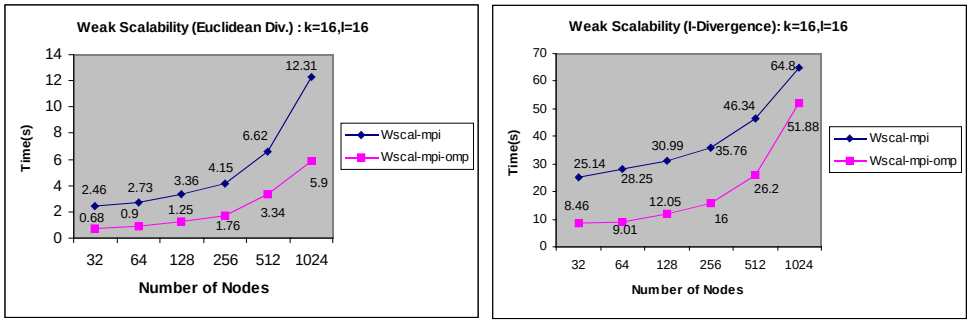
(a) (b)

Figure 4: Strong Scalability: (a) Euclidean divergence. (b) I-divergence

6.1.2 Weak Scalability

Fig. $5(a)$ displays the weak scalability for Euclidean distance based co-clustering as the number of nodes (P_0) increases from 32 to 1024 and the training data increases from 3.125% to 100% of the full Netflix dataset (with $k = 16$, $l = 16$). Here, the hybrid algorithm performs consistently better compared to the base algorithm: $3.61\times$ better at

$P_0 = 32$ and $2.1\times$ better at $P_0 = 1024$. The total time for the hybrid algorithm increases by $8.67\times$ as the number of nodes increase from 32 to 1024. This is due to the compute time increase by $2.91\times$ and also increase in load imbalance. Fig. $5(b)$ illustrates the weak scalability of the hybrid algorithm for I-divergence: with $32\times$ increase in the data and number of nodes, the training time only increases by $6.13\times$. Further, the hybrid algorithm performs consistently better than the base algorithm.

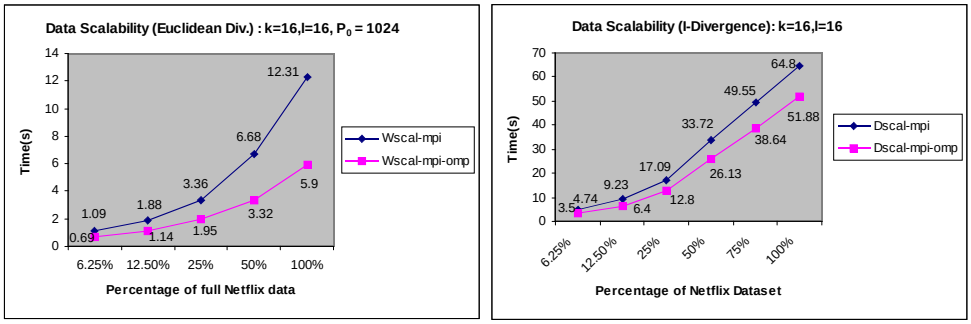


(a) (b)

Figure 5: Weak Scalability: (a) Euclidean divergence. (b) I-divergence

6.1.3 Data Scalability

Fig. $6(a)$ displays the data scalability for Euclidean distance based co-clustering as the training data increases from 6.25% to 100% of the full Netflix dataset, while $P_0 = 1024$. The training time for the hybrid algorithm increases by $8.55\times$ with $16\times$ increase in data, while that for the base algorithm increases by $11.3\times$. Thus, the hybrid algorithm shows better than linear data scalability and also better data scalability as compared to the base algorithm. The hybrid algorithm also performs better than the base by $1.58\times$ at $P_0 = 32$ and $2.1\times$ better at $P_0 = 1024$. Fig. $6(b)$ illustrates the data scalability for the hybrid algorithm with I-divergence as the training time increases only by $14.8\times$ with $16\times$ increase in data, while the number of nodes is kept constant at $P_0 = 1024$.

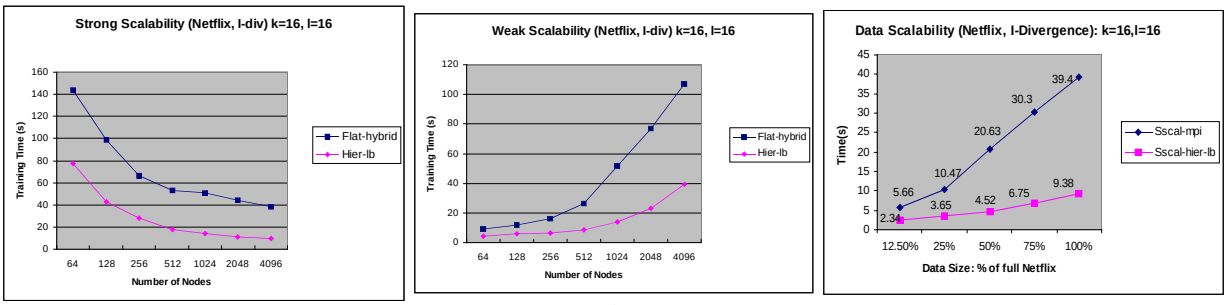


(a) (b)

Figure 6: Data Scalability: (a)Euclidean divergence. (b) I-divergence

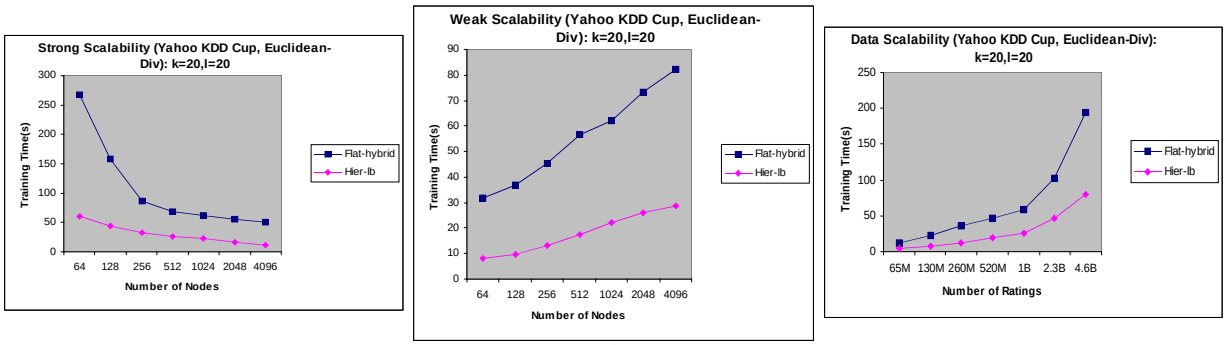
6.2 Batch Hierarchical Algorithm results

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 over $480K$ users and $17K$ movies) , and, *Yahoo KDD Cup* ($252M$ training ratings and $4M$ validation ratings over $1M$ users and $624K$ songs) datasets were used to evaluate and compare the performance and scalability of these distributed algorithms. The experiments were performed on the Blue gene/P (MPP) architecture. 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



(a) (b) (c)

Figure 7: Netflix (I-div/C6): (a) Strong Scalability. (b) Weak Scalability. (c) Data Scalability



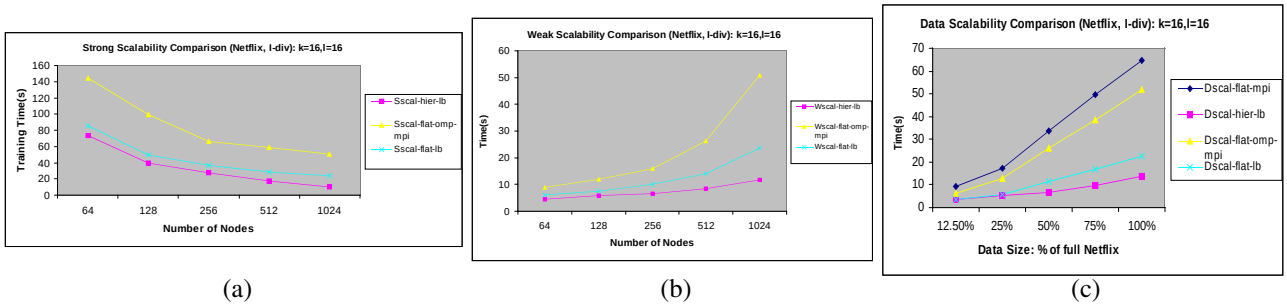
(a) (b) (c)

Figure 8: Yahoo-KDD (Euclidean/C6): (a) Strong Scalability. (b) Weak Scalability. (c) Data Scalability

KDD Cup data. 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 3). 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.

6.2.1 Strong Scalability

Fig. 7(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. 8(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 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



(a) (b) (c)

Figure 9: Detailed Comparison(Netflix): (a) Strong Scalability. (b) Weak Scalability. (c) Data Scalability

optimizing the merge phase in the hierarchical algorithm.

6.2.2 Weak Scalability

Fig. 7(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. $8(b)$ 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 $2.9\times$ ($28.5s$ vs $82.4s$) at 4096 nodes.

6.2.3 Data Scalability

Fig. 7(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. 8(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.

6.2.4 Detailed Scalability Comparison

In this section we present detailed comparison of the gains obtained by the hierarchical algorithm and load balancing. Fig. $9(a)$ 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. $9(b)$ 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. $9(c)$ 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.

6.2.5 Performance vs Accuracy Trade-off

Fig. 10 illustrates the variation of RMSE and training time for the hierarchical algorithm with the increase in the number of clusters, for the Yahoo KDD dataset with I-divergence. Here, as the number of clusters increases from 16 to 128, the time increases from $33s$ to $266s$ while the RMSE first goes down to the lowest value of 27.95 for 20 clusters and then increases monotonically to 30.13 RMSE. Thus, the RMSE has a sweet spot with respect to the number of clusters. This trade-off curve is better than for the flat hybrid algorithm since the time increase is higher for similar behavior in RMSE change(the plot has been omitted for brevity).

Fig. 11 illustrates the variation of RMSE and training time for the hierarchical algorithm with the increase in the number of iterations, for the Yahoo KDD dataset with I-divergence. Here, as the number of iterations increases from 12 to 20, the time increases from $62s$ to $86s$ while the RMSE decreases from 29.34 to 28.88.

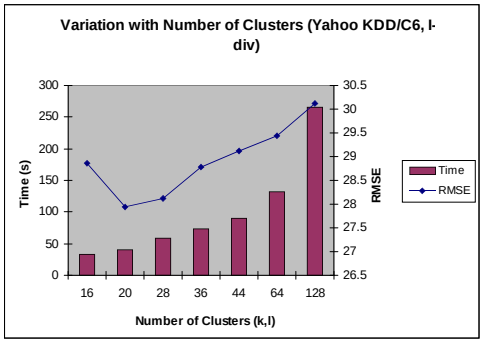


Figure 10: Variation with Number of Clusters (Yahoo KDD /I-div/ C6)

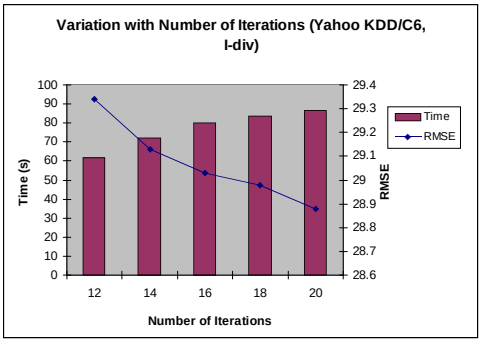
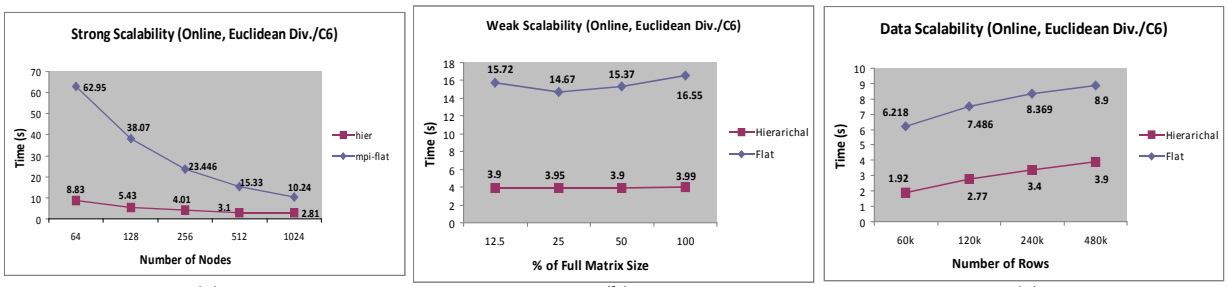


Figure 11: Variation with Number of Iterations (Yahoo/I-div/ C6)

6.3 Online Hierarchical results



(a) (b) (c)

Figure 12: Netflix Online(Euclidean/C6): (a) Strong Scalability. (b) Weak Scalability. (c) Data Scalability

6.3.1 Strong Scalability

Fig. $12(a)$ 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.

6.3.2 Weak Scalability

Fig. $12(b)$ 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 $4\times$.

6.3.3 Data Scalability

Fig. $12(c)$ 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 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 $3.24\times$.

6.3.4 RMSE vs Performance Trade-offs

In this section we present the trade-offs between RMSE and performance. Fig. 13 presents RMSE and training time for the online mode hierarchical distributed algorithm using Euclidean divergence with C6 constraints and 4% change in input data. The X-axis shows the height in the hierarchical computation tree at which the full iterations are started. The height is measured from the leaf of the hierarchical computation tree. As expected, when the height increases from 1 to 4, the online training time redudes from around $4s$ to $2s$. Simultaneously, the RMSE also increases from 0.88 to 0.905. Thus, a $2\times$ gain in performance comes at the cost of 0.02 increase in RMSE.

Fig. 14 presents the RMSE and training time for the online mode hierarchical distributed algorithm using I-divergence with C6 constraints and 4% change in input data. The X-axis shows the height in the hierarchical computation tree at which the full co-clustering iterations are started. As expected, when the height increases from 1 to 4, the online training time redudes from around $13.04s$ to $6.44s$. Simultaneously, the RMSE also increases from 0.897 to 0.92. Thus, a $2\times$ gain in performance comes at the cost of 0.023 increase in RMSE.

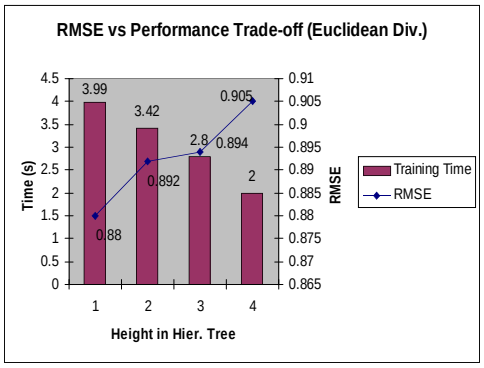


Figure 13: RMSE vs Performance Trade-off(Euclidean Div./ C6)

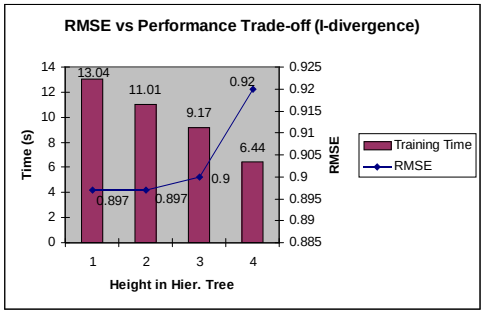


Figure 14: RMSE vs. Performance Trade-off(I-divergence / C6)

7 Conclusions & Future Work

Soft real-time co-clustering and collaborative filtering with high prediction accuracy are computationally challenging problems. We have presented novel performance optimizations for distributed co-clustering and a hierarchical algorithm with soft real-time performance over highly sparse massive data sets. Using pipelined parallelism and compute communication overlap optimizations our hierarchical algorithm outperforms all known prior results for collaborative filtering while maintaining high accuracy. Theoretical time complexity analysis proves the efficacy of our approachs. We demonstrated soft real-time parallel collaborative filtering using the Netflix Prize and Yahoo KDD Cup datasets on a multi-core cluster architecture. We also demonstrated strong, weak and data scalability of our all approaches for multi-core cluster architectures. We delivered the best known training time of $9.38s$ with I-div for the full Netflix dataset and the best known prediction of $2us$ per rating for $1.4M$ ratings with high prediction accuracy, RMSE value of 0.87 ± 0.02, using $4K$ nodes of BG/P. The I-div training time is $4\times$ better than flat hybrid algorithm using same number of nodes. Further, we demonstrate strong performance on $900M$ ratings from the Netflix dataset and $4.6B$ ratings from the Yahoo KDD Cup dataset. In future, we intend to investigate theoretical analysis of convergence for this hierarchical algorithm and to performance analysis using queuing theoretic models for large scale systems.

References

[1] Nicholas Ampazis. Collaborative filtering via concept decomposition on the netflix dataset. In *ECAI*, pages 143–175, 2008.

[2] Arindam Banerjee, Sugato Basu, and Srujana Merugu. Multi-way clustering on relation graphs. In *SDM*, 2007.

[3] Arindam Banerjee, Inderjit Dhillon, Joydeep Ghosh, Srujana Merugu, and Dharmendra S. Modha. A generalized maximum entropy approach to bregman co-clustering and matrix approximation. *Journal of Machine Learning Research*, 8(1):1919 – 1986, Aug. 2007.

[4] J. Bennett and S. Lanning. The netflix prize. In *KDD-Cup and Workshop at the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2007.

[5] M. Brand. Fast online svd revisions for lightweight recommender systems. In *SIAM International Conference on Data Mining*, pages 37–48, 2003.

[6] 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*, pages 43–52, 1998.

[7] Y. Cheng and G. M. Church. Biclustering of expression data. In *Proceedings of the 8th International Conference on Intelligent Systems for Molecular Biology*, pages 93–103, 2000.

[8] 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*, pages 1115–1124, 2009.

[9] Alexandre de Spindler, Moira C. Norrie, Michael Grossniklaus, and Beat Signer. Spatio-temporal proximity as a basis for collaborative filtering in mobile environments. In *UMICS*, 2006.

[10] Inderjit S. Dhillon and Dharmendra S. Modha. Concept decompositions for large sparse text data using clustering. In *Machine Learning*, pages 143–175, 1999.

[11] Michael R. Garey and David S. Johnson. *Computers & Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[12] Thomas George and Srujana Merugu. A scalable collaborative filtering framework based on co-clustering. In *Fifth International Conference on Data Mining*, pages 625–628, 2005.

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

[14] Shahzaib Hassan and Zeeshan Syed. From netflix to heart attacks: collaborative filtering in medical datasets. In *International Health Informatics Symposium (IHI)*, pages 128–134, 2010.

[15] Kuo-Wei Hsu, Arindam Banerjee, and Jaideep Srivastava. I/o scalable bregman co-clustering. In *Proceedings of the 12th Pacific-Asia conference on Advances in knowledge discovery and data mining*, pages 896–903, 2008.

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

[17] Dino Ienco, Ruggero G. Pensa, and Rosa Meo. Parameter-free hierarchical co-clustering by n-ary splits. In *ECML/PKDD (1)*, pages 580–595, 2009.

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

[19] Bongjune Kwon and Hyuk Cho. Scalable co-clustering algorithms. *Algorithms and Architectures for Parallel Processing, Lecture Notes in Computer Science*, 6081(10):32–43, 2010.

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

[21] A. Narang, A. Srivastava, and P.K. Katta. Distributed scalable collaborative filtering algorithm. In *EuroPar 2011*, France, 2011.

[22] Ruggero G. Pensa and Jean-François Boulicaut. Constrained co-clustering of gene expression data. In *SDM*, pages 25–36, 2008.

[23] Paul Resnick and Hal R. Varian. Recommender systems - introduction to special section. *Comm. ACM*, 40(3):56–58, 1997.

[24] R.M.Bell and Y Koren. Scalable collaborative filtering with jointly derived neighbourhood interpolation weights. In *ICDM*, pages 43–52, 2007.

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

[26] Badrul M. Sarwar, George Karypis, Joseph A. Konstan, and John Riedl. Analysis of recommendation algorithms for e-commerce. In *ACM Conference on Electronic Commerce*, pages 158–167, 2000.

[27] J. Ben Schafer, Joseph A. Konstan, and John Riedi. Recommender systems in e-commerce. In *ACM Conference on Electronic Commerce*, pages 158–166, 1999.

[28] N. Srebro and T. Jaakkola. Weighted low rank approximation. In *Twentieth International Conference on Machine Learning*, pages 720–728, 2003.

[29] 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.