Fully Scalable Methods for Distributed Tensor Factorization

Kijung Shin, Lee Sael, and U Kang

Abstract—Given a high-order large-scale tensor, how can we decompose it into latent factors? Can we process it on commodity computers with limited memory? These questions are closely related to recommender systems, which have modeled rating data not as a matrix but as a tensor to utilize contextual information such as time and location. This increase in the order requires tensor factorization methods scalable with both the order and size of a tensor. In this paper, we propose two distributed tensor factorization methods, CDTF and SALS. Both methods are scalable with all aspects of data and show a trade-off between convergence speed and memory requirements. CDTF, based on coordinate descent, updates one parameter at a time, while SALS generalizes on the number of parameters updated at a time. In our experiments, only our methods factorized a 5-order tensor with 1 billion observable entries, 10M mode length, and 1K rank, while all other state-of-the-art methods failed. Moreover, our methods required several orders of magnitude less memory than their competitors. We implemented our methods on MAPREDUCE with two widely-applicable optimization techniques: local disk caching and greedy row assignment. They speeded up our methods up to 98.2× and also the competitors up to 5.9×.

Index Terms—Tensor Factorization, Tensor Completion, Distributed Computing, MapReduce, Hadoop

1 Introduction

Recommendation problems can be viewed as completing a partially observable user-item matrix whose entries are ratings. Matrix factorization (MF), which decomposes the input matrix into a user factor matrix and an item factor matrix such that their product approximates the input matrix, is one of the most widely-used methods for matrix completion [1], [2], [3]. To handle web-scale data, efforts were made to find distributed methods for MF [3], [4], [5].

On the other hand, there have been attempts to improve the accuracy of recommendation by using additional contextual information such as time and location. A straightforward way to utilize such extra factors is to model rating data as a partially observable tensor where additional modes correspond to the extra factors. Similar to the matrix completion, tensor factorization (TF), which decomposes the input tensor into multiple factor matrices and a core tensor, has been used for tensor completion [6], [7], [8], [9].

TABLE 1: Summary of scalability results. The factors which each method is scalable with are checked. CDTF and SALS are the only methods scalable with all the factors.

	CDTF, **SALS** (Proposed)	ALS [14], [3]	PSGD [10]	FLEXIFACT [11]
Order	✓	✓	✓	
Observations	✓	✓	✓	✓
Mode Length	✓			✓
Rank	✓			✓
Machines	✓	✓		

As more extra information becomes available, a necessity for TF algorithms scalable with the order as well as the number of entries in a tensor has arisen. A promising way to find such algorithms is to extend distributed MF algorithms to higher orders. However, the extensions of existing methods [3] [10] [11] have limited scalability, as we explain in Section 2.3.

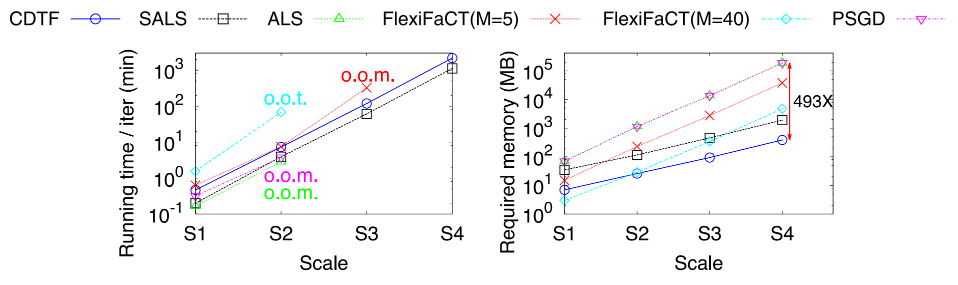
In this paper, we propose Coordinate Descent for Tensor Factorization (CDTF) and Subset Alternating Least Square (SALS), distributed tensor factorization methods scalable with all aspects of data. CDTF applies coordinate descent, which updates one parameter at a time, to TF. SALS, which includes CDTF as a special case, generalizes on the number of parameters updated at a time. These two methods have distinct advantages: CDTF is more memory efficient and applicable to more loss functions, while SALS converges faster to a better solution.

Our methods can be used in any applications handling large-scale partially observable tensors, including social network analysis [12] and Web search [13].

The main contributions of our study are as follows:

- *Kijung Shin is with Computer Science Department, Carnegie Mellon University, USA, E-mail: kijungs@cs.cmu.edu*
- *Lee Sael is with the Department of Computer Science, State University of New York Korea, Republic of Korea, E-mail: sael@sunykorea.ac.kr*
- *U Kang is with the Department of Computer Science and Engineering, Seoul National University, Republic of Korea, E-mail: ukang@snu.ac.kr (corresponding author)*

This work was supported by the Basic Science Research Program through the National Research Foundation of Korea funded by the Ministry of Science, ICT and Future Planning (grants No. 2013R1A1A1064409 and NRF-2015R1C1A2A01055739), and by the KEIT Korea under the "Global Advanced Technology Center" (10053204). The Institute of Engineering Research at Seoul National University provided research facilities for this work. The ICT at Seoul National University provided research facilities for this study.



(a) Running time (b) Memory requirements

Fig. 1: Scalability comparison of tensor factorization methods on the Hadoop cluster. o.o.m. : out of memory, o.o.t. : out of time (takes more than a week). Only our proposed CDTF and SALS scaled up to the largest scale S4 (details are in Table 4), and they showed a trade-off: CDTF was more memory-efficient, and SALS ran faster. The scalability of ALS, PSGD, and FLEXIFACT with five reducers was limited due to their high memory requirements. In particular, ALS and PSGD required 186GB for S4, which is 493× of 387MB that CDTF required. The scalability of FLEXIFACT with 40 reducers was limited because of its rapidly increasing communication cost.

- **Algorithm.** We propose CDTF and SALS, scalable tensor factorization algorithms. Their distributed versions are the only methods scalable with all the following factors: the order and size of data, the number of parameters, and the number of machines (see Table 1).
- **Analysis.** We analyze our methods and their competitors in the general N-order setting in the following aspects: computational complexity, communication complexity, memory requirements, and convergence speed (see Table 3).
- **Optimization.** We implement our methods on MAPREDUCE with two widely-applicable optimization techniques: local disk caching and greedy row assignment. They speeded up not only our methods (up to 98.2×) but also the competitors (up to 5.9×) (see Figure 10).
- **Experiment.** We empirically confirm the superior scalability of our methods and their several orders of magnitude less memory requirements than the competitors. Only our methods successfully analyzed a 5-order tensor with 1 billion observable entries, 10M mode length, and 1K rank, while all others failed (see Figure 1).

Our open-sourced code and the data we used are available at **http://www.cs.cmu.edu/~kijungs/codes/cdtf**. In Section 2, we present preliminaries for tensor factorization. In Section 3, we describe our proposed CDTF and SALS methods. In Section 4, we present the optimization techniques used in our implementations on MAPREDUCE. In Section 5, we provide experimental results. After reviewing related work in Section 6, we conclude in Section 7.

2 NOTATIONS AND PRELIMINARIES

In this section, we describe notations (summarized in Table 2); and the preliminaries of tensor factorization and its distributed algorithms.

2.1 Tensor and the Notations

Tensors are multi-dimensional arrays that generalize vectors (1-order tensors) and matrices (2-order tensors) to higher orders. Like rows and columns in a matrix, an N-order tensor has N modes, whose lengths are denoted by I_1 through I_N, respectively. We denote tensors with variable order N by boldface Euler script letters (e.g., $\boldsymbol{\mathcal{X}}$). Matrices and vectors are denoted by boldface capitals (e.g., $\mathbf{A}$) and boldface lowercases (e.g., $\mathbf{a}$), respectively. We denote the entry of a tensor by the symbolic name of the tensor with its indices in subscript. For example, the (i_1, i_2)th entry of $\mathbf{A}$ is denoted by $a_{i_1 i_2}$, and the $(i_1, ..., i_N)$th entry of $\boldsymbol{\mathcal{X}}$ is denoted by $x_{i_1...i_N}$. The i_1th row of $\mathbf{A}$ is denoted by $\mathbf{a}_{i_1 *}$, and the i_2th column of $\mathbf{A}$ is denoted by $\mathbf{a}_{* i_2}$.

TABLE 2: Table of symbols.

Symbol	Definition
$\boldsymbol{\mathcal{X}}$	input tensor ($\in \mathbb{R}^{I_1 \times I_2 ... \times I_N}$)
$x_{i_1...i_N}$	$(i_1, ..., i_N)$th entry of $\boldsymbol{\mathcal{X}}$
N	order of $\boldsymbol{\mathcal{X}}$
I_n	length of the nth mode of $\boldsymbol{\mathcal{X}}$
$\mathbf{A}^{(n)}$	nth factor matrix ($\in \mathbb{R}^{I_n \times K}$)
$a^{(n)}_{i_n k}$	(i_n, k)th entry of $\mathbf{A}^{(n)}$
K	rank of $\boldsymbol{\mathcal{X}}$
Ω	set of indices of observable entries of $\boldsymbol{\mathcal{X}}$
$\Omega^{(n)}_{i_n}$	subset of Ω whose nth mode's index is i_n
${}_mS_n$	set of rows of $\mathbf{A}^{(n)}$ assigned to machine m
$\boldsymbol{\mathcal{R}}$	residual tensor ($\in \mathbb{R}^{I_1 \times I_2 ... \times I_N}$)
$r_{i_1...i_N}$	$(i_1, ..., i_N)$th entry of $\boldsymbol{\mathcal{R}}$
M	number of machines (reducers on MAPREDUCE)
T_{out}	number of outer iterations
T_{in}	number of inner iterations
$\lambda (= \lambda_{\mathbf{A}})$	regularization parameter for factor matrices
$\lambda_{\mathbf{b}}$	regularization parameter for bias terms
C	number of parameters updated at a time
η_0	initial learning rate

2.2 Tensor Factorization

Our definition of tensor factorization is based on PARAFAC Decomposition [15], the most popular decomposition method. We use L_2 regularization, whose weighted form has been used in many recommender systems [1], [2], [3]. Other decomposition and regularization methods are also considered in Section 3.5.

Definition 1 (Partially Observable Tensor Factorization): Given an N-order tensor $\boldsymbol{\mathcal{X}} (\in \mathbb{R}^{I_1 \times I_2 ... \times I_N})$ with observable entries $\{x_{i_1...i_N} | (i_1, ..., i_N) \in \Omega\}$, the rank K factorization of $\boldsymbol{\mathcal{X}}$ is to find factor matrices $\{\mathbf{A}^{(n)} \in \mathbb{R}^{I_n \times K} | 1 \leq n \leq N\}$ that minimize (1).

TABLE 3: Summary of distributed tensor factorization algorithms for partially observable tensors. The performance bottlenecks that prevent each algorithm from handling web-scale data are marked by asterisks (*). Only our proposed SALS and CDTF methods have no bottleneck. Communication complexity is measured by the number of parameters that each machine exchanges with the others. For simplicity, we assume that workload of each algorithm is equally distributed across machines, that the length of every mode is equal to I, and that T_{in} of SALS and CDTF is set to one.

Algorithm	Computational complexity (per iteration)	Communication complexity (per iteration)	Memory requirements	Convergence speed
CDTF	$O(\|\Omega\|N^2K/M)$	$O(NIK)$	$O(NI)$	Fast
SALS	$O(\|\Omega\|NK(N+C)/M + NIKC^2/M)$	$O(NIK)$	$O(NIC)$	Fastest
ALS [14], [3]	$O(\|\Omega\|NK(N+K)/M + NIK^3/M)$*	$O(NIK)$	$O(NIK)$*	Fastest
PSGD [10]	$O(\|\Omega\|NK/M)$	$O(NIK)$	$O(NIK)$*	Slow*
FlexiFaCT [11]	$O(\|\Omega\|NK/M)$	$O(M^{N-2}NIK)$*	$O(NIK/M)$	Fast

$$L(\mathbf{A}^{(1)},...,\mathbf{A}^{(N)}) = \sum_{(i_1,...,i_N)\in\Omega}\left(x_{i_1...i_N} - \sum_{k=1}^{K}\prod_{n=1}^{N}a^{(n)}_{i_nk}\right)^2 + \lambda\sum_{n=1}^{N}\|\mathbf{A}^{(n)}\|_F^2 \tag{1}$$

The loss function (1) depends only on the observable entries. Each factor matrix $\mathbf{A}^{(n)}$ corresponds to the latent feature vectors of the objects that the nth mode of $\mathfrak{X}$ represents, and $\sum_{k=1}^{K}\prod_{n=1}^{N}a^{(n)}_{i_nk}$ corresponds to the interaction among the features. Note that this problem is non-identifiable in that (1) may have many global minima corresponding to different factor matrices.

2.3 Distributed Methods for Tensor Factorization

In this section, we explain how widely-used distributed optimization methods are applied to partially observable tensor factorization (see Section 6 for distributed methods applicable only to fully observable tensors). Their performances are summarized in Table 3. Note that only our proposed CDTF and SALS methods, which are described in Sections 3 and 4, have no bottleneck in any aspects. The preliminary version of CDTF and SALS appeared in [16].

2.3.1 *ALS: Alternating Least Square*

ALS is a widely-used technique for fully observable tensor factorization [14] and was used also for partially observable tensors [17]. However, previous ALS approaches for partially observable tensors have scalability issues since they repeatedly estimate all unobservable entries, which can be far more than observed entries depending on data. Fortunately, a recent ALS approach for partially observable matrices [3] is extensible for tensors, and is parallelizable. This extended method is explained in this section.

ALS updates factor matrices one by one while keeping all other matrices fixed. When all other factor matrices are fixed, minimizing (1) is analytically solvable in terms of the updated matrix. We update each factor matrix $\mathbf{A}^{(n)}$ row by row, by the following rule, exploiting the independence between rows:

$$\begin{aligned}[a^{(n)}_{i_n1},...,a^{(n)}_{i_nK}]^T &\leftarrow \underset{[a^{(n)}_{i_n1},...,a^{(n)}_{i_nK}]^T}{\arg\min} L(\mathbf{A}^{(1)},...,\mathbf{A}^{(N)}) \\ &= (\mathbf{B}^{(n)}_{i_n} + \lambda\mathbf{I}_K)^{-1}\mathbf{c}^{(n)}_{i_n},\end{aligned} \tag{2}$$

where $\mathbf{B}^{(n)}_{i_n}$ is a K by K matrix whose entries are

$$(\mathbf{B}^{(n)}_{i_n})_{k_1k_2} = \sum_{(i_1,...,i_N)\in\Omega^{(n)}_{i_n}}\left(\prod_{l\neq n}a^{(l)}_{i_lk_1}\prod_{l\neq n}a^{(l)}_{i_lk_2}\right), \forall k_1, k_2,$$

$\mathbf{c}^{(n)}_{i_n}$ is a length K vector whose entries are

$$(\mathbf{c}^{(n)}_{i_n})_k = \sum_{(i_1,...,i_N)\in\Omega^{(n)}_{i_n}}\left(x_{i_1...i_N}\prod_{l\neq n}a^{(l)}_{i_lk}\right), \forall k,$$

and $\mathbf{I}_K$ is the K by K identity matrix. $\Omega^{(n)}_{i_n}$ denotes the subset of Ω whose nth mode's index is i_n. This update rule is a special case of (9), which is proved in Theorem 1 in Section 3.3.1, since ALS is a special case of SALS (see Section 3.2).

Updating a row, $\mathbf{a}^{(n)}_{i_n*}$ for example, using (2) takes $O(|\Omega_{i_n}|K(N+K)+K^3)$, which consists of $O(|\Omega^{(n)}_{i_n}|NK)$ to calculate $\prod_{l\neq n}a^{(l)}_{i_l1}$ through $\prod_{l\neq n}a^{(l)}_{i_lK}$ for all the entries in $\Omega^{(n)}_{i_n}$, $O(|\Omega^{(n)}_{i_n}|K^2)$ to build $\mathbf{B}^{(n)}_{i_n}$, $O(|\Omega^{(n)}_{i_n}|K)$ to build $\mathbf{c}^{(n)}_{i_n}$, and $O(K^3)$ to invert $\mathbf{B}^{(n)}_{i_n}$. Thus, updating every row of every factor matrix once, which corresponds to a full ALS iteration, takes $O(|\Omega|NK(N+K)+K^3\sum_{n=1}^{N}I_n)$.

In distributed environments, updating each factor matrix can be parallelized without affecting the correctness of ALS by distributing the rows of the factor matrix across machines and updating them simultaneously. The parameters updated by each machine are broadcast to all other machines. The number of parameters each machine exchanges with the others is $O(KI_n)$ for each factor matrix $\mathbf{A}^{(n)}$ and $O(K\sum_{n=1}^{N}I_n)$ per iteration. The memory requirements of ALS, however, cannot be distributed. Since the update rule (2) possibly depends on any entry of any fixed factor matrix, every machine is required to load all the fixed matrices into its memory. This high memory requirements of ALS, $O(K\sum_{n=1}^{N}I_n)$ memory space per machine, have been noted as a scalability bottleneck even in matrix factorization [4], [5].

2.3.2 *PSGD: Parallelized Stochastic Gradient Descent*

PSGD [10] is a distributed algorithm based on stochastic gradient descent (SGD). In PSGD, the observable entries of $\mathfrak{X}$ are randomly divided into M machines which run SGD independently using the assigned entries. The updated parameters are averaged after each iteration. For each observable entry $x_{i_1...i_N}$, $a^{(n)}_{i_nk}$ for all n and k, whose number is NK, are updated at once by the following rule :

$$a^{(n)}_{i_nk} \leftarrow a^{(n)}_{i_nk} - 2\eta\left(\frac{\lambda a^{(n)}_{i_nk}}{|\Omega^{(n)}_{i_n}|} - r_{i_1...i_N}\left(\prod_{l\neq n}a^{(l)}_{i_lk}\right)\right) \tag{3}$$

where $r_{i_1...i_N} = x_{i_1...i_N} - \sum_{s=1}^{K}\prod_{l=1}^{N}a^{(l)}_{i_ls}$. It takes $O(NK)$ to calculate $r_{i_1...i_N}$ and $\prod_{l=1}^{N}a^{(l)}_{i_lk}$ for all k.

Once they are calculated, since $\prod_{l \neq n} a^{(l)}_{i_l k}$ can be calculated as $\left(\prod_{l=1}^{N} a^{(l)}_{i_l k}\right) / a^{(n)}_{i_n k}$, calculating (3) takes $O(1)$, and thus updating all the NK parameters takes $O(NK)$. If we assume that $\mathcal{X}$ entries are equally distributed across machines, the computational complexity per iteration is $O(|\Omega|NK/M)$. Averaging parameters can also be distributed, and in the process, $O(K\sum_{n=1}^{N} I_n)$ parameters are exchanged by each machine. Like ALS, the memory requirements of PSGD cannot be distributed, i.e., all the machines are required to load all the factor matrices into their memory. Thus, $O(K\sum_{n=1}^{N} I_n)$ memory space is required per machine. Moreover, PSGD tends to converge slowly due to the non-identifiability of (1).

2.3.3 FlexiFaCT: Flexible Factorization of Coupled Tensors

FLEXIFACT [11] is another SGD-based algorithm that remedies the high memory requirements and slow convergence of PSGD. FLEXIFACT divides $\mathcal{X}$ into M^N blocks. Each M disjoint blocks not sharing common fibers (i.e., rows in a general nth mode) compose a stratum. FLEXIFACT processes one stratum of $\mathcal{X}$ at a time by distributing the M blocks composing a stratum across machines and processing them independently. The update rule is the same as (3), and the computational complexity per iteration is $O(|\Omega|NK/M)$, as in PSGD. However, contrary to PSGD, averaging is unnecessary since a set of parameters updated by each machine is disjoint with the sets updated by the others. In addition, the memory requirements of FLEXIFACT are distributed among the machines. Each machine only needs to load the parameters related to the block it processes, whose number is $(K\sum_{n=1}^{N} I_n)/M$, into its memory at a time. However, FLEXIFACT suffers from high communication cost. After processing one stratum, each machine sends the updated parameters to the machine which updates them using the next stratum. Each machine exchanges at most $(K\sum_{n=2}^{N} I_n)/M$ parameters per stratum and $M^{N-2}K\sum_{n=2}^{N} I_n$ per iteration where M^{N-1} is the number of strata. Thus, the communication cost increases exponentially with the order of $\mathcal{X}$ and polynomially with the number of machines.

3 PROPOSED METHODS

In this section, we propose two scalable tensor factorization algorithms: CDTF (Section 3.1) and SALS (Section 3.2). After analyzing their theoretical properties (Section 3.3), we discuss how these methods are parallelized in distributed environments (Section 3.4) and applied to diverse loss functions (Section 3.5).

3.1 Coordinate Descent for Tensor Factorization

3.1.1 Update Rule

Coordinate descent for tensor factorization (CDTF) is a tensor factorization algorithm based on coordinate

Algorithm 1: Serial version of CDTF

```
Input  : $\mathcal{X}$, $K$, $\lambda$
Output: $\mathbf{A}^{(n)}$ for all $n$
1 initialize $\mathcal{R}$ and $\mathbf{A}^{(n)}$ for all $n$
2 for outer iter = 1..$T_{out}$ do
3     for $k$ = 1..$K$ do
4         compute $\hat{\mathcal{R}}$ using (5)
5         for inner iter = 1..$T_{in}$ do
6             for $n$ = 1..$N$ do
7                 for $i_n$ = 1..$I_n$ do
8                     update $a^{(n)}_{i_n k}$ using (6)
9         update $\mathcal{R}$ using (7)
```

descent and extends CCD++ [5] to higher orders. Coordinate descent (CD) is a widely-used optimization technique for minimizing a multivariate function. CD begins with an initial guess of parameters (i.e., variables) and iterates over parameters for updating each parameter at a time. When CD updates a parameter, it minimizes the loss function with respect to the updated parameter, while fixing all other parameters at their current values. This one-dimensional minimization problem is usually solved easier than the original problem. CD usually iterates over parameters multiple times, decreasing the loss function monotonically until convergence. CDTF applies coordinate descent to tensor factorization, whose parameters are the entries of factor matrices (i.e., $\mathbf{A}^{(1)}$ through $\mathbf{A}^{(N)}$) and loss function is (1). CDTF updates an entry of a factor matrix at a time by assigning the optimal value minimizing (1). Equation (1) becomes a quadratic equation in terms of the updated parameter when all other parameters are fixed. This leads to the following update rule for each parameter $a^{(n)}_{i_n k}$:

$$a^{(n)}_{i_n k} \leftarrow \underset{a^{(n)}_{i_n k}}{\arg\min}\, L(\mathbf{A}^{(1)}, ..., \mathbf{A}^{(N)})$$
$$= \frac{\sum\limits_{(i_1,...,i_N)\in\Omega^{(n)}_{i_n}} \left(\left(r_{i_1...i_N} + \prod\limits_{l=1}^{N} a^{(l)}_{i_l k}\right) \prod\limits_{l\neq n} a^{(l)}_{i_l k}\right)}{\lambda + \sum_{(i_1,...,i_N)\in\Omega^{(n)}_{i_n}} \prod_{l\neq n} (a^{(l)}_{i_l k})^2} \quad (4)$$

where $r_{i_1...i_N} = x_{i_1...i_N} - \sum_{s=1}^{K}\prod_{l=1}^{N} a^{(l)}_{i_l s}$. Computing (4) from the beginning takes $O(|\Omega^{(n)}_{i_n}|NK)$, but we can reduce it to $O(|\Omega^{(n)}_{i_n}|N)$ by maintaining a residual tensor $\mathcal{R}$ up-to-date instead of calculating its entries every time. To maintain $\mathcal{R}$ up-to-date, after updating each parameter $a^{(n)}_{i_n k}$, we update the entries of $\mathcal{R}$ in $\{r_{i_1...i_N} | (i_1, ..., i_N) \in \Omega^{(n)}_{i_n}\}$ by the following rule:

$$r_{i_1...i_N} \leftarrow r_{i_1...i_N} + \left((a^{(n)}_{i_n k})^{old} - a^{(n)}_{i_n k}\right) \prod\nolimits_{l\neq n} a^{(l)}_{i_l k},$$

where $(a^{(n)}_{i_n k})^{old}$ is the old parameter value.

3.1.2 Update Sequence

Column-wise Order. The update sequence of parameters is another important part of coordinate descent. CDTF adopts the following column-wise order:

$$\Big(\underbrace{(a^{(1)}_{11} ... a^{(1)}_{I_1 1})}_{\mathbf{a}^{(1)}_{*1}} ... \underbrace{(a^{(N)}_{11} ... a^{(N)}_{I_N 1})}_{\mathbf{a}^{(N)}_{*1}}\Big) ... \Big(\underbrace{(a^{(1)}_{1K} ... a^{(1)}_{I_1 K})}_{\mathbf{a}^{(1)}_{*K}} ... \underbrace{(a^{(N)}_{1K} ... a^{(N)}_{I_N K})}_{\mathbf{a}^{(N)}_{*K}}\Big)$$

where we update the entries in, for example, the kth column of a factor matrix and then move on to the

kth column of the next factor matrix. After the kth columns of all the factor matrices are updated, the $(k+1)$th columns of the matrices are updated.

In this column-wise order, we can reduce the number of $\mathfrak{R}$ updates. Updating, for example, the kth columns of all the factor matrices is equivalent to the rank-one factorization of tensor $\hat{\mathfrak{R}}$ whose $(i_1, ..., i_N)$th entry is $\hat{r}_{i_1...i_N} = x_{i_1...i_N} - \sum_{s \neq k} \prod_{l=1}^{N} a^{(l)}_{i_l s}$. $\hat{\mathfrak{R}}$ can be computed from $\mathfrak{R}$ by the following rule:

$$\hat{r}_{i_1...i_N} \leftarrow r_{i_1...i_N} + \prod_{n=1}^{N} a^{(n)}_{i_n k}, \tag{5}$$

which takes $O(N)$ for each entry and $O(|\Omega|N)$ for entire $\hat{\mathfrak{R}}$. Once $\hat{\mathfrak{R}}$ is computed, we can compute the rank-one factorization (i.e., updating $a^{(n)}_{i_n k}$ for all n and i_n) without updating $\hat{\mathfrak{R}}$. This is because the entries of $\hat{\mathfrak{R}}$ do not depend on the parameters updated during the rank-one factorization. Each parameter $a^{(n)}_{i_n k}$ is updated in $O(|\Omega^{(n)}_{i_n}|N)$ by the following rule:

$$a^{(n)}_{i_n k} \leftarrow \frac{\sum_{(i_1,...,i_N) \in \Omega^{(n)}_{i_n}} \left(\hat{r}_{i_1...i_N} \prod_{l \neq n} a^{(l)}_{i_l k} \right)}{\lambda + \sum_{(i_1,...,i_N) \in \Omega^{(n)}_{i_n}} \prod_{l \neq n} (a^{(l)}_{i_l k})^2}. \tag{6}$$

We need to update $\mathfrak{R}$ only once per rank-one factorization by the following rule:

$$r_{i_1...i_N} \leftarrow \hat{r}_{i_1...i_N} - \prod_{n=1}^{N} a^{(n)}_{i_n k}, \tag{7}$$

which takes $O(|\Omega|N)$ for entire $\mathfrak{R}$ as in (5).

Inner Iterations. The number of updates on $\mathfrak{R}$ can be reduced further by repeating the update of, for example, the kth columns multiple times before moving on to the $(k+1)$th columns. Since the repeated computation (i.e., rank-one factorization) does not require to update $\mathfrak{R}$, if the number of the repetitions is T_{in}, $\mathfrak{R}$ needs to be updated only once per T_{in} times of rank-one factorization. With T_{in} times of iteration, the update sequence at column level changes as follows:

$$(\mathbf{a}^{(1)}_{*1}...\mathbf{a}^{(N)}_{*1})^{T_{in}} (\mathbf{a}^{(1)}_{*2}...\mathbf{a}^{(N)}_{*2})^{T_{in}} ... (\mathbf{a}^{(1)}_{*K}...\mathbf{a}^{(N)}_{*K})^{T_{in}}.$$

Algorithm 1 describes the serial version of CDTF with T_{out} times of outer iteration. We initialize the entries of $\mathbf{A}^{(1)}$ to zero and those of all other factor matrices to random values, which makes the initial value of $\mathfrak{R}$ equal to $\mathfrak{X}$. Instead of computing $\hat{\mathfrak{R}}$ (line 4) before rank-one factorization, the entries of $\hat{\mathfrak{R}}$ can be computed while computing (6) and (7). This can result in better performance on a disk-based system like MAPREDUCE by eliminating disk I/O operations required to compute and store $\hat{\mathfrak{R}}$.

3.2 Subset Alternating Least Square

Subset alternating least square (SALS), another scalable tensor factorization algorithm, generalizes CDTF in terms of the number of parameters updated at a time. Figure 2 depicts the difference among CDTF, SALS, and ALS. Unlike CDTF, which updates each column of factor matrices entry by entry, and ALS, which updates all K columns row by row, SALS updates each C $(1 \leq C \leq K)$ columns row by row.

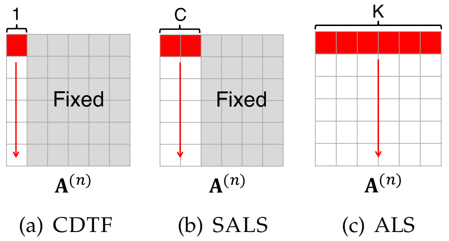


Fig. 2: Update rules of CDTF, SALS, and ALS. CDTF updates each column of factor matrices entry by entry, SALS updates each C $(1 \leq C \leq K)$ columns row by row, and ALS updates all K columns row by row.

Algorithm 2: Serial version of SALS

```
Input  : $\mathfrak{X}$, $K$, $\lambda$
Output: $\mathbf{A}^{(n)}$ for all $n$
1  initialize $\mathfrak{R}$ and $\mathbf{A}^{(n)}$ for all $n$
2  for outer iter = 1..$T_{out}$ do
3      for split iter = 1..$\lceil \frac{K}{C} \rceil$ do
4          choose $k_1, ..., k_C$ (from columns not updated yet)
5          compute $\hat{\mathfrak{R}}$ using (8)
6          for inner iter = 1..$T_{in}$ do
7              for $n$ = 1..$N$ do
8                  for $i_n$ = 1..$I_n$ do
9                      update $a^{(n)}_{i_n k_1}, ..., a^{(n)}_{i_n k_C}$ using (9)
10         update $\mathfrak{R}$ using (10)
```

SALS contains CDTF $(C = 1)$ and ALS $(C = K)$ as special cases. In other words, SALS and ALS are similar in that, when they update a factor matrix, they fix all other factor matrices at the current values and update the factor matrix in a row-wise manner by assigning the row vector minimizing (1) given all other parameters. However, SALS and ALS are different in that, when they update a row of a factor matrix, ALS updates all the K entries in the row at a time, while SALS updates only $C(\leq K)$ entries in the row fixing the other $(K-C)$ entries. Our experimental results in Section 5 show that, with proper C, SALS enjoys both the high scalability of CDTF and the fast convergence of ALS although SALS requires more memory space than CDTF.

Algorithm 2 describes the procedure of SALS. Line 4, where C columns are randomly chosen, and line 9, where C parameters are updated simultaneously, are the major differences from CDTF. Updating C columns $(k_1, ..., k_C)$ of all the factor matrices (lines 7 through 9) is equivalent to the rank-C factorization of $\hat{\mathfrak{R}}$ whose $(i_1, ..., i_N)$th entry is $\hat{r}_{i_1...i_N} = x_{i_1...i_N} - \sum_{k \notin \{k_1,...,k_C\}} \prod_{n=1}^{N} a^{(n)}_{i_n k}$. $\hat{\mathfrak{R}}$ can be computed from $\mathfrak{R}$ by the following rule (line 5):

$$\hat{r}_{i_1...i_N} = r_{i_1...i_N} + \sum_{c=1}^{C} \prod_{n=1}^{N} a^{(n)}_{i_n k_c}, \tag{8}$$

which takes $O(NC)$ for each entry and $O(|\Omega|NC)$ for entire $\hat{\mathfrak{R}}$. After computing $\hat{\mathfrak{R}}$, the parameters in the C columns are updated row by row as follows (line 9):

$$[a^{(n)}_{i_n k_1}, ..., a^{(n)}_{i_n k_C}]^T \leftarrow \underset{[a^{(n)}_{i_n k_1},...,a^{(n)}_{i_n k_C}]^T}{\arg\min} L(\mathbf{A}^{(1)}, ..., \mathbf{A}^{(N)}) = (\mathbf{B}^{(n)}_{i_n} + \lambda \mathbf{I}_C)^{-1} \mathbf{c}^{(n)}_{i_n} \tag{9}$$

where $\mathbf{B}_{i_n}^{(n)}$ is a C by C matrix whose entries are

$$(\mathbf{B}_{i_n}^{(n)})_{c_1c_2} = \sum_{(i_1,\ldots,i_N)\in\Omega_{i_n}^{(n)}} (\prod_{l\neq n} a_{i_lk_{c_1}}^{(l)} \prod_{l\neq n} a_{i_lk_{c_2}}^{(l)}), \forall c_1, c_2$$

$\mathbf{c}_{i_n}^{(n)}$ is a length C vector whose entries are

$$(\mathbf{c}_{i_n}^{(n)})_c = \sum_{(i_1,\ldots,i_N)\in\Omega_{i_n}^{(n)}} (\hat{r}_{i_1\ldots i_N} \prod_{l\neq n} a_{i_lk_c}^{(l)}), \forall c$$

and $\mathbf{I}_C$ is the C by C identity matrix. The correctness of this update rule is proved in Section 3.3.1. Computing (9) takes $O(|\Omega_{i_n}^{(n)}|C(N+C)+C^3)$, which consists of $O(|\Omega_{i_n}^{(n)}|NC)$ to compute $\prod_{l\neq n} a_{i_lk_1}^{(l)}$ through $\prod_{l\neq n} a_{i_lk_C}^{(l)}$ for all the entries in $\Omega_{i_n}^{(n)}$, $O(|\Omega_{i_n}^{(n)}|C^2)$ to build $\mathbf{B}_{i_n}^{(n)}$, $O(|\Omega_{i_n}^{(n)}|C)$ to build $\mathbf{c}_{i_n}^{(n)}$, and $O(C^3)$ to compute the inverse. Instead of computing the inverse, the Cholesky decomposition, which also takes $O(C^3)$, can be used for speed and numerical stability. After repeating this rank-C factorization T_{in} times (line 6), $\mathfrak{R}$ is updated by the following rule (line 10):

$$r_{i_1\ldots i_N} \leftarrow \hat{r}_{i_1\ldots i_N} - \sum_{c=1}^{C}\prod_{n=1}^{N} a_{i_nk_c}^{(n)}, \quad (10)$$

which takes $O(|\Omega|NC)$ for the entire $\mathfrak{R}$ as in (8).

3.3 Theoretical Analysis

3.3.1 Convergence Analysis

We prove that CDTF and SALS monotonically decrease the loss function (1) until convergence.

Theorem 1 (Correctness of SALS*):* The update rule (9) minimizes (1) with respect to the updated parameters. That is,

$$\underset{[a_{i_nk_1}^{(n)},\ldots,a_{i_nk_C}^{(n)}]^T}{\arg\min} L(\mathbf{A}^{(1)},\ldots,\mathbf{A}^{(N)}) = (\mathbf{B}_{i_n}^{(n)} + \lambda\mathbf{I}_C)^{-1}\mathbf{c}_{i_n}^{(n)}.$$

Proof:

$$\frac{\partial L}{\partial a_{i_nk_c}^{(n)}} = 0, \forall c, 1\leq c\leq C$$

$$\Leftrightarrow \sum_{(i_1,\ldots,i_N)\in\Omega_{i_n}^{(n)}} \left(\left(\sum_{s=1}^{C}\prod_{l=1}^{N} a_{i_lk_s}^{(l)} - \hat{r}_{i_1\ldots i_N}\right)\prod_{l\neq n} a_{i_lk_c}^{(l)}\right) + \lambda a_{i_nk_c}^{(n)} = 0, \forall c$$

$$\Leftrightarrow \sum_{(i_1,\ldots,i_N)\in\Omega_{i_n}^{(n)}} \left(\sum_{s=1}^{C}\left(a_{i_nk_s}^{(n)}\prod_{l\neq n} a_{i_lk_s}^{(l)}\right)\prod_{l\neq n} a_{i_lk_c}^{(l)}\right) + \lambda a_{i_nk_c}^{(n)} = \sum_{(i_1,\ldots,i_N)\in\Omega_{i_n}^{(n)}} (\hat{r}_{i_1\ldots i_N}\prod_{l\neq n} a_{i_lk_c}^{(l)}), \forall c$$

$$\Leftrightarrow (\mathbf{B}_{i_n}^{(n)} + \lambda\mathbf{I}_c)[a_{i_nk_1}^{(n)},\ldots,a_{i_nk_C}^{(n)}]^T = \mathbf{c}_{i_n}^{(n)}. \quad \square$$

This theorem also proves the correctness of CDTF, which is a special case of SALS, and leads to the following convergence property.

Theorem 2 (Convergence of CDTF *and* SALS*):* In CDTF and SALS, the loss function (1) decreases monotonically.

Proof: By Theorem 1, every update in CDTF and SALS minimizes (1) in terms of updated parameters. Thus, the loss function never increases. □

3.3.2 Complexity Analysis

In this section, we analyze the computational and space complexity of CDTF and SALS.

Theorem 3 (Computational complexity of SALS*):* The computational complexity of SALS (Algorithm 2) is $O(T_{out}|\Omega|NT_{in}K(N+C) + T_{out}T_{in}KC^2\sum_{n=1}^{N} I_n)$.

Proof: As explained in Section 3.2, updating each C parameters $(a_{i_nk_1}^{(n)},\ldots,a_{i_nk_C}^{(n)})$ using (9) (line 9 of Algorithm 2) takes $O(|\Omega_{i_n}^{(n)}|C(C+N)+C^3)$; and both computing $\hat{\mathfrak{R}}$ (line 5) and updating $\mathfrak{R}$ (line 10) take $O(|\Omega|NC)$. Thus, updating all the entries in C columns (lines 8 through 9) takes $O(|\Omega|C(C+N) + I_nC^3)$, and the rank C factorization (lines 7 through 9) takes $O(|\Omega|NC(N+C)+C^3\sum_{n=1}^{N} I_n)$. As a result, an outer iteration, which repeats the rank C factorization $T_{in}K/C$ times and both $\hat{\mathfrak{R}}$ and $\mathfrak{R}$ updates K/C times, takes $O(|\Omega|NT_{in}K(N+C)+T_{in}KC^2\sum_{n=1}^{N} I_n)+O(|\Omega|NK)$, where the second term is dominated. □

Theorem 4 (Space complexity of SALS*):* The memory requirement of SALS (Algorithm 2) is $O(C\sum_{n=1}^{N} I_n)$.

Proof: Since $\hat{\mathfrak{R}}$ computation (line 5 of Algorithm 2), rank C factorization (lines 7 through 9), and $\mathfrak{R}$ update (line 10) all depend only on the C columns of the factor matrices, the number of whose entries is $C\sum_{n=1}^{N} I_n$, the other $(K-C)$ columns need not be loaded into the memory. Thus, the columns of the factor matrices can be loaded by turns depending on $(k_1,\ldots,k_C)$ values. Moreover, updating C columns (lines 8 through 9) can be processed by streaming the entries of $\hat{\mathfrak{R}}$ from disk and processing them one by one instead of loading them all at once because the entries of $\mathbf{B}_{i_n}^{(n)}$ and $\mathbf{c}_{i_n}^{(n)}$ in (9) are the sum of the values calculated independently from each $\hat{\mathfrak{R}}$ entry. Likewise, $\hat{\mathfrak{R}}$ computation and $\mathfrak{R}$ update can also be processed by streaming $\mathfrak{R}$ and $\hat{\mathfrak{R}}$, respectively. □

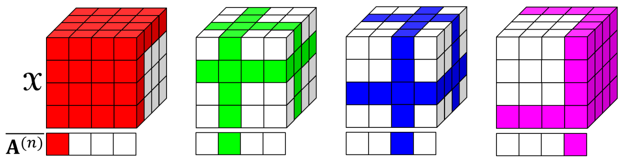
These theorems are applied to CDTF, which is a special case of SALS.

Theorem 5 (Computational complexity of CDTF*):* The computational complexity of CDTF (Algorithm 1) is $O(T_{out}|\Omega|N^2T_{in}K)$.

Theorem 6 (Space complexity of CDTF*):* The memory requirement of CDTF (Algorithm 1) is $O(\sum_{n=1}^{N} I_n)$.

3.4 Parallelization in Distributed Environments

In this section, we describe the distributed versions of CDTF and SALS. We assume a distributed environment where machines do not share memory, such as in MAPREDUCE. The extension to shared-memory systems, where the only difference is that we do not need to broadcast updated parameters, is straightforward. The method to assign the rows of the factor matrices to machines (i.e., to decide ${}_mS_n$ for all n and m) is explained in Section 3.4.3, and until then, we assume that the row assignments are given.



(a) Machine 1 (b) Machine 2 (c) Machine 3 (d) Machine 4

Fig. 3: Work and data distribution of CDTF and SALS in distributed environments when an input tensor is a 3-order tensor and the number of machines is four. We assume that the rows of the factor matrices are assigned to the machines sequentially. The colored region of $\overline{\mathbf{A}^{(n)}}$ (the transpose of $\mathbf{A}^{(n)}$) in each sub-figure corresponds to the parameters updated by each machine, resp., and that of $\mathcal{X}$ corresponds to the data distributed to each machine.

Algorithm 3: Distributed version of CDTF

```
Input : $\mathcal{X}$, $K$, $\lambda$, ${}_mS_n$ for all $m$ and $n$
Output: $\mathbf{A}^{(n)}$ for all $n$
1  distribute the ${}_m\Omega$ entries of $\mathcal{X}$ to each machine $m$
2  Parallel (P): initialize the ${}_m\Omega$ entries of $\mathcal{R}$
3  P: initialize $\mathbf{A}^{(n)}$ for all $n$
4  for outer iter = 1..$T_{out}$ do
5      for $k$ = 1..$K$ do
6          P: compute the ${}_m\Omega$ entries of $\hat{\mathcal{R}}$
7          for inner iter = 1..$T_{in}$ do
8              for $n$ = 1..$N$ do
9                  P: update ${}_m\mathbf{a}^{(n)}_{*k}$ using (6)
10                 P: broadcast ${}_m\mathbf{a}^{(n)}_{*k}$
11         P: update the ${}_m\Omega$ entries of $\mathcal{R}$ using (7)
```

Algorithm 4: Distributed version of SALS

```
Input : $\mathcal{X}$, $K$, $\lambda$, ${}_mS_n$ for all $m$ and $n$
Output: $\mathbf{A}^{(n)}$ for all $n$
1  distribute the ${}_m\Omega$ entries of $\mathcal{X}$ to each machine $m$
2  Parallel (P): initialize the ${}_m\Omega$ entries of $\mathcal{R}$
3  P: initialize $\mathbf{A}^{(n)}$ for all $n$
4  for outer iter = 1..$T_{out}$ do
5      for split iter = 1..$\lceil \frac{K}{C} \rceil$ do
6          choose $(k_1, ..., k_C)$ (from columns not updated yet)
7          P: compute the ${}_m\Omega$ entries of $\hat{\mathcal{R}}$
8          for inner iter = 1..$T_{in}$ do
9              for $n$ = 1..$N$ do
10                 P: update $\{a^{(n)}_{i_n k_c} | i_n \in {}_mS_n, 1 \le c \le C\}$
                   using (9)
11                 P: broadcast $\{a^{(n)}_{i_n k_c} | i_n \in {}_mS_n, 1 \le c \le C\}$
12         P: update the ${}_m\Omega$ entries of $\mathcal{R}$ using (10)
```

3.4.1 Distributed Version of CDTF

Work and Data Distribution. Since the update rule (6) for each parameter $a^{(n)}_{i_n k}$ does not depend on the other parameters in its column $\mathbf{a}^{(n)}_{*k}$, updating the parameters in a column can be distributed across multiple machines and processed simultaneously without affecting the updated results and thus correctness (Theorem 2). For each column $\mathbf{a}^{(n)}_{*k}$, machine m updates the assigned parameters, ${}_m\mathbf{a}^{(n)}_{*k} = \{a^{(n)}_{i_n k} | i_n \in {}_mS_n\}$. At matrix level, ${}_mS_n$ rows of each factor matrix $\mathbf{A}^{(n)}$ are updated by machine m. The entries of $\mathcal{X}$ necessary to update the assigned parameters are distributed to each machine. That is, $\mathcal{X}$ entries in ${}_m\Omega = \bigcup_{n=1}^{N} \left(\bigcup_{i_n \in {}_mS_n} \Omega^{(n)}_{i_n} \right)$ are distributed to each machine m, and the machine maintains and updates $\mathcal{R}$ entries in ${}_m\Omega$.

The sets of $\mathcal{X}$ entries sent to the machines are not disjoint (i.e., ${}_{m_1}\Omega \cap {}_{m_2}\Omega \neq \emptyset$), and each entry of $\mathcal{X}$ can be sent to N machines at most. Thus, the computational cost for computing $\hat{\mathcal{R}}$ and updating $\mathcal{R}$ increases up to N times in distributed environments. However, since this cost is dominated by the others, the overall asymptotic complexity remains the same. Figure 3 shows an example of work and data distribution.

Communication. In order to update the parameters, for example, in the kth column of a factor matrix using (6), the kth columns of all other factor matrices are required. Therefore, after updating the kth column of a factor matrix, the updated parameters are broadcast to all other machines so that they can be used to update the kth column of the next factor matrix. The broadcast parameters are also used to update the entries of $\mathcal{R}$ using (7). Therefore, after updating the assigned parameters in $\mathbf{a}^{(n)}_{*k}$, each machine m broadcasts $|{}_mS_n|$ parameters and receives $(I_n - |{}_mS_n|)$ parameters from the other machines. The number of parameters each machine exchanges with the other machines is $\sum_{n=1}^{N} I_n$ per rank-one factorization and $KT_{in} \sum_{n=1}^{N} I_n$ per outer iteration. Algorithm 3 depicts the distributed version of CDTF.

3.4.2 Distributed Version of SALS

SALS is also parallelized in distributed environments without affecting its correctness. The work and data distribution of SALS is exactly the same as that of CDTF, and the only difference between two methods is that in SALS each machine updates and broadcasts C columns at a time.

Algorithm 4 describes the distributed version of SALS. After updating the assigned parameters in C columns of a factor matrix $\mathbf{A}^{(n)}$ (line 10), each machine m sends $C|{}_mS_n|$ parameters and receives $C(I_n - |{}_mS_n|)$ parameters (line 11), and thus each machine exchanges $C \sum_{n=1}^{N} I_n$ parameters during rank C factorization (lines 9 through 11). Since this factorization is repeated $T_{in}K/C$ times, the total number of parameters each machine exchanges is $KT_{in} \sum_{n=1}^{N} I_n$ per outer iteration, as in CDTF.

3.4.3 Row Assignment

The running time of the parallel steps in the distributed versions of CDTF and SALS depends on the longest running time among all machines. Specifically, the running time of lines 6, 9, and 11 in Algorithm 3 and lines 7, 10, and 12 in Algorithm 4 are proportional to $\max_m |{}_m\Omega^{(n)}|$ where ${}_m\Omega^{(n)} = \bigcup_{i_n \in {}_mS_n} \Omega^{(n)}_{i_n}$. Line 10 in Algorithm 3 and line 11 in Algorithm 4 are proportional to $\max_m |{}_mS_n|$. Therefore, it is important to assign the rows of the factor matrices to the machines (i.e., to decide ${}_mS_n$) such that $|{}_m\Omega^{(n)}|$ and $|{}_mS_n|$ are

Algorithm 5: Greedy row assignment in CDTF and SALS

```
Input  : $\mathfrak{X}$, $M$
Output: ${}_mS_n$ for all $m$ and $n$
1  initialize $|{}_m\Omega|$ to 0 for all $m$
2  for $n$ = 1..$N$ do
3      initialize ${}_mS_n$ to $\emptyset$ for all $m$
4      initialize $|{}_m\Omega^{(n)}|$ to 0 for all $m$
5      calculate $|\Omega^{(n)}_{i_n}|$ for all $i_n$
6      foreach $i_n$ (in decreasing order of $|\Omega^{(n)}_{i_n}|$) do
7          find $m$ with $|{}_mS_n| < \lceil \frac{I_n}{M} \rceil$ and the smallest $|{}_m\Omega^{(n)}|$
8          (in case of a tie, choose the machine with smaller $|{}_mS_n|$,
           and if still a tie, choose one with smaller $|{}_m\Omega|$)
9          add $i_n$ to ${}_mS_n$
10         add $|\Omega^{(n)}_{i_n}|$ to $|{}_m\Omega^{(n)}|$ and $|{}_m\Omega|$
```

evenly distributed among all the machines. For this purpose, we design a greedy assignment algorithm which aims to minimize $\max_m |{}_m\Omega^{(n)}|$ under the condition that $|{}_mS_n|$ is completely even (i.e., $|{}_mS_n| = I_n/M$ for all n where M is the number of machines). For each factor matrix $\mathbf{A}^{(n)}$, we sort its rows in the decreasing order of $|\Omega^{(n)}_{i_n}|$ and assign the rows one by one to the machine m that satisfies $|{}_mS_n| < \lceil I_n/M \rceil$ and has the smallest $|{}_m\Omega^{(n)}|$ currently. Algorithm 5 describes the details of the algorithm. The effects of greedy row assignment on actual running times are described in Section 5.5. In the section, the greedy row assignment is compared with the two baselines:

- Sequential row assignment : Indices of the rows in each mode are divided into M sequential ranges, and the rows whose indices belong to the same range are assigned to the same machine (i.e., ${}_mS_n = \{i_n \in \mathbb{N} | \frac{I_n \times (m-1)}{M} < i_n \leq \frac{I_n \times m}{M}\}$).
- Random row assignment : Each row in each mode is assigned to a randomly chosen machine.

3.5 Loss Functions and Updates

In this section, we discuss how CDTF and SALS are applied to minimize various loss functions which have been found useful in previous studies. We specifically consider the PARAFAC decomposition with L_1 and weighted-L_2 regularization and non-negativity constraints. Coupled tensor factorization and factorization using the bias model are also considered.

3.5.1 PARAFAC with L_1 Regularization

L_1 regularization, which leads to sparse factor matrices, can be used in the PARAFAC decomposition instead of L_2 regularization. Sparser factor matrices are easier to interpret as well as require less storage. With L_1 regularization, we use the following loss function instead of (1):

$$L_{Lasso}(\mathbf{A}^{(1)}, ..., \mathbf{A}^{(N)}) = \sum_{(i_1,...,i_N)\in\Omega} \left(x_{i_1...i_N} - \sum_{k=1}^{K} \prod_{n=1}^{N} a^{(n)}_{i_nk} \right)^2 + \lambda \sum_{n=1}^{N} \|\mathbf{A}^{(n)}\|_1. \tag{11}$$

To minimize (11), CDTF (see Section 3.1), which updates one parameter at a time while fixing the others, uses the equation (12), instead of the original equation (6), for updating each parameter $a^{(n)}_{i_nk}$:

$$a^{(n)}_{i_nk} \leftarrow \arg\min_{a^{(n)}_{i_nk}} L_{Lasso}(\mathbf{A}^{(1)}, ..., \mathbf{A}^{(N)}) = \begin{cases} (\lambda - g)/d & \text{if g} > \lambda \\ -(\lambda + g)/d & \text{if g} < \text{-}\lambda \\ 0 & \text{otherwise} \end{cases} \tag{12}$$

where $g = -2\sum_{(i_1,...,i_N)\in\Omega^{(n)}_{i_n}} \left(\hat{r}_{i_1...i_N} \prod_{l\neq n} a^{(l)}_{i_lk} \right)$ and $d = 2\sum_{(i_1,...,i_N)\in\Omega^{(n)}_{i_n}} \prod_{l\neq n} (a^{(l)}_{i_lk})^2$. The proof of this update rule can be found in Theorem 7 in [18]. If we simply replace (6) with (12), Algorithms 1 and 3 can be used to minimize (11) without other changes.

However, SALS is not directly applicable to (11) since a closed form solution for minimizing the loss function with respect to C parameters dose not exist if $C \geq 2$. Although an iterative or suboptimal update rule can be used instead of (9), this is beyond the scope of this paper.

3.5.2 PARAFAC with Weighted L_2 Regularization

Weighted L_2 regularization [3] has been used in many recommender systems based on matrix factorization [1], [2], [3] since it effectively prevents overfitting. Applying this method to the PARAFAC decomposition results in the following loss function:

$$L(\mathbf{A}^{(1)}, ..., \mathbf{A}^{(N)}) = \sum_{(i_1,...,i_N)\in\Omega} \left(x_{i_1...i_N} - \sum_{k=1}^{K} \prod_{n=1}^{N} a^{(n)}_{i_nk} \right)^2 + \lambda \sum_{n=1}^{N} \sum_{i_n=1}^{I_n} |\Omega^{(n)}_{i_n}| \cdot \|\mathbf{a}^{(\mathbf{n})}_{\mathbf{i_n}*}\|_2^2. \tag{13}$$

CDTF and SALS can be used to minimize (13) if λ in (6) and (9) is replaced with $\lambda|\Omega^{(n)}_{i_n}|$. This can be proved also by replacing λ in Theorem 1 and its proof with $\lambda|\Omega^{(n)}_{i_n}|$. With these changes, Algorithms 1, 2, 3, and 4 can be used to minimize (13) without other changes.

3.5.3 PARAFAC with Non-negativity Constraint

The constraint that factor matrices have only non-negative entries has been adopted in many applications [11], [19] because the resulting factor matrices are easier to inspect. To minimize (1) under this non-negativity constraint, CDTF (see Section 3.1) updates each parameter $a^{(n)}_{i_nk}$ by assigning the nonnegative value minimizing (1) when the other parameters are fixed at their current values. That is,

$$a^{(n)}_{i_nk} \leftarrow \arg\min_{a^{(n)}_{i_nk}\geq 0} L(\mathbf{A}^{(1)}, ..., \mathbf{A}^{(N)}) = \max\left(\frac{-g}{d+2\lambda}, 0 \right), \tag{14}$$

where $g = -2\sum_{(i_1,...,i_N)\in\Omega^{(n)}_{i_n}} \left(\hat{r}_{i_1...i_N} \prod_{l\neq n} a^{(l)}_{i_lk} \right)$ and $d = 2\sum_{(i_1,...,i_N)\in\Omega^{(n)}_{i_n}} \prod_{l\neq n} (a^{(l)}_{i_lk})^2$. See Theorem 8 in [18] for the proof of this update rule. Algorithms 1 and 3 can be used to minimize (1) under the non-negativity constraint if we simply replace (6) with (14).

However, SALS cannot be directly used with the non-negativity constraint since a closed form solution for minimizing (1) with respect to C parameters does not exist under the constraint if $C \geq 2$. Although an iterative or suboptimal update rule can be used instead of (9), this is beyond the scope of this paper.

3.5.4 Coupled Tensor Factorization

The joint factorization of tensors sharing common modes (e.g., user-movie rating data and a social network among users) can provide a better understanding of latent factors [11], [20]. Suppose that an N_x-order tensor $\mathfrak{X}$ and an N_y-order tensor $\mathfrak{Y}$ share their first mode without loss of generality. The coupled factorization of them decomposes $\mathfrak{X}$ into factor matrices ${}_x\mathbf{A}^{(1)}$ through ${}_x\mathbf{A}^{(N_x)}$ and $\mathfrak{Y}$ into ${}_y\mathbf{A}^{(1)}$ through ${}_y\mathbf{A}^{(N_y)}$ under the condition that ${}_x\mathbf{A}^{(1)} = {}_y\mathbf{A}^{(1)}$. The loss function of the coupled factorization is the sum of the loss function in each separate factorization (i.e., equation (1)) as follows:

$$L_{Coupled}({}_x\mathbf{A}^{(1)},...,{}_x\mathbf{A}^{(N_x)},{}_y\mathbf{A}^{(1)},...,{}_y\mathbf{A}^{(N_y)}) = L({}_x\mathbf{A}^{(1)},...,{}_x\mathbf{A}^{(N_x)}) + L({}_y\mathbf{A}^{(1)},...,{}_y\mathbf{A}^{(N_y)}) - \lambda\|{}_x\mathbf{A}^{(1)}\|_F^2 \quad (15)$$

where $-\lambda\|{}_x\mathbf{A}^{(1)}\|_F^2$ prevents the shared matrix from being regularized twice.

The update rule for the entries in the non-shared factor matrices are the same as that in separate factorization. When updating the entries in the shared matrix, however, we should consider both tensors. In SALS (see Section 3.2), if $L({}_x\mathbf{A}^{(1)},...,{}_x\mathbf{A}^{(N_x)})$ is minimized at $[a^{(1)}_{i_1k_1},...,a^{(1)}_{i_1k_C}]^T = ({}_x\mathbf{B}^{(1)}_{i_1} + \lambda\mathbf{I}_C)^{-1}{}_x\mathbf{c}^{(1)}_{i_1}$ and $L({}_y\mathbf{A}^{(1)},...,{}_y\mathbf{A}^{(N_y)})$ is minimized at $({}_y\mathbf{B}^{(1)}_{i_1} + \lambda\mathbf{I}_C)^{-1}{}_y\mathbf{c}^{(1)}_{i_1}$ (see (9) for the notations), $L_{Coupled}$ is minimized at $({}_x\mathbf{B}^{(1)}_{i_1} + {}_y\mathbf{B}^{(1)}_{i_1} + \lambda\mathbf{I}_C)^{-1}({}_x\mathbf{c}^{(1)}_{i_1} + {}_y\mathbf{c}^{(1)}_{i_1})$, given the other parameters. That is, to minimize (15), SALS updates each C parameter $[a^{(1)}_{i_1k_1},...,a^{(1)}_{i_1k_C}]^T$ in the shared factor matrix by the following rule:

$$[a^{(1)}_{i_1k_1},...,a^{(1)}_{i_1k_C}]^T \leftarrow \underset{[a^{(1)}_{i_1k_1},...,a^{(1)}_{i_1k_C}]^T}{\arg\min}\; L_{Coupled}({}_x\mathbf{A}^{(1)},...,{}_x\mathbf{A}^{(N_x)},{}_y\mathbf{A}^{(1)},...,{}_y\mathbf{A}^{(N_y)})$$
$$= ({}_x\mathbf{B}^{(1)}_{i_1} + {}_y\mathbf{B}^{(1)}_{i_1} + \lambda\mathbf{I}_C)^{-1}({}_x\mathbf{c}^{(1)}_{i_1} + {}_y\mathbf{c}^{(1)}_{i_1}) \quad (16)$$

See Theorem 9 and Algorithm 6 in [18] for the proof of this update rule and the pseudocode for coupled tensor factorization. CDTF, a special case of SALS, also can be used for minimizing (15) in the same way.

3.5.5 Bias Model

The PARAFAC decomposition (see Section 2.2) captures interactions among modes that lead to different entry values in $\mathfrak{X}$. In recommendation problems, however, much of the variation in entry values (i.e., rates) is explained by each mode solely. For this reason, [3] added bias terms, which capture the effect of each mode, in their matrix factorization model. Likewise, we can add bias vectors $\{\mathbf{b}^{(n)} \in \mathbb{R}^{I_n} | 1 \leq n \leq N\}$ to (1) resulting in the following loss function:

$$L_{Bias}(\mathbf{A}^{(1)},...,\mathbf{A}^{(N)},\mathbf{b}^{(1)},...,\mathbf{b}^{(N)}) = \sum_{(i_1,...,i_N)\in\Omega}\left(x_{i_1...i_N} - \mu - \sum_{n=1}^{N} b^{(n)}_{i_n} - \sum_{k=1}^{K}\prod_{n=1}^{N} a^{(n)}_{i_nk}\right)^2 + \lambda_{\mathbf{A}}\sum_{n=1}^{N}\|\mathbf{A}^{(n)}\|_F^2 + \lambda_{\mathbf{b}}\sum_{n=1}^{N}\|\mathbf{b}^{(n)}\|_2^2 \quad (17)$$

where μ denotes the average entry value of $\mathfrak{X}$, and $\lambda_{\mathbf{A}}$ and $\lambda_{\mathbf{b}}$ denote the regularization parameters for factor matrices and bias terms, respectively.

With (17), each $(i_1,...i_N)$th entry of the residual tensor $\mathfrak{R}$ is redefined as $r_{i_1...i_n} = x_{i_1...i_N} - \mu - \sum_{n=1}^{N} b^{(n)}_{i_n} - \sum_{k=1}^{K}\prod_{n=1}^{N} a^{(n)}_{i_nk}$. At each outer iteration, we first update factor matrices by either CDTF or SALS and then update bias vectors in the CDTF manner. That is, we update each parameter $b^{(n)}_{i_n}$ at a time, while fixing the other parameters at their current values, by the following update rule:

$$b^{(n)}_{i_n} \leftarrow \underset{b^{(n)}_{i_n}}{\arg\min}\; L_{Bias}(\mathbf{A}^{(1)},...,\mathbf{A}^{(N)},\mathbf{b}^{(1)},...,\mathbf{b}^{(N)}) = \sum_{(i_1,...,i_N)\in\Omega^{(n)}_{i_n}} \bar{r}_{i_1...i_N}/(\lambda_{\mathbf{b}} + |\Omega^{(n)}_{i_n}|), \quad (18)$$

where $\bar{r}_{i_1...i_N} = r_{i_1...i_N} + b^{(n)}_{i_n}$. The proof of this update rule can be found in Theorem 10 in the supplementary document [18]. After updating each $b^{(n)}_{i_n}$, the entries of $\mathfrak{R}$ in $\{r_{i_1...i_N} | (i_1,...,i_N) \in \Omega^{(n)}_{i_n}\}$ should be updated by the following rule:

$$r_{i_1...i_N} \leftarrow r_{i_1...i_N} + (b^{(n)}_{i_n})^{old} - b^{(n)}_{i_n} \quad (19)$$

where $(b^{(n)}_{i_n})^{old}$ is the old parameter value. Algorithm 7 in [18] gives the detailed pseudocode of SALS for the bias model. It includes CDTF for the bias model as a special case.

4 Optimization on MapReduce

In this section, we describe the optimization techniques used to implement CDTF and SALS on MapReduce, which is one of the most widely used distributed platforms. Theses techniques can also be applied to diverse distributed algorithms, including ALS, FlexiFact, and PSGD. The effect of these techniques on actual running time is in Section 5.5.

4.1 Local Disk Caching

The typical MapReduce implementation of CDTF and SALS without local disk caching runs each parallel step (e.g., parameter $(a^{(n)}_{*k})$ update, $\mathfrak{R}$ update, and $\hat{\mathfrak{R}}$ computation in Algorithm 3) as a separate MapReduce job, which incurs redistributing data across machines. Repeatedly redistributing data is extremely inefficient in SALS and CDTF due to their highly iterative nature. We avoid this inefficiency by caching data to local disk once they are distributed. In our implementation of CDTF and SALS with local disk caching, $\mathfrak{X}$ entries are distributed across machines and cached in their local disk during the map and reduce stages (see Algorithm 8 in the supplementary document [18] for the detailed pseudocode); and the rest of CDTF and SALS runs in the close stage (cleanup stage in Hadoop) using the cached data.

4.2 Direct Communication

In MapReduce, it is generally assumed that reducers run independently and do not communicate directly

TABLE 4: Scale of synthetic datasets. B: billion, M: million, K: thousand. The length of every mode is equal to I.

	S1	**S2** (default)	**S3**	**S4**
N	2	3	4	5
I	300K	1M	3M	10M
$\|\Omega\|$	30M	100M	300M	1B
K	30	100	300	1K

with each other. However, CDTF and SALS require updated parameters to be broadcast to other reducers. We circumvent this problem by adapting the direct communication method used in FlexiFaCT [11]. Each reducer writes the parameters that it updated to the distributed file system, and the other reducers read these parameters from the distributed file system. See Section 2.2 of [18] for details including a pseudocode.

4.3 Greedy Row Assignment

We also apply the greedy row assignment, explained in Section 3.4.3, to our MAPREDUCE implementations. See Section 2.3 of [18] for the detailed MAPREDUCE implementation of the greedy row assignment.

5 EXPERIMENTS

Our experimental results answer the questions below:

- **Q1: Data scalability (Section 5.2).** How do CDTF, SALS, and their competitors scale with regard to the order, number of observations, mode length, and rank of an input tensor?
- **Q2: Machine scalability (Section 5.3).** How do CDTF, SALS, and their competitors scale with regard to the number of machines?
- **Q3: Convergence (Section 5.4).** How quickly and accurately do CDTF, SALS, and their competitors factorize real-world tensors?
- **Q4: Optimization (Section 5.5).** How much do local disk caching and greedy row assignment improve the speed of CDTF and SALS? Can these techniques be applied to other methods?
- **Q5: Effects of C and T_{in} (Section 5.6)** How do the number of columns updated at a time (C) and the number of inner iterations (T_{in}) affect the convergence of SALS and CDTF?

All experiments are focused on distributed methods, which are the most suitable to achieve our purpose of handling large-scale data. We compared our methods with ALS, FLEXIFACT, and PSGD, all of which can be parallelized in distributed environments, as explained in Section 2.3. Serial methods (e.g., [17], [21]) and methods not directly applicable to partially observable tensors (e.g., [22], [23]) were not considered.

5.1 Experimental Settings

We ran experiments on a 40-node Hadoop cluster. Each node had an Intel Xeon E5620 2.4GHz CPU. The maximum heap size per reducer was set to 8GB.

We used both synthetic (Table 4) and real-world (Table 5) datasets most of which are available at **http://www.cs.cmu.edu/~kijungs/codes/cdtf**. Synthetic tensors were created as synthetic matrices were created in [24]. That is, we first created ground-truth factor matrices with random entries. Then, we randomly sampled $|\Omega|$ entries of the created tensor and set each sampled entry $x_{i_1 \ldots i_N}$ to $\delta + \sum_{k=1}^{K} \prod_{n=1}^{N} a_{i_n k}^{(n)}$ where δ denotes random noise with mean 0 and variance 1. The real-world tensors are preprocessed as follows:

TABLE 5: Summary of real-world datasets.

	Movielens$_4$	**Netflix$_3$**	**Yahoo-music$_4$**
N	4	3	4
I_1	71,567	2,649,429	1,000,990
I_2	65,133	17,770	624,961
I_3 / I_4	169 / 24	74 / -	133 / 24
$\|\Omega\|$	9,301,274	99,072,112	252,800,275
$\|\Omega\|_{test}$	698,780	1,408,395	4,003,960
K	20	40	80
$\lambda(=\lambda_{\mathbf{A}})$	0.01	0.02	1.0
$\lambda_{\mathbf{b}}$	1.0	0.02	1.0
η_0	0.01	0.01	10^{-5} (FLEXIFACT) 10^{-4} (PSGD)

- **Movielens$_4$[1]:** Movie rating data from MovieLens, an online movie recommender service. We converted them into a four-order tensor where the third and fourth modes correspond to (year, month) and hour-of-day when the movie was rated, respectively. The rates range from 1 to 5.
- **Netflix$_3$[2]:** Movie rating data used in Netflix prize. We regarded them as a three-order tensor where the third mode is (year, month) when the movie was rated. The rates range from 1 to 5.
- **Yahoo-music$_4$[3]:** Music rating data used in KDD CUP 2011. We converted them into a four-order tensor in the same way as we did for Movielens$_4$. The rates range from 0 to 100.

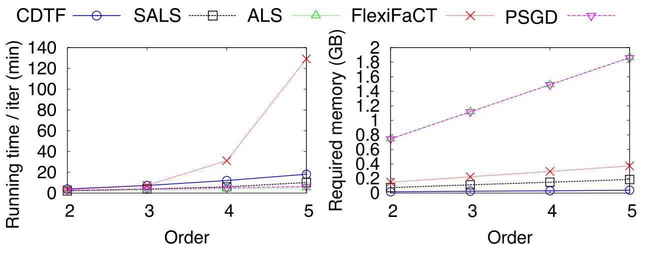
All the methods were implemented in Java with Hadoop 1.0.3. The local disk caching, the direct communication, and the greedy row assignment (see Section 4) were applied to all the methods if possible. All our implementations used weighted L_2 regularization (see Section 3.5.2). For CDTF and SALS, T_{in} was set to 1 and C was set to 10, unless otherwise stated. The learning rate of FLEXIFACT and PSGD at tth iteration was set to $2\eta_0/(1+t)$, as in the open-sourced FLEXIFACT[4]. The number of reducers was set to 5 for FLEXIFACT, 20 for PSGD, and 40 for the other methods, each of which leaded to the best performance on the machine scalability test in Section 5.3.

5.2 Data Scalability

5.2.1 Scalability with Each Factor (Figures 4-7)

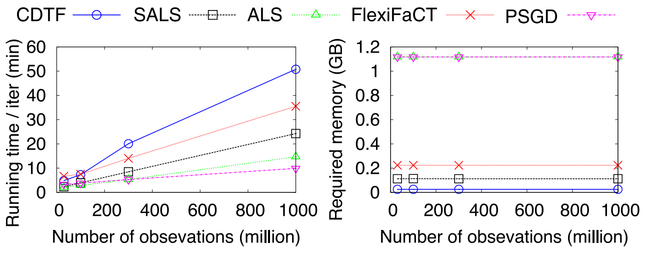
We measured the scalability of CDTF, SALS, and the competitors with regard to the order, number of observations, mode length, and rank of an input tensor. When measuring the scalability with regard to

1. http://grouplens.org/datasets/movielens
2. http://www.netflixprize.com
3. http://webscope.sandbox.yahoo.com/catalog.php?datatype=c
4. http://alexbeutel.com/l/flexifact/



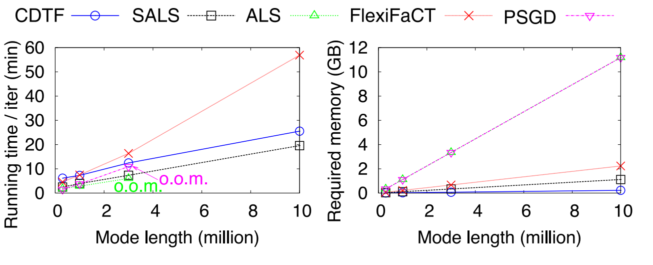
(a) Running time (b) Memory requirements

Fig. 4: Scalability w.r.t. order. FLEXIFACT was not scalable with order, while the other methods, including CDTF and SALS, were scalable with order.



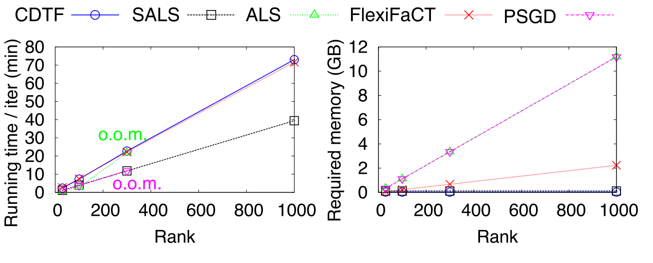
(a) Running time (b) Memory requirements

Fig. 5: Scalability w.r.t. the number of observations. All methods, including CDTF and SALS, were scalable with the number of observable entries.



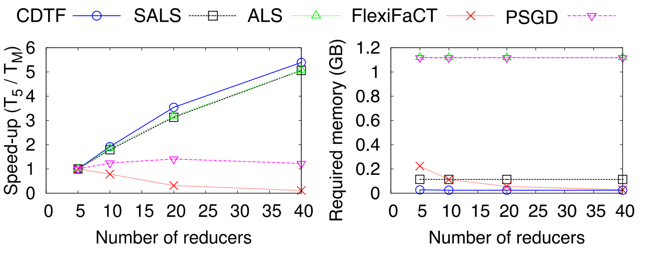
(a) Running time (b) Memory requirements

Fig. 6: Scalability w.r.t. mode length. o.o.m.: out of memory. ALS and PSGD did not scale with mode length, while CDTF, SALS, and FLEXIFACT did.



(a) Running time (b) Memory requirements

Fig. 7: Scalability w.r.t. rank. o.o.m.: out of memory. ALS and PSGD did not scale with rank, while CDTF, SALS, and FLEXIFACT did.



(a) Speed-up (b) Memory requirements

Fig. 8: Machine scalability on the S2 scale dataset. FLEXIFACT and PSGD did not scale with the number of machines, while CDTF, SALS, and ALS did.

a factor, the factor was scaled up from S1 to S4 while all other factors were fixed at S2 in Table 4.

As seen in Figure 4, FLEXIFACT did not scale with the order because of its communication cost, which increases exponentially with the order. ALS and PSGD were not scalable with the mode length and the rank due to their high memory requirements as Figures 6 and 7 show. They required up to 11.2GB, which is 48× of 234MB that CDTF required and 10× of 1,147MB that SALS required. Moreover, the running time of ALS increased rapidly with rank owing to its cubically increasing computational cost.

Only SALS and CDTF were scalable with all the factors, as summarized in Table 1. Their running times increased linearly with all the factors except the order, with which they increased slightly faster due to the quadratically increasing computational cost.

5.2.2 *Overall scalability (Figure 1 in Section 1)*

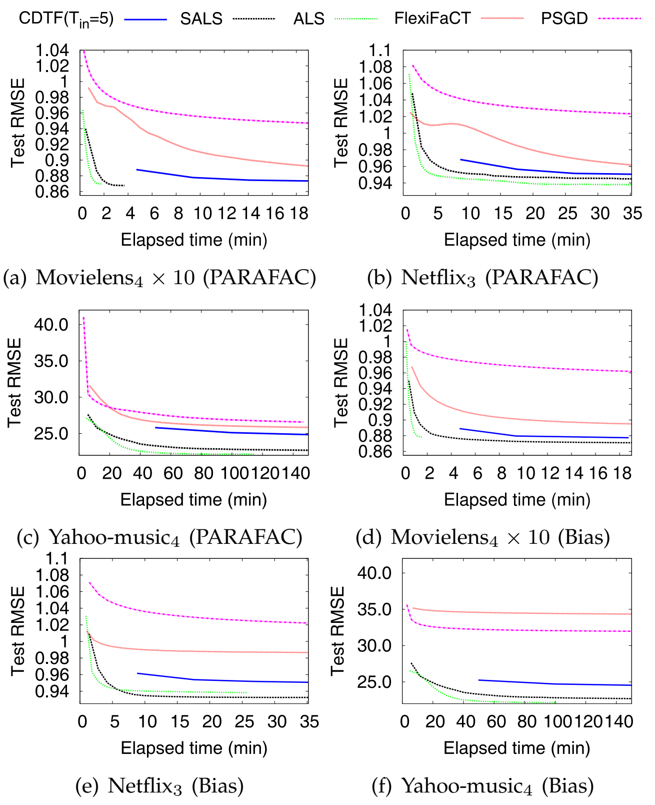
We measured the scalability of the methods by scaling up all the factors simultaneously from S1 to S4. The scalability of FLEXIFACT with five machines, ALS, and PSGD was limited owing to their high memory requirements. ALS and PSGD required about 186GB to handle S4, which is 493× of 387MB that CDTF required and 100× of 1,912MB that SALS required. FLEXIFACT with 40 machines did not scale over S2 due to its rapidly increasing communication cost. Only CDTF and SALS scaled up to S4, and there was a trade-off between them: CDTF was more memory-efficient, and SALS ran faster.

5.3 Machine Scalability (Figure 8)

We measured the speed-ups (T_5/T_M where T_M is the running time with M reducers) and memory requirements of the methods on the S2 scale dataset by increasing the number of reducers. The speed-ups of CDTF, SALS, and ALS increased linearly at the beginning and then flattened out slowly owing to their fixed communication cost, which does not depend on the number of reducers (see Table 3). The speed-up of PSGD flattened out fast, and PSGD even slightly slowed down at 40 reducers because of the increased overhead. FLEXIFACT slowed down as the number of reducers increased because of its rapidly increasing communication cost. The memory requirements of FLEXIFACT decreased as the number of reducers increased, while the other methods required the fixed amounts of memory.

5.4 Convergence (Figure 9)

We compared how quickly and accurately each method factorizes real-world tensors using the PARAFAC model (Section 2.2) and the bias model (Section 3.5.5). Accuracies were measured per iteration by root mean square error (RMSE) on a held-out test set, which is commonly used by recommender



(a) Movielens$_4$ × 10 (PARAFAC) (b) Netflix$_3$ (PARAFAC) (c) Yahoo-music$_4$ (PARAFAC) (d) Movielens$_4$ × 10 (Bias) (e) Netflix$_3$ (Bias) (f) Yahoo-music$_4$ (Bias)

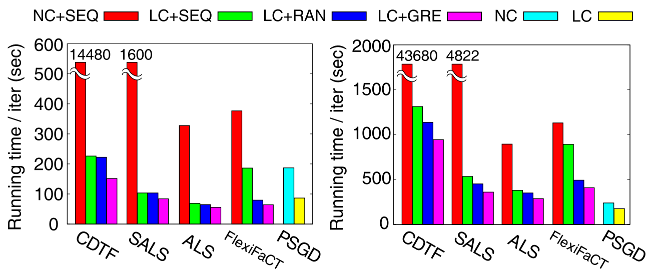
Fig. 9: Convergence speed on real-world datasets. (a)-(c) show the results with the PARAFAC model, and (d)-(e) show the results with the bias model (Section 3.5.5). SALS and ALS converged fastest to the best solution, and CDTF followed them. CDTF and SALS, however, have much better scalability than ALS as shown in Section 5.2.

systems. We used cross validation for K, λ, and η_0, whose values used are in Table 5. Since the Movielens$_4$ dataset (183MB) is too small to run on Hadoop, we increased its size by duplicating each user 10 times. Due to the non-convexity of (1), the methods converged to local minima with different accuracies.

In all datasets, SALS was comparable with ALS, which converged fastest to the best solution, and CDTF followed them. CDTF and SALS, however, have much better scalability than ALS as shown in Section 5.2. PSGD converged slowest to the worst solution because of the non-identifiability of (1).

5.5 Optimization (Figure 10)

We measured how our proposed optimization techniques, specifically the local disk caching and the greedy row assignment, affect the running time of CDTF, SALS, and the competitors on real-world datasets. The local disk caching speeded up CDTF up to 65.7×, SALS up to 15.5×, and the competitors up to 4.8×. The speed-ups of SALS and CDTF were the most significant because of their highly iterative nature. Additionally, the greedy row assignment speeded up CDTF up to 1.5×; SALS up to 1.3×; and the competitors up to 1.2× compared with the second best one. It is not applicable to PSGD, which does not distribute parameters in a row-wise manner.



(a) Netflix$_3$ (b) Yahoo-music$_4$

Fig. 10: Effects of the optimization techniques on running times. NC: no caching, LC: local disk caching, SEQ: sequential row assignment, RAN: random row assignment, GRE: greedy row assignment (see Section 3.4.3 for the row assignment methods). Our proposed optimization techniques (LC+GRE) significantly speeded up CDTF, SALS, and also their competitors.

5.6 Effects of C and T_{in} (Figures 11 and 12)

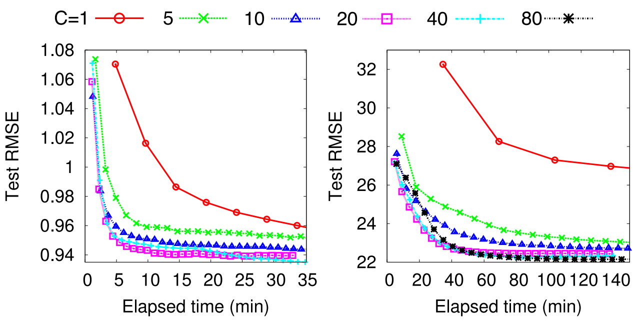
Figure 11 compares the convergence properties of SALS with different C values when T_{in} is fixed to one. As C increased, SALS tended to converge to better solutions (with lower test RMSE) although it required more memory space proportional to C (see Theorem 4). When considering this trade-off, we suggest that C be set to the largest value where memory requirements do not exceed the available memory.

We also compared the convergence properties of CDTF with different T_{in} values in Figure 12. Although there was an exception (T_{in}=1 in the Netflix$_3$ dataset), CDTF tended to converge to better solutions (with lower test RMSE) more stably as T_{in} increased. In SALS with larger C values, however, the effects of inner iterations on its convergence property were marginal (see Figure 14 in the supplementary document [18] for the detailed experimental results). We suggest that C should be set first as described above. Then, if $C = 1$ (or equivalently CDTF is used), T_{in} should be set to a high enough value, which was ten in our experiments. Otherwise, T_{in} can be set to one.

6 Related Work

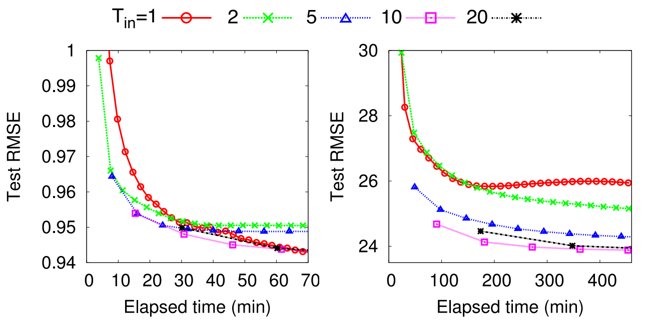
Matrix Factorization. Matrix factorization (MF) has been successfully used in many recommender systems [1], [2], [3]. The underlying intuition of MF for recommendation can be found in [2]. Two major approaches for large-scale MF are alternating least squares (ALS) and stochastic gradient descent (SGD). ALS is inherently parallelizable [3] but has high memory requirements and computational cost [5]. Efforts were made to parallelize SGD [4], [10], [25], including distributed stochastic gradient descent (DSGD) [4], which divides a matrix into disjoint blocks and processes them simultaneously. Recently, coordinate descent was also applied to large-scale MF [5].

Fully Observable Tensor Factorization. Fully observable tensor factorization is a basis for many applications, including chemometrics [14] and signal processing [26]. Comprehensive survey on tensor factorization can be found in [15], [27]. To factorize large-



(a) Netflix$_3$ (b) Yahoo-music$_4$

Fig. 11: Effects of the number of columns updated at a time (C) on the convergence of SALS. SALS tended to converge to better solutions as C increased.



(a) Netflix$_3$ (b) Yahoo-music$_4$

Fig. 12: Effects of inner iterations (T_{in}) on the convergence of CDTF (or equivalently SALS with $C = 1$). CDTF tended to converge stably to better solutions as T_{in} increased.

scale tensors, several approaches have been proposed. [22], [23], [28], [29] carefully reorder operations in the standard ALS or GD algorithm to reduce intermediate data, while [30] modifies the standard ALS algorithm itself for the same purpose. In [31], [32], a tensor is divided into small subtensors, and each subtensor is factorized. Then, the factor matrices of the entire tensor are reconstituted from those of subtensors. In [26], [33], a tensor is compressed before being factorized. Among these methods, [22], [23], [28], [29], [31], [32] run on multiple machines in a distributed way. However, all these methods assume that input tensors are fully observable without missing entries. Thus, they are not directly applicable to partially observable tensors (e.g., rating data where most entries are missing), which we consider in this work.

Partially Observable Tensor Factorization. The factorization of partially observable tensors (i.e., tensors with missing entries) has been used in many fields such as computer vision [34], chemometrics [17], social network analysis [12], and Web search [13]. Recently, it also has been used in recommender systems to utilize additional contextual information such as time and location [6], [7], [8], [9]. Moreover, even when input tensors are fully observable, [35] suggests that converting them to partially observable tensors by random sampling and factorizing the converted tensors can be computationally and space efficient. [17] proposes two approaches for partially observable tensor factorization: (1) combining imputation with the standard ALS algorithm, and (2) fitting the model only to observable entries. Since the former approach has scalability issues, the latter approach has been taken in gradient-based optimization methods [11], [21] as well as our methods. [11] is the state-of-the-art distributed algorithm for partially observable tensor factorization, but it has limited scalability w.r.t. the order and the number of machines as shown in Section 5. Our methods overcome these limitations by using coordinate descent and its generalization instead of gradient-based optimization methods. Both our methods and [11] supports coupled tensor factorization, and [28] also can be used for joint factorization with a partially observable matrix (see Appendix of [28]) although [28] itself is a fully observable tensor factorization method.

MAPREDUCE, Hadoop, and Alternative Frameworks. In this work, we implement our methods on Hadoop, an open-source implementation of MAPREDUCE [36], because of its wide availability. Hadoop is one of the most widely-used distributed computing frameworks and offered by many cloud services (EC2, Azure, Google Cloud, etc.) so that it can be used without any hardware or setup expertise. Due to the same reason, a variety of data mining algorithms, including matrix and tensor factorization [4], [11], have been implemented on Hadoop. However, its limited computational model and inefficiency due to the repeated redistribution of data also have been pointed out as problems [5], [37]. We address this problem by implementing local disk caching and broadcast communication on the application level without modifying Hadoop itself. Another, possibly simpler, solution is to use other implementations of MAPREDUCE [38], [39] or Spark [40], which support local disk caching and broadcast communication as native features. Especially, Spark has gained rapid adoption due to its fast speed and rich features.

7 CONCLUSION

In this paper, we propose two distributed algorithms for high-order and large-scale tensor factorization: CDTF and SALS. They decompose a tensor with a given rank through a series of lower rank factorization. CDTF and SALS have advantages over each other in terms of memory usage and convergence speed, respectively. We compared our methods with other state-of-the-art distributed methods both analytically and experimentally. Only CDTF and SALS are scalable with all aspects of data (i.e., order, the number of observable entries, mode length, and rank) and successfully factorized a 5-order tensor with 1B observable entries, 10M mode length, and 1K rank with up to 493× less memory requirement. We implemented our methods on top of MAPREDUCE with two widely-applicable optimization techniques, which accelerated not only CDTF and SALS (up to 98.2×), but also the competitors (up to 5.9×).

References

[1] P.-L. Chen, C.-T. Tsai, Y.-N. Chen, K.-C. Chou, C.-L. Li, C.-H. Tsai, K.-W. Wu, Y.-C. Chou, C.-Y. Li, W.-S. Lin, *et al.*, "A linear ensemble of individual and blended models for music rating prediction," *KDDCup 2011 Workshop*, 2011.

[2] Y. Koren, R. Bell, and C. Volinsky, "Matrix factorization techniques for recommender systems," *Computer*, vol. 42, no. 8, pp. 30–37, 2009.

[3] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan, "Large-scale parallel collaborative filtering for the netflix prize," in *AAIM*, pp. 337–348, 2008.

[4] R. Gemulla, E. Nijkamp, P. Haas, and Y. Sismanis, "Large-scale matrix factorization with distributed stochastic gradient descent," in *KDD*, 2011.

[5] H.-F. Yu, C.-J. Hsieh, S. Si, and I. S. Dhillon, "Scalable coordinate descent approaches to parallel matrix factorization for recommender systems," in *ICDM*, 2012.

[6] A. Karatzoglou, X. Amatriain, L. Baltrunas, and N. Oliver, "Multiverse recommendation: N-dimensional tensor factorization for context-aware collaborative filtering," in *RecSys*, 2010.

[7] A. Nanopoulos, D. Rafailidis, P. Symeonidis, and Y. Manolopoulos, "Musicbox: Personalized music recommendation based on cubic analysis of social tags," *IEEE TASLP*, vol. 18, no. 2, pp. 407–412, 2010.

[8] V. W. Zheng, B. Cao, Y. Zheng, X. Xie, and Q. Yang, "Collaborative filtering meets mobile recommendation: A user-centered approach.," in *AAAI*, 2010.

[9] H. Lamba, V. Nagarajan, K. Shin, and N. Shajarisales, "Incorporating side information in tensor completion," in *WWW Companion*, 2016.

[10] R. McDonald, K. Hall, and G. Mann, "Distributed training strategies for the structured perceptron," in *HLT-NAACL*, 2010.

[11] A. Beutel, A. Kumar, E. E. Papalexakis, P. P. Talukdar, C. Faloutsos, and E. P. Xing, "Flexifact: Scalable flexible factorization of coupled tensors on hadoop.," in *SDM*, 2014.

[12] D. M. Dunlavy, T. G. Kolda, and E. Acar, "Temporal link prediction using matrix and tensor factorizations," *ACM TKDD*, vol. 5, no. 2, p. 10, 2011.

[13] J.-T. Sun, H.-J. Zeng, H. Liu, Y. Lu, and Z. Chen, "Cubesvd: a novel approach to personalized web search," in *WWW*, 2005.

[14] A. Smilde, R. Bro, and P. Geladi, *Multi-way analysis: applications in the chemical sciences*. John Wiley & Sons, 2005.

[15] T. Kolda and B. Bader, "Tensor decompositions and applications," *SIAM review*, vol. 51, no. 3, 2009.

[16] K. Shin and U. Kang, "Distributed methods for high-dimensional and large-scale tensor factorization," in *ICDM*, 2014.

[17] G. Tomasi and R. Bro, "Parafac and missing values," *Chemometr. Intell. Lab. Syst.*, vol. 75, no. 2, pp. 163–180, 2005.

[18] "Supplementary document (proofs, implementation details, and additional experiments)." Available at http://www.cs.cmu.edu/~kijungs/codes/cdtf/supple.pdf.

[19] A. Cichocki and P. Anh-Huy, "Fast local algorithms for large scale nonnegative matrix and tensor factorizations," *IEICE Trans. Fundamentals*, vol. 92, no. 3, pp. 708–721, 2009.

[20] E. Acar, T. G. Kolda, and D. M. Dunlavy, "All-at-once optimization for coupled matrix and tensor factorizations," in *MLG*, 2011.

[21] E. Acar, D. M. Dunlavy, T. G. Kolda, and M. Mørup, "Scalable tensor factorizations for incomplete data," *Chemometr. Intell. Lab. Syst.*, vol. 106, no. 1, pp. 41–56, 2011.

[22] I. Jeon, E. E. Papalexakis, U. Kang, and C. Faloutsos, "Haten2: Billion-scale tensor decompositions," in *ICDE*, 2015.

[23] U. Kang, E. Papalexakis, A. Harpale, and C. Faloutsos, "Gigatensor: scaling tensor analysis up by 100 times-algorithms and discoveries," in *KDD*, 2012.

[24] F. Niu, B. Recht, C. Ré, and S. J. Wright, "Hogwild!: A lock-free approach to parallelizing stochastic gradient descent," *NIPS*, 2011.

[25] M. Zinkevich, M. Weimer, A. J. Smola, and L. Li, "Parallelized stochastic gradient descent," in *NIPS*, 2010.

[26] N. D. Sidiropoulos and A. Kyrillidis, "Multi-way compressed sensing for sparse low-rank tensors," *IEEE Signal Processing Letters*, vol. 19, no. 11, pp. 757–760, 2012.

[27] L. Sael, I. Jeon, and U. Kang, "Scalable tensor mining," *Big Data Research*, vol. 2, no. 2, pp. 82–86, 2015.

[28] J. H. Choi and S. Vishwanathan, "Dfacto: Distributed factorization of tensors," in *NIPS*, 2014.

[29] B. Jeon, I. Jeon, L. Sael, and U. Kang, "Scout: Scalable coupled matrix-tensor factorization - algorithm and discoveries," in *ICDE*, pp. 811–822, 2016.

[30] A. H. Phan and A. Cichocki, "Parafac algorithms for large-scale problems," *Neurocomputing*, vol. 74, no. 11, pp. 1970–1984, 2011.

[31] A. L. De Almeida and A. Y. Kibangou, "Distributed large-scale tensor decomposition," in *ICASSP*, 2014.

[32] A. L. de Almeida and A. Y. Kibangou, "Distributed computation of tensor decompositions in collaborative networks," in *CAMSAP*, pp. 232–235, 2013.

[33] J. E. Cohen, R. C. Farias, and P. Comon, "Fast decomposition of large nonnegative tensors," *IEEE Signal Processing Letters*, vol. 22, no. 7, pp. 862–866, 2015.

[34] J. Liu, P. Musialski, P. Wonka, and J. Ye, "Tensor completion for estimating missing values in visual data," *IEEE TPAMI*, vol. 35, no. 1, pp. 208–220, 2013.

[35] N. Vervliet, O. Debals, L. Sorber, and L. De Lathauwer, "Breaking the curse of dimensionality using decompositions of incomplete tensors: Tensor-based scientific computing in big data analysis," *IEEE Signal Processing Magazine*, vol. 31, no. 5, pp. 71–79, 2014.

[36] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[37] C. Teflioudi, F. Makari, and R. Gemulla, "Distributed matrix completion.," in *ICDM*, 2012.

[38] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "Haloop: efficient iterative data processing on large clusters," *PVLDB*, vol. 3, no. 1-2, pp. 285–296, 2010.

[39] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: a runtime for iterative mapreduce," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pp. 810–818, ACM, 2010.

[40] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *USENIX*, 2010.

Kijung Shin is a Ph.D. student in the Computer Science Department of Carnegie Mellon University. He received B.S. in Computer Science and Engineering at Seoul National University. His research interests include graph mining and scalable machine learning.

Lee Sael, aka Sael Lee, holds a joint position as an assistant professor in the Department of Computer Science at SUNY Korea and an assistant research professor in the Department of Computer Science at Stony Brook University. She received her Ph.D. in Computer Science from Purdue University, West Lafayette, IN in 2010, and her B.S. in Computer Science from Korea University, Seoul, Republic of Korea in 2005.

U Kang is an assistant professor in the Department of Computer Science and Engineering of Seoul National University. He received his Ph.D. in Computer Science at Carnegie Mellon University, and his B.S. in Computer Science and Engineering at Seoul National University. He won 2013 SIGKDD Doctoral Dissertation Award, 2013 New Faculty Award from Microsoft Research Asia, and two best paper awards. His research interests include data mining.