Speeding up Distributed Low-rank Matrix Factorization

Chengjie Qin
EECS, School of Engineering
University of California, Merced
5200 N Lake Road
Merced, CA, USA 95343
cqin3@ucmerced.edu

Florin Rusu
EECS, School of Engineering
University of California, Merced
5200 N Lake Road
Merced, CA, USA 95343
frusu@ucmerced.edu

***Abstract*—Distributed solution for solving low-rank matrix factorization (LMF), an important problem in recommendation system, has recently been studied a lot in order to better deal with the exploding data under the context of Big Data. Stochastic gradient descent is a general technique to solve a large class of convex optimization problems and it is often been chosen to solve problems that deals with large data sets in particular. In this work, we summarize the existing distributed solutions of LMF problem using stochastic gradient descent. We then proposed a novel distributed solution for LMF problem and our solution is able to achieve the best convergence rate as well as fastest execution time when compared with existing solutions. When deployed in the cloud, our solution has the potential to dramatically reduce the cost of complex analytics over massive datasets.**

I. Introduction

Matrix Completion has been widely used in Recommender Systems [11]. In essential, matrix completion tries to infer the whole original matrix while only given a partially observed matrix. Low-rank Matrix Factorization (LMF) is a technique to solve matrix completion by trying to represent the observed matrix as a product of two low-rank matrices with the minimum error. Then, in return, when the two low-rank matrices are multiplied together, the unobserved entries of the original matrix are approximated by the product of corresponding row and column of the two low-rank matrices. The resulting matrix of the product is taken as an inference of the original matrix.

In the age of Big Data, the amount of data that matrix completion tasks are dealing with has exploded. Large network companies, such as Netflix [3] and Yahoo! [8], have data that contains user ratings of more than 100 millions and even billions respectively. Netflix is also running its entire service on Amazon EC2, a cloud service of Amazon, where the billing is based on usage. Considering such a scale of the LMF problem it needs to deal with and the pay-as-you-go billing policy in the cloud, efficient distributed solver of LMF problem is eagerly needed.

Researchers have done quite a few work in running LMF tasks in parallel [12], [9], [14], [18], [10]. `Stochastic Gradient Descent (SGD)`, as a general optimization algorithm to solve LMF problems, is given great favor over other algorithms by all these work due to several nice properties of it. First, SGD requires very cheap steps in updating the parameters so that the execution time of SGD scales linearly with data size. This is extremely useful when the amount of data to be processed is huge. Other optimization algorithms which requires expensive (in computation) updates fail to have this linear scalability. Second, SGD only requires one or a few data points to perform an update. This property allows SGD to be distributed executed over multiple cores or multiple machines concurrently.

Current solutions for distributed LMF can be categorized into three categorise. The first line of works focus on running share-memory parallel SGD to solve LMF problems. [13] proves that when the given matrix is sparse, LMF can be ran parallel in a share-memory environment without any locks. Feng et al. [9] applies this method in to Bismarck, a framework for running machine learning tasks inside databases. The second line of works [20], [14] solve LMF in a share-nothing environment. In this case, the data is first partitioned into multiple nodes. Then multiple SGDs are ran separately on different partitions. In the end all the low-rank matrices from different partitions are merged together. [20] runs LMF parallel in a MapReduce setting. [14] further pushes the same idea to the single node level in a distributed parallel database, i.e. running multiple SGDs even on a single node. The third line of works [10], [12], [18] try to separate the data in a way that the model (two low-rank matrices) can also be also partitioned accordingly in order to reduce data transmission. In this case, each node only keeps a portion of the model and only concatenating is needed when models from different nodes together are merged together. Different from the former two method, the third method exploits the special characteristics of LMF task and is therefore not applicable to other problems.

There are, however, different limitations of current solutions. The first share-memory method needs the observation matrix to be sparse to work. Otherwise locks need to be applied in order to guarantee convergence. Applying locks fails to scale with the data size when the concurrent updating increases, because the possible contention will increase when every thread (or process) try to update the same parameter. Also, the first method requires the whole model (two low-rank matrices) to be fit in memory which sometimes is not the case. The second method avoid the updating contention problem by having duplicates of models for different partitions of data. Each updating thread will have its own copy of the model and the updating will be performed only on its own copy, so there is no updating contention. An overall model is got be averaging all the copies of the model. Using the second method, [14] is able to push the computation to share-nothing environment with linear speed-up. But the problem of the second method is that it compromises the convergence rate by having the merges. In other words, the more merges it has, the slower the overall problem converges. Moreover, the second method has the same problem as the first method in that the model it can run is limited by the memory size. Even wore, since the second method duplicates the model, it will suffer more than the first method when the model size grows. The third method overpasses the memory problem by partitioning both the data and the model at the same time. By taking advantage of the special structure of LMF problem, the third method is able to avoid the model averaging and only needs model concatenating in the end. However, single node parallelism was not exploited enough so that it suffers from slow execution compared with the former two methods.

In this work, we seek to find a solution that overcomes all limitations of current methods and has the fastest speed as well as preserving the best convergence rate. In other words, our goal is to come up with an solutions satisfying following conditions. First, it should be a distributed method that can be ran in a share-nothing cluster so that it does not limited by the number of cores as share-memory methods. Second, it should be able to partition the model instead of to be limited by the memory size to allow for running potentially large datasets. Third, it should be able to utilize all the resources it is given and push the execution speed to the maximum. Our specific contributions are:

- We summarize advantages and limitations of existing parallel LMF solutions in details.
- We propose a novel distributed solutions for solving large-scale LMF problems by taking a nice combination of existing solutions. We exam the technique details needed for implementing our solution and discuss the difference between other solutions.
- We run comparisons between existing solutions and our solution over a 1 million by 1 million matrix with 1.7 billion non-zero entries over a 9-node cluster. Our solution converges fastest within 100 seconds. Moreover, when the model size grows larger, our solution is 2∼4× faster than other solutions, i.e. our solution scales the best with larger tasks.

The implications of our solution has on big data analytics in the cloud are dramatic. Given the pay-as-you-go billing policy in the cloud, our solution uses considerably less time compared other which results in a lot of economical benefits.

The remainder of this paper is structured as follows. We explain the backgroud of low-rank matrix factorization problem and standard stochastic gradient descent method in Section II. In Section III, we summarize the existing distributed and parallel solutions for LMF problems and their limitations. In Section IV, we describe our solutions for distributed LMF problem. In Section V, we demonstrate the efficiency and scalability of our solutions and show that we achieve a 2.5× speed-up compared with the fastest distributed LMF solutions. The related work is discussed in Section VI and we conclude our work in Section VII.

$$\begin{array}{c} \\ Jay \\ Anna \\ John \end{array} \begin{array}{c} \begin{array}{ccc} Inception & StarWars & Avatar \end{array} \\ \left(\begin{array}{ccc} 1 & ? & 3 \\ ? & 4 & 5 \\ 3 & 5 & ? \end{array}\right) \end{array}$$

Fig. 1: A user-movie rating matrix

II. Preliminaries

In this section, we explain how Low-rank Matrix Factorization (LMF) is used in recommender systems and how stochastic gradient descent is used to solve LMF problem.

A. Low-rank Matrix Factorization Problem

We take "Netflix Problem" as an example of LMF tasks. Netflix is an on-line movie provider who wants to give recommendations of movies based on users' feedback (ratings of movies). The ratings of different movies from different users can be represented as a matrix as shown in Figure 1. Each row corresponds to one user and each column corresponds to one movie. Users rate the movies by numbers from 1 to 5. Apparently the rating matrix is not complete since only some of the users have rated some of the movies. The goal of a recommender system is to predict the '?' in rating matrix to guess what is the rating a user might give based on his rating of other movies and other people's rating of the same movie.

Formally, let us assume there are m users and n movies. We denote the $m \times n$ rating matrix as $\boldsymbol{V}$. The goal of LMF is, given a small rank r where $r \ll \text{rank}(\boldsymbol{V})$, to find an $m \times r$ matrix $\boldsymbol{L}$ and an $r \times n$ $\boldsymbol{R}$ matrix such that $\boldsymbol{V} \approx \boldsymbol{LR}$. In other words, we seek to approximate the original rating matrix $\boldsymbol{V}$ by the product of two low-rank matrices—$\boldsymbol{L}$ and $\boldsymbol{R}$. To evaluate how good the approximation is, we have the basic squared loss function f:

$$f(L, R) = \sum_{(i,j) \in Z} (\boldsymbol{V}_{ij} - \boldsymbol{L}_{i*}\boldsymbol{R}_{*j})^2 \tag{1}$$

where Z is all the non-zero entries in $\boldsymbol{V}$, i.e. the training set. There is usually a regularization term added to the loss function (1). The most common one is L_p regularization [17]. In this work, we only look at the basic loss function without any regularization term since any regularization can be applied to the loss function and the training process by stochastic gradient descent will be still the same.

Given the loss function (1), the goal of LMF is to find two matrices $\boldsymbol{L}^*$ and $\boldsymbol{R}^*$, such that:

$$(\boldsymbol{L}^*, \boldsymbol{R}^*) = \operatorname*{argmin}_{\boldsymbol{L},\boldsymbol{R}} f(\boldsymbol{L}, \boldsymbol{R}) \tag{2}$$

There are lots of methods available solve the above optimization problem. But when it comes to large data sets, stochastic gradient descent generally outperforms other methods in high runtime performance which results in fast convergence in real time.

B. Gradient Methods

Consider the following optimization problem with a decomposable objective function (i.e. summation):

$$\min_{w\in\mathbb{R}^d} \sum_{i=1}^{N} f\left(w, z_i\right) + P(w) \tag{3}$$

in which a d-dimensional vector $w \in \mathbb{R}^d$, $d \geq 1$ has to be found such that the objective function is minimized. The constants z_i, $1 \leq i \leq N$ correspond to tuples in a database table—materialized or obtained as the result of a query. Essentially, each term in the objective function corresponding to a tuple z_i can be viewed as a separate function $f_i(w) = f\left(w, z_i\right)$.

Gradient descent: Gradient descent is an iterative method to solve (3). The main idea is to start from an arbitrary vector $w^{(0)}$ and then to determine iteratively new vectors $w^{(k+1)}$ such that the objective function at each iteration decreases, i.e., $f(w^{(k+1)}) < f(w^{(k)})$ (we consider $P\left(w\right) = 0$ for simplicity). The way how gradient decent guarantees the decrease is by taking the opposite direction of current gradient which formally can be written as:

$$w^{(k+1)} = w^{(k)} - \alpha_k \nabla f\left(w^{(k)}\right) \tag{4}$$

where $\alpha_k \geq 0$ is the step size and $\nabla f(w^{(k)})$ is the current gradient of function f at $w^{(k)}$.

Stochastic Gradient Descent: Stochastic gradient descent (SGD) differs from standard gradient descent in that instead of computing the exact gradient $\nabla f(w^{(k)})$ at iteration k, SGD takes $\nabla f_i(w^{(k)})$, an approximation of $\nabla f(w^{(k)})$ base on one or a few $f_i(w^{(k)})$. So in SGD the updates of w is:

$$w^{(k+1)} = w^{(k)} - \alpha_k \nabla f_{\eta(k)}\left(w^{(k)}\right) \tag{5}$$

Taking steps based on the tuple-based approximation instead of the actual gradient is the characteristic property of SGD. Typically, $\alpha_k \to 0$ as $k \to \infty$ and $\eta(k) \in \{1, \ldots, N\}$ represents a random permutation, i.e., all f_i have to be considered before any term is repeated.

How can we determine when the method converges? And how do we know that the method converges to a global minimum of the objective function? The standard method to check for convergence is to compare the objective function for $w^{(k+1)}$ and $w^{(k)}$ obtained in consecutive iterations. If the difference is smaller than a given threshold, convergence can be assumed. An alternative is to fix the number of iterations ahead of time and take the last w as the optimal solution. Convergence to a global optimal solution is theoretically guaranteed when the functions $\sum_{i=1}^{N} f_i(w)$ are convex.

The advantage of SGD is that SGD's updates only need one or a few data points which is much cheaper than gradient descent which needs to go through the whole dataset to do one update. Though the updates of SGD is noisy, it converges much faster in time when the current position is far away from the optimal [4]. This is extremely helpful when dealing with huge data size since the update gets even more expensive (computationally) for gradient descent. Chances are that SGD already converges while standard gradient descent has not even finished one update.

The loss function of LMF satisfies the characteristics of objective function (3) where $w = (\boldsymbol{L}_{i*}, \boldsymbol{R}_{*j})$ and $z_i = \boldsymbol{V}_{ij}$. So the loss function (1) of LMF can be minimized by SGD. Particularly, when there is a regularization term for LMF, i.e. when $P\left(w\right) \neq 0$. SGD still proceeds in the same way by adding the gradient of regularization term $P'\left(w\right)$ to each update in (5). There are a lot ways to circumvent the cases when $P\left(w\right)$ is non-differentiable as well [4].

III. Parallel Stochastic Gradient Descent for Low-rank Matrix Factorization

Considering that SGD only requires one or a few tuples for each update, the model updating of SGD is potentially parallelizable by itself. Moreover, LMF problem contains certain structure that allows for even more parallelism. In this section, we explore the special structure of LMF problem and summarize the existing parallel solutions for LMF.

A. Share-memory LMF

According to equation (1), while updating each point $\boldsymbol{V}_{ij}$, only $\boldsymbol{L}_{i*}$ and $\boldsymbol{R}_{*j}$ get updated. Which means all the $\boldsymbol{L}_{i'*}$ $(i' \neq i)$ and $\boldsymbol{R}_{*j'}$ $(j \neq j')$ are not affected by the updating of $\boldsymbol{V}_{ij}$. Therefore, updating of $\boldsymbol{V}_{ij}$ and $\boldsymbol{V}_{kl}(i \neq k$ and $j \neq l)$ can be concurrently done.

One way to parallel LMF is to keep the model ($\boldsymbol{L}$ and $\boldsymbol{R}$) in the shared memory and have multiple threads updating the model at the same time [15]. Finer grain locks can be applied to specific rows and columns that are been updated, thus allows for parallel updating of the rows and columns as long as they do not overlap. Hogwild! [13] proves that when the matrix $\boldsymbol{V}$ is sparse, it does not even need to lock on rows and columns. Allowing contention can also provide good enough convergence rate in the sparse case.

The number of parallelism of share-memory updating, however, is limited by the number of cores that one node has. Even if there are more cores available, having more cores will increase the possibility of updating contention waiting which compromises the execution time. Moreover, share-memory LMF requires the model to be fit in memory, so it fails to support the case when the model size exceeds the memory size.

B. Distributed LMF

Distributed SGD is an alternative to overpass the updating contention problem of share-memory. We summarize two existing distributed solutions for LMF—*model averaging* and *model concatenating*.

1) ***Model Averaging:*** Model averaging [20] avoids the updating contentions by randomly partitioning the data into different nodes and running multiple SGDs distributed on each data partitions. The number of parallelism of model averaging equals to the number of data partitions, i.e. the number of nodes. When all the data partition finish updating, an overall model is generated by averaging all the models from different partitions. The averaging operation is a weighted averaging for each and every dimension of the model and the weight is the portion of data each partition contains. [14] pushes the parallelism of model averaging inside each node by further dividing each data partition into chunks and scheduling the chunks to multiple updating threads within one node. [14] achieves linear speed-up in terms of cores until hitting the IO-bound of the physical disks and therefore it is the state-of-the-art in terms of single iteration execution time.

Model averaging LMF solves the updating contention problem by keeping copies of models in memory. Considering model it even keeps multiple model within one node, it memory limitation of model averaging LMF is higher than share-memory LMF. Also, the averaging operation introduces variation of the model which hurts the convergence rate in terms of number of epochs needed to converge. In other words, model averaging LMF takes more epochs to converge to the same value as share-memory. But since model averaging can be significantly faster in epoch execution time, it is still achieves faster convergence rate in terms of real time.

*2) **Model Concatenating**:* Model concatenating [10] overpasses the averaging operation in model averaging by partitioning the data into *interchangeable sets* instead of random sets. We take the definition of *interchangeable set* from [10]:

Definition 1. *Two training points $z_1, z_2 \in Z$ (Z is training the set) are interchangeable with respect to a loss function f having summation form (3) if for any w, and $\alpha > 0$,*

$$\begin{aligned} & f'_{z_1}(w) = f'_{z_2}(w - \alpha f'_{z_2}(w)) \\ \text{and} \quad & f'_{z_2}(w) = f'_{z_2}(w - \alpha f'_{z_1}(w)) \end{aligned} \tag{6}$$

Definition 2. *Two disjoint sets of training points Z_1, $Z_2 \subset Z$ are interchangeable with respect to f if z_1 and z_2 are interchangeable for every $z_1 \in Z_1$ and $z_2 \in Z_2$ with respect to f.*

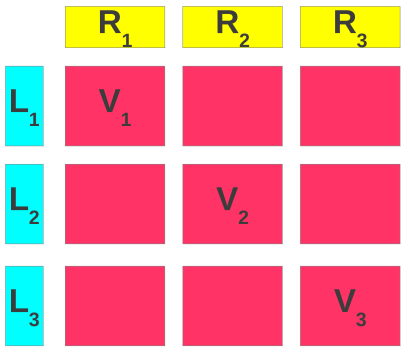


Fig. 2: interchangeable sets and their corresponding model partitions

By partitioning data into interchangeable sets, *model concatenating* is able to partition the model accordingly. As shown in Figure 2, since V_1, V_2, and V_3 are three interchangeable sets each of which only updates an non-overlapped portion of model (L and R), the model can also be partitioned into $L_1 \sim L_3$, $R_1 \sim R_3$ according. The advantages of partitioning interchangeable sets are as follows. First, from the definition we know that the order in which interchangeable sets are processed does not change the convergence of original problem. Thus $\boldsymbol{V_1}$, $\boldsymbol{V_2}$, and $\boldsymbol{V_3}$ can be processed concurrently without affecting the convergence rate. Second, only model concatenation needs to be performed when merging models from different interchangeable sets. This avoids the harmful averaging operation as in model averaging LMF. Last, being able to also partition the model gives lose the memory limitation of the size of model that one can run. This is a critical advantage especially in the Big Data era where datasets are exploding. Notice that a group of interchangeable sets take up only a portion of matrix V, multiple sub-epochs over different interchangeable sets are needed to ran to fully cover the whole matrix V.

The parallelism of *model concatenating* comes from the concurrent processing of interchangeable sets. So the number of parallelism comes with the number of *interchangeable sets* one can get. The limitation of *model concatenating* is that to achieve higher parallelism, a lot of partitions need to be done. Unlike *model averaging* where the partition is random, finer granularity structural partition will incur unbalance workload. In other word, different interchangeable sizes will end of with very different size. This causes long waiting time between sub-epochs since nodes are waiting each other for the corresponding portion of model.

[18] alleviates the waiting between sub-epochs by further partition each data portion into two parts. In essential, this is a pipeline mechanism where one part of model is being sent to other nodes when the other part is getting updated. As shown in Figure 3, when partition V_{11} finishes on $node_1$, instead of also waiting V_{12} to finish it sends R_{11} directly to $node_2$ while V_{12} is still executing. Utilizing the pipeline mechanism solves the waiting problem of *model concatenating* but it still suffers low speed-up considering the number of cores available for each node. [18] also only aims for in-memory LMF.

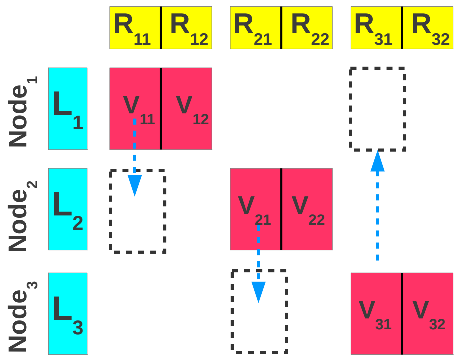


Fig. 3: pipeline mechanism within each partition

IV. Turbo Distributed LMF

In this section, we propose **T**urbo **D**istributed **LMF** (TD-LMF), an efficient distributed solution for solving large-scale LMF problem. TD-LMF achieves the fastest computation speed possible as well as preserving the highest convergence rates. TD-LMF's advantages over existing solutions are summarized as follows:

- Unlike single node *share-memory LMF*, TD-LMF is a distributed solution which can be scaled out to the a large number of nodes in the cloud.

- TD-LMF is able to partition the model so that it is not limited by the memory size as *model averaging LMF* methods.
- TD-LMF adopts the notion of *interchangeable sets* which does not incur any penalty on loss while merging models from different data partitions together.
- The parallelism of TD-LMF is not limited by the number of partitions as *model concatenating LMF* methods are. TD-LMF fully utilizes the cores available in each node to speed-up the execution. TD-LMF is able to beat the fastest distributed LMF in per epoch execution time. The dominance is even more and more obvious when the model size grows larger and larger.
- Different from in-memory methods, TD-LMF reads data from the disks thus imposing no limit on number of nodes should be ran in order to keep all the training data in memory.

In a nutshell, TD-LMF follows the data partition of *model concatenating LMF* at cluster-level and performs share-memory updating at node-level. Models are concatenated together after each iteration. In the following sections, we describe what techniques are used in TD-LMF to accomplish its amazing performance.

A. Data & Model Partition

The data partition in TD-LMF is done randomly column-wise. Each node gets the same number of random columns from the training data. In such a large granularity, the partitions are less likely to be unbalanced. So generally each and every node gets the same workload from this cluster-level data partition. The data in each partition is further divided randomly row-wise into blocks. The number of blocks equals to the number of nodes in the cluster. Note that after the data partition, rows and columns can be re-name to natural number since there is no order in the training matrix anyway. An example of 3 node data partition is shown is shown in Figure 4. The right side of Figure 4 shows the blocks on each node after re-naming the row and column numbers. Thus, two blocks that do not share any rows and columns are *interchangeable sets*.

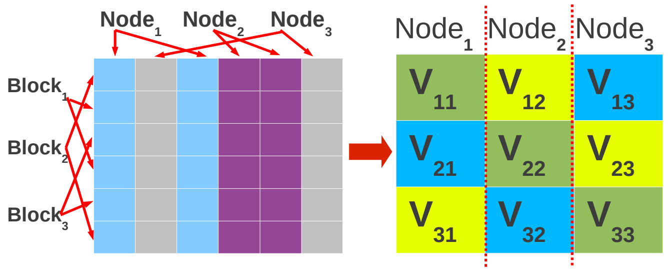


Fig. 4: Data partition and sub-epoch scheduling.

After the data is partitioned as above, the model can also be partitioned accordingly as shown in Figure 2. Thus each node only needs to store part of the model which allows for huge tasks where the model does not fit into the memory of one node.

B. Sub-epoch Scheduling

One epoch (iteration) in TD-LMF is further divided into multiple sub-epochs. Each sub-epoch processes one block of training data from the partition. The execution of sub-epoch over different nodes is fully parallelized on *interchangeable sets*. The number of sub-epochs equals to the number of blocks on each nodes. Figure 4 shows the sub-epoch scheduling of TD-LMF, the blocks in the same color are interchangeable sets and are thus in the same sub-epoch, e.g. V_{11}, V_{22}, V_{33} are in same sub-epoch. There are three sub-epochs in this case.

C. Share-memory Update

When one block is processed during a sub-epochs, share-memory updating is used within each node. This share-memory updating differs from *share-memory LMF* in that instead of linearly scanning through the training data and having tuple-level parallelism, share-memory updating in TD-LMF parallelize the update in chunk level. A chunk is nothing but a branch of tuples (training data points) and a block contains multiple chunks. Multiple chunks are processed concurrently in TD-LMF and this allows for reducing contention which we will explain in the following section IV-D.

To further speed up the TD-LMF, instead of using mutexes, we use *CompareAndSwap* atomic instruction to lock only on the rows and columns that are being updated.

D. Reducing Contention

To reduce the contention of share-memory updating during a sub-epoch, the training data is sorted by the coordinates within each node partition. In essential, the intuition of sorting the data is to put the points which are more likely to have contention together. And then serially process these contention points. This kind of assembled contention unit is a chunk in TD-LMF. Since the parallelism comes from concurrent processing of multiple chunks, the contention across chunks is minimized. Note that we still preserve tuple-level randomization and chunk-level randomization to ensure the SGD is performed in a random way across iterations. The randomization is guaranteed by random shuffle the chunks and tuples within one chunk while processing. Details about the randomization in can be found in [14].

E. Model Transmission

Since the model is partitioned during each sub-epoch, model transmission is needed at the end of each sub-epoch to allow other nodes to work on the same portion of model in turns. In TD-LMF, model transmission is done by node-to-node communication since there is no cluster-level synchronization needed.

V. Empirical Evaluation

In this section, we presents how TD-LMF is implemented in GLADE, a distributed parallel database. We evaluate the efficiency and scalability of TD-LMF on large dataset. To show comparisons, we implement *model averaging*, *model concatenating*, and TD-LMF in GLADE and we test these solutions by a large dataset. For efficiency, we measure the per-epoch execution time of all these methods. To test scalability, we run LMF with different ranks to see how well every methods scale with the model size.

A. Implementation

GLADE: We implement TD-LMF in GLADE. GLADE is a parallel data processing system executing any computation specified as a Generalized Linear Aggregate (GLA) [6] using a merge-oriented strategy supported by a push-based storage manager that drives the execution. Essentially, GLADE provides an infrastructure abstraction for parallel processing that decouples the algorithm from the runtime execution. The algorithm has to be specified in terms of a clean interface, while the runtime takes care of all the execution details including data management, memory management, and scheduling. Contrary to existent parallel data processing systems designed for a target architecture, typically shared-nothing, GLADE is architecture-independent. GLADE hits the physical limitations both on shared-disk servers as well as on shared-nothing clusters—see [16] for experimental evidence. The reason for this is the exclusive use of thread-level parallelism inside a processing node while process-level parallelism is used only across nodes. There is no difference between these two in the GLADE infrastructure abstraction.

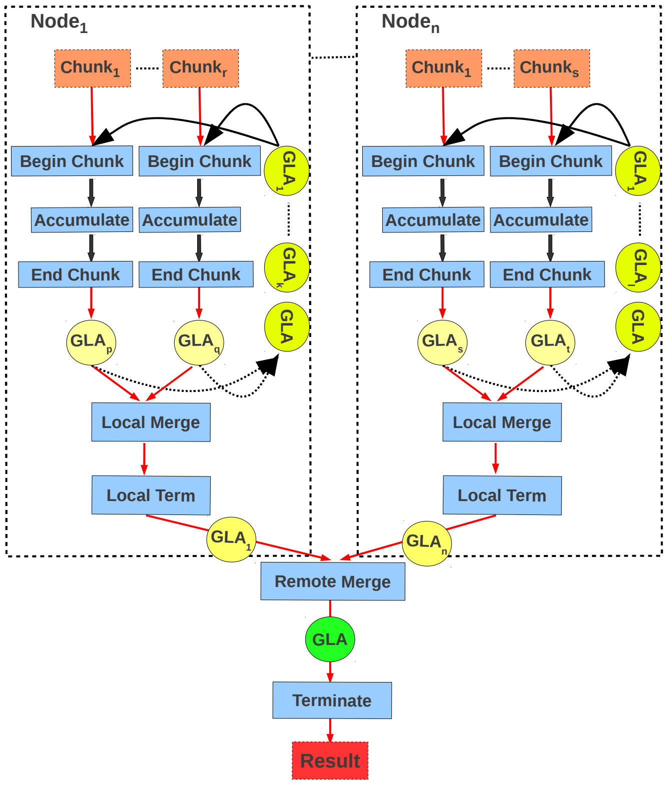


Fig. 5: GLADE architecture for chunk-at-a-time parallel processing using a merge-oriented execution strategy.

GLADE consists of two types of entities: a coordinator and one or more executor processes. The coordinator is the interface between the user and the system. Since it does not manage any data except the catalog metadata, the coordinator does not execute any data processing task. These are the responsibility of the executors, typically one for each physical processing node. It is important to notice that the executors act as completely independent entities, in charge of their data and of the physical resources. Each executor runs an instance of the DataPath [2] execution engine enhanced with a GLA metaoperator for the execution of arbitrary user code specified using the GLA interface.

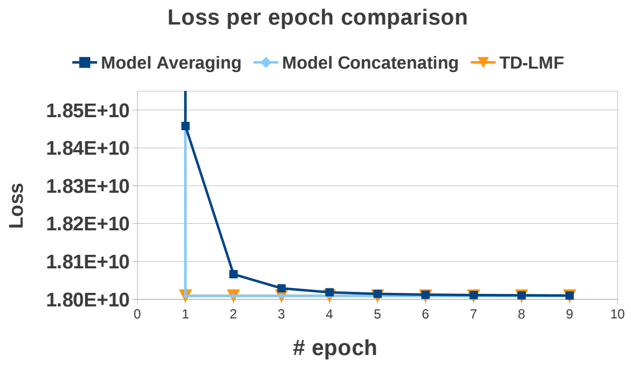
Figure 5 depicts the stages of the GLADE execution strategy expressed in terms of the GLA interface abstraction [16]. The GLA interface extends the common UDAs implemented in every major RDBMS [9] to parallel chunk-at-a-time or vectorized execution—the processing strategy in GLADE. A chunk contains a set of tuples. It is the basic I/O and processing unit in GLADE. Intuitively, GLAs are objects corresponding to the state of the aggregate upon which the methods in the GLA interface are invoked following a well-defined pattern. The semantic of the UDA methods is standard and is presented elsewhere [7]. `BeginChunk` and `EndChunk` provide access to the entire chunk – the unit of processing – before and after the GLA state is updated. An important usage scenario for these methods in our case is to reorder the tuples in the chunk such that different orders are used across iterations. `Merge` and `Terminate` are split into local (thread-level) and remote (process-level) instances to make the distinction between node- and cluster-level processing clear. This allows for different model merging strategies to be applied inside the node and between nodes.

There are several reasons make us to implement TD-LMF in GLADE. First, GLADE uses static scheduling is ideal for iterative tasks like LMF. Hadoop is notorious for low efficiency in running iterative tasks. Some variance of Hadoop (e.g. Spark [19], Haloop [5]) works better but still suffers when the data does not fit in memory. Second, unlike Hadoop-based systems, GLADE has node-level multi-thread parallelism which allows us to run parallel LMF within each node. Third, SGD is already supported in GLADE with maximum speed (I/O bounded) [14]. More details about how SGD is realized in GLADE can be find in [14].

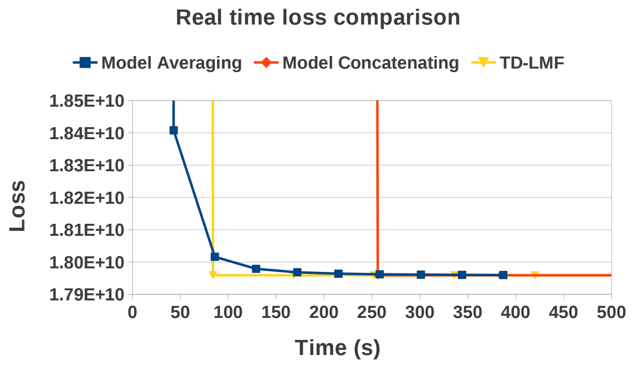
Minimizing Disk IO: Intuitively, the sub-epoch control can be done by imposing selection conditions to the coordinates in the query. Doing this, however, incurs huge overhead on disk IO. The reason is the classic way of checking selection condition in database is post-reading which means the the data is first read in memory and then the selection conditions are tested. If the classic way is followed, the database needs to read the whole dataset once at each sub-epoch. We minimize the disk IO by adding the max and min value to the meta-date of chunks. In the case of LMF, the added value are the max and min coordinates of the training tuples in a chunk. Every time before one chunk is read, TD-LMF will first check the meta-data to see whether it matches the selection condition so that the selection condition is checked before actual data reading starts.

B. Setup

In our experiments, we used a 9-node cluster running Ubuntu SMP 11.04 with Linux kernel 2.6.38-14. One node is configured as the GLADE coordinator while the other eight are workers. Each worker node has 16 cores, 16GB of RAM, and 4 1TB disks. Striping, i.e., RAID-0, is implemented directly



(a)



(b)

Fig. 6: (a) loss value as a function of the number of epoch; (b) loss value as a function of time

in GLADE, not configured in the OS. Notice that only the workers are executing data processing tasks. All the tasks read data from disks with cold caches. We report the average execution time per iteration for the model computation phase. We plot the loss function value to show convergence rate.

Dataset	Dimension	# Examples	Dataset Size
`Matrix10B`	1 million $\times$ 1 million	1.7 billion	36 GB

Rank of L&R	Model Size
10	160 MB
50	800 MB
100	1.6 GB

TABLE I: dataset and model size of different ranks

C. Dataset

We run experiments over the largest synthetic dataset `matrix10B`[1] from [9]. `matrix10B` is a sparse matrix with 1 million rows and 1 million columns containing 1.7 billion non-empty cells. The size of the chunk is set to 2^{20} tuples. We also experiment with different rank size to test the scalability of each solutions. Table V-B shows the statistics of `matrix10B`.

D. Efficiency

We measure the loss over number of epochs and loss over real time for the converging efficiency of all the methods. Figure 6a depicts the loss value as a function of number of epochs. The ranks of L and R are both 10. As we can see from the figure, TD-LMF achieve the same loss value as *model concatenating* in same epochs while *model averaging* converges slower due to the effect of averaging models. Figure 6b depicts the loss comparison over the real time. We can see that TD-LMF first converges, then *model averaging*, and last is *model concatenating*. This is because the *model averaging* is much faster in per epoch execution time. So even tough *model averaging* can not get as good loss as *model concatenating* in same number of epochs, it can run a lot more iterations in the same period of time to get better loss. *model averaging* still can not beat TD-LMF since TD-LMF much faster than *model concatenating*.

[1]We thank the Bismarck authors for providing us the datasets.

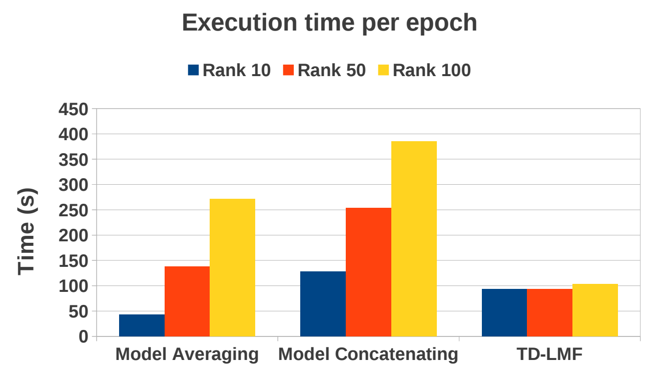


Fig. 7: Comparison of execution time per epoch.

E. Scalability to Model Size

To test scalability to dealing with large tasks, we experiment with different rank to see how each methods scale with different model size. Figure 7 depicts execution time per epoch with different rank sizes. When the rank is 10, i.e. model size 160 MB, *model averaging* is the fastest one. This is because *model averaging* achieves full parallelism and no synchronization is involved. However, as the rank increases, i.e. the model size increases, the size of model starts to play a role. As we can see, *model averaging* almost triple the execution time, while *model concatenating* almost double the execution time and TD-LMF almost stays the same. The reasons are explained as follows. *Model averaging* keeps copies of model for parallelism. When the model size grows, *model averaging* can not keep as many as copies as before since there is not enough memory. In other words, the amount of parallelism of *model averaging* decreases as the model grows. The execution time of *Model concatenating* increases mainly because higher rank requires more computation and each chunk takes longer to process (the dot product operation is more expensive). TD-LMF, however, is almost free from the effect since the increased workload is alleviated to all running threads (16 in our case) so that it is not affect is really small. Note that, if the advantage of TD-LMF will keep increasing if the rank grows even larger. This shows the nice scalability of TD-LMF to deal with large tasks.

F. Discussion

We have shown that TD-LMF is able to achieve the same convergence rate in terms of number of epochs as *model concatenating* and that TD-LMF is faster than *model averaging* in terms of execution time. This is a remarkable result since *model concatenating* is state-of-the-art in convergence rate and *model averaing* is state-of-the-art in computation speed of SGD-based distributed LMF tasks. Moreover, when the model size grows larger, TD-LMF has a 2$\sim$4$\times$ speed-up compared with existing solutions. We expect to have even higher speed-up if the model size keeps growing.

VI. Related Work

There is a plethora of work about how to better support large-scale machine learning tasks under Big Data context. The iterative feature of machine learning tasks post challenges on conventional MapReduce implementation, e.g. Hadoop. Variants of Hadoop which include HaLoop [5] and Spark [19] have been proposed to better deal with iterative task. These systems all target at tasks that fit in the aggregated memory of a large cluster which limits the parallelism of a single node. GLADE [16] reads data from the disk which allows full utilization of the cores that one node has. Also, GLADE has been shown to be significantly faster than Hadoop [6]. We therefore choose GLADE as our starting point to implement TD-LMF.

As to parallel solutions of LMF, [10] and [12] run LMF on Hadoop and Spark respectively. Both of these work only achieve cluster-level parallelism in which the number of concurrent worker equals to the number of nodes in a cluster. No node-level parallelism is explored, i.e. there is no multi-threading and only single worker running in one node.

[18] further partitions the data inside one node into small *interchangeable sets* to allow for thread level parallelism on MPI [1]. But further partitioning the data into smaller granularity introduce high variance in workload for different threads. The reason is that the available training data V_{ij} is unlikely to uniformly distributed over the original matrix. Small granularity data partition is prone to be unbalance and has to deal with the load-balancing issue. MPI also suffers from heavy implementation overheads, while in GLADE all the work is expressed in a set of user-defined functions. [18] also requires the dataset to be fit in the aggregated memory of cluster.

Hogwild! [13] shows how SGD can be implemented without any locks when the given matrix is sparse. This work inspires our work in bring in the parallelism at node-level, but we end up still having locks to allow for more general cases.

One closely related work is [14] where LMF is implemented in GLADE through *model averaging*. The data partition in [14] is random and the model communication is done by every node sending the whole model to the coordinator. The coordinator needs to broadcast the averaged model back to every node after each epochs. In TD-LMF, model transmission is done by node-to-node communication since there is no cluster-level synchronization needed. Also, one epoch in [14] is further broken down to several sub-epochs to accomplish non-averaging parallelism in TD-LMF.

VII. Conclusions

In this paper, we propose TD-LMF, a novel scalable and efficient parallel distributed solution for low-rank matrix factorization problem. TD-LMF overcomes drawbacks of different existing solutions and achieve best performance in both convergence rate as well as execution time. When the tasks grows larger, e.g. when the model size increases, TD-LMF shows even more advantages over existing solutions in that its much less sensitive to the model size. This gives potential to run LMF over at the maximum scale possible. In the future, we plan to extend this work to more machine learning problems with the same parallelism as well as convergence rate.

References

[1] http://www.mpich.org/.

[2] S. Arumugam and al. The DataPath System: A Data-Centric Analytic Processing Engine for Large Data Warehouses. In *SIGMOD 2010.*

[3] J. Bennett, C. Elkan, B. Liu, P. Smyth, and D. Tikk. Kdd cup and workshop 2007. *SIGKDD Explor. Newsl.*, 2007.

[4] D. P. Bertsekas. Incremental Gradient, Subgradient, and Proximal Methods for Convex Optimization: A Survey. MIT 2010.

[5] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. HaLoop: Efficient Iterative Data Processing on Large Clusters. *PVLDB*, 3(1), 2010.

[6] Y. Cheng, C. Qin, and F. Rusu. GLADE: Big Data Analytics Made Easy. In *SIGMOD 2012*.

[7] S. Cohen. User-Defined Aggregate Functions: Bridging Theory and Practice. In *SIGMOD 2006*.

[8] G. Dror, N. Koenigstein, Y. Koren, and M. Weimer. The Yahoo! Music Dataset and KDD-Cup'11. *JMLR Workshop and Conference Proceedings*, pages 3–18, 2012.

[9] X. Feng, A. Kumar, B. Recht, and C. Ré. Towards a Unified Architecture for in-RDBMS Analytics. In *SIGMOD 2012*.

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

[11] Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *Computer*, pages 30–37, 2009.

[12] B. Li, S. Tata, and Y. Sismanis. Sparkler: supporting large-scale matrix factorization. EDBT, 2013.

[13] F. Niu, B. Recht, C. Ré, and S. J. Wright. A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. In *NIPS 2011*.

[14] C. Qin and F. Rusu. Scalable i/o-bound parallel incremental gradient descent for big data analytics in glade. DanaC, 2013.

[15] B. Recht and C. R. Parallel stochastic gradient algorithms for large-scale matrix completion. *Mathematical Programming Computation*, 5(2), 2013.

[16] F. Rusu and A. Dobra. GLADE: A Scalable Framework for Efficient Analytics. *OS Review*, 46(1), 2012.

[17] A. P. Singh and G. J. Gordon. A unified view of matrix factorization models. ECML PKDD '08, 2008.

[18] C. Teflioudi, F. Makari, and R. Gemulla. Distributed matrix completion. *IEEE International Conference on Data Mining*, 2012.

[19] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. HotCloud, 2010.

[20] M. Zinkevich, M. Weimer, A. Smola, and L. Li. Parallelized Stochastic Gradient Descent. In *NIPS 2010*.