Distributed Methods for High-dimensional and Large-scale Tensor Factorization

Kijung Shin
Dept. of Computer Science and Engineering
Seoul National University
Seoul, Republic of Korea
koreaskj@snu.ac.kr

U Kang
Dept. of Computer Science
KAIST
Daejeon, Republic of Korea
ukang@cs.kaist.ac.kr

***Abstract*—Given a high-dimensional and 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 recommendation systems exploiting context information such as time and location. They require tensor factorization methods scalable with both the dimension and size of a tensor. In this paper, we propose two distributed tensor factorization methods, SALS and CDTF. Both methods are scalable with all aspects of data, and they show an interesting trade-off between convergence speed and memory requirements. SALS updates a subset of the columns of a factor matrix at a time, and CDTF, a special case of SALS, updates one column at a time. On our experiment, only our methods factorize a 5-dimensional tensor with 1B observable entries, 10M mode length, and 1K rank, while all other state-of-the-art methods fail. Moreover, our methods require several orders of magnitude less memory than the competitors. We implement our methods on MAPREDUCE with two widely applicable optimization techniques: local disk caching and greedy row assignment.**

***Keywords*-Tensor factorization; Recommender system; Distributed computing; MapReduce**

I. Introduction and Related Work

The recommendation problem 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 so that their multiplication approximates the input matrix, is one of the most widely used methods [2, 7, 14]. To handle web-scale data, efforts have been made to find distributed ways for MF, including ALS [14], DSGD [4], and CCD++ [12].

On the other hand, there have been attempts to improve the accuracy of recommendation by using additional 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 dimensions correspond to the extra factors. As in the matrix case, tensor factorization (TF), which decomposes the input tensor into multiple factor matrices and a core tensor, has been used [5, 9, 13].

As the dimension of web-scale recommendation problems increases, a necessity for TF algorithms scalable with the dimension as well as the size of data has arisen. A promising way to find such algorithms is to extend distributed MF algorithms to higher dimensions. However, the scalability of existing methods including ALS [14], PSGD [8], and FLEXIFACT [1] is limited as summarized in Table I.

Table I: Summary of scalability results. The factors which each method is scalable with are checked. Our proposed SALS and CDTF are the only methods scalable with all the factors.

	CDTF	**SALS**	ALS	PSGD	FLEXIFACT
Dimension	✓	✓	✓	✓	
Observations	✓	✓	✓	✓	✓
Mode Length	✓	✓			✓
Rank	✓	✓			✓
Machines	✓	✓	✓		

In this paper, we propose SALS and CDTF, distributed tensor factorization methods scalable with all aspects of data. SALS updates a subset of the columns of a factor matrix at a time, and CDTF, a special case of SALS, updates one column at a time. Our methods have distinct advantages: SALS converges faster, and CDTF is more memory-efficient. They can also be applied to any application handling large-scale and partially observable tensors, including social network analysis [3] and Web search [11].

The main contributions of our study are as follows:

- **Algorithm.** We propose two tensor factorization algorithms: SALS and CDTF. Their distributed versions are the only methods scalable with all the following factors: the dimension and size of data; the number of parameters; and the number of machines (Table I and Table II).
- **Optimization.** We implement our methods on MAPREDUCE with two novel optimization techniques: local disk caching and greedy row assignment. They speed up not only our methods (up to 98.2×) but also their competitors (up to 5.9×) (Figure 6).
- **Experiment.** We empirically confirm the superior scalability of our methods and their several orders of magnitude less memory requirements than their competitors. Only our methods analyze a 5-dimensional tensor with 1B observable entries, 10M mode length, and 1K rank, while all others fail (Figure 4(a)).

The binary codes of our methods and several datasets are available at http://kdm.kaist.ac.kr/sals. The rest of this paper

Table II: Summary of distributed tensor factorization algorithms. The performance bottlenecks which prevent each algorithm from handling web-scale data are colored red. Only our proposed SALS and CDTF 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]	$O(\|\Omega\|NK(N+K)/M + NIK^3/M)$	$O(NIK)$	$O(NIK)$	Fastest
PSGD [8]	$O(\|\Omega\|NK/M)$	$O(NIK)$	$O(NIK)$	Slow
FLEXIFACT [1]	$O(\|\Omega\|NK/M)$	$O(M^{N-2}NIK)$	$O(NIK/M)$	Fast

Table III: Table of symbols.

Symbol	Definition
$\mathcal{X}$	input tensor ($\in \mathbb{R}^{I_1 \times I_2 \ldots \times I_N}$)
N	dimension of $\mathcal{X}$
I_n	length of the nth mode of $\mathcal{X}$
$\mathbf{A}^{(n)}$	nth factor matrix ($\in \mathbb{R}^{I_n \times K}$)
K	rank of $\mathcal{X}$
Ω	set of indices of observable entries of $\mathcal{X}$
$\Omega^{(n)}_{i_n}$	subset of Ω whose nth mode's index is equal to i_n
${}_mS_n$	set of rows of $\mathbf{A}^{(n)}$ assigned to machine m
$\mathcal{R}$	residual tensor ($\in \mathbb{R}^{I_1 \times I_2 \ldots \times I_N}$)
M	number of machines (reducers on MAPREDUCE)
T_{out}	number of outer iterations
T_{in}	number of inner iterations
λ	regularization parameter
C	number of parameters updated at a time
η_0	initial learning rate

is organized as follows. Section II presents preliminaries for tensor factorization. Section III describes our proposed SALS and CDTF. Section IV presents the optimization techniques for them on MAPREDUCE. After providing experimental results in Section V, we conclude in Section VI.

II. PRELIMINARIES: TENSOR FACTORIZATION

In this section, we describe the preliminaries of tensor factorization and its distributed algorithms.

A. Tensor and the Notations

A tensor is a multi-dimensional array which generalizes a vector (1-dimensional tensor) and a matrix (2-dimensional tensor) to higher dimensions. Like rows and columns in a matrix, an N-dimensional tensor has N modes whose lengths are I_1 through I_N, respectively. We denote tensors with variable dimension N by boldface Euler script letters, e.g., $\mathcal{X}$. 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 $\mathcal{X}$ is denoted by $x_{i_1 \ldots i_N}$. Table III lists the symbols used in this paper.

B. Tensor Factorization

There are several ways to define tensor factorization, and our definition is based on PARAFAC decomposition, which is one of the most popular decomposition methods, and nonzero squared loss with L2 regularization, whose weighted form has been successfully used in many recommendation systems [2, 7, 14]. Details about PARAFAC decomposition can be found in [6].

Definition 1 (Tensor Factorization): Given an N-dimensional tensor $\mathcal{X}(\in \mathbb{R}^{I_1 \times I_2 \ldots \times I_N})$ with observable entries $\{x_{i_1 \ldots i_N} | (i_1, ..., i_N) \in \Omega\}$, the rank K factorization of $\mathcal{X}$ is to find factor matrices $\{\mathbf{A}^{(n)} \in \mathbb{R}^{I_n \times K} | 1 \le n \le N\}$ which minimize the following loss function:

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

Note that the loss function depends only on the observable entries. Each factor matrix $\mathbf{A}^{(n)}$ corresponds to the latent feature vectors of nth mode instances, and $\sum_{k=1}^{K} \prod_{n=1}^{N} a^{(n)}_{i_n k}$ corresponds to the interaction between the features.

C. Distributed Methods for Tensor Factorization

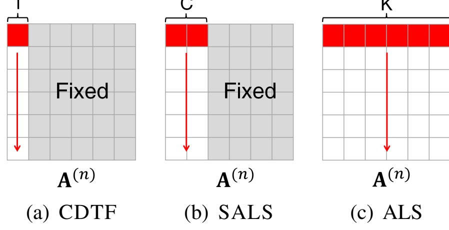
Table II summarizes the performances of several tensor factorization algorithms suitable for distributed environments. Detailed explanation, including update rule and complexity analysis of each method, can be found in [10].

III. PROPOSED METHODS

In this section, we propose subset alternating least square (SALS) and coordinate descent for tensor factorization (CDTF). They are scalable algorithms for tensor factorization, which is essentially an optimization problem whose loss function is (1) and parameters are the entries of factor matrices, $\mathbf{A}^{(1)}$ through $\mathbf{A}^{(N)}$. Figure 1 depicts the difference among CDTF, SALS, and ALS. Unlike ALS, which updates each K columns of factor matrices row by row, SALS updates each C $(1 \le C \le K)$ columns row by row, and CDTF updates each column entry by entry. CDTF can be seen as an extension of CCD++ [12] to higher dimensions. Since SALS contains CDTF ($C = 1$) as well as ALS ($T_{in} = 1, C = K$) as a special case, we focus on SALS and additionally explain optimization schemes for CDTF.

A. Update Rule and Update Sequence

Algorithm 1 describes the procedure of SALS. $\mathcal{R}$ denotes the residual tensor where $r_{i_1 \ldots i_N} = x_{i_1 \ldots i_N} - \sum_{k=1}^{K} \prod_{n=1}^{N} a^{(n)}_{i_n k}$. We initialize the entries of $\mathbf{A}^{(1)}$ to zeros and those of all other factor matrices to random values so that the initial value of $\mathcal{R}$ is equal to $\mathcal{X}$ (line 1). In



(a) CDTF (b) SALS (c) ALS

Figure 1: 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 each K columns row by row.

Algorithm 1: Serial version of SALS

```
Input  :  𝔛, K, λ
Output:  A^(n) for all n
1  initialize ℜ and A^(n) for all n
2  for outer iter = 1..T_out do
3      for split iter = 1..⌈K/C⌉ do
4          choose k_1, ..., k_C (from columns not updated yet)
5          compute ℜ̂
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 (2)
10         update ℜ using (3)
```

every iteration (line 2), SALS repeats choosing C columns, k_1 through k_C, randomly without replacement (line 4) and updating them while keeping the other columns fixed, which is equivalent to the rank C factorization of $\hat{\mathfrak{R}}$ where $\hat{r}_{i_1 \ldots i_N} = r_{i_1 \ldots i_N} + \sum_{c=1}^{C} \prod_{n=1}^{N} a_{i_n k_c}^{(n)}$. Once $\hat{\mathfrak{R}}$ is computed (line 5), updating C columns of factor matrices matrix by matrix (line 7) is repeated T_{in} times (line 6). For each factor matrix, since its rows are independent of each other in minimizing (1) when the other factor matrices are fixed, the entries are updated row by row (line 8) as follows:

$$[a_{i_n k_1}^{(n)}, \ldots, a_{i_n k_C}^{(n)}]^T \leftarrow \underset{[a_{i_n k_1}^{(n)}, \ldots, a_{i_n k_C}^{(n)}]^T}{\arg\min} L(\mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(N)})$$
$$= (\mathbf{B}_{i_n}^{(n)} + \lambda \mathbf{I}_C)^{-1} \mathbf{c}_{i_n}^{(n)}, \quad (2)$$

where the (c_1, c_2)th entry of $\mathbf{B}_{i_n}^{(n)} (\in \mathbb{R}^{C \times C})$ is

$$\sum_{(i_1, \ldots, i_N) \in \Omega_{i_n}^{(n)}} \left(\prod_{l \neq n} a_{i_l k_{c_1}}^{(l)} \prod_{l \neq n} a_{i_l k_{c_2}}^{(l)} \right),$$

the cth entry of $\mathbf{c}_{i_n}^{(n)} (\in \mathbb{R}^{C})$ is

$$\sum_{(i_1, \ldots, i_N) \in \Omega_{i_n}^{(n)}} \left(\hat{r}_{i_1 \ldots i_N} \prod_{l \neq n} a_{i_l k_c}^{(l)} \right),$$

and $\mathbf{I}_C$ is a C by C identity matrix. $\Omega_{i_n}^{(n)}$ denotes the subset of Ω whose nth mode's index is equal to i_n. The proof of this update rule can be found in [10]. Since $\mathbf{B}_{i_n}^{(n)}$ is symmetric, Cholesky decomposition can be used for its inversion. After this rank C factorization, the entries of $\mathfrak{R}$ are updated by the following rule (line 10):

$$r_{i_1 \ldots i_N} \leftarrow \hat{r}_{i_1 \ldots i_N} - \sum_{c=1}^{C} \prod_{n=1}^{N} a_{i_n k_c}^{(n)}. \quad (3)$$

In CDTF, instead of computing $\hat{\mathfrak{R}}$ before rank one factorization, containing the computation in (2) and (3) results in better performance on a disk-based system like MapReduce by significantly reducing disk I/O operations. Moreover, updating columns in a fixed order instead of choosing them randomly converges faster in CDTF in our experiments.

B. Complexity Analysis

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

Proof: Computing $\hat{\mathfrak{R}}$ (line 5) and updating $\mathfrak{R}$ (line 10) take $O(|\Omega|NC)$. Updating C parameters using (2) (line 9) takes $O(|\Omega_{i_n}^{(n)}|C(C+N) + C^3)$, which consists of $O(|\Omega_{i_n}^{(n)}|NC)$ to calculate $\prod_{l \neq n} a_{i_l k_1}^{(l)}$ through $\prod_{l \neq n} a_{i_l k_C}^{(l)}$ 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 invert $\mathbf{B}_{i_n}^{(n)}$. See the full proof in [10]. ■

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

Proof: All the computation including $\hat{\mathfrak{R}}$ computation (line 5), rank C factorization (lines 7 through 9), and $\mathfrak{R}$ update (line 10) can be performed by loading only $(k_1, \ldots, k_C)$ columns of factor matrices into memory and streaming the entries of $\hat{\mathfrak{R}}$ and $\mathfrak{R}$ from disk. Thus, SALS only needs to load C columns of factor matrices, the number of whose entries is $O(C\sum_{n=1}^{N} I_n)$, into memory by turns depending on $(k_1, \ldots, k_C)$ values. See the full proof in [10]. ■

C. Parallelization in Distributed Environments

In this section, we describe how to parallelize SALS in distributed environments such as MapReduce where machines do not share memory. Algorithm 2 depicts the distributed version of SALS.

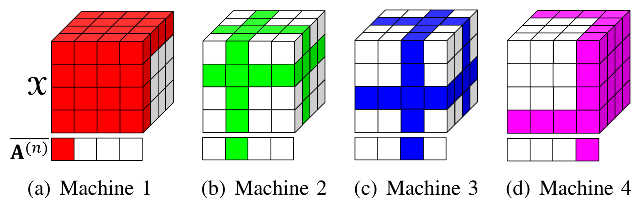
Since update rule (2) for each row (C parameters) of a factor matrix does not depend on the other rows in the matrix, rows in a factor matrix can be distributed across machines and updated simultaneously without affecting the correctness of SALS. Each machine m updates ${}_mS_n$ rows of $\mathbf{A}^{(n)}$ (line 10), and for this, the ${}_m\Omega = \bigcup_{n=1}^{N} \left(\bigcup_{i_n \in {}_mS_n} \Omega_{i_n}^{(n)} \right)$ entries of $\mathfrak{X}$ are distributed to machine m in the first stage of the algorithm (line 1). Figure 2 shows an example of work and data distribution in SALS.

After the update, parameters updated by each machine are broadcasted to all other machines (line 11). Each machine m broadcasts $C|{}_mS_n|$ parameters and receives $C(I_n - |{}_mS_n|)$ parameters from the other machines after each update. The total number of parameters each machine exchanges with the other machines is $KT_{in}\sum_{n=1}^{N} I_n$ per outer iteration.

Algorithm 2: Distributed version of SALS

Input : $\mathcal{X}$, K, λ, ${}_mS_n$ 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 ${}_m\Omega$ 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 (2)
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 (3)



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

Figure 2: Work and data distribution of SALS in distributed environments with a 3-dimensional tensor and 4 machines. 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.

The running time of parallel steps in Algorithm 2 depends on the longest running time among all machines. Specifically, the running time of lines 7, 10, and 12 is proportional to $\max_m |{}_m\Omega^{(n)}|$ where ${}_m\Omega^{(n)} = \bigcup_{i_n \in {}_mS_n} \Omega^{(n)}_{i_n}$, and that of line 11 is proportional to $\max_m |{}_mS_n|$. Therefore, it is necessary to assign the rows of the factor matrices to the machines (i.e., to decide ${}_mS_n$) so that $|{}_m\Omega^{(n)}|$ and $|{}_mS_n|$ are even among all the machines. The greedy assignment algorithm described in Algorithm 3 aims to minimize $\max_m |{}_m\Omega^{(n)}|$ under the condition that $\max_m |{}_mS_n|$ is minimized (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 which satisfies $|{}_mS_n| < \lceil I_n/M \rceil$ and has the smallest $|{}_m\Omega^{(n)}|$ currently. The effects of this greedy row assignment on actual running times are described in Section V-E.

IV. Optimization on MapReduce

In this section, we describe two optimization techniques used to implement SALS and CDTF on MapReduce, which is one of the most widely used distributed platforms.

Algorithm 3: Greedy row assignment in SALS

Input : $\mathcal{X}$, 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 the one with smaller $|{}_m\Omega|$)
9.   add i_n to ${}_mS_n$
10.   add $|\Omega^{(n)}_{i_n}|$ to $|{}_m\Omega^{(n)}|$ and $|{}_m\Omega|$

A. *Local Disk Caching*

Typical MapReduce implementations (e.g., one described in [10]) repeat distributing data from a distributed file system to machines at every iteration. This repetition is inefficient for SALS and CDTF due to their highly iterative nature. SALS and CDTF require $\mathcal{R}$ (or $\hat{\mathcal{R}}$) to be distributed $T_{out}K(T_{in}N + 2)/C$ times and $T_{out}K(T_{in}N + 1)$ times, respectively. Our implementation reduces this inefficiency by caching data to local disk once they are distributed. Our implementation streams the cached data, ${}_m\Omega^{(n)}$ entries of $\mathcal{R}$ for example, from the local disk instead of distributing entire $\mathcal{R}$ from the distributed file system when updating the columns of $\mathbf{A}^{(n)}$. The effect of this local disk caching on actual running time is described in Section V-E.

B. *Direct Communication*

In MapReduce, it is generally assumed that reducers run independently and do not communicate directly with each other. However, we adapt the direct communication method using the distributed file system in [1] to broadcast parameters among reducers efficiently. Our method is described in detail in [10].

V. Experiments

A. *Experimental Settings*

We run experiments on a 40-node Hadoop cluster. Each node has an Intel Xeon E5620 2.4GHz CPU. The maximum heap memory size per reducer is set to 8GB. The real-world tensors and the synthetic tensors used in our experiments are summarized in Table IV and Table V. Most of them are available at http://kdm.kaist.ac.kr/sals and explained in detail in [10]. All methods in Table II are implemented using Java with Hadoop 1.0.3. Local disk caching, direct communication, and greedy row assignment are applied to all the methods if possible. All our implementations use weighted-λ-regularization [14]. For SALS and CDTF, T_{in} is set to 1, and C is set to 10, unless otherwise stated. The learning rate of FlexiFaCT and PSGD at tth iteration is set to $2\eta_0/(1+t)$, which follows the open-sourced FlexiFaCT

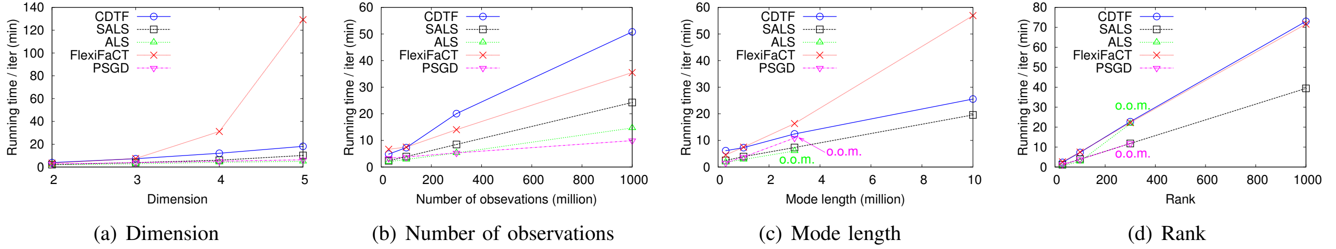


Figure 3: Scalability on each aspect of data. o.o.m. : out of memory. Only SALS and CDTF scale with all the aspects.

Table IV: Summary of real-world datasets.

	Movielens$_4$	**Netflix**$_3$	**Yahoo-music**$_4$
N	4	3	4
I_1	715,670	2,649,429	1,000,990
I_2	65,133	17,770	624,961
I_3	169	74	133
I_4	24	-	24
$\lvert\Omega\rvert$	93,012,740	99,072,112	252,800,275
K	20	40	80
λ	0.01	0.02	1.0
η_0	0.01	0.01	10^{-5} (FLEXIFACT) 10^{-4} (PSGD)

Table V: 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
$\lvert\Omega\rvert$	30M	100M	300M	1B
K	30	100	300	1K

implementation (http://alexbeutel.com/l/flexifact/). The number of reducers is set to 5 for FLEXIFACT, 20 for PSGD, and 40 for the others, each of which leads to the best performance on the machine scalability test in Section V-C, unless otherwise stated.

B. Data Scalability

1) Scalability on Each Factor (Figure 3): We measure the scalability of CDTF, SALS, and the competitors with regard to the dimension, number of observations, mode length, and rank of an input tensor. When measuring the scalability with regard to a factor, the factor is scaled up from S1 to S4 while all other factors are fixed at S2 as summarized in Table V. As seen in Figure 3(a), FLEXIFACT does not scale with dimension because of its communication cost, which increases exponentially with dimension. ALS and PSGD are not scalable with mode length and rank due to their high memory requirements as Figures 3(c) and 3(d) show. They require up to 11.2GB, which is 48× of 234MB that CDTF requires and 10× of 1,147MB that SALS requires. Moreover, the running time of ALS increases rapidly with rank owing to its cubically increasing computational cost. Only SALS and CDTF are scalable with all the factors as summarized in Table I. Their running times increase linearly with all the factors except dimension, with which they increase slightly faster due to the quadratically increasing computational cost.

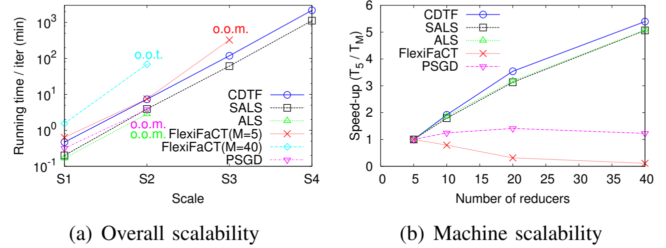


Figure 4: (a) Overall scalability. o.o.m. : out of memory, o.o.t. : out of time (takes more than a week). Only SALS and CDTF scale up to the largest scale S4. (b) Machine scalability. Computations of SALS and CDTF are efficiently distributed across machines.

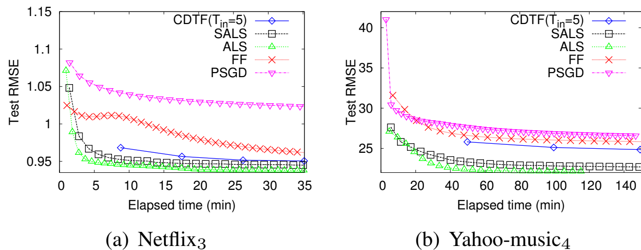
2) Overall Scalability (Figure 4(a)): We measure the scalability of the methods by scaling up all the factors together from S1 to S4. The scalability of ALS and PSGD, and FLEXIFACT with 5 machines is limited owing to their high memory requirements. ALS and PSGD require almost 186GB to handle S4, which is 493× of 387MB that CDTF requires and 100× of 1,912MB that SALS requires. FLEXIFACT with 40 machines does not scale over S2 due to its rapidly increasing communication cost. Only SALS and CDTF scale up to S4, and there is a trade-off between them: SALS runs faster, and CDTF is more memory-efficient.

C. Machine Scalability (Figure 4(b))

We measure the speed-ups (T_5/T_M where T_M is the running time with M reducers) of the methods on the S2 scale dataset by increasing the number of reducers. The speed-ups of CDTF, SALS, and ALS increase linearly at the beginning and then flatten out slowly owing to their fixed communication cost which does not depend on the number of reducers. The speed-up of PSGD flattens out fast, and PSGD even slightly slows down in 40 reducers because of increased overhead. FLEXIFACT slows down as the number of reducers increases because of its rapidly increasing communication cost.

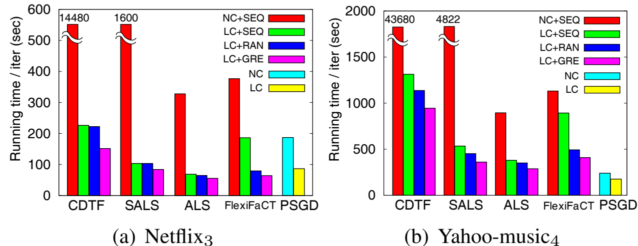
D. Convergence (Figure 5)

We compare how quickly and accurately each method factorizes real-world tensors. Accuracies are calculated at each iteration by root mean square error (RMSE) on a held-out test set, which is a measure commonly used by recommendation systems. Table IV describes K, λ, and η_0 values used for each dataset. They are determined by cross validation. Owing to the non-convexity of (1), each



(a) Netflix_3 (b) Yahoo-music_4

Figure 5: Convergence speed on real-world datasets. SALS is comparable with ALS, which converges fastest to the best solution, and CDTF follows them.



(a) Netflix_3 (b) Yahoo-music_4

Figure 6: Effects of optimization techniques on running times. NC: no caching, LC: local disk caching, SEQ: sequential row assignment[1], RAN: random row assignment, GRE: greedy row assignment. Our proposed optimization techniques (LC+GRE) significantly accelerate CDTF, SALS, and also their competitors.

algorithm may converge to local minima with different accuracies. In all datasets (results on the Movielens_4 dataset are omitted for space reasons), SALS is comparable with ALS, which converges the fastest to the best solution, and CDTF follows them. PSGD converges the slowest to the worst solution due to the non-identifiability of (1) [4]. Extra experiments regarding the effect of C and T_{in} values on the convergence of SALS and CDTF are described in [10].

E. Optimization (Figure 6)

We measure how our proposed optimization techniques, local disk caching and greedy row assignment, affect the running time of CDTF, SALS, and the competitors on real-world datasets. The direct communication method explained in Section IV-B is applied to all the implementations if necessary. Local disk caching speeds up CDTF up to $65.7\times$, SALS up to $15.5\times$, and the competitors up to $4.8\times$. The speed-ups of SALS and CDTF are the most significant because of the highly iterative nature of SALS and CDTF. Additionally, greedy row assignment speeds up CDTF up to $1.5\times$; SALS up to $1.3\times$; and the competitors up to $1.2\times$ compared with the second best one. It is not applicable to PSGD, which does not distribute parameters row by row.

VI. Conclusion

In this paper, we propose SALS and CDTF, distributed algorithms for high-dimensional and large-scale tensor factorization. They are scalable with all aspects of data (dimension, the number of observable entries, mode length, and rank) and show a trade-off: SALS has an advantage in terms of convergence speed, and CDTF has one in terms of memory usage. Local disk caching and greedy row assignment, two proposed optimization schemes, significantly accelerate not only SALS and CDTF but also their competitors.

[1] ${}_mS_n = \{i_n \in \mathbb{N} | \frac{I_n \times (m-1)}{M} < i_n \leq \frac{I_n \times m}{M}\}$

Acknowledgments

This work was supported by AFOSR/AOARD under the Grant No. FA2386-14-1-4036, and by the National Research Foundation of Korea (NRF) Grant funded by the Korean Government (MSIP) (No. 2013R1A1A1064409)

References

[1] 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.

[2] 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.

[3] D. M. Dunlavy, T. G. Kolda, and E. Acar. Temporal link prediction using matrix and tensor factorizations. *ACM TKDD*, 5(2):10, 2011.

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

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

[6] T. Kolda and B. Bader. Tensor decompositions and applications. *SIAM review*, 51(3), 2009.

[7] Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, 2009.

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

[9] A. Nanopoulos, D. Rafailidis, P. Symeonidis, and Y. Manolopoulos. Musicbox: Personalized music recommendation based on cubic analysis of social tags. *IEEE TASLP*, 18(2):407–412, 2010.

[10] K. Shin and U. Kang. Distributed methods for high-dimensional and large-scale tensor factorization. *arXiv preprint arXiv:1410.5209*, 2014.

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

[12] C.-J. H. S. S. Yu, Hsiang-Fu and I. S. Dhillon. Parallel matrix factorization for recommender systems. *Knowl. Inf. Syst.*, pages 1–27, 2013.

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

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