LFTF: A Framework for Efficient Tensor Analytics at Scale

Fan Yang Fanhua Shang Yuzhen Huang
James Cheng Jinfeng Li Yunjian Zhao Ruihao Zhao
Department of Computer Science and Engineering
The Chinese University of Hong Kong
{fyang,fhshang,yzhuang,jcheng,jfli,yjzhao,rhzhao}@cse.cuhk.edu.hk

ABSTRACT

Tensors are higher order generalizations of matrices to model multi-aspect data, e.g., a set of purchase records with the schema (user_id, product_id, timestamp, feedback). Tensor factorization is a powerful technique for generating a model from a tensor, just like matrix factorization generates a model from a matrix, but with higher accuracy and richer information as more attributes are available in a higher-order tensor than a matrix. The data model obtained by tensor factorization can be used for classification, recommendation, anomaly detection, and so on. Though having a broad range of applications, tensor factorization has not been popularly applied compared with matrix factorization that has been widely used in recommender systems, mainly due to the high computational cost and poor scalability of existing tensor factorization methods. Efficient and scalable tensor factorization is particularly challenging because real world tensor data are mostly sparse and massive. In this paper, we propose a novel distributed algorithm, called Lock-Free Tensor Factorization (LFTF), which significantly improves the efficiency and scalability of distributed tensor factorization by exploiting asynchronous execution in a re-formulated problem. Our experiments show that LFTF achieves much higher CPU and network throughput than existing methods, converges at least 17 times faster and scales to much larger datasets.

1. INTRODUCTION

Matrix factorization, also called *matrix decomposition*, is a traditional technique for data analysis. One of the representative applications of matrix factorization is recommender systems. For example, matrix factorization has been shown to be the superior *collaborative filtering* approach than neighborhood methods according to the Netflix Prize competition [5]. However, matrix factorization only models the relation between two aspects (or *columns*) of the data (typically between *users* and *items*), while there are other columns of the data that may provide richer information.

Proceedings of the VLDB Endowment, Vol. 10, No. 7

Tensor factorization, also called *tensor decomposition*, generalizes the problem of matrix factorization to factorize a K-th order tensor instead of a matrix.

As more information is taken into account, tensor factorization is able to picture the relations in a more accurate way than matrix factorization [16]. The richer information possessed by a tensor also leads to a broader range of applications than matrix factorization. Below we list some common use cases of tensor factorization.

Application I: Recommender Systems. Tensor factorization is a typical technique for *context-aware collaborative filtering* [7, 16], as an improvement to the factorization of a matrix that does not integrate other aspects (i.e., the context) into the model. For example, a product retailer may want to model users' ratings on products over a third aspect, e.g., *time*. He may select three columns (user_id, product_id, timestamp) from a database table, and perform a third order tensor factorization. If the location of the purchase is also of interest, he may choose to construct a fourth order tensor (user_id, product_id, location, timestamp) [7]. The resulting latent model obtained by tensor factorization can be used for product recommendation, customer classification, product categorization, etc., in a way similar to the matrix-based collaborative filtering but in a much more accurate and contextually informative way.

Application II: Anomaly Detection. A network security analyst may want to extract (source-ip, target-ip, port, timestamp) information from massive network intrusion logs, and construct a model using tensor factorization [15, 30]. The resulting model is useful in understanding network usage patterns and in identifying network anomalies.

Application III: Social Network Analysis. Tensor factorization is very useful in studying social networks. For example, factorizing a (user_id, user_id, timestamp) tensor can be used to analyze the change of community structures over time, where the first two attributes form a who-friends-whom network. A social network can also be interpreted using a tensor (user_id, user_id, relation_type), in which the type of relations can be employer-employee, friends, family and so on. Tensor factorization in this case facilitates relation mining in social networks [25].

Tensor factorization can also be employed to attack different data mining problems, e.g., Latent Dirichlet Allocation (a commonly used topic model), with greatly improved efficiency [2,3]. Real-world tensors can be enormous in size and often very sparse. For example, the Amazon Movie reviews corpora (used in our experiments) can be mapped to a third

order tensor with size $(889,167 \times 253,059 \times 1,762,532)$ and having 720 million non-zeros entries. Therefore, it is important to develop efficient and scalable tools for processing the massive sparse tensors of today and of the future.

Existing Methods. Existing tools such as the Matlab Tensor Toolbox and the Matlab N-way Toolbox solve tensor factorization in Matlab with reasonable efficiency. However, they operate strictly on data that can fit in main memory of a single machine, and are thus not scalable solutions. Distributed solutions have been proposed [6, 10, 14, 15, 29] in order to perform tensor factorization over large-scale data, but existing distributed solutions are not efficient and are thus not practical, especially for interactive data analytics which are widely preferred by data scientists today. The low efficiency and poor scalability of existing tensor factorization methods have severely restricted the use of tensor factorization in real-world applications.

Technical Challenges. While modeling data in a more informative way using tensor is attractive, it is challenging to design an efficient and scalable method for solving tensor factorization, especially in terms of distributed execution. Since tensor factorization involves more aspects of data than matrix factorization, it is inherently a more difficult problem. Two commonly used methods are *Alternating Least Squares* (*ALS*) and *Gradient Descent* (*GD*). However, they introduce the intermediate data explosion problem because of the unnecessary materialization of Khatri-Rao products [10, 15]. *Stochastic Gradient Descent* (*SGD*) has been used as an alternative to avoid this drawback. However, due to the use of BSP (Bulk Synchronous Parallel) style distributed computation, SGD-based tensor factorization suffers from another problem, i.e., frequent global synchronization barriers to avoid conflicting SGD updates, resulting in poor scalability.

Proposed Solution. We found that by carefully reformulating the original tensor factorization problem, we obtain a new form that allows us to plug in distributed asynchronous execution, which has the potential to completely remove the synchronization overhead of BSP execution. In this way, we obtain *a new solution that enjoys the benefits of the state-of-the-art ALS-based and SGD-based solutions (i.e., massive parallelism/light computation), yet without their performance limitations (i.e., time-consuming network transmission)*. We name the new solution as *LFTF* (**L**ock **F**ree **T**ensor **F**actorization).

The LFTF algorithm decomposes the original K-th order tensor factorization problem into $(K-1)$ matrix factorization subproblems, and solves them in stages. Each subproblem is solved in one stage. Inside a stage, we design an asynchronous algorithm (called `AsyncPSGD`) to carry out *distributed SGD* in order to solve the corresponding matrix factorization subproblem. Across the stages, LFTF alternates the updates among different modes of the tensor. The design of the asynchronous algorithm unleashes the full power of distributed SGD. Our algorithm only requires light synchronization (with negligible overhead) between consecutive stages, where the algorithm switches to update different modes of the tensor. The asynchronous SGD inside a stage ensures that updates are performed efficiently. Processing in stages and synchronization between consecutive stages guarantee the eventual convergence of the model.

Thanks to the lock-free design, LFTF suits well with the hybrid characteristics of modern computing clusters, where computing nodes are connected in a shared nothing architecture, while inside each computing node multiple CPU cores share the same main memory. In Section 5, we will show that in contrast to existing solutions [6, 10, 14, 15, 29], the design of LFTF is able to exploit much more benefits from such a computing architecture to achieve much higher performance, while producing high-quality factorizations comparable with existing methods [6, 10, 14, 15, 29].

Contributions. In summary, our main contributions are as follows.

1. **Efficiency:** The high performance provided by the LFTF framework makes tensor analytics fast and practical. Extensive experiments show that our new approach is 17 times to orders of magnitude faster than existing solutions [6, 10, 14, 15, 29].

2. **Generality and Usability:** Ensured by high performance, the proposed framework is able to carry out large-scale tensor analytics at interactive speeds. The framework is also handy for model parameter tuning (e.g., regularization terms).

3. **Scalability:** The LFTF algorithm is highly scalable. Our experiments show that LFTF is able to linearly scale to 10 billion rows of data, which is significantly larger than the size of data used in prior works [6, 10, 14, 15, 29].

Paper Organization. We first introduce some background in Section 2. In Sections 3 and 4, we present the LFTF algorithm and its theoretical analysis. We evaluate LFTF in Section 5, followed by related work in Section 6 and conclusions in Section 7.

2. BACKGROUND

In this section, we introduce the basic notations and background related to tensor factorization. We also introduce some standard techniques and highlight the challenges of applying them for factorizing large-scale tensors. We refer readers to [17] for a detailed review of tensor factorization.

Table 1 lists some commonly used notations. Scalars are denoted by lower-case letters such as x, y, z, and vectors by bold lower-case letters such as $\mathbf{x}$, $\mathbf{y}$, $\mathbf{z}$. Matrices are denoted by upper-case letters, e.g., X, and their entries by lower-case letters, e.g., x_{ij}, which is the entry corresponds to the i-th row and j-th column of X. A K-th order tensor is denoted by a calligraphic letter, e.g., $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_K}$, where I_i is the size of the i-th mode of the tensor. An entry in a K-th order tensor $\mathcal{X}$ is denoted by $x_{i_1,\ldots,i_K}$.

2.1 Tensor Rank and CP Factorization

With an exact analogue to the definition of the matrix rank, the rank of a tensor $\mathcal{X}$, denoted by $\text{rank}(\mathcal{X})$, is defined as follows.

DEFINITION 1. *The rank of a tensor is the smallest number of rank-one tensors that generate the tensor as their sum, i.e., the smallest R such that*

$$\mathcal{X} \approx \sum_{i=1}^{R} \mathbf{a}_i^1 \circ \mathbf{a}_i^2 \circ \cdots \circ \mathbf{a}_i^K = [\![A_1, A_2, \ldots, A_K]\!], \quad (1)$$

Table 1: Commonly used notations

Symbol	Description
$\mathcal{X}$, X, $\mathbf{x}$, x	Tensor, matrix, vector, scalar
$\mathbf{a}_i, \mathbf{b}_j, \mathbf{c}_k$	The i-th, j-th or k-th row of A, B, C
$\mathcal{X}_{(k)}$	Mode-k unfolding of tensor $\mathcal{X}$
$\mathcal{X} \times_k A$	Mode-k product
$\mathcal{C}, A_k, k=1,\ldots,K$	Core tensor and factor matrices
$[\![A_1, \ldots, A_K]\!]$	Full outer product, $A_1 \circ \ldots \circ A_K$
$[\![\mathcal{C}; A_1, \ldots, A_K]\!]$	Full multilinear product, $\mathcal{C} \times_1 A_1 \cdots \times_K A_K$
$\otimes$, $\odot$, $\circ$	Kronecker, Hadamard, and outer products
Ω, $\lvert\Omega\rvert$	Index set of observed entries, cardinality of Ω
$\lVert\cdot\rVert_F$, $\lVert\cdot\rVert_2$	Frobenius norm, spectral norm

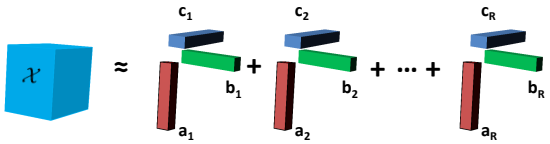


Figure 1: Illustration of an R-component CP model for a third order tensor

where $\circ$ *denotes the outer product of vectors, i.e.,* $(\mathbf{a}_i^1 \circ \mathbf{a}_i^2 \circ \cdots \circ \mathbf{a}_i^K)_{i_1,i_2\cdots,i_K} = [\mathbf{a}_i^1]_{i_1}[\mathbf{a}_i^2]_{i_2}\cdots[\mathbf{a}_i^K]_{i_K}$, *and* $A_i = [\mathbf{a}_1^i, \mathbf{a}_2^i \ \ldots \mathbf{a}_R^i]$.

The tensor factorization form in Equation (1) is called the CANDECOMP/PARAFAC (CP) factorization for K-th order tensors. For example, the CP factorization for third order tensors is expressed as

$$\mathcal{X} \approx [\![A, B, C]\!] = \sum_{i=1}^{R} \mathbf{a}_i \circ \mathbf{b}_i \circ \mathbf{c}_i, \tag{2}$$

where A, B and C are referred to as the *factor matrix* which is the combination of the vectors from the rank-one components, e.g., $A = [\mathbf{a}_1\ \mathbf{a}_2 \ldots \mathbf{a}_R] \in \mathbb{R}^{I_1 \times R}$, as shown in Figure 1. In the following discussion, we will also use the term *latent vector* to refer to a row of a factor matrix.

In this sense, the approximation generally requires significantly less storage (i.e., $O(R(I_1 + I_2 + \cdots + I_K))$ than the original tensor (i.e., $O(I_1 \times I_2 \times \cdots \times I_K)$). Thus, in this paper, we focus on the CP factorization for large-scale sparse tensor completion and decomposition problems as follows:

$$\min_{A,B,C} \frac{1}{2} \sum_{(i,j,k)\in\Omega} (\mathcal{X}_{ijk} - \sum_{r=1}^{R} a_{ir} b_{jr} c_{kr})^2, \tag{3}$$

where $\mathcal{X}$ is the partially observed tensor (usually very sparse) and Ω denotes the set of indices observed.

2.2 Alternating Least Squares

The principled way to decompose large-scale sparse tensors is the CP factorization, as stated in Equation (3). In [1], Acar *et al.* presented a weighted batch least squares algorithm for sparse tensor factorization. However, this algorithm suffers from high computational complexity and is not applicable for large-scale datasets.

To factorize a large tensor, the *Alternating Least Squares* (*ALS*) strategy is used (e.g., in SALS [29], DFacTo [10] and HaTen2 [14]) to update alternately various factor matrices, e.g., A, B and C, as follows:

$$\begin{aligned} A^+ &\longleftarrow \mathcal{X}_{(1)}(B \odot C)(B^T B C^T C)^\dagger, \\ B^+ &\longleftarrow \mathcal{X}_{(2)}(C \odot A)(C^T C A^T A)^\dagger, \\ C^+ &\longleftarrow \mathcal{X}_{(3)}(A \odot B)(A^T A B^T B)^\dagger, \end{aligned} \tag{4}$$

where $A^\dagger$ denotes the Moore-Penrose pseudo-inverse of A. Note that both DFacTo and HaTen2 utilize all entries of tensor data to address the tensor factorization problem and thus have poor performance as illustrated in Section 5, while SALS (and also our method) considers only the observed entries.

The ALS strategy is also used in the weighted batch Tucker decomposition method, which also suffers from high computational complexity and is not scalable. We present its model as follows, which is used in our analysis in Section 4:

$$\min_{\mathcal{C},A_k} \lVert \mathcal{W} * (\mathcal{X} - \mathcal{C} \times_1 A_1 \times_2 \cdots \times_K A_K) \rVert_F^2, \tag{5}$$

where $A_k \in \mathbb{R}^{I_n \times R_k}$, $\mathcal{C} \in \mathbb{R}^{R_1 \times \cdots \times R_K}$ is a core tensor with the given Tucker rank $(R_1, \ldots, R_K)$, and $\times_k$ denotes the mode-k product, $*$ denotes the Hadamard (elementwise) product, and $\mathcal{W}$ is a nonnegative weight tensor with the same size as $\mathcal{X}$,

$$w_{i_1,i_2\cdots,i_K} = \begin{cases} 1 & \text{if } x_{i_1,i_2\cdots,i_K} \text{ is known,} \\ 0 & \text{otherwise.} \end{cases}$$

Another alternative is the stochastic gradient descent strategy [6], which we discuss in details in the following subsection.

2.3 Stochastic Gradient Descent

Stochastic Gradient Descent (*SGD*) is a core building block in solving matrix/tensor factorization used by many algorithms [6, 11, 34, 35]. Consider a third order tensor. According to Equation (3), using the Frobenius Norm Regularization as an example, we can formulate the model as follows:

$$\min_{A,B,C} \sum_{(i,j,k)\in\Omega} [(x_{ijk} - \sum_{r=1}^{K} a_{ir} b_{jr} c_{kr})^2 + \lambda(\lVert\mathbf{a}_i\rVert^2 + \lVert\mathbf{b}_j\rVert^2 + \lVert\mathbf{c}_k\rVert^2)], \tag{6}$$

where $\lambda > 0$ is a regularization parameter. The update steps are given as follows:

$$\begin{aligned} \mathbf{a}_i &\leftarrow \mathbf{a}_i - \eta_t[(\mathbf{a}_i(\mathbf{b}_j. * \mathbf{c}_k)^T - x_{ijk})(\mathbf{b}_j. * \mathbf{c}_k) + \lambda \mathbf{a}_i], \\ \mathbf{b}_j &\leftarrow \mathbf{b}_j - \eta_t[(\mathbf{a}_i(\mathbf{b}_j. * \mathbf{c}_k)^T - x_{ijk})(\mathbf{a}_i. * \mathbf{c}_k) + \lambda \mathbf{b}_j], \\ \mathbf{c}_k &\leftarrow \mathbf{c}_k - \eta_t[(\mathbf{a}_i(\mathbf{b}_j. * \mathbf{c}_k)^T - x_{ijk})(\mathbf{a}_i. * \mathbf{b}_j) + \lambda \mathbf{c}_k], \end{aligned} \tag{7}$$

where η_t denotes the learning rate at time t (readers may refer to [10, 34] for some strategies of setting η_t), and $.*$ represents the element-wise product.

Challenges of Applying SGD. As ALS and Gradient Descent methods suffer from intermediate data explosion problem [10, 15], SGD has been considered as a better alternative. However, parallelizing SGD is tricky primarily due to the *conflicting update* problem, i.e., threads across the cluster may need to update the same sets of parameters, but these updates are conflicting and the model may converge slowly or even diverge. As a result, parallelization of SGD needs to involve fine-grained or coarse-grained locking across the network in order to avoid such conflicting updates. The locking and waiting severely affect the efficiency and degrade the overall performance.

In addition to the conflicting update problem, the layout of the factor matrices also needs to be carefully designed, since each machine in the cluster may need to update potentially all the entries of the factor matrices based on its partition of the training data. Thus, a bad layout of the factor matrices can lead to very high communication overheads

due to updating the entries of the factor matrices through the network.

One straightforward solution is to implement SGD upon the Parameter Server framework [20, 32], which is a general framework for distributed machine learning. Parameter Server keeps the factor matrices in a distributed key-value store. However, before each update, a worker needs to fetch the latent vectors from the key-value store; and after the update, it has to commit the update through the network to the key-value store, resulting in significant overhead. The state-of-the-art SGD-based solution is FlexiFaCT [6], which addresses the conflicting update problem by training disjoint sets of training data points. Two sets of points, P_1 and P_2, are disjoint if for any point $x_{ijk} \in P_1$ and $y_{lmn} \in P_2$, we have $i \neq l$, $j \neq m$ and $k \neq n$. FlexiFaCT trains disjoint sets of points in parallel in each iteration, then invokes a global barrier, and switches to next set in the next iteration. The problem of such a partitioning scheme is that these disjoint sets are usually small, resulting in shortlived iterations and frequent global barriers. In each iteration, all threads have to wait for the slowest one (usually the one getting most training points), resulting in poor scalability.

In the following sections, we introduce the LFTF algorithm, which is a novel lock-free solution to address the conflicting update problem with the following key advantages: (1) it does not have the network transmission bottleneck as in Parameter Server, (2) it does not suffer from the poor scalability problem due to disjoint sets as in FlexiFaCT, and (3) it removes the global barrier in SGD parallelization by asynchronous processing.

3. THE LFTF ALGORITHM

In this section, we present the LFTF algorithm. We first give an overview of the algorithm, and then we discuss the detailed aspects of the algorithm including the data layout, the non-blocking training process in each stage, and the implementation details.

Following existing works on tensor factorization [6, 10, 14, 15, 29], in this paper we also focus on factorizing 3-order tensors, while briefly describing the extension of the LFTF algorithm to handle higher order tensors as well as streaming data.

3.1 Overview

In LFTF, each mode of a K-th order tensor $\mathcal{X}$ may have one of the three different *roles*, namely, ρ_m, ρ_p and ρ_r, where the subscripts refer to "***migrating***", "***partitioned***", and "***replicated***", respectively, the function of each role will be discussed shortly. For simplicity, our discussion assumes that $\mathcal{X}$ is a third order tensor, and we will discuss how to extend to higher order later.

LFTF proceeds in stages. Among the three roles, ρ_m is fixed in all stages and we assign a mode in $\mathcal{M}$ to ρ_m before the first stage, where $\mathcal{M} = \{1, 2, 3\}$. Assume w.l.o.g. that we assign $\rho_m = 1$ (which means the first mode is assigned the "migrating" role), then ρ_m is fixed to 1 throughout the whole training process. After fixing ρ_m, the algorithm cycles ρ_p among the remaining modes, while ρ_r is the next mode of ρ_p in the cycle. In our example of $\mathcal{M} = \{1, 2, 3\}$ and $\rho_m = 1$, we cycle ρ_p among $\mathcal{M} \setminus \{\rho_m\} = \{2, 3\}$, that is: $\rho_p = 2$ in the first stage (which indicates $\rho_r = 3$), and $\rho_p = 3$ (and $\rho_r = 2$) in the next stage, and then the cycle repeats in subsequent stages. Thus, to be more precisely, LFTF proceeds in cycles of stages.

Algorithm 1: Overall Process of LFTF

Input : Tensor $\mathcal{X}_\Omega \in \mathbb{R}^{I \times J \times K}$, maximum number of stages T, and regularization parameter λ
Output: A, B, and C

```
1 begin
2     Let P = [A, B, C]; Randomize A, B and C
3     Choose a ρ_m ∈ M
4     Choose a ρ_p ∈ M \ {ρ_m}
      /* Compute stage by stage                    */
5     for t = 1, ..., T do
6         AsyncPSGD (X, P, ρ_m, ρ_p)
7         Cycle ρ_p to the next mode in M \ {ρ_m}
8         Synchronize the updated entries in P^{ρ_p}
```

A high-level view of LFTF is shown in Algorithm 1. Before the training actually starts, the three factor matrices A, B and C of a 3D tensor are randomly initialized. Here, we use P to represent a set of all the three factor matrices, P^i represents the factor matrix corresponding to the i-th mode, and P^i_r refers to the r-th row of the factor matrix P^i. In each stage, LFTF first picks ρ_p, and then feeds the factor matrices corresponding to the ρ_m-th mode and ρ_p-th mode to an asynchronous SGD algorithm, `AsyncPSGD` (to be described in later subsections). After that, updates to the factor matrix P^{ρ_p} are synchronized among the machines, and then ρ_p cycles to another mode.

3.2 Data Layout

Now we explain how the three different roles relate to the data layout. Suppose that A corresponds to the ρ_m-th mode, B corresponds to the ρ_p-th mode, and C corresponds to the ρ_r-th mode, then during the training in a particular stage:

1. B is evenly and horizontally ***partitioned*** across the machines.
2. All rows of C are ***replicated*** among all machines.
3. All rows of A are ***migrating*** asynchronously among the machines.

The above data layout shows that in each stage, a row of A or B can be found in exactly only one of the M machines, while all rows of C are available in any machine. When the role of a mode is switching from ρ_p to ρ_r, the partition of the factor matrix B in each machine needs to be broadcast and synchronized among all the machines (i.e., B will correspond to the replicated mode in the next stage).

While the layout of the factor matrices are changing during the LFTF computation, the training data are partitioned once only before the computation starts. First, one copy of the training data is partitioned by the ρ_p-th mode (i.e., the ρ_p-th attribute) among the machines in the cluster, and we denote this partitioning by Ω_1. Then, a second copy of the training data is partitioned by the ρ_r-th mode (i.e., the ρ_r-th attribute) among the machines, denoted by Ω_2. Note that the ρ_p-th and ρ_r-th mode here are the modes chosen for ρ_p and ρ_r before the computation by stages starts.

Each training point pt of the training data in a 3D tensor $\mathcal{X}$ is indexed by (i, j, k), such that x_{ijk} is the entry of pt in

Algorithm 2: AsyncPSGD

Input : A tensor $\mathcal{X}_\Omega \in \mathbb{R}^{I\times J\times K}$, P, ρ_m, and ρ_p
Output: Factor matrices A, B, and C

1 **begin**
  /* N_{lt} concurrent threads being executed */
2 **ParallelFor** $t \in \{1, \ldots, N_{lt}\}$
3 **while** *switch signal is not received* **do**
4 **if** *queue[t] not empty* **then**
5 $P_r^{\rho_m}$ = queue[t].pop()
  /* let μ be the id of this machine */
6 **for** $(\mathbf{a}_i, \mathbf{b}_j, \mathbf{c}_k) \in \Omega^\mu_{\rho_p} \bowtie P_r^{\rho_m}$ **do**
7 Perform PartialGD
  /* migrate among the machines */
8 migrate $P_r^{\rho_m}$ to the next machine

$\mathcal{X}$. We say a training point pt *corresponds to* a row P_r^d if $pt_d = r$, where $d \in \{1, 2, 3\}$, and pt_d is the d-th index of pt. We use Ω_i^μ to denote the partition of Ω_i (where $i \in \{1, 2\}$) stored in machine μ, and $\Omega_d^\mu \bowtie P_r^{\rho_m}$ to represent all the training points in Ω_d^μ that correspond to $P_r^{\rho_m}$.

3.3 Training

The AsyncPSGD (short for *Asynchronous Partial Stochastic Gradient Descent*), an important component of LFTF, is given in Algorithm 2. During the AsyncPSGD process inside a stage, the rows of P^{ρ_m} keep migrating among machines (Line 8). Upon arrival at a machine, a latent vector $P_r^{\rho_m}$ will perform gradient descent over the training data in that machine (Lines 2-7).

There are N_{lt} simultaneous training loops, where N_{lt} is the number of local threads in use, each is called a trainer. Each trainer is associated with a queue to receive incoming latent vectors. At the beginning of a training loop, the trainer will pop latent vectors from its queue. Upon the availability of a new latent vector $P_r^{\rho_m}$, the trainer fetches all the training points corresponding to $P_r^{\rho_m}$ from the machine, iterates through the points, and updates $P_r^{\rho_m}$ and the latent vector of P^{ρ_p} that the current point corresponds to. Assume that $\mathbf{a}_i$, $\mathbf{b}_j$ and $\mathbf{c}_k$ are the i-th, j-th and k-th row of A, B and C, respectively, involved in the update, where C is replicated, then we perform the update, denoted by PartialGD, as follows:

$$\begin{aligned}\mathbf{a}_i &\leftarrow \mathbf{a}_i - \eta_t[(\mathbf{a}_i(\mathbf{b}_j. * \mathbf{c}_k)^T - x_{ijk})(\mathbf{b}_j. * \mathbf{c}_k) + \lambda \mathbf{a}_i],\\ \mathbf{b}_j &\leftarrow \mathbf{b}_j - \eta_t[(\mathbf{a}_i(\mathbf{b}_j. * \mathbf{c}_k)^T - x_{ijk})(\mathbf{a}_i. * \mathbf{c}_k) + \lambda \mathbf{b}_j].\end{aligned} \tag{8}$$

The name PartialGD is shorthand for *partial gradient descent*, since the latent vectors of only two modes are updated (compared with Equation (7)) and hence we refer to the update process as a *partial gradient descent* process.

After iterating through all the training points corresponding to $P_r^{\rho_m}$, $P_r^{\rho_m}$ migrates to the next machine in a round-robin manner, and is pushed into the queue of a trainer in that machine. The trainer to process $P_r^{\rho_m}$ is also chosen in a round-robin manner. The training process ends when a *switch* signal is received, which is issued by a global master process. Upon receiving the *switch* signal, the AsyncPSGD process ends, and ρ_p will cycle to the next mode.

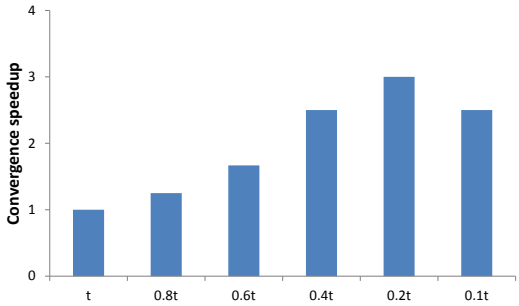


Figure 2: Converge speed vs. training duration

3.3.1 *Faster LFTF Convergence*

The AsyncPSGD process in Algorithm 2 terminates when the *switch* signal is received. If the *switch* signal is not used and the while-loop condition in Line 3 is always true, then AsyncPSGD terminates when the asynchronous SGD process in the current stage converges (let t be the time taken), after sampling enough training points. However, we found that if we terminate AsyncPSGD earlier before the asynchronous SGD process converges in a stage (i.e., sampling less training points), it leads to faster convergence to the overall LFTF computation with similar error. As a micro benchmark, Figure 2 shows the overall LFTF convergence speedup when running AsyncPSGD for a training duration of $c \cdot t$, by varying c from 0.1 to 1. We observe a speedup of 3 times when we terminate AsyncPSGD at $0.2t$, though terminating AsyncPSGD earlier leads to slower convergence likely because the overhead of stage switching starts to nullify the benefit.

We explain the above finding as follows. Let err be the error (e.g., *root-mean-square error*) obtained by running the asynchronous SGD process in a stage until it converges. Often, AsyncPSGD achieves $(err + \delta)$ for a small constant δ at time $c \cdot t$ for a small $c < 1.0$. In other words, the majority of the training time in each stage is used in improving $(err + \delta)$ to err, while $(err + \delta)$ is in fact already a satisfactory error that will lead to much faster overall LFTF convergence and still achieve a comparable final error. Based on this observation, we design an optimization to bring faster convergence to the overall LFTF computation as follows.

We can use a parameter c to set the training duration at each stage, i.e., a master process issues a *switch* signal to terminate AsyncPSGD after a duration of $c \cdot t$. Similar to the setting of the learning rate in SGD, it is difficult to come up with a strong theoretical result for the choice of c, but a value between 0.1 and 0.4 always gives a good overall LFTF convergence speedup with comparable final error in all the cases we tested. We set $c = 0.2$ as default. However, t changes from stage to stage, in particular t is much smaller in later stages. Thus, we cannot determine t. Instead, we use the number of updates performed in each stage.

Recall that we cycle ρ_p among $\mathcal{M} \setminus \{\rho_m\}$, and thus each cycle consists of $(|\mathcal{M}| - 1)$ stages. Let $N[i][j]$ be the number of updates performed in the j-th stage in the i-th cycle. Although t changes, $N[i][j]$ does not change much for different i. Thus, we can measure $N[1][j]$ for $1 \le j < |\mathcal{M}|$ in the 1st cycle of the LFTF computation. Then, starting from the 2nd cycle, we terminate AsyncPSGD after $c \cdot N[1][j]$ updates have been performed in the j-th stage of each cycle, for $1 \le j < |\mathcal{M}|$.

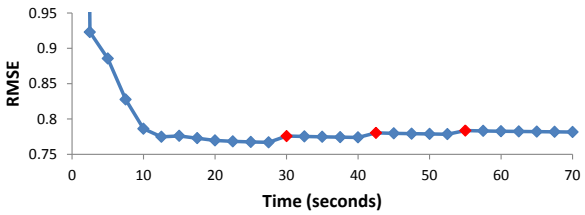


Figure 3: Incremental learning test

3.3.2 *Choice of ρ_m*

LFTF picks a mode for ρ_m that satisfies the following conditions:

1. The size of this mode should be as small as possible.
2. The size of this mode should be at least larger than the total number of cores in use in the cluster.

The first condition is intended to reduce network traffic, by migrating a smaller amount of latent vectors in P^{ρ_m} during `AsyncPSGD` computation. The second condition is to ensure that during the training, each core is busy processing a migrating latent vector.

3.3.3 *Handling of Skew Data Distribution*

It is possible that the training data are highly skewed, in which case the training data should no longer be partition evenly. A simple and effective way is to sample the training data and then partition the data in different ranges so that each contains roughly the same amount of rows. In this case, each migrating parameter will have roughly the same amount of computation in each machine, and thus we can mitigate the problem of stragglers.

3.4 Extending to Higher Order Tensor

Although our discussion so far focuses on a 3-mode tensor, the LFTF algorithm can be extended to factorize a higher order tensor. For factorizing a K-th order tensor with $K > 3$, we still have 3 roles but make the following change only: only one mode acts as ρ_m, another mode acts as ρ_p, and the remaining $(K-2)$ mode are ρ_r. There are $(K-1)$ copies of training data, corresponding to all the mode in $\mathcal{M} \setminus \{\rho_m\}$, partitioned in a way similar to the third order setting described in Section 3.2. The training process is also similar to the process discussed in Section 3.3, and in each stage only the latent vectors of ρ_m and ρ_p are updated.

3.5 Extending to Streaming Data

Another merit of LFTF is that it can be naturally extended to process streaming data, unlike most existing solutions [6, 10, 14, 15, 29]. Suppose there is a new training data point coming in, we simply merge it into the corresponding $K-1$ sets of partitions. This process will not interrupt the training, and the trainers just keep working until reaching the new convergence.

Figure 3 shows a microbenchmarks on the Netflix (user_id, movie_id, time) tensor data. We use only 70% of the original data in the training, and each red dot represents streaming 10% of the original data to join the training. In fact, such process is formally called Incremental Learning.

3.6 Implementation Details

LFTF is implemented based on a master-worker architecture, and runs one process on each worker. The master issues commands to workers, e.g., cycling ρ_p to a different mode (i.e., issuing a *switch* signal) or ending the training process. Each worker process keeps one communicator thread, which is responsible for receiving commands from the master and migrating rows (i.e., latent vectors) from other workers.

After receiving the migrating rows, the communicator thread assigns them to the local trainers in a round-robin manner. In this sense, the communicator also generates tasks (as an incoming latent vector means a pending round of training), and distributes them evenly to the local trainers. A process spawns N_{lt} local trainer threads, which keep polling incoming latent vectors from their queues during the training process. Queues are implemented using a circular array and are lock-free in the setting of single consumer and single producer (as each trainer is associated with its own queue). Our empirical experience shows that setting N_{lt} to $1.5N_c$ results in good performance as it keeps all the CPU cores busy, where N_c is the total number of physical cores in a computing node.

During the training process in a stage, all the executions are asynchronous. The communicator threads keep receiving parameters and the trainer threads keep sending out parameters, while the trainer threads perform the training and update the parameters after receiving them. In this case, the CPU computation and the network communication are both kept busy at the same time, unlike many existing synchronous solutions [6, 10, 14, 15, 29] where the CPUs are idle during massive network transmission.

3.7 Integration with General Systems

To show the value of the LFTF framework in practice, we build an LFTF algorithm package on Husky (http://www.husky-project.com/) [33], which is a general-purpose distributed computing platform. The non-blocking asynchronous execution of Husky greatly facilitates the implementation of LFTF (as LFTF requires some parameters to be continuously migrated across the cluster), and its Python binding makes access to the core functionality of LFTF much easier. For example, the following statement trains a model based on several columns of a relational table.

```
model = LFTF.train(
    cols=(my_table['user'],
          my_table['product'],
          my_table['location']),
    measure=my_table['rating'])
```

A data scientist may explicitly require a specific type of constraint (e.g., Non-Negative Constraint), and use the following command,

```
model = LFTF.train(
    cols=(my_table['user'],
          my_table['product'],
          my_table['location']),
    measure=my_table['rating'],
    constraint=LFTF.constraints.NonNeg())
```

When this model is prepared, LFTF provides two useful methods, `topk_sim` and `topk_rec`, for mining useful information from the model. The `topk_sim` method identifies

the top k similar entities, and `topk_rec` provides the top k recommendations with potentially high measurement given other columns. These are two basic but commonly used functions. Other methods can also be easily added to do more advanced analytics based on the model. The following code shows how to use `topk_sim` and `topk_rec`.

```
# return a list of 5 similar users
LFTF.topk_sim(key='Alice', dim=0, num=5)
# return a list of top 10 recommended products
# for Alice in London
LFTF.topk_rec(('Alice', '*', 'London'), num=10)
```

4. THEORETICAL ANALYSIS OF LFTF

In this section, we analyze the complexity and completion guarantee of the LFTF algorithm.

4.1 Complexity Analysis

Let Ω be the set of training data points and K be the order of the input tensor (i.e., a K-th order tensor). Storing the training data according to the layout in Section 3.2 requires $O((K-1)|\Omega|)$ space. While the space requirement may be high for a single machine, it is quite affordable in distributed computing. For example, suppose M machines are used, then each machine uses $O(|\Omega|(K-1)/M)$ space. Let R be the rank, then the space taken to store a latent vector is $O(R)$. Thus, the overall space complexity per machine is $O(|\Omega|(K-1)/M + (I_1+I_2+\cdots+I_K)R/M)$, where I_i is size of the i-th mode of the K-th order tensor.

We follow a similar way to the time complexity analysis in [34]. Let I^m be the size of the "migrating" mode, and I^{max} be the maximum size of any mode of the tensor. Assume that transmitting a latent vector requires $d \cdot R$ time and performing an SGD update needs $c \cdot R$ time, where d and c represent constants related to network latency and CPU computation, respectively. Consider a single latent vector that travels through all machines b times. The complexity related to CPU computation is $c \cdot O(bR\frac{|\Omega|}{I^m})$, and the complexity related to network transmission is $d \cdot O(bRM)$. An upper bound of the time for broadcasting at the end of each stage is $d \cdot O(RI^{max})$. In this case, when $c \cdot O(bR\frac{|\Omega|}{I^m})$ is much larger than $d \cdot O(bRM)$ (which is usually true when the conditions in Section 3.3.2 are satisfied), a stage of LFTF computation is essentially CPU-bound. When the training time inside a stage is much larger than $d \cdot O(RI^{max})$, the whole LFTF computation is CPU-bound and scales linearly. We verify our analysis by the following experimental results.

Figure 4 plots the network usage in factorizing the Netflix tensor data obtained in our experiment. We can see that most of the time the network usage is low, meaning that the LFTF computation is CPU-bound. Also, the synchronization of only one mode is observable (i.e., the spikes) in the figure, since the cardinality of its associated attribute is much larger than that of the other modes. During the short periods when the spikes happen, `AsyncPSGD` is also working on the updated values along with the synchronization without being idle. The same observation also applies to other datasets used in our experiments.

Next, we give an analysis to justify our design of fixing one mode as the "migrating" mode in LFTF. An interesting phenomenon we observed is that even if we cycle the "migrating" status among the other modes, the result is still no

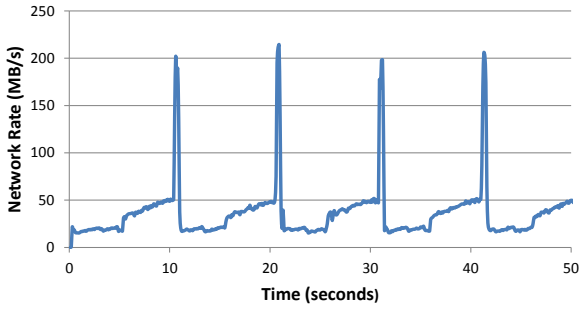


Figure 4: Network usage plot for factorizing the Netflix tensor

better than fixing a "migrating" mode. The following is an analysis to explain this phenomenon.

Consider a third-order tensor, $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$, assume without loss of generality that $I \leq J \leq K$. According to Section 3.3.2, we choose the 1-st mode as the "migrating" mode. To simplify the analysis, we assume that the 3-rd mode is selected as the "replicated" mode, and the entries of the input tensor are all observed. Let A, B and C be the three factor matrices of the tensor, corresponding to the 1-st, 2-nd and 3-rd mode, respectively. Then, the update steps for A and B are formulated as follows:

$$\begin{aligned} A^+ &\leftarrow A + \eta_t \Big\{ \Big[A(C \odot B)^T - \mathcal{X}_{(1)} \Big] (C \odot B) + \lambda A \Big\}, \\ B^+ &\leftarrow B + \eta_t \Big\{ \Big[B(C \odot A)^T - \mathcal{X}_{(2)} \Big] (C \odot A) + \lambda B \Big\}. \end{aligned} \tag{9}$$

LEMMA 1. *[9] Let M be an $n_1 \times n_2$ incoherent matrix of rank r, and $n = \max\{n_1, n_2\}$. Suppose that we observe m entries of M with locations sampled uniformly at random. Then, there exist constants C and c such that if*

$$m \geq Cn^{6/5} r(\beta \log n),$$

for some $\beta > 2$, then the minimizer to the nuclear norm model is equal to M with probability at least $1 - cn^{-\beta}$.

According to Lemma 1, the sampling complexity of the three mode-k unfoldings completion problems is $O((JK)^{1.2} r \log(JK))$, $O((IJ)^{1.2} r \log(IJ))$ and $O((IK)^{1.2} r \log(IK))$. Thus, since $I \leq J \leq K$, completing the 2nd and 3rd mode unfoldings requires lower measurements than completing the 1st mode unfolding.

4.2 Completion Guarantee

To provide the completion guarantee for factorizing higher-order tensors, we extend the partial observation estimation theorem for matrices as in [8, 31] to the higher-order tensor case, which mainly involves a tensor covering number argument and Hoeffding inequality for sampling without replacement [28]. In addition, we also analyze the decomposition and completion performance of our sparse tensor completion and decomposition model (6). For simplicity of discussion, we assume that the 3rd-order tensor $\mathcal{X}$ is of size $I \times I \times I$, and has tensor rank R, though our analysis can be easily extended to the more general case, i.e., $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_K}$.

4.2.1 Tensor Covering Number Argument

We first extend the covering number argument for low-rank matrices in [8, 31] to third-order tensors [23], as stated in the following theorem and lemma.

THEOREM 1. *Let* $\mathcal{S}_R = \{\mathcal{X} \in \mathbb{R}^{I\times I\times I} | \text{ Tucker-rank}(\mathcal{X}) \preceq (R,R,R), \|\mathcal{X}\|_F \leq \nu\}$. *Then there exists an* ϵ*-net* $\overline{\mathcal{S}}_R$ *with the covering number* $|\overline{\mathcal{S}}_R|$ *for the Frobenius norm obeying*

$$|\overline{\mathcal{S}}_R| \leq (12\nu/\epsilon)^{R^3+3RI}. \tag{10}$$

To prove Theorem 1, we first give the following lemma, which uses the triangle inequality to characterize the combined effects of perturbations in the factors of the higher-order singular value decomposition (HOSVD) [18] form for the true tensor $\mathcal{T}$:

$$\mathcal{T} = [\![A,B,C]\!] = [\![\mathcal{C};U_1,U_2,U_3]\!]. \tag{11}$$

LEMMA 2. *Let* $\mathcal{C},\mathcal{C}' \in \mathbb{R}^{R_1\times R_2\times R_3}$ *and* $U_k, U_k' \in \mathbb{R}^{I_k\times R_k}$ *with* $U_k^T U_k = I_{R_k}$, $U_k'^T U_k' = I_{R_k}$, $\|\mathcal{C}\|_F \leq \nu$ *and* $\|\mathcal{C}'\|_F \leq \nu$. *Then*

$$\begin{aligned}&\|[\![\mathcal{C};U_1,U_2,U_3]\!] - [\![\mathcal{C}';U_1',U_2',U_3']\!]\|_F \\ \leq &\|\mathcal{C}-\mathcal{C}'\|_F + \nu\sum_{k=1}^{3}\|U_k - U_k'\|_2.\end{aligned} \tag{12}$$

The lemma is in essence the same as Lemma 2 in [24], where the main difference is the ranges of $\|\mathcal{C}\|_F$ and $\|\mathcal{C}'\|_F$: instead of having $\|\mathcal{C}\|_F = 1$ and $\|\mathcal{C}'\|_F = 1$, we have $\|\mathcal{C}\|_F \leq \nu$ and $\|\mathcal{C}'\|_F \leq \nu$, where $\nu > 0$ is any positive number. Using Lemma 2, we construct an ε-net for $\mathcal{S}_R$ by building $\epsilon/4$-nets for each of the 4 factors $\{U_k\}$ and $\mathcal{C}$.

4.2.2 Error Bound

We give the partial observation theorem for higher-order tensor completion, which involves the covering number argument in Theorem 1 and the Hoeffding inequality for sampling without replacement [28], stated as follows.

THEOREM 2. *Assume* $\max_{i_1,i_2,i_3}|\mathcal{X}_{i_1i_2i_3}| \leq \gamma$. *Let* $\mathcal{L}(\mathcal{X}) = \frac{1}{\sqrt{I^3}}\|\mathcal{X}-\mathcal{X}'\|_F$ *and* $\mathcal{L}'(\mathcal{X}) = \frac{1}{\sqrt{|\Omega|}}\|\mathcal{X}_\Omega - \mathcal{X}'_\Omega\|_F$ *be the actual and empirical loss function, respectively. Then for all tensors* $\mathcal{X}$ *with* Tucker-rank $(\mathcal{X}) \preceq (R,R,R)$, *with probability greater than* $1-2\exp(-I)$, *there exists a fixed constant* C *such that*

$$\sup_{\mathcal{X}\in\mathcal{S}_R}\left|\mathcal{L}(\mathcal{X}) - \mathcal{L}'(\mathcal{X})\right| \leq C\gamma\left(\frac{(R^3+3RI)\log(\sqrt{I^3})}{|\Omega|}\right)^{1/4}.$$

The proof can be derived by following the proof of Theorem 2 in [31], where the main difference is that the covering number argument in Theorem 1 for third-order tensors is used to replace that of Lemma A2 in [31] for low-rank matrices.

In the following, we show that when sufficiently many entries are sampled, the solution of our model recovers a tensor close to the ground-truth one. We assume that the observed tensor $\mathcal{X}' \in \mathbb{R}^{I\times I\times I}$ can be decomposed as a true tensor $\mathcal{T}$ with Tucker-rank (r,r,r) and a random gaussian noise $\mathcal{E}$ whose entries are independently drawn from $\mathcal{N}(0,\sigma^2)$, i.e., $\mathcal{X}' = \mathcal{T} + \mathcal{E}$.

DEFINITION 2. *The root mean square error (RMSE) is a frequently used measure of the difference between the solution* $\mathcal{X} = [\![A,B,C]\!]$ *and the true tensor* $\mathcal{T}$:

$$\text{RMSE} := \frac{1}{\sqrt{I^3}}\|\mathcal{T} - [\![A,B,C]\!]\|_F. \tag{13}$$

Using the partial observation theorem for third-order tensors, as stated in Theorem 2, we give the error bound for our model as follows.

THEOREM 3. *Let* $\mathcal{X} = [\![A,B,C]\!]$ *be the solution of our sparse tensor completion and decomposition model (6). Then there exists two absolute constants* C_1 *and* C_2, *such that with probability at least* $1-2\exp(-I)$,

$$\text{RMSE} \leq C_1\gamma\left(\frac{(R^3+3RI)\log(\sqrt{I^3})}{|\Omega|}\right)^{\frac{1}{4}} + \frac{\|\mathcal{E}_\Omega\|_F}{C_2\sqrt{|\Omega|}} + \frac{\|\mathcal{E}\|_F}{\sqrt{I^3}},$$

where $\gamma = \max_{i_1,i_2,i_3}|\mathcal{X}_{i_1i_2i_3}|$.

The proof can be derived in a way similar to that in [22].

From Theorem 3, we can see that when the sample complexity $|\Omega| \gg (R^3+3RI)\log(\sqrt{I^3})$, the first term diminishes, and the RMSE is essentially bounded by the average magnitude of entries of the noise tensor $\mathcal{E}$. In other words, our algorithm is stable. If $\mathcal{E} = 0$, i.e., without any noise, our formulation needs only $O(R^3+3RI)$ observations to exactly recover all $\mathcal{X} \in \mathcal{S}_R$, as stated in Theorem 1 in [24], while much more observations are required for recovering the true tensor by the other models [13,21,24].

Due to space limitation, we put the detailed proofs of the theorems in an online appendix of this paper: http://www.cse.cuhk.edu.hk/proj-h/pub/LFTF-appendix.pdf.

5. EXPERIMENTAL EVALUATION

We studied the performance of LFTF on 3 large real-world datasets, comparing with the best existing methods [6,10,14,29]. We also evaluated the scalability of LFTF, using five large synthetic datasets, with the number of rows ranging from 2 billions to 10 billions. *The results show that LFTF has much superior performance, in terms of both convergence rate and training error, and handles datasets much larger than existing methods.*

5.1 Experiment Settings

We compared with the following methods: HaTen2 [14], CDTF [29], SALS [29], DFacTo-ALS [10], DFacTo-GD [10]. Among these methods, HaTen2 strictly adheres to the Map-Reduce programming paradigm, while others do not and hence the processes of different machines can directly communicate. The implementations of these methods are from their authors and were carefully tuned so that their performance match the results in their original papers.

We ran all the experiments on a Linux computing cluster, where each machine has 48GB RAM, two 2.0GHz Intel(R) Xeon(R) CPU (12 physical cores in total), a 450GB SATA disk (6Gb/s, 10k rpm, 64MB cache), and a Broadcom Gigabit Ethernet NIC. The Hadoop version we used is Hadoop-2.6.0, and for MPI it is MPICH-3.1.4. Unless otherwise stated, we used 18 machines to run the experiments, and we made sure that all programs involved in the experiments had access to all the available computing resources.

5.2 Convergence Analysis

We first analyze the convergence rate, the training error obtained at convergence, and other convergence characteristics of LFTF and the existing methods.

In this set of experiments, we used three large real-world datasets: Netflix, YahooMusic, and AmazonMovie. The

Table 2: Summary of real-world datasets

Dataset	I	J	K	Observations
Netflix	480,189	17,770	2,182	100,480,507
YahooMusic	1,823,179	136,736	9,442	699,640,226
AmazonMovie	889,167	253,059	1,762,532	720,766,541

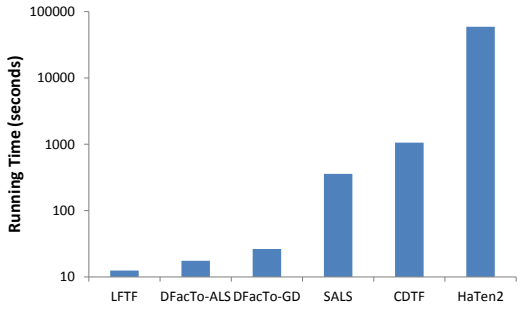


Figure 5: Running time on Netflix (in log scale)

Netflix dataset has three columns, (*user_id*, *movie_id*, *date*), and the YahooMusic dataset is in the form of (*user_id*, *music_id*, *artist_id*). The AmazonMovie dataset is converted from the reviews of movies on Amazon into the form of (*user_id*, *movie_id*, *word*), by splitting the review text into words and removing stop words. Ratings by the users are used as the measure. Table 2 gives a summary of the datasets.

5.2.1 *Running Time Comparison*

We ran all the methods to convergence and measured their running time as the metric to evaluate the efficiency of each algorithm. Figure 5 plots the running time (in log scale) of the algorithms on the Netflix dataset. We first observe that HaTen2 is two orders of magnitude slower than the other algorithms, which we explain as follows. HaTen2 is implemented on Hadoop and performs computation in rounds of MapReduce jobs. Each iteration of HaTen2 involves hundreds of MapReduce jobs; thus, by considering the overhead of starting each MapReduce job alone, this can take several thousands of seconds to complete an iteration. We omitted the results of HaTen2 in the following study because Haten2 took hours or even days to finish a complete run.

Like HaTen2, SALS and CDTF are also implemented on Hadoop. However, SALS and CDTF do not strictly follow the MapReduce programming paradigm: when they enter the Reduce phase, Reducers directly communicate with each other and use message passing to perform distributed computing. This allows them to get rid of the costly "context switch" (i.e., dumping to HDFS and loading from it) between consecutive MapReduce jobs, and thus enjoy much higher efficiency than HaTen2. Another algorithm, FlexiFaCT [6], is also implemented in this way, but it was shown in [29] that FlexiFaCT is clearly beaten by both SALS and CDTF, and thus SALS and CDTF serve as better baselines for our comparison.

Figure 6 reports the running time of all the algorithms except HaTen2 on the three real-world datasets. LFTF is the fastest algorithm on all the three datasets. DFacTo-ALS and DFacTo-GD are also fast, but they have poorer convergence quality as to be shown in Section 5.2.2. SALS and CDTF are at least 17 times slower than LFTF, which reveals the advantage of using SGD in our LFTF approach, compared

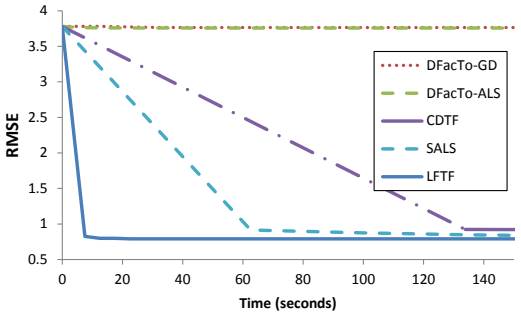


Figure 7: RMSE plot of Netflix (best viewed in color)

with the ALS approach adopted in SALS and CDTF. The key difference is that LFTF performs updates using the lock-free asynchronous SGD approach, and fully utilizes all the CPU cores at all time.

5.2.2 *Convergence Quality Comparison*

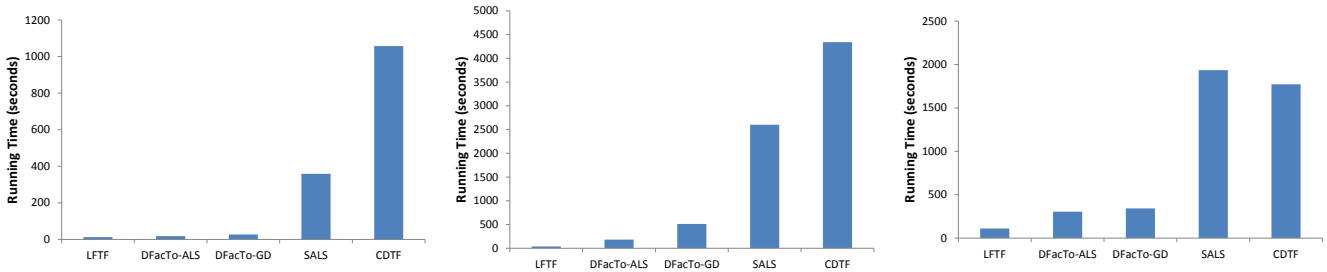
Next, we plot the training root-mean-square error (RMSE) over time to verify the correctness and monotonic decrease of training error. We are interested in finding which algorithm can produce lower RMSE within the same time period.

We first show in Figure 7 that DFacTo-ALS and DFacTo-GD do not obtain reasonable RMSE for the Netflix dataset (similar patterns also obtained for the other datasets), as they only converge to a high RMSE value. Note that although the two lines of DFacTo-ALS and DFacTo-GD look flat in Figure 7, we actually ran them until convergence, i.e., the decrease in their RMSE value is less than a threshold (which is 0.0001 for all the algorithms). As a rating in the Netflix dataset ranges from 1 to 5, an RMSE value of greater than 3 implies that it is even worse than simply setting 3 as the predicted rating for all entries. As the authors of DFacTo-ALS and DFacTo-GD only reported the training time but not the training error in [10], we also could not make cross comparison of the results. Thus, we omit their results in the following analysis.

Figure 8 shows the RMSE plot for all the datasets. LFTF is able to attain a lower RMSE much faster than the other methods. But we note that given sufficient time, the other method such as SALS also attains similar RMSE as LFTF.

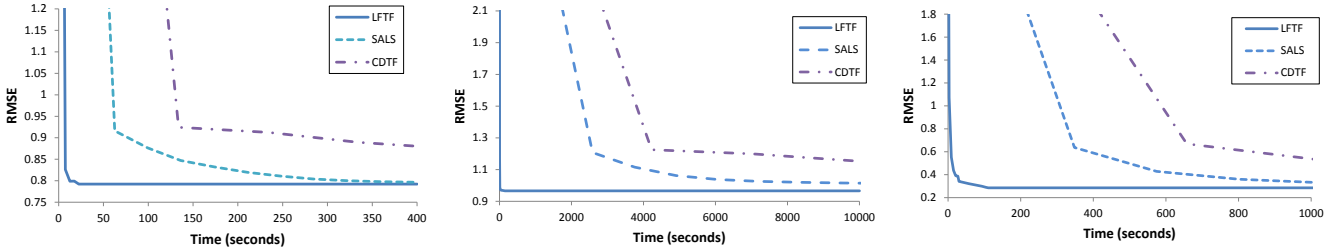
Analysis on performance improvement. LFTF is a new solution that enjoys the benefits of the state-of-the-art ALS-based and SGD-based solutions (i.e., massive parallelism/light computation), yet without their performance limitations (i.e., time-consuming network transmission). Thus, LFTF shares similar convergence characteristics as ALS and SGD. However, we remark that, in the case of distributed computing, a faster theoretical convergence rate does not necessarily imply shorter wall-clock running time, as the overhead of data transmission often leads to long overall running time. Thus, the primary focus of our work, as well as that of the other existing distributed methods [6, 10, 14, 15, 29] for the same reason, is not to improve the convergence quality or theoretical convergence rate, but to improve the efficiency of distributed tensor factorization.

Compared with existing distributed methods, LFTF is able to attain a lower RMSE much faster than these methods, as reported in Figures 5-8, for the following reasons. Existing distributed methods have to shuffle all parameters across the machines in the cluster, leading to $O(R(I_1 + I_2 +$



(a) Netflix **(b)** YahooMusic **(c)** AmazonMovie

Figure 6: Running time on different real-world datasets



(a) Netflix **(b)** YahooMusic **(c)** AmazonMovie

Figure 8: RMSE plot of different real-world datasets

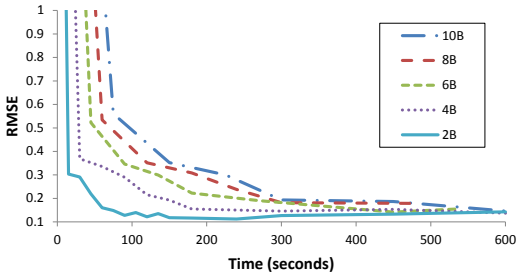


Figure 9: Scalability by varying training set size

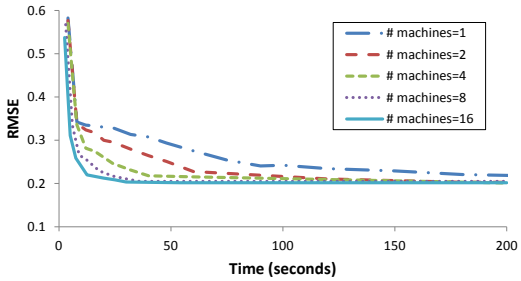


Figure 10: Scalability by varying machine number

$\cdots + I_K)$) network transmission complexity (needed for shuffling all the factor matrices) for each single iteration, during which the CPUs are being idle. In contrast, in LFTF the CPU is busy performing computation all the time. As the discussion in Section 4.1 shows that $(c \cdot O(bR\frac{|\Omega|}{I^m})) \gg (d \cdot O(bRM))$ holds for LFTF, meaning that the network transmission time is negligible compared with the CPU computation time. Thus, LFTF is able to do much more real computation work within the same time frame.

5.3 Scalability Test

In this set of experiments, we tested the scalability of LFTF using synthetic datasets based on the characteristics of Netflix. Let $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$ be the Netflix tensor with O observations, where I, J, K, and O are given in Table 2. We generated 5 datasets $\mathcal{X}_i \in \mathbb{R}^{(i \cdot f \cdot I) \times (i \cdot f \cdot J) \times (i \cdot f \cdot K)}$ with $(i \cdot f \cdot O)$ observations, where $i \in \{2, 4, 6, 8, 10\}$ and $f = 10$. In other words, we enlarge the cardinality of each attribute by $i \cdot f$ times, and generated 5 tensors with 2, 4, 6, 8 and 10 billion observations, respectively. The largest tensor used here is larger than the largest in any existing work.

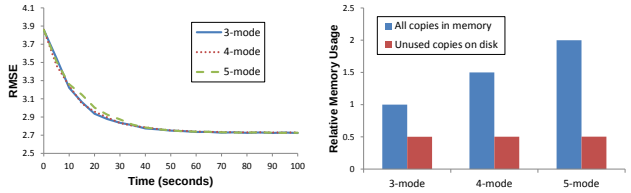
We measured the time needed for LFTF to converge for the 5 synthetic datasets, as plotted in Figure 9. We can see that the time taken by LFTF to converge increases gracefully and linearly when more training data are used.

We also tested the horizontal scalability of LFTF by fixing the dataset but changing the number of machines. The dataset we used has only 100 million observations so that it can easily fit in a single machine. Figure 10 shows that the time to reach the same RMSE is almost halved when the number of machines is doubled, which verifies the horizontal scalability of LFTF.

5.4 Training Higher-Order Tensor Models

Although our work focuses on 3-mode tensors, we also tested LFTF on higher-order tensors to investigate its convergence characteristics. We generated three tensor datasets, with the number of modes ranging from 3 to 5. The size of training data, i.e., the number of rows, was fixed at 1 billion and each mode corresponds to 100,000 parameters. Using $\eta = 0.0002$ and $\lambda = 0.8$, we obtained the convergence characteristics as shown in Figure 11a.

Figure 11a shows that LFTF can effectively train tensors with higher order to convergence with comparable quality and efficiency. As LFTF is a methods that trades memory for faster speed, an important trade-off here is that LFTF



(a) Convergence characteristics **(b)** Memory consumption

Figure 11: Training higher-order tensor models

needs to replicate $(K-1)$ copies of training data, which may limit its scalability when the tensor order increases, as shown by the blue bars in Figure 11b. However, we observed that LFTF only needs one copy of the training data in each stage of training. Thus, we can keep the unused copies in secondary storage, and prefetch only the required copy of training data into main memory when we are about to start the next stage. In this way, we can significantly reduce the total memory usage, as shown by the red bars in Figure 11b, while keeping comparable running speed with prefetching.

6. RELATED WORK

Alternating Least Squares (ALS) and Gradient Descent (GD). Bader and Kolda [4] developed an efficient algorithm to solve sparse Tensor Factorization model, in which they re-arrange the update formula in order to avoid intermediate data explosion problem caused by the materialization of very large, unnecessary intermediate Khatri-Rao products. Similar strategies are used in DFacTo-ALS and DFacTo-GD [10], which re-arrange the update formula in a different way. These works focus on shorter iteration time, but their update rules are very coarse-grained and do not consider the sparsity of data, resulting in problematic convergence behavior. In addition, the MapReduce-based GigaTensor [15] and HaTen2 [14] involve many rounds of MapReduce jobs and are hence slow.

SALS [29] and CDTF [29] are two better ALS-based methods, which consider the sparsity of data in designing update rules. Although ALS itself is easy to parallelize, it has high computational complexity and cannot effectively interleave CPU computation with network transmission. In contrast, LFTF enjoys much higher efficiency than SALS and CDTF on the same hardware, as shown in our experiments.

Stochastic Gradient Descent (SGD). The problem of ALS and GD, which are adopted by the aforementioned methods, is that they cannot reach a reasonable convergence in an efficient manner like SGD. FlexiFaCT [6] implements distributed SGD on the MapReduce framework. Although SGD completely avoids the data explosion problem of ALS and GD, the difficulty of designing SGD algorithm is how to resolve conflicting updates. To tackle this problem, FlexiFaCT performs SGD in a synchronous manner. Although in this case there is no conflicting update, it introduces extensive synchronization and locking. As one round of SGD is usually fast with linear time complexity, the overhead of frequent execution barriers overtakes the benefit of SGD, and significantly slows down the overall process.

Asynchronous ML Algorithms. In recent years we have seen growing interests in the development of highly-efficient asynchronous ML algorithms. HogWild! [26] uses a method to exploit concurrency for SGD in the shared-memory setting, based on the assumption that the training data is sparse enough, and conflicting updates rarely happen. However, the same techniques cannot apply in the shared-nothing setting, where updates to global parameters are no longer timely enough. In order to achieve higher throughput for the distributed setting, Ho *et al.* proposed a framework called *Stale Synchronous Parallel* (*SSP*) [12], which attempts to improve performance by reusing stale parameters on a local machine when updated values (stored in a key-value store) are not yet available. Using stale parameters is known to make convergence slower, while reading/writing data from/to the key-value store also incurs high communication overheads.

Large-Scale Matrix Factorization (MF). Large-scale MF has been extensively studied as MF is an important building block in a recommender system. The two state-of-the-art lock-free MF algorithms are FPSGD [35] and NOMAD [34], both employ different strategies to avoid conflicting SGD updates in a lock-free manner. However, compared with HogWild!, the two algorithm does not exploit the data sparsity as HogWild! does in a local machine. Twitter developed the Factorbird system [27] dedicated for MF, which employs partitioning as most lock-free approaches do, and performs asynchronous updates locally using the HogWild!-style SGD. Many existing methods for solving MF using SGD have excellent performance, and Factorbird combines these contributions into a highly usable system. However, all these methods cannot be naturally generalized to tensor data while retaining their efficiency.

7. CONCLUSIONS AND FUTURE WORK

We presented LFTF, a novel distributed solution for large-scale tensor factorization. LFTF re-formulates the original problem, so that it becomes possible to exploit distributed asynchronous execution, and thus gets rid of the BSP synchronization overhead. We verified by experiments that LFTF converges significantly faster than the best existing methods, with a lower training error.

Our work also reveals several possible future research directions: (1) More high-level data mining tools can be built upon LFTF in order to enjoy much faster tensor analytics. (2) The use of asynchrony should extend to wider areas in machine learning and data mining, and there are much more possibilities of using Husky's programming framework to naturally expresses asynchrony and develop highly efficient algorithms (the work in this paper is just one example). (3) Researchers can make use of Husky's expressive yet simple enough framework to experiment and develop novel distributed algorithms to solve many big data problems (e.g., machine learning, graph analytics, and even coarse-grained workloads [19, 33]), as over-simplified frameworks (e.g., MapReduce or Parameter Servers) may limit the design space as well as algorithm performance, while the use of low-level tools (e.g., MPI) significantly increases the development cost. For future work, we aim to develop more efficient and scalable algorithms that follow the above three directions.

Acknowledgment. The work was supported by the HK GRF 14206715 (2150851) and 14222816 (2150895), Grant 3132821 funded by the Research Committee of CUHK.

8. REFERENCES

[1] E. Acar, D. Dunlavy, T. Kolda, and M. Mørup. Scalable tensor factorizations with missing data. In *SDM*, pages 701–711, 2010.

[2] A. Anandkumar, R. Ge, D. Hsu, S. M. Kakade, and M. Telgarsky. Tensor decompositions for learning latent variable models. *JMLR*, 15(1):2773–2832, 2014.

[3] A. Anandkumar, R. Ge, D. Hsu, S. M. Kakade, and M. Telgarsky. Tensor decompositions for learning latent variable models (A survey for ALT). In *ALT*, pages 19–38, 2015.

[4] B. W. Bader and T. G. Kolda. Efficient MATLAB computations with sparse and factored tensors. *SIAM J. Sci. Comput.*, 30(1):205–231, 2007.

[5] J. Bennett, S. Lanning, and N. Netflix. The netflix prize. In *In KDD Cup and Workshop in conjunction with KDD*, 2007.

[6] A. Beutel, P. P. Talukdar, A. Kumar, C. Faloutsos, E. E. Papalexakis, and E. P. Xing. Flexifact: Scalable flexible factorization of coupled tensors on hadoop. In *SDM*, pages 109–117, 2014.

[7] P. Bhargava, T. Phan, J. Zhou, and J. Lee. Who, what, when, and where: Multi-dimensional collaborative recommendations using tensor factorization on sparse user-generated data. In *WWW*, pages 130–140, 2015.

[8] E. Candès and Y. Plan. Tight oracle inequalities for low-rank matrix recovery from a minimal number of noisy random measurements. *IEEE Trans. Inf. Theory*, 57(4):2342–2359, 2011.

[9] E. Candès and B. Recht. Exact matrix completion via convex optimization. *Found. Comput. Math.*, 9(6):717–772, 2009.

[10] J. H. Choi and S. Vishwanathan. Dfacto: Distributed factorization of tensors. In *NIPS*, pages 1296–1304, 2014.

[11] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *SIGKDD*, pages 69–77, 2011.

[12] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. R. Ganger, and E. P. Xing. More effective distributed ML via a stale synchronous parallel parameter server. In *NIPS*, pages 1223–1231, 2013.

[13] P. Jain and S. Oh. Provable tensor factorization with missing data. In *NIPS*, pages 1431–1439, 2014.

[14] I. Jeon, E. E. Papalexakis, U. Kang, and C. Faloutsos. Haten2: Billion-scale tensor decompositions. In *ICDE*, pages 1047–1058, 2015.

[15] U. Kang, E. E. Papalexakis, A. Harpale, and C. Faloutsos. Gigatensor: scaling tensor analysis up by 100 times - algorithms and discoveries. In *SIGKDD*, pages 316–324, 2012.

[16] A. Karatzoglou, X. Amatriain, L. Baltrunas, and N. Oliver. Multiverse recommendation: n-dimensional tensor factorization for context-aware collaborative filtering. In *RecSys*, pages 79–86, 2010.

[17] T. G. Kolda and B. W. Bader. Tensor decompositions and applications. *SIAM Rev.*, 51(3):455–500, 2009.

[18] L. Lathauwer, B. Moor, and J. Vandewalle. A multilinear singular value decomposition. *SIAM J. Matrix Anal. Appl.*, 21(4):1253–1278, 2000.

[19] J. Li, J. Cheng, Y. Zhao, F. Yang, Y. Huang, H. Chen, and R. Zhao. A comparison of general-purpose distributed systems for data processing. In *IEEE International Conference on Big Data*, pages 378–383, 2016.

[20] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B. Su. Scaling distributed machine learning with the parameter server. In *OSDI*, pages 583–598, 2014.

[21] J. Liu, P. Musialski, P. Wonka, and J. Ye. Tensor completion for estimating missing values in visual data. *IEEE Trans. Pattern Anal. Mach. Intell.*, 35(1):208–220, 2013.

[22] Y. Liu, F. Shang, W. Fan, J. Cheng, and H. Cheng. Generalized higher-order orthogonal iteration for tensor decomposition and completion. In *NIPS*, pages 1763–1771, 2014.

[23] Y. Liu, F. Shang, W. Fan, J. Cheng, and H. Cheng. Generalized higher order orthogonal iteration for tensor learning and decomposition. *IEEE Trans. Neural Netw. Learn. Syst.*, 2015.

[24] C. Mu, B. Huang, J. Wright, and D. Goldfarb. Square deal: Lower bounds and improved relaxations for tensor recovery. In *ICML*, pages 73–81, 2014.

[25] M. Nickel, V. Tresp, and H. Kriegel. A three-way model for collective learning on multi-relational data. In *ICML*, pages 809–816, 2011.

[26] B. Recht, C. Re, S. J. Wright, and F. Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *NIPS*, pages 693–701, 2011.

[27] S. Schelter, V. Satuluri, and R. Zadeh. Factorbird - a parameter server approach to distributed matrix factorization. *CoRR*, abs/1411.0602, 2014.

[28] R. J. Serfling. Probability inequalities for the sum in sampling without replacement. *Ann. Statist.*, 2:39–48, 1974.

[29] K. Shin and U. Kang. Distributed methods for high-dimensional and large-scale tensor factorization. In *ICDM*, pages 989–994, 2014.

[30] J. Sun, D. Tao, and C. Faloutsos. Beyond streams and graphs: Dynamic tensor analysis. In *KDD*, pages 374–383, 2006.

[31] Y. Wang and H. Xu. Stability of matrix factorization for collaborative filtering. In *ICML*, 2012.

[32] E. P. Xing, Q. Ho, W. Dai, J. K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu. Petuum: A new platform for distributed machine learning on big data. In *SIGKDD*, pages 1335–1344, 2015.

[33] F. Yang, J. Li, and J. Cheng. Husky: Towards a more efficient and expressive distributed computing framework. *PVLDB*, 9(5):420–431, 2016.

[34] H. Yun, H. Yu, C. Hsieh, S. V. N. Vishwanathan, and I. S. Dhillon. NOMAD: nonlocking, stochastic multi-machine algorithm for asynchronous and decentralized matrix completion. *PVLDB*, 7(11):975–986, 2014.

[35] Y. Zhuang, W. Chin, Y. Juan, and C. Lin. A fast parallel SGD for matrix factorization in shared memory systems. In *RecSys*, pages 249–256, 2013.