Large-scale and Scalable Latent Factor Analysis via Distributed Alternative Stochastic Gradient Descent for Recommender Systems

Article · February 2020

DOI: 10.1109/TBDATA.2020.2973141

CITATIONS
83

READS
261

5 authors, including:

Qiang He
Swinburne University of Technology
222 PUBLICATIONS **4,901** CITATIONS

SEE PROFILE

Xin Luo
Shantou University
107 PUBLICATIONS **3,513** CITATIONS

SEE PROFILE

Ming Sheng Shang
Chongqing Institute of Green and Intelligent Technology, Chinese Academy of Sci…
134 PUBLICATIONS **5,054** CITATIONS

SEE PROFILE

Large-scale and Scalable Latent Factor Analysis via Distributed Alternative Stochastic Gradient Descent for Recommender Systems

Xiaoyu Shi, *Member*, *IEEE*, Qiang He, *Senior Member*, *IEEE*, Xin Luo, *Senior Member*, *IEEE*, Yanan Bai, and Mingsheng Shang

Abstract—Latent factor analysis (LFA) via stochastic gradient descent (SGD) is highly efficient in discovering user and item patterns from high-dimensional and sparse (HiDS) matrices from recommender systems. However, most LFA-based recommender systems adopt a standard SGD algorithm, which suffers limited scalability when addressing big data. On the other hand, most existing parallel SGD solvers are either under the memory-sharing framework designed for a bare machine or suffering high communicational costs, which also greatly limits their applications in large-scale systems. To address the above issues, this paper proposes a distributed alternative stochastic gradient descent (DASGD) solver for an LFA-based recommender. Its training-dependences among latent features are decoupled via alternatively fixing one-half of the features to learn the other half following the principle of SGD but in parallel. It's distribution mechanism consists of efficient data partition, allocation and task parallelization strategies, which greatly reduces its communicational cost for high scalability. Experimental results on three large-scale HiDS matrices generated by real-world applications demonstrate that the proposed DASGD algorithm outperforms state-of-the-art distributed SGD solvers for recommender systems in terms of prediction accuracy as well as scalability. Hence, it is highly useful for training LFA-based recommenders on large scale HiDS matrices with the help of cloud computing facilities.

Index Terms—Recommender System, Latent Factor Analysis, High-Dimensional and Sparse Matrices, Alternative Stochastic Gradient Descent, Distributed Computing

1 Introduction

PROMOTED by the rapid advances in clouding computing and Internet of Things (IoT) technologies, the last decade has witnessed a data explosion in the human world [1-5]. It becomes increasingly difficult for people to retrieve their truly desired information from such a big amount of data. In this context, recommender systems, as one of the most promising approaches for alleviating this information overload pain, have been widely employed in many domains. Recommender systems can provide personalized services [6-7], including information retrieval, to users by analyzing their historical behaviors [8-10]. Taking this advantage, many major Internet companies, e.g., Amazon [11], YouTube [12] and Yahoo! Music [13], have extensively employed recommender systems to improve their users' experiences in information retrieval.

- *X. Shi, X. Luo, Y. Bai and M. Shang are with the Chongqing Engineering Research Center of Big Data Application for Smart Cities, and Chongqing Key Laboratory of Big Data and Intelligent Computing, Chongqing Institute of Green and Intelligent Technology, Chinese Academy of Sciences, Chongqing 400714, China (e-mail:{xiaoyushi, luoxin21, yannanbai, msshang}@cigit.ac.cn).*
- *Q. He is with the School of Software and Electrical Engineering, Swinburne University of Technology, Melbourne, 3122, Australia. (email: qhe@swin.edu.au)*
- *X. Luo is also with the Hengrui (Chongqing) Artificial Intelligence Research Center, Department of Big Data Analyses Techniques, Cloudwalk, Chongqing 401331, and the Department of Computing, Hong Kong Polytechnic University, Kowloon, Hong Kong 999077, China.*
- *X. Shi, Q. He and Y. Bai contributed equally to this work and should be considered co-first authors.*
- ***Corresponding author: Xin Luo.***

Collaborative filtering (CF) is one of the most powerful and widely-used methods in recommender systems, for its high prediction accuracy and ease of implementation [14-17]. CF-based recommendation models can be further categorized into the neighborhood-based models (NBM) and latent factor analysis-based models (LFA). NBM makes recommendations based on entity relations like the user-item relations. Its main idea is to first identify each entities' neighborhood based on historical data, and then predict users' potential preferences based on their neighbors [11].

On the other hand, LFA makes recommendations based on dimensionality reduction techniques [18-23]. Compared with traditional matrix-factorization (MF) methods that deal with full matrices, LFA specializes in high-dimensional and sparse matrices (HiDS) that can be found in many domains, e.g., social network services [24], bioinformatics [25], business intelligence [26], etc. In general, LFA maps both users and items into the same low-rank latent factor (LF) space, trains desired LFs based on the given data, and then estimates unknown data based on the obtained LFs to make recommendations.

The core components of an LFA are the objective model and the solver, which directly impact the prediction accuracy and efficiency of the recommender system. The objective function of an LFA is usually a function related to LFs (i.e., the input to the model) and ratings (i.e., the output of the model). There are a lot of literature that is devoted into the construction of the objective model, e.g., the incremental MF model with bias term and regularization term [27],

the singular value decomposition (SVD) ++ model [28], the probabilistic MF model [29], and the Nonnegative MF (NMF) [30-32]. The solver of an LFA is usually implemented based on a pyramid of optimization techniques. In recent years, alternating least square (ALS) and stochastic gradient descent (SGD) are the most popular optimization technique for the design of recommender systems for their unique advantages, i.e., SGD's high efficiency [44] and ALS's scalability on distributed computing facilities [33].

LFA has received extensive attention for its ability to deal with the cold-start and sparsity issues faced by recommender systems. However, traditional LFA-based recommender systems are challenged by large-scale real-world applications. For example, recommender systems for web services such as recommendation services for TencentVideo (Tencent Inc. is one of the largest Internet companies in China), which generates nearly 4 billion user actions and more than 1 PB data every day. Those user actions and data must be processed by the recommender systems on-the-fly. Obviously, it is imperative for a recommender system to leverage a cluster of machines to accelerate the recommendation process.

The rapid development and advances of cloud computing provide mega-scale hardware resources with elastic expansion ability for processing large-scale datasets. However, the main obstacle to parallelize SGD is over-writing problem caused by the inner-dependence among the user and item latent feature pair, which leads to the computation conflict during a training epoch. As a result, it prevents LFA recommenders with serial SGD solvers from fully leveraging the power of cloud computing.

From this point of view, an efficient parallel SGD solver to LFA should carefully address the inner-dependence among involved latent features. Current approaches can be further divided into two categories:

1) Memory-sharing. In such a method, the raw rating data are wholly loaded into the main memory. Each solving units shares the main memory (i.e., the rating set and latent feature matrices). Thus, it actually focuses on the collaborative computing between multiple processors, making it only suitable for a standalone server with multiple computational cores. Hogwild! [49] and alternative SGD (ASGD) [50] are the representative algorithms of this kind. Nonetheless, with the exponential growth of data volume in real-world applications, the scale of rating data in a recommender system can grow far beyond the capacity of a single server.
2) Distribution. A method of this kind focuses on distributing an algorithm on a cluster of multiple computing nodes. Taking DSGD [51] for example, it splits the rating matrix into several independent blocks and then assigns them to different computational node. The obvious benefit of such a design is that it can fully leverage the power of a cluster to handle big data.

In today's big data era, the communication cost more expensive than computation cost, since the scale of computing increase quickly (e.g., the data centers with millions of chips). As a result, the data communication cost directly determines the efficiency of parallel and distributed algorithms in the cloud computing environment. However, existing distribution-based parallel SGD solvers like DSGD fail in well balancing data exchange and synchronization, preventing them to achieve satisfactory speedup even with the help of large-scale cloud-computing facilities.

To address the above issues, this study proposes a distributed alternating stochastic gradient descent (DASGD) solver for performing scalable LFA on a recommender system. Its essential idea is two-fold: 1) alternatively learning desired LFs for decoupling their inner-dependences like in an ASGD algorithm [50]; and 2) optimizing its data partition and computation through a hybrid approach combining dual and parallel optimization. The main contributions of this paper include:

1) A DASGD algorithm, including its parallel formulation, algorithm design, and theoretical analysis;
2) A hybrid optimization approach to DASGD on large-scale cloud-computing facilities, which combines dual-parallel optimization for data partition and computation. For data partition optimization, a vertical partitioning method and a horizontal partitioning method are proposed to reduce the data communication cost among multiple nodes. For computation, a directed acyclic graph (DAG) for learning tasks in DASGD-based LFA is constructed via a resilient distributed dataset (RDD) model.
3) Extensive experiments conducted on three industrial datasets in real cloud environment.

Note that an ASGD model is initially designed under the memory–sharing framework for a bare machine. It does not consider the communication cost among multiple computational nodes. Hence, it cannot take the advantage of large-scale cloud computing facilities, making its scalability limited. In contrast, a DASGD model mostly focuses on the communicational design and dual-parallel optimization. It is highly scalable and more suitable to address large-scale LFA problems emerged from real applications. According to the authors' best knowledge, such efforts are never seen in any previous studies. Note that detailed discussions regarding the proposed model and related studies are given in the supplementary file of this paper.

The rest of this paper is organized as follows. Section 2 gives the preliminaries. Section 3 describes a DASGD algorithm, and the distribution optimization of DASGD on Spark. Section 4 empirically evaluates DASGD. Section 5 reviews the related works and the principle comparison among state-of-the-art parallel SGD approaches. Finally, Section 6 discusses and concludes this paper.

2 Preliminaries

2.1 Problem Statement

A CF-based recommender system generally makes user-specific recommendations of items depend on a user-item matrix that contains information on user-item interactions, e.g., rating, purchase, clicks.

Definition 1. User-Item matrix (R). Formally, R is an inner

product of user set U and item set I, which can be represented as $|U|\times|I|$. $r_{u,i}$ in R is user u's rating on item i.

Definition 2. The problem of CF. Given Λ and Γ as the sets of known and unknown rating in R, where $|\Lambda|<<|\Gamma|$ for a HiDS matrix, the problem of CF is to construct an estimator $\hat{R}$ to R to achieve or approximate the following objective,

$$\arg\min\left(\sum_{(u,i)\in\Lambda}\left|\hat{r}_{u,i}-r_{u,i}\right|\right) \tag{1}$$

where $\hat{r}_{u,i}$ is the approximation to $r_{u,i}$ in R.

Definition 3. Latent factor analysis (LFA). LFA aims to factorize the original set of known ratings Λ into two low-rank matrices, $P^{|U|\times f}$ and $Q^{|I|\times f}$, where $f<<\min\{|U|,|I|\}$. The predicted rating $\hat{r}_{u,i}$ is defined as the inner product of the corresponding user-item feature vector pair given by:

$$\hat{r}_{u,i}=p_u q_i^T \tag{2}$$

In (2) the key part is the parameter estimation (i.e., p_u and q_i). A common way is to directly model only on the set of known ratings Λ with a Tikhonov regularization term to address the issue of overfitting problems. The regularized squared error (RES) is minimized as

$$\min_{P,Q}\sum_{(u,i)\in\Lambda}\left(r_{u,i}-p_u q_i^T\right)^2+\lambda\left(\left\|p_u\right\|^2+\left\|q_i\right\|^2\right), \tag{3}$$

where λ controls the degree of regularization. Due to the fact that both of p_u and q_i are unknown in (3), it is a non-convex optimization function. Therefore, finding the global minimum is intractable. Fortunately, optimization techniques like alternating least square and stochastic gradient descent are feasible for it.

2.2 SGD-based training

Stochastic gradient descent is a popular optimization algorithm for large-scale machine learning [37], due to its low requirement for memory and high execution efficiency. A SGD solver runs several iterations through the set of known ratings Λ until the stop criteria, e.g., the maximum number of iterations or the error threshold, is fulfilled. At each iteration, SGD traverses the rating records in the form of <user, item, rating>, and updates parameters p_u and q_i at the same time using the following equations:

$$\begin{aligned} p_u &= p_u+\gamma(e_{u,i}q_i-\lambda p_u)\\ q_i &= q_i+\gamma(e_{u,i}p_u-\lambda q_i)\end{aligned} \tag{4}$$

where γ is the learning rate, and $e_{u,i}$ is the prediction error (i.e., $e_{u,I}=r_{u,i}-\hat{r}_{u,i}$).

For SGD, based on the rating $r_{u,i}$ it updates involved p_u and q_i, yielding the computation complexity at $O(f)$. Hence, its computation complexity of one iteration is $O(|\Lambda|f)$.

3 A DASGD Algorithm

3.1 Parallelization

3.1.1 Parameter interdependence in SGD

A SGD-based LFA is an iterative learning model. All of rating records are learned sequentially during each training iteration. The processes for updating p_u and q_i are dependent on each other. To parallelize the SGD-based LFA model so that it can be implemented in a distributed manner, we first relax the inner dependent constraints between p_u and q_i during the updating processes. The general training process of LFA with a SGD solver includes the prediction rule (2), the RES calculation (3) and the latent factors updating rule (4). Let $\Lambda(U=u)$ denotes the subset of Λ that contains all rating records of a specific user u. Each training instance in $\Lambda(U=u)$ can be orderly marked as $r_{u,1}$, $r_{u,2}$, $r_{u,3}$,$\cdots$,$r_{u,k}$, where k is the total number of items rated by user u. i.e., $k=|\Lambda(U=u)|$. In each training iteration, let $p_u^{(0)}$ denotes the initial value of user feature p_u, and $p_u^{(1)}$ is the temporary result after rating $r_{u,1}$ is trained. From the user feature updating rule (4), we know that:

$$\begin{aligned} p_u^{(1)} &= p_u^{(0)}+\gamma(e_{u,1}q_1^{temp}-\lambda p_u^{(0)})\\ &= (1-\lambda\gamma)p_u^{(0)}+\gamma(e_{u,1}q_1^{temp})\end{aligned} \tag{5}$$

After that, we can obtain the expression $p_u^{(2)}$ after training the rating value $r_{u,2}$:

$$\begin{aligned} p_u^{(2)} &= p_u^{(1)}+\gamma(e_{u,2}q_2^{temp}-\lambda p_u^{(1)})\\ &= (1-\lambda\gamma)p_u^{(1)}+\gamma(e_{u,2}q_2^{\text{temp}})\\ &= (1-\lambda\gamma)^2 p_u^{(0)}+(1-\lambda\gamma)\gamma(e_{u,1}q_1^{temp})+\gamma(e_{u,2}q_2^{temp})\end{aligned} \tag{6}$$

Similarly, we drive the expression of p_u after the sequential training on the rating value from $r_{u,3}$ to $r_{u,k}$. Hence, the final form of $p_u^{(k)}$ is:

$$\begin{aligned} p_u^{(k)} &= p_u^{(k-1)}+\gamma(e_{u,k}q_k^{temp}-\lambda p_u^{(k-1)})\\ &= (1-\lambda\gamma)p_u^{(k-1)}+\gamma(e_{u,k}q_k^{temp})\\ &= \ldots\\ &= (1-\lambda\gamma)^k p_u^{(0)}+(1-\lambda\gamma)^{k-1}\gamma(e_{u,1}q_1^{temp})+\ldots\\ &\quad +(1-\lambda\gamma)\gamma(e_{u,k-1}q_{k-1}^{temp})+\gamma(e_{u,k}q_k^{temp})\end{aligned} \tag{7}$$

Let $w=1-\lambda\gamma$, (7) can then be transformed into:

$$\begin{aligned} p_u^{(k)} &= w^k p_u^{(0)}+w^{k-1}\gamma(e_{u,1}q_1^{temp})+w^{k-2}\gamma(e_{u,2}q_2^{temp})\\ &\quad +\ldots+w\gamma(e_{u,k-1}q_{k-1}^{temp})+\gamma(e_{u,k}q_k^{temp})\end{aligned} \tag{8}$$

where q_k^{temp} is the temporal state of q_k when training the kth item rated by user u.

For the re-computing process for updating Q, let $\Lambda(I=i)$ denotes the subset of Λ that contains all the rating records on item i, i.e., $r_{1,i}$, $r_{2,i}$, $\cdots$, $r_{H,i}$, where $H=|\Lambda(I=i)|$. Then, the final form of q_i after using a serial of rating values $r_{1,i}$, $r_{2,i}$, $\cdots$, $r_{H,i}$ is:

$$\begin{aligned} q_i^{(H)} &= w^H q_i^{(0)}+w^{H-1}\gamma(e_{1,i}p_1^{temp})+w^{H-2}\gamma(e_{2,i}p_2^{temp})\\ &\quad +\ldots+w\gamma(e_{H-1,i}p_{H-1}^{temp})+\gamma(e_{H,i}p_H^{temp})\end{aligned} \tag{9}$$

where p_H^{temp} is the temporal state of p_H.

3.1.2 Interdependence Decoupling

From the above analysis, we see that the parameter updating process of LFA is not only dependent on the initial value, but also on the state of related features. For example, the value of $p_u^{(k)}$ depends on the value of q_i if user u has rated on item i. Because of the inter-dependence of user features and item features, the training process of LFA cannot be separated into independent processes. Thus, once

we remove the inter-dependence among the parameters, we can parallelize each epoch. Actually, the final outputs after a training epoch are only relevant to the initial values of user/item features, if they are not dependent on the temporal state of each other, which has been proved theoretically in Section 3.1.1. This can be achieved by separating each epoch into two parts. In one of the two parts, one kind of latent features is trained with the other fixed.

ALGORITHM 1 SERIAL VERSION OF DASGD-BASED LFA TO TRAINING PROCESS

Input: Known rating set Λ, initial feature matrices P and Q, the parameters λ and γ, the max training epochs $Iter_{max}$, the number of users m and the number of items n.

Output: the trained feature matrices P and Q

	Code	Computing Cost
1:	w=1-$\lambda\gamma$	O(1)
2:	**For** *iter* from 1 to $Iter_{max}$ **do**	×$Iter_{max}$
3:	**For** i from 0 to m-1 **do** % fix with Q	×m
4:	**For** $j \in \Lambda(U=i)$ **do**	×$\lvert\Lambda(U=i)\rvert$
5:	$\hat{r}_{i,j} \leftarrow \sum_{k=0}^{f-1} p_{i,k}q_{k,j}$	O(f)
6:	$e_{u,i} = r_{u,i} - \hat{r}_{i,j}$	O(1)
7:	**For** k from 0 to f-1 **do**	×f
8:	$p_{i,k} = wp_{i,k} + \gamma(e_{i,j}q_{k,j})$	O(1)
9:	**End For**	--
10:	**End For**	--
11:	**End For**	--
12:	**For** j from 0 to n-1 do %fix with P	×n
13:	**For** $i \in \Lambda(I=j)$ **do**	×$\lvert\Lambda(I=j)\rvert$
14:	$\hat{r}_{i,j} \leftarrow \sum_{k=0}^{f-1} p_{i,k}q_{k,j}$	O(f)
15:	$e_{u,i} = r_{u,i} - \hat{r}_{i,j}$	O(1)
16:	**For** k from 0 to f-1 **do**	×f
17:	$q_{k,j} = wq_{k,j} + \gamma(e_{i,j}p_{i,k})$	O(1)
18:	**End For**	--
19:	**End For**	--
20:	**End For**	--
21:	**End For**	--
22:	**Return** P and Q	--

Inspired by the alternating updating method from ALS solver, DASGD solves this problem via the below process:

1) Initialize feature matrices P and Q as P^0 , Q^0, t = 0;
2) Repeat until a pre-defined convergence criterion is fulfilled;
3) Fix P^t, solve Q^{t+1} by minimizing the RES in (3), i.e., RES(P^t, Q^{t+1})<RES(P^t, Q^t), and updating each item latent feature using (9);
4) Fix Q^{t+1}, solve P^{t+1} by minimizing the RES in (3), i.e., RES(P^{t+1}, Q^{t+1})<RES(P^t, Q^{t+1}), and updating each user latent feature using (8);

DASGD decouples the inner-dependent between user features and item features, and updates different parameters simultaneously. The pseudo code for the DASGD solver is summarized in ***Algorithm 1***.

Let $Iter_{max}$ and f denote the number of training epochs and the dimension of the latent factor space. Note that in ***Algorithm 1,*** $\lvert\Lambda(U=i)\rvert$ represents the number of non-zero elements (i.e., ratings by user i) in the ith row of R. Hence, with R being sparse, ($m\lvert\Lambda(U=i)\rvert$) comes to $\lvert\Lambda\rvert$ instead of $\lvert U\rvert\times\lvert I\rvert$, where the former is the number of known entries in R while the latter is the full size of R. Hence, the total cost of updating P in each epoch (Lines 3-8 in ***Algorithm 1***) is $O(m\lvert\Lambda(U=i)\rvert f)=O(\lvert\Lambda\rvert f)\ll O(mnf)$. Similarity, the total computational complexity of updating Q (Lines 12-17 in ***Algorithm 1***) is $O(n\lvert\Lambda(I=j)\rvert f)=O(\lvert\Lambda\rvert f)\ll O(mnf)$. Thus, the total computational complexity of each iteration in DASGD is $O(\lvert\Lambda\rvert f)$.

3.1.3 Parallelized Training Rule

The over-writing problem is the main barrier to parallelizing SGD in recommender systems. The essence of over-writing problem is that the inner-dependence on the user latent feature and the item latent feature pair, leading to the computation conflict during a training epoch. Based on the proposed decouple method, we parallelize the training process of DASGD. It effectively overcomes the main barrier to parallelizing SGD. As analyzed in Section 3.1.1, the process for updating P and Q in one iteration of DASGD can be formulated as follows:

$$\begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ p_j \\ \vdots \\ p_{|U|} \end{bmatrix} = \begin{bmatrix} \sum_{k\in\Lambda(U=1)} (1-\lambda\gamma)p_1 + \gamma\left(r_{1,k} - p_1 q_k^T\right)q_k \\ \sum_{k\in\Lambda(U=2)} (1-\lambda\gamma)p_2 + \gamma\left(r_{2,k} - p_2 q_k^T\right)q_k \\ \cdots \\ \sum_{k\in\Lambda(U=j)} (1-\lambda\gamma)p_j + \gamma\left(r_{j,k} - p_j q_k^T\right)q_k \\ \cdots \\ \sum_{k\in\Lambda(U=|U|)} (1-\lambda\gamma)p_{|U|} + \gamma\left(r_{j,k} - p_{|U|} q_k^T\right)q_k \end{bmatrix} \tag{10}$$

$$\begin{bmatrix} q_1 \\ q_2 \\ \vdots \\ q_j \\ \vdots \\ q_{|I|} \end{bmatrix} = \begin{bmatrix} \sum_{k\in\Lambda(I=1)} (1-\lambda\gamma)q_1 + \gamma\left(r_{k,1} - p_k q_1^T\right)p_k \\ \sum_{k\in\Lambda(I=2)} (1-\lambda\gamma)q_2 + \gamma\left(r_{k,2} - p_k q_2^T\right)p_k \\ \cdots \\ \sum_{k\in\Lambda(I=j)} (1-\lambda\gamma)q_j + \gamma\left(r_{k,j} - p_k q_j^T\right)p_k \\ \cdots \\ \sum_{k\in\Lambda(I=|I|)} (1-\lambda\gamma)q_{|I|} + \gamma\left(r_{j,k} - p_k q_{|I|}^T\right)p_k \end{bmatrix} \tag{11}$$

Algorithm 2 presents the pseudo code of the parallelization version of DASGD-based LFA. In ***Algorithm 2,*** the total computational complexity of updating P (Lines 3-8) is $O\left(\lceil\lvert U\rvert/C\rceil\times\lvert\Lambda(U=i)\rvert\times f\right)\approx O\left(\lceil\lvert\Lambda\rvert/C\rceil\times f\right)$. Similarity, the total computational complexity of updating Q (Lines 9-14) is $O\left(\lceil\lvert I\rvert/C\rceil\times\lvert\Lambda(I=j)\rvert\times f\right)\approx O\left(\lceil\lvert\Lambda\rvert/C\rceil\times f\right)$. Therefore, the total computational complexity of each iteration for the parallelization version of DASGD is $O\left(\lceil\lvert\Lambda\rvert/C\rceil\times f\right)$. Hence, the computation time of serial DASGD can be effectively reduced with the parallelization version with multiple computational cores. It demonstrates that the computational complexity of a full training of DASGD is inversely proportional to the total number of CPU cores C.

ALGORITHM 2 PARALLELIZATION TRAINING PROCESS OF DASGD-BASED LFA

Input: Known rating set Λ, initial feature matrices P and Q, the parameters λ and γ, the max training epochs $Iter_{max}$ and the total number of CPU cores C.

Output: the trained latent feature matrices P and Q.

	Code	Computing Cost
1:	w=1-$\lambda\gamma$	O(1)
2:	**For** *iter* from 1 to $Iter_{max}$ **do**	×$Iter_{max}$
3:	**For** *loop* from 0 to $\lceil |U|/C \rceil - 1$ **do**	× $|U|/C$
4:	**parallel**: % with fixed $\boldsymbol{Q}$	--
5:	Select $C_{id} \in \{0,\cdots,C\text{-}1\}$	O(1)
6:	Select $i \in \{0,\cdots,|U|\text{-}1\}$	O(1)
7:	Dispatch each C_{id} to train a set of ratings $r_{i,j} \in \{i,j \mid j \in \Lambda(U{=}i)\}$ for updating $\boldsymbol{p}_i$ independently	×($|\Lambda(U{=}i)|$×f)
8:	**End For**	--
9:	**For** *loop* from 0 to $\lceil |I|/C \rceil - 1$ do	× $|I|/C$
10:	**parallel:**	--
11:	Select $C_{id} \in \{0,\cdots,C\text{-}1\}$	O(1)
12:	Select $j \in \{0,\cdots,|I|\text{-}1\}$	O(1)
13:	Dispatch each C_{id} to train a set of ratings $r_{i,j} \in \{i,j \mid i \in \Lambda(I{=}j)\}$ for updating $\boldsymbol{q}_j$ independently	×($|\Lambda(I{=}j)|$×f)
14:	**End For**	--
15:	**End For**	--
16:	**Return** P and Q	--

Based on the LFA with DASGD solver, we consider the following issues in a distributed computing environment:

1) How to allocate sparse data onto cluster memory with the lowest communication cost?
2) How to implement the most efficient distribution for an DASGD-LFA recommender on a cluster?

3.2 Distribution

To improve its efficiency and mitigate the communication cost when handling large-scale datasets, we design a distribution approach of the DASGD-LFA recommender on Apache Spark. Spark is selected as the platform for building the distributed recommendation system for the following major reasons [56-57]: 1) Spark supports fast in-memory data processing; 2) Spark's Resilient Distributed Datasets (RDDs) enable multiple operators in memory, while Hadoop has to write interim results to disks; 3) Spark is highly scalable and allows nodes to be added and deleted dynamically; and 4) it is fault-tolerant and stateless which allows fast recovery from failures.

Based on the Spark platform, our DASGD-LFA recommender system is optimized with a hybrid parallel approach combining computation parallelization and data parallelization. The data-parallel optimization is realized with a data-aware partition method that follows the updating rules of DASGD. It can significantly reduce data transmission operations in a distributed environment. Taking advantage of the rich operations supported by Spark, an updating task DAG of LFA is constructed based on the RDD model, which is used to optimize the computation task execute on distributed cluster nodes.

3.2.1 Data-aware partition

With the inner-dependence of latent feature vectors relaxed by DASGD, a data-aware partition method is designed for training the LFA model with a large-scale dataset. The training dataset is vertically/horizontally split into several latent feature subsets, and each feature subset is individually allocated to the Spark cluster.

During the process for training LFA, the updating tasks of each user/item latent feature vector consume most of the training time. However, these tasks are only related to the target rating values and the corresponding latent feature variables. For example, updating p_u is only related to $\{q_j, r_{u,j} \mid j \in \Lambda(U{=}u)\}$ as shown in ***Algorithm* 2**. Therefore, to reduce the volume of data and the data communication cost, we propose a data-aware partition method for distributed LFA, which split the ratings and latent factors into several subsets in a vertical/horizontal way.

Since the updating process of P and Q are symmetric, here we only discuss the data partition process for updating Q. Assuming that the size of $|\Lambda(I{=}j)|$ is $H(j)$, i.e., there are a total of $H(j)$ users that rated item j, uid_0, $\cdots$, $uid_{H(j)}$ represent the users who rated item j. Ap_{uid0},$\cdots$, $Ap_{uidH(j)}$ are the location information of corresponding user latent feature vectors storage in the cluster. Then, each physical address of user feature vector Ap_{uid0} and the rating value $r_{uid0,j}$ are selected to generate a latent factor partition LF_{qj}. It can be expressed as:

$$LF_{q_j} = \left\{ \begin{bmatrix} uid_0, & uid_1, & ..., & uid_{H(j)} \end{bmatrix} \begin{bmatrix} Ap_{uid_0}, & Ap_{uid_1}, & ..., & Ap_{uid_{H(j)}} \end{bmatrix} \begin{bmatrix} r_{uid_0,j}, & r_{uid_1,j}, & ..., & r_{uid_{H(j)},j} \end{bmatrix} \right\} \tag{12}$$

Each partition is then loaded as a block object that is independent of the other subsets. Note that the user index, the addresses of latent feature subset and the rating subset are re-organized as arrays rather than a tuple with three values in the forms of <uid_0, Ap_{uid0}, $r_{uid0,j}$>. As a result, it greatly reduces the costs of saving instances and transmitting data with continuous memory, which substantially reduces memory fragmentation.

Similarity, the content of latent feature partition LF_{pi} is represented as:

$$LF_{P_i} = \left\{ \begin{bmatrix} iid_0, & iid_1, & ..., & iid_{K(i)} \end{bmatrix} \begin{bmatrix} Aq_{iid_0}, & Aq_{iid_1}, & ..., & Aq_{iid_{K(i)}} \end{bmatrix} \begin{bmatrix} r_{i,iid_0}, & r_{i,iid_1}, & ..., & r_{i,iid_{K(i)}} \end{bmatrix} \right\}, \tag{13}$$

where iid is the index of the item rated by user i, and $K(i)$ is the length of the item subset rated by user i, i.e., $K(i)=|\Lambda(U{=}i)|$. Aq_{iid} is the location information of the latent feature vector of item iid. Note that each rating data is saved into two LF partitions respectively but from different partition indexes. Therefore, two copies are saved in the Spark cluster for each rating record. This effectively avoids the data transmission between the machines in the Spark cluster during the iterative process for proposed DASGD.

To implement our data-aware partition method, the correlation between P and Q based on the raw known rating set Λ needs to be saved. Hence, we first create a P-Q table. As presented in Table I, each element in the table is saved in a two-tuple form (i.e., < *relationFlag*, *blockIds*>), where *relationFlag* is a single value and *blockIds* is also a tuple with three values. In detail, *relationFlag* represents the relationship between the specified user latent feature and specified item latent feature based on whether the user rated on the item, while *blockIds* records the location index of the blocks that save the rating instance, specified user and item latent feature in the Spark cluster. For example, the value of p_1-q_1 is (1, $blockIdsr_{1,1}$), where 1 means that p_1 and q_1 are related, $blockIdsr_{1,1}$ records the block indexes of rating instance $r_{1,1}$, p_1 and q_1 saved in the Spark cluster. Note that block is the basic physical storage unit in the cluster.

TABLE I
P-*Q* TABLE BASED ON RATING MATRIX

ID	q_1	q_2	…	q_n
p_1	(1, $blockIdsr_{1,1}$)	(0, *NULL*)	…	(1,$blockIdsr_{1,n}$)
p_2	(0, *NULL*)	(1,$blockIdsr_{1,2}$)	…	(0, *NULL*)
…	…	…	…	…
p_{m-1}	(1,$blockIdsr_{m-1,1}$)	(0, *NULL*)	…	(0, *NULL*)
p_m	(0, *NULL*)	(0, *NULL*)	…	(0, *NULL*)

Second, because the DASGD solver is an iterative algorithm, the P-Q table is saved in the cache space to further accelerate the computation process by avoiding repetitive calculation in every iteration. Our data-aware partitioning method for DASGD-LFA is presented in Fig. S1. According to the updating process of DASGD, the updating user/item latent feature is related to the target rating values and the corresponding latent feature variables. For example, updating p_u is only related to $\{q_j, r_{u,j} | j \in \Lambda(U=u)\}$ as shown in ***Algorithm 2***. Then, we select user latent factor p_u, a subset of rating values rated by user u, and the location information of the corresponding item latent factors in one partition (i.e, LFp_u) by using the P-Q table, where P-Q table saves the relationship between each pair of $<p_u,q_j>$ based on $r_{u,j}$, and the storage addresses in cluster.

As shown in Fig. S1, a DASGD algorithm write the location information of desired LFs into the proposed P-Q table. Then the P-Q table is duplicated to the local memory of all computational nodes. Based on such a design, in a DASGD algorithm each partition only includes related target ratings and location information of corresponding LFs, besides the target latent feature. Thus, the training process is taken on each computational node without unnecessary data exchange during each training iteration of a DASGD algorithm, thereby greatly reducing the communicational cost as well as improving the storage accessing efficiency.

3.2.2 Data allocation

To address the data imbalance in an HiDS matrix and migrate the global communication, we adopt a static data scheduling strategy to automatically dispatch those partitions with varying sizes. Note that a Spark cluster usually contains a master node and a set of slave nodes. The master node is responsible for the task assignment and coordinated communication between different slaves that execute the tasks. Because of the different volumes of each latent feature partition, the workloads of all computing tasks during the training process will also vary. Our allocation strategy determines which latent feature partition should be allocated to which slave node(s), according to the partition size. A partition (e.g., LF_{pu}) includes many pairs of ($r_{u,i}$, q_i) that are saved in a slave according to its available memory. As shown in Fig. S2, there are three scenarios in the data allocation strategy:

1) When the available storage capacity of a slave equals to the volume of the *LF Partition*, the *LF Partition* will be assigned to this slave node.
2) When the available storage capacity of a slave is larger than the size of the *LF Partition*, several *LF Partitions* will be accommodated to this slave node.
3) When the available storage capacity of a slave is smaller than the size of the *LF Partition*, this *LF Partition* will be allocated to several slave nodes at similar physical locations.

ALORITHM 3 HORITIONAL DATA PARTITION AND DATA ALLOCATION STRATEGY OF DASGD-BASED LFA

Input:
RatingRDD: an RDD object of the original rating dataset.
Output:
LFList: a list of the indexes of each latent feature subset and the allocated slave nodes.

```
For i from 0 to |U|-1 do
  LFp_i ← horizontalPartition(RatingRDD, i)
  slaves ← findAviableNodes().sortbyIP()
  id ← 0
  If LFp_i.size < slaves [id].aviableSpace then
    slaves[id].aviableSpace ← slaves[id].aviableSpaces - LFp_i.size
    LFList ← < LFp_i , slaves[id].IP >
    LFp_i.persist()
    If slaves[id].aviableSpace == null then
      id ← id + 1
    End If
  Else
    while LFp_i .size ≠ null do
      (LF'p_i, LF''p_i) ← dividedLF(LFp_i, slaves[id].aviableSpaces)
      assignLFtoSlave(LF'p_i, slaves[id])
      LF'p_i.persist()
      slavesIPs.add(slaves[id].IP)
      id ← id +1
      LFp_i ← LF''p_i
    End while
    LFList ← < LFp_i, slavesIPs>
  End If
End For
Return LFList
```

In cases 1) and 2), no extra data transmission cost incurs across different slave nodes during the subsequent training computation process, which can find the required latent features directly in the target LF partition. In case 3), the data communication operations occur only in the local

communication among several slave nodes at similar physical locations and not the global communication across the cluster. In addition, this does not impact the updating of LF partition *(e.g., LFp_u)*, since the updating process of p_u is sequential but each latent feature of P is updated in parallel with Q fixed.

In summary, the proposed data allocation strategy assigns the LF partitions according to the storage capacity available and the physical locations of each computational node. The advantage of this method is that it effectively avoids global communication during the training process. Note that in ***Algorithm 3***, the rating dataset is first split into ($|U|$-1) LF partitions by the horizontal partition function. Then, according to the data allocation strategy described in Fig. S2, each LF partition is automatically assigned to slaves dependent on their available spaces and the size of the LF partition. For improving the computational efficiency, the LF partitions are saved into the main memory for reuse via the *persist*() function.

3.2.3 Task parallelization

Based on the results of the data-aware partitioning, we propose a task-parallel approach for DASGD-LFA recommender on Spark. Specifically, n data partition tasks are built in parallel at the first level of the training process of LFA, where n is the number of cluster slavers. And c user/item latent features in each slaver are calculated concurrently at the second level of the training process, where c is the number of CPU cores in each slaver.

According to the updating process of DASGD and the dependence of each RDD object, a task DAG of LFA is constructed based on the RDD model, which is used to perform the computation task in parallel on distributed nodes. Each job of the program of the DASGD-LFA training process is split into a serial of stages. An example of the task DAG of DASGD-LFA is described in Fig. S3.

In stage 1, n T_{DPV} tasks are generated for the vertical-partitioning task of Λ in the form of *RatingRDD*. First, each record in Λ is randomly dispatched to each slave node with hash operations. Then, the rating records are re-organized to *LF Partitions* with the map operation, according to the vertical-partitioning method and static data allocation strategy. After stage 1, the data required for calculating q_i (i.e., LF_{qi}) is packaged in T_{calcQi}, and c subtasks are also generated for updating c item latent features (e.g., q_1, …, q_c) simultaneously. These T_{calcQi} subtasks compute the corresponding subsets of latent feature matrix Q with the map operation and submit their results to $T_{updateQ}$ with the groupBy operation. Since the subtasks T_{DPVi} and T_{calcqi} are conducted on the same slave node, the cost of data transmission within the Spark cluster is reduced significantly. According to the DASGD algorithm shown in Section 3.1.3, the updated $LFRDD_Q$ is combined with *RatingRDD* in the form of RDD to produce the $LFRDD_P$. Similar to the calculation process for Q, both n T_{DPH} and the c T_{calcP} subtasks are generated for updating P. Finally, the T_{out} task is created for combining the result of T_{calcQ} and T_{calcP}.

3.2.4 Additional cost during distribution

Storage. Owing to the proposed data-aware partitioning strategy, the intermediate states of desired LFs are reused effectively during the training process of a DASGD algorithm. Given c, $|\Lambda|$ and f, the storage cost is $[2\times|\Lambda|+(|U|+|I|)\times f]$ only, which is easy to resolve practically and linear with the scale of a target problem.

Communication. In an DASGD algorithm, data communication happens in the data allocation and training process. Given c slaves and training data size at $[2\times|\Lambda|+(|U|+|I|)\times f]$, for data allocation the average cost of data communication is $[2\times|\Lambda|+(|U|+|I|)\times f]/c$. During one training iteration, all of required latent features can be find directly in the target LF partition, since the location information of those latent features are saved in corresponding LF partition. If an LF partition is assigned to multiple slaves, then extra local communication of the subsequent computing tasks among these slaves happens, i.e., scenario 3) in Fig. S2. If one or more LF partitions are assigned to one slave (i.e., scenario 1) and 2) of Fig. S2), no extra data communication incurs across different nodes during the subsequent computation tasks. Hence, the data communicational cost in DASGD is very small.

In summary, the data-aware partition and allocation strategies significantly reduce the data communication cost in the distributed computing environment. Hence, it effectively improves the scalability of a DASGD algorithm.

4 Experimental Evaluation

In this section, we first introduce the experiment setup, then evaluate the performance of the LFA model with our DASGD solver in terms of prediction accuracy, execution time, rank varying of P and Q, data volume, data communication cost and scalability on both real-world and synthetic datasets.

4.1 Experiment Setup

The experiments are conducted on a Spark cluster of 18 physical machines in the datacenter hosted at the Chongqing Institute of Green and Intelligent Technology, Chinese Academy of Sciences. Each physical machine is equipped with two Intel Xeon(R) E5-2680 2.4 GHz processors with 28 cores, 512GB RAM. The Spark cluster consists of one master node and a set of slave nodes connected by high-speed Gigabit network. Each node is allocated 4 vCPUs and 20GB RAM. The operation system running on each node is CentOS 7.3 with Linux core 3.10. Meanwhile, Hadoop version 2.6.3 and Spark version 1.6.2 are also installed, respectively. The prototype DASGD solver is written in Scala 2.10.

Two series of experiments are conducted, one on real-world datasets and the other on a synthetic dataset. The real-world datasets are collected from three well-known industrial recommender systems, i.e., Douban [1], MoiveLens[2] and Netflix[3]. Furthermore, we also generate several very large-scale and high-dimensional matrices with varying number of rows (i.e., m) and columns (i.e., n). In

[1] https://www.cse.cuhk.edu.hk/irwin.king.new/pub/data/douban
[2] https://grouplens.org/datasets/movielens/
[3] https://www.netflixprize.com/

the synthetic matrices, the known entries are random integer numbers uniformly selected within the range of [1, 5]. The details of the datasets are described in Table II.

TABLE II
DATASET IN DETAILS

Dataset	#Users	#Items	#Ratings	Sparsity (%)
Douban	129,490	58,541	16,830,839	0.54%
MovieLens	138,493	26,744	20,000,263	0.22%
Netflix	480,189	98,212	100,480,507	0.21%
Syn-*m*-*n*	*m*	*n*	0.02**m***n*	0.2%

Please note that on all datasets, we adopt 5-fold cross-validations. The evaluation is performed as follows: (1) each dataset is randomly split into five disjoint subsets, each containing 20% of the data; (2) four subsets are selected as a training set and the remaining one as a testing set; (3) each tested model is built on the training set and its performance is evaluated on the testing set; (4) steps 2 and 3 are sequentially repeated for five times to ensure that each subset is used as the testing set once and only once; and (5) the results obtained from the five runs of experiments are averaged to produce the final results.

The Root Mean Squared Error (RMSE) is calculated to evaluate the accuracy of the LFA model with different solvers:

$$RMSE = \sqrt{\frac{\sum_{u,i \in T} (r_{u,i} - \hat{r}_{u,i})^2}{|\Gamma|}} \tag{14}$$

where Γ denotes the size of the testing data. In order to evaluate the parallelization performance, the speedup is calculated:

$$S(N) = T(1)/T(N) \tag{15}$$

where $T(N)$ is the execution time taken by the solver running on N slave nodes, and $T(1)$ is the execution time taken by the solver running on a single node.

To perform a fair comparison, the grid search method is used to find the optimal parameters for each solver. The rank is set to 50. On all training sets, we also adopt the 5-fold cross validation setting for grid search. In addition, to find the optimal parameters for each model, i.e., regularization parameter λ and learning rate γ, the value of λ and γ are [2^{-1}, 2^{-2}, 2^{-3}, 2^{-4}, 2^{-5}, 2^{-6}, 2^{-7}, 2^{-8}, 2^{-9}].

To demonstrate the performance efficiency of DASGD, we compare it with three commonly adopted LFA solvers. First of all, a serial SGD (SSGD) algorithm is chosen as the baseline. Then a DSGD algorithm [54] and MLlib-ALS algorithm [58] are compared with the proposed DASGD algorithm. Note that there are two state-of-the-art parallel SGD solvers are excluded in the experiments, i.e., a Hogwild! solver [49] and an ASGD solver [50]. The reason why we did not compare them with the proposed DASGD model is that both of them are based on the memory-sharing framework, and cannot work in a distributed computational environment.

All compared algorithms are written in the scala 2.10 program. For SSGD, it is implemented on a single computational node with the configuration of 8VCPU and 20GB RAM. DSGD, MLlib-ALS and DASGD are deployed on the same Spark cluster. Note that for fully utilizing the power of cluster resources, the raw rating dataset is assigned to different slave nodes in the Spark platform. Hence, the rating dataset cannot be loaded into the memory as a whole in the Spark platform. The adopted cluster contains one master node and 12 slave nodes.

5.2 Running Time vs. RMSE

We firstly analysis the prediction accuracy of DASGD, DSGD, SSGD and MLlib-ALS on three industrial datasets. The experimental results are shown in Fig. S4 and Fig. S5, respectively.

Fig. S4 shows the running time taken by each solver on the three real-world datasets. Clearly, the running times of the LFA with distributed solvers are less than that of the SSGD solver. Taking MovieLens for example, DSGD is about 4.2x faster than SSGD in each iteration. For DASGD it is 58x faster than SSGD, and MLlib-ALS is 46x faster than SSGD on average. For different distributed solvers, the running times vary from dataset to dataset. On NetFlix, the updating rules of DASGD and DSGD are based on a set of independent blocks. However, DASGD is approximately 64x faster than DSGD in each iteration on average. The same is observed on Douban and MovieLens. The main reasons for this phenomenon are twofold. (1) DSGD requires the synchronization of P and Q after each stratum during one iteration. As a result, it leads to extra data communication cost on nodes. DASGD can effectively reduce the fetching time of P and Q with DASGD. The dispatched subset of P and Q resides consistently on nodes, until one iteration is completed. (2) The workload of DSGD on different nodes may be unbalanced, because of the non-even distribution of the rating matrix. However, the data-aware partition scheme of DASGD can effectively address the workload balance issue. Thus, we can conclude that, compared with DSGD, DASGD achieves higher computing efficiency in each iteration.

Fig. S5 compares the running time versus the RMSE obtained by LFA with the three distributed solvers and the sequential SGD. As shown, the performance of DASGD is better than the DSGD and MLlib-ALS on all three datasets in general. First, the prediction accuracy of DASGD is close to the serial version of the SGD solver on all three datasets. Taking Douban for example, DASGD achieves an RMSE of 0.715, versus SSGD's 0.714. On NetFlix, the RMSE achieved by DASGD is 0.829, a bit higher SGD's 0.827. Second, the prediction accuracy of DASGD is higher than that of DSGD by 6.5% on average, and 7.2% higher in the best case on Netflix. In general, DASGD converges faster than DSGD. Therefore, it demonstrates that an LFA model with a DASGD solver can achieve a higher prediction accuracy than DSGD. The main reason is that DASGD iterates twice on each rating record in each training iteration, compared with DSGD which only iterates once on each rating record. We also see that the accuracy of SSGD is slightly higher than that of DASGD. The main reason is their different orders of updating different parameters. DASGD updates each element of P and Q by using all ratings about the target user or item continuously, while the SSGD's process for updating P and Q are random. On the other hand, DASGD

and MLlib-ALS achieve almost the same prediction accuracy at the same running time. Hence, it demonstrates that these two solvers perform well with different rate of training time.

Fig. S6 plots the mean and standard deviation of LFA with three different solvers in terms of RMSE. Compared to DSGD, the DASGD solver obtains a higher prediction accuracy. Hence, it demonstrates that DAGSD will not cause a negative effect on prediction accuracy. On the other hand, all the three distributed solvers' prediction accuracies fluctuate slightly. The corresponding variant range of mean RMSE is less than 0.001. This shows the stability of DSGD, DASGD and MLlib-ALS. It demonstrates that all the methods have solved the over-writing problem effectively. For decoupling the training process of LFA, the DSGD adopts the blocking method with the interchangeable theorem, while the DASGD and MLlib-ALS choose the alternating updating method.

5.3 Running Time vs. Rank

We then measure how DASGD performs under different settings of the rank f. Fig. S7 shows the time taken in each iteration on average for different solvers with a varying size of rank f. It can be seen that the times taken by all the solvers increase as f increases, and the running times of MLlib-ALS and DSGD increase much faster than that of DASGD. Taking dataset Douban for example, MLlib-ALS takes 283 seconds to finish one iteration when f increases to 400, a significant increase from 16 seconds when f = 50. The difference is approximately 17 times. The same phenomena can be observed on the MovieLens and Netflix datasets. In summary, DASGD is 5x faster than MLlib-ALS on average when f = 400. The main reason is the difference in their computational complexity. The computational complexity of ALS is proportional to the $O(f^3)$, when the size of the rating matrix is constant. For the SGD-based solvers, their computational complexity is only proportional to $O(f)$.

Thus, MLlib-ALS's running time increases dynamically when f increases from 50 to 400. Furthermore, the increase in DASGD's running time is less than 4 times when the rank increases from 50 to 400, while DSGD's increase ratio is only less than 2 times. The main reason is that, for the communication-intensive distributed solver DSGD, the increase of f mainly affects the execution time on each node/core, while it has little impact on the data transmission cost. Compared to DSGD, DASGD is computationally intensive because it iterates each rating record twice in one iteration, while the DSGD only calculates each rating record once. However, the computational efficiency of DASGD is much higher than DSDG in the same environment.

5.4 Running Time vs. Data Sizes

We also measure how DASGD performs with matrix R of different sizes. In the experiments, we generate a series of synthetic datasets with different numbers of items n (i.e., columns) with a fixed number of users m (i.e., rows). Fig. S8 shows the average per-iteration running times of DSGD, DASGD and MLlib-ALS on datasets of different sizes. We observe that the times taken by all three solvers increase as the dataset size increases. The increase in DSGD's running time is the most significant. When n = 100k, DASGD is 38x faster than DSGD, and 1.3x faster than MLlib-ALS. The main reasons are that for the communication-intensive DSGD, the data transmission cost incurred between different nodes significantly increases when R increases.

5.5 Communication Cost of DASGD and DSGD

We conduct a set of experiments to analysis the data communication costs of DSGD and DASGD. In one iteration of the training process, the shuffle read size of a slave node represents the volume of data obtained from other slave nodes via the network transmission. Fig. S9 is the comparison results of shuffle read size on different datasets. From Fig. S9, it shows that the shuffle read sizes of DASGD are less than that of DSGD in all three datasets. For DSGD, it proposes the stratum mechanism to update the latent features for utilizing the power of a cluster of slave nodes, but the strong data synchronization for latent features after each stratum makes the computation tasks have to frequent access data across different slave nodes. As a result, the shuffle read size of DSGD increases from 3.45GB to 34.7GB, when the size of the dataset changes from 270MB (Douban) to 1.86GB (NetFlix). Compared with DSGD, the shuffle read size of DASGD increases from 1.54GB to 8.74GB with the dataset varying. Taking dataset NetFlix for example, the data communication size of DSGD is almost 19x larger than the raw rating dataset, while it is only 5x larger than the raw rating dataset for DASGD. Hence, it demonstrates that our proposed optimization methods of data partition and allocation can significantly reduce the data communication across different slaves in the distributed environment.

Fig. S10 shows the results of the time taken by DSGD and DASGD on different datasets. In detail, the setup time mainly represents the time spends on loading the raw rating data into Spark cluster, while the execution time is the time spends on performing the training process with including the data communication. From Fig. S10, it is clear that our DASGD has a superiority execution efficiency over the DSGD on all three datasets. The inner reason is that the stratum mechanism of DSGD creates more data communication operations across slave nodes, which leads to more execution time in one training iteration.

Furthermore, the comparison results of data communication costs with varying slave nodes are presented in Fig. S11. From Fig. S11, it is clear that the data communication costs of DASGD are less than that of DSGD in all cases. Most importantly, the shuffle read size of DSGD grows linearly with an increasing number of employed slave nodes. For DASGD, the proposed data-aware partition method and the data allocation strategy make the most of computation tasks access data from the local slave node, reducing the frequency of data communication across different nodes. As a result, the data communication costs of DASGD only increases from 1.42GB to 8.74GB with increasing the slaves from 2 to 12, while the data communication costs of DSGD increases from 5.8GB to 34.7GB in the same cases. Therefore, DASGD can effectively minimize

the data communication cost of training process in a distributed environment, and the expansion of cluster scale does not lead to an obvious increase in data communication cost. It demonstrates that our DASGD enjoys a notable and superiority advantage over the DSGD in terms of computational efficiency and scalability.

5.6 Scalability

We also evaluate the performance of LFA with the DASGD solver running on the Spark cluster with numbers of slave nodes. In the experiments, we use dataset Syn-1M-50K, which includes 1 million rows, 50 thousand columns, and 1 billion nonzero elements. Each solver is tested on different numbers of slave nodes, ranging from 6 to 60. Fig. S12 demonstrates the results.

As shown in Fig. S12, the running times of the three solvers gradually decrease with the increase in the number of slave nodes in the Spark cluster. When the number of slave nodes increases from 6 to 12, the average running time of DASGD decreases from 900 to 477 seconds. Furthermore, the running time of DASGD drops relatively slowly from 209 to 144 when the number of slave nodes increases from 36 to 60. That is because the number of the cores in the Spark cluster is larger than the number of the blocks of the LF subsets. Because each LF subset is allocated to multiple slave nodes, it takes more time to transmit data among these slave nodes. The same phenomena are observed for DSGD and MLlib-ALS.

Fig. S13 shows the acceleration achieved by the three solvers. Since the large-scale dataset cannot be loaded into the memory of a single node, the speedup achieved by each solver using N slave nodes can be defined as $S(N) = T(N)/T(6)$, where $T(N)$ is the running time of DASGD on N slave nodes. In Fig. S10, the improvement on the efficiency of DASGD via parallelization is significant. Specifically, with 12 slave nodes, the speedup achieved by DASGD reaches 1.89, very close to the theoretical maximum value of 2.00. The speedup achieved by DASGD is within the range of [4.9, 6.1] as the number of slave nodes increases from 36 to 60. This is caused by the time taken by the non-parallel parts of the algorithm, e.g., the data transmission, application submission and configuration, etc. In summary, we can clearly see that DASGD significantly outperforms both DSGD and MLlib-ALS in terms of speedup. Therefore, it demonstrates that DASGD is not only fast without accuracy loss, but also scalable to very large datasets.

6 Conclusions

In this paper, we proposed an LFA recommender system with a distributed SGD solver named DASGD for large-scale recommendations, aiming to accelerate the training process. To achieve this objective, we firstly analyzed the parameter training process of LFA, and pointed out that the main reason for disabling the parallelization of LFA is that the user latent features and the item latent features are dependent on the temporal state of each other during each training epoch. Based on the analytical results, we adopted the DASGD solver with user/item wise updating methods to train the latent features, thus it can update user/item latent feature simultaneously by proposed parallelization algorithm. Furthermore, to handle large-scale datasets, we implemented the DASDG solver on the Spark, which provided a distributed platform for building the LFA recommender system.

To accelerate the training process of LFA with DASGD on Spark, a hybrid optimization of DASGD that combines data parallelization and task parallelization optimization is proposed. Taking advantage of user/item wise updating methods, the training dataset is split into two ways, vertically and horizontally, and then allocated to a Spark cluster by proposed static data allocation Strategy. As a result, the data transmission cost is significantly reduced. Benefitting from the rich operations offered by Spark, a task parallel approach is proposed by building the task DAG. The results of experiments conducted on the industrial cloud environment with both real-world and synthetic datasets indicate the superiority and notable advantages of LFA with the DASGD solver over the state-of-the-art methods, including MLlib-ALS and DSGD, in terms of both prediction accuracy and efficiency. When handling very large datasets, DASGD also demonstrates higher scalability compared with DSGD and MLlib-ALS.

In the experiments, we create a synthetic matrix with the data generating method as mentioned in [61], whose feasibility has also been proven in [51, 54]. In our future work, we will investigate how a synthetic data generation method will impact the performance of an LFA model. This will allow us to obtain guidance for model structure design.

Acknowledgment

This work is supported in part by the National Natural Science Foundation of China under grants 61602434, 61772493, 91646114 and 61702475, in part by the Natural Science Foundation of Chongqing (China) under grant cstc2019jcyjjqX0013, in part by the Chongqing Research Program of Technology Innovation and Application under grants cstc2019jscx-zdztzxX0019, cstc2018jszx-cyztzxX0025, Chongqing research program of key standard technologies innovation of key industries under grant cstc2017zdcy-zdyfX0076, in part by Youth Innovation Promotion Association CAS No.2017393, and in part by the Pioneer Hundred Talents Program of Chinese Academy of Sciences.

References

[1] J. Gubbi, R. Buyya, S. Marusic, M. Palaniswami,"Internet of Things (IoT): A vision, architectural elements, and future directions," *Future Generation Computer Systems*, vol. 29, no. 7, pp. 1645-1660, Dec. 2013.

[2] M. Armbrust, A. Fox, R. Griffith, A.D. Joseph, R. Katz, A. konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, M. Zaharia, "A view of cloud computing," *Communcation of the ACM*, vol. 53, no. 4, pp. 50-58, April. 2010.

[3] Y. Jiang, Z. Huang, D. H. K. Tsang," Towards Max-Min Fair Resource Allocation for Stream Big Data Analytics in Shared Clouds," *IEEE Trans on Big Data*, vol. 4, no. 1, pp. 130-137, 2018.

[4] S. Tuarob, S. Bhatia, P. Mitra, C. L. Giles," AlgorithmSeer: A System for Extracting and Searching for Algorithms in Scholarly Big Data," *IEEE Trans on Big Data*, vol. 2, no. 1, pp. 3-17, Mar. 2016.

[5] X. Wu, X. Zhu, G.-Q. Wu, and W. Ding, "Data mining with big data," *IEEE Trans on Knowledge and Data Engineering*, vol. 26, no. 1, pp. 97-107, 2014.

[6] L. Qi, X. Xu, X . Zhang, W. Dou, C. Hu, Y. Zhou, J. Yu," Structural Balance Theory-Based E-Commerce Recommendation over Big Rating Data," *IEEE Trans on Big Data*, vol. 4, no. 3, pp. 301-312, Sept. 2018.

[7] S. Jiang, X. Qian, T. Mei, Y. Fu, "Personalized Travel Sequence Recommendation on Multi-Source Big Social Media," *IEEE Trans on Big Data*, vol. 2, no. 1, pp. 43-56, Mar. 2016.

[8] P. Resnick and H. R. Varian, "Recommender systems," *Communications of the ACM,* vol. 40, no. 3, pp. 56-58, Mar. 1997.

[9] G. Adomavicius and A. Tuzhilin, "Toward the next generation of recommender systems: a survey of the state-of-the-art and possible extensions," *IEEE Trans. on Knowledge and Data Engineering*, vol. 17, pp. 734-749, Jun. 2005.

[10] J. B. Schafer, J. A. Konstan, and J. Riedl, "E-Commerce recommendation applications," *Data Mining and Knowledge Discovery*, vol. 5, pp. 115-153, Jan. 2001.

[11] G. Linden, B. Smith, and J. York, "Amazon.com recommendations: Item-to-item Collaborative Filtering," *IEEE Internet Computing*, vol. 7, no. 1, pp. 76-80, Jan. 2003.

[12] J. Davidson, B. Liebald, J. Liu, P. Nandy, T. V. Vleet, U. Gargi, S. Gupta, Y. He, M. Lambert, B. Livingston, and D. Sampath, "The YouTube video recommendation system," In *Proceedings of the fourth ACM conference on Recommender systems*, Barcelona, Spain, Sept. 2010, pp. 293-296.

[13] N. Koenigstein, G. Dror, Y. Kroen, "Yahoo! Music recommendations: modeling music ratings with temporal dynamical and item taxonomy," In *Proceedings of the 5th ACM conference on Recommender systems*, Chicago, Illinois, USA, Oct. 2011, pp.165-172.

[14] Y. Cai, H.-f. Leung, Q. Li, H. Min, J. Tang, and J. Li, "Typicality-based collaborative filtering recommendation," *IEEE Trans. on Knowledge and Data Engineering*, vol. 26, no. 3, pp. 766-779, Mar. 2014.

[15] M-S. Shang, Z. Zhang, T. Zhou, and Y-C. Zhang, "Collaborative filtering with diffusion-based similarity on tripartite graphs," *Physica A: Statistical Mechanics and its Applications*, vol. 389, no. 6, pp. 1259-1264, Jan. 2010.

[16] H. Li, R. Hong, S. Zhu, Y. Ge, "Point-of-Interest recommender systems: a separate-space perspective," In *Proceedings of 15th IEEE International Conference on Data Mining*, Atlantic City, NJ, USA, Nov. 2015, pp. 231-240.

[17] X. Luo, M.Shang, S. Li, "Efficient extraction of non-negative latent factors from high-dimensional and sparse matrices in industrial applications," In *Processings of 16th IEEE International Conference on Data Mining*, Barcelona, Spain, Dec. 2016, pp. 311-319.

[18] Q. Gu, J. Zhou, and C. Ding, " Collaborative - filtering: weighted nonnegative matrix - factorization incorporating user and item graphs," In *Proceedings of the SIAM International Conference on Data Mining*, Columbus, OH, Apr. 2010, pp. 199 - 210.

[19] W. Luan, G. Liu, C. Jiang, and L.Qi, "Partition-based collaborative tensor factorization for POI recommendation," *IEEE/CAA Journal of Automatica Sinica*, vol. 4, no. 3, pp. 437-446, 2017.

[20] X. Luo, M.-C. Zhou, S. Li, Z.-H. You, Y. –N. Xia and Q.-S. Zhu,"A Nonegative Latent Factor Model for Large-Scale Sparse Matrices in Recommender Systems via Alternating Direction Method," *IEEE Trans. on Neural Networks and Learning Systems*, vol. 27, pp. 524-537, 2016.

[21] M. Shang, X. Luo, Z. Liu, J. Chen, Y. Yuan, and M. Zhou,"Randomized Latent Factor Model for High-dimensional and Sparse Matrices from Industrial Applications," *IEEE/CAA Journal of Automatica Sinica*, DOI:10.1109/JAS.2018.7511189.

[22] Y. Cong, J, Liu, B. Fan, P. Zeng, H. Yu, J. Luo," Online Similarity Learning for Big Data with Overfitting," *IEEE Trans on Big Data*, vol. 4, no. 1, pp. 78-89, Mar. 2018.

[23] X. Luo, M. Zhou, Y. Xia, and Q.Zhu, "An Incremental-and-Static-combined scheme for matrix-factorization-based collaborative filtering," *IEEE Trans. on Automation Science and Engineering*, vol. 13, no. 1, pp. 333-343, Jan. 2016.

[24] Z. Gharamani, "Probabbilistic machine learning and artificaial intelligence," *Nature*, vol. 521, no. 7553, pp. 452-459, 2015.

[25] M. Hofree, J. –P. Shen, H.Carter, A. Gross, and T. Ideker, "Network-based stratification of tumor mutations," *Nature Methods*, vol.10, no. 11, pp. 1108-1115, 2013.

[26] Y. Li, B. Cao, L. Xu, J. Yin, S. Deng, Y. Yin, and Z. Wu "An Efficient Recommendation Method for Improving Bussiness Process Modeling," *IEEE Transactions on Industrial Informatics*, vol. 10, no. 1, pp. 502-513, 2014.

[27] Y. Koren, R. Bell, C. Volinsky, "Matrix factorization techniques for Recommender Systems", *IEEE Journals & Magazines*, vol. 42, no. 8, pp. 30-37, 2009.

[28] A. Paterek, "Improving regularized singular value decomposition for collaborative filtering," In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, California, USA, ACM, 2007, pp.5-8.

[29] R. R. Salakhutdinov and A. Mnih, "Probabilistic matrix factorization," In *Proceedings of the 20th Interational Conference on Neural Information Processing Systems*, Vancouver, British Columbia, Canada, Dec.2011, pp. 1–8.

[30] X. Luo, M. Zhou, Y. Xia and Q. Zhu, "An efficient non-negative matrix-factorization-based approach to collaborative filtering for recommender systems," *IEEE Trans. on Industrial Informatics*, vol.10, no.2, pp.1273-1284, May. 2014.

[31] Y. Chen, H. Zhang, J. Wu, X. Wang, R. Liu, M. Lin, "Modeling emerging, evolving and fading Topics Using Dynamic Soft Orthogonal NMF with Sparse Representation," In *Proceedings of 15th IEEE Interational Conference on Data Mining*, Atlantic City, NJ, USA, 2015, pp.61-70.

[32] L. –X. Li, L. Wu, H.-S. Zhang, and F.-X. Wu,"A Fast Algorithm for Non-Negative Matrix-Factroization-Based Approach to Collaborative Filtering for Recommender Systems," *IEEE Trans. On Neural Network Learning System*, vol. 25, no. 10, pp. 1855-1863, Jan. 2014.

[33] G. Takács, I. Pilászy, B. Németh, and D. Tikk, "Scalable collaborative filtering approaches for large recommender systems," *Journal of Machine Learning Research*, vol. 10, pp. 623-656, Mar. 2009.

[34] T. George and S. Merugu. "A scalable collaborative filtering framework based on co-clustering," In Proceedings of the 5th IEEE International Conference on Data Mining, Houston, TX, USA, Nov. 2005, pp. 625-628.

[35] A. Narang, R. Gupta, A. Joshi, V.K. Garg. "Highly scalable parallel collaborative filtering algorithm," In *Proceedings of 2010 International Conference on High Performance Computing*, Dona Paula, India, Dec. 2010, pp.1-10.

[36] M. Papagelis, I. Rousidis, D. Plexousakis, and E. Theoharopoulos.

"Incremental collaborative filtering for highly-scalable recommendation algorithms," *Foundations of Intelligent Systems*, vol. 3488, pp. 553–561, 2005.

[37] X. Luo, Y. Xia, Q. Zhu, "Boosting the K-Nearest-Neighborhood based incremental collaborative filtering," *Knowledge-Based System*, vol.53, pp.90-99, 2013.

[38] D. -D. Zhao, M. -S. Shang," User-based collaborative-filtering recommendation algorithms on Hadoop," In *Proceedings of 3th International Conference on Knowledge Discovery and Data Mining*, Phuket, Thailand, 2010, pp. 478-481.

[39] C. Li, K. He, "CBMR: An optimized MapReduce for Item-Based CF recommendation with empirical analysis," *Concurrency and Computation Practice and Experience*, vol. 29, no. 10, pp: e4092, 2017.

[40] Y. Huang, B. Cui, W. Zhang, J. Jiang, Y. Xu, "TencentRec: Real-time stream recommendation in patrice," In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, MelBourne, Victoria, Australia, May. 2015, pp.227-238.

[41] D. Srivatsava, M.M. Nena, W. Matt, G. Joydeep, "Pervasive parallelism in data mining: dataflow solution to co-clustering large and sparse netflix data," In *Processings of 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. Paris, France, 2009, pp.1115-1144.

[42] Y. Zhou, D. Wilkinson, R. Schreiber, R. Pan," Large-scale parallel collaborative filtering for the Netflix prize", In *Proceedings of the 4th International Conference on Algorithmic Aspects in Information and Management*, Berlin, Heidelberg, 2008, pp.337-348.

[43] S. Schelter, C. Boden, M. Schenck, A. Alexandrov, V. Markl, " Distributed Matrix Factorization with MapReduce using a series of Broadcast-Joins", In *Proceddings of the 7th ACM Conference on Recommender Systems*, Hong Kong, China, 2013, pp. 281-284.

[44] L. Bottou," Large-scale machine learning with stochastic gradient descent," In *Proceedings of 19th International Conference on Computational Statistics*, Paris France, 2010, pp. 177-186.

[45] O. Dekel, R.-G. Bacjracj, O. Shamir, L. Xiao, "Optimal Distributed Online Prediction Using Mini-Bathes", *Journal of Machine Learning Research*, vol. 13, pp.165-202, 2012.

[46] J. Langford, A. Smola, and M. Zinkevich, "Slow learners are fast," In *Proceedings of the 22th Adavnces in Neural Information Processing Systems*, Vancouver, British Columbia, Canada, 2009, pp.2331-2339.

[47] A. Agarwal and J. C. Duchi, "Distributed delayed stochastic optimization", In *Proceedings of the 24th Adavnces in Neural Information Processing Systems*, Granada, Spain, 2011, pp. 873-881.

[48] M. Zinkevich, M. Weimer, A. Smola, and L. Li, " Parallelized stochastic gradient descent," In *Proceedings of 23th Adavances in Neural Information Processing Systems*, Vancouver, Canada , Dec.2010, pp. 2595-2603.

[49] F. Niu, B. Recht, C. Re, and S. J. Wright, "HogWild!: A lock-free approach to parallelizing stochastic gradient descent, " In *Proceedings of 24th Adavances in Neural Information Processing Systems* , Granada, Spain, 2011, pp. 693-701.

[50] X. Luo, H. Liu, G. Gou, Y. Xia, Q. Zhu, "A parallel matrix factorization based recommender by alternating stochastic gradient decent", *Engineering Applications of Artificial Intelligence*, vol. 25, pp: 1403-1412, 2012.

[51] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismaanis, "Large-scale matrix factorization with distributed stochastic gradient descent," In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge discovery and data mining*, San Diego, California, USA, Aug. 2011, pp. 69-77.

[52] C. Telioudi, F. Makari, and R. Gemulla, " Distributed matrix completion," In *Proceedings of 12th Interantional Conference on Data Mining*, Brussels, Belgium, 2012, pp. 655-664.

[53] Y. Huang, B. Cui, J. Jiang, K. Hong, W. Zhang, and Y. Xie, "Real-time video recommendation exploration," In *Proceedings of the 2016 International Conference on Management of Data*, San Francisco, CA, USA, 2016, pp. 35-46.

[54] H-F. Yu, C-J. Hsieh, S. SI, I. Dhillon, "Scalable Coordinate Descent Approaches to Parallel Matrix Factorization for Recommender Systems," In *Proceedings of 12th International Conference on Data Mining*, Brussels, Belgium, Dec. 2012, pp. 765-774.

[55] J. Yin, L. Gao, Z. Zhang, "Scalable distributed nonnegative matrix factorization with block-wise updates", *IEEE Trans. on knowledge and data engineering*, vol. 30, no. 6, pp.1136-1149, 2018.

[56] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, SAN JOSE, CA, 2012, pp. 15–28.

[57] W. Shi, Y. Zhu, P.S. Yu, J. Zhang, T. Huang, C. Wang, Y. Chen," Effective Prediction of Missing Data on Apache Spark over Multivariable Time Series," *IEEE Trans on Big Data*, vol. 4, no. 4, pp. 473-486, Dec. 2018.

[58] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tasi, M. Made, S. Owen, D. Xin. R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, A. Talwalkar, "MLlib: machine learning in apache spark", *Journal of Machine Learning Research*, vol. 17, no. 1, pp. 1235-1241, 2016.

[59] W.-S. Chin, Y. Zhuang, Y.-C. Juan, and C.-J. Lin, "A fast parallel stochastic gradient method for matrix factorization in shared memory systems," *ACM Transactions on Intelligent Systems and Technology*, vol. 6, no. 1, pp. 2, 2015.

[60] H. Yun, H.-F. Yu, C.-J. Hsieh, S. Vishwanathan, and I. Dhillon, "Nomad: Non-locking, stochastic multi-machine algorithm for asynchronous and decentralized matrix completion," *Proceedings of the VLDB Endowment*, vol. 7, no. 11, pp. 975–986, 2014.

[61] H. Li, K. Li, J.An, and K. Li,"MSGD: A novel matrix factorization approach for large-scale collaborative filtering recommender systems on GPUs", *IEEE transaction on Parallel and Distributed Computing*, Vol. 29, no. 7, pp: 1530-1544, 2018.

Xiaoyu Shi (M'19) received the B.S. degree in computer science from the PLA Information Engineering University, Zhengzhou, China, in 2007, and the Ph.D. degree in computer science from the University of Electronic Science and Technology of China, Chengdu, China, in 2015. He joined the Chongqing Institute of Green and Intelligent Technology, Chinese Academy of Sciences, Chongqing, China in 2015, as a associate professor of computer science and engineering. His research interests include recommender system, cloud computing, artificial intelligence and big data applications.

Qiang He received his first PhD degree from Swinburne University of Technology, Australia, in 2009 and his second PhD degree in computer science and engineering from Huazhong University of Science and Technology, China, in 2010. He is a senior lecturer at Swinburne. His research interests include service computing, software engineering, cloud computing and edge computing. More details about his research can be found at https://sites.google.com/site/heqiang/.

Xin Luo (M'14–SM'17) received the B.S. degree in computer science from the University of Electronic Science and Technology of China, Chengdu, China, in 2005 and the Ph.D. degree in computer science from Beihang University, Beijing, China, in 2011. In 2016, he joined the Chongqing Institute of Green and Intelligent Technology, Chinese Academy of Sciences, Chongqing, China, as a Professor of computer science and engineering. He is currently also a Distinguished Professor of computer science with the Dongguan University of Technology, Dongguan, China. His current research interests include big data analysis and intelligent control. He has published over 100 papers (including over 40 IEEE TRANSACTIONS papers) in the above areas. Dr. Luo was a recipient of the Hong Kong Scholar Program jointly by the Society of Hong Kong Scholars and China Post-Doctoral Science Foundation in 2014, the Pioneer Hundred Talents Program of Chinese Academy of Sciences in 2016, and the Advanced Support of the Pioneer Hundred Talents Program of Chinese Academy of Sciences in 2018. He is currently serving as an Associate Editor for the IEEE/CAA JOURNAL OF AUTOMATICA SINICA, IEEE ACCESS, and *Neurocomputing*. He has received the Outstanding Associate Editor reward of IEEE ACCESS in 2018. He has also served as the Program Committee Member for over 20 international conferences.

Yanan Bai received the M.S. degree in computer science from the Guizhou University, Guizhou, China, in 2010. She joined the University of Pingdingshan, Henan, China, in 2010. She is currently a Ph.D. candidate in computer science from the University of Chinese Academy of Sciences, Chongqing, China, in 2017. Her research interests are in information safety and artificial intelligence.

Mingsheng Shang received his Ph.D Degree in Computer Science from University of Electronic Science and Technology of China (UESTC). He joined the Chongqing Institute of Green and Intelligent Technology, Chinese Academy of Sciences, Chongqing, China, in 2015, and is currently a Professor of computer science and engineering. His research interests include data mining, complex networks and cloud computing.