Collaborative Filtering as a Case-Study for Model Parallelism on Bulk Synchronous Systems

Ariyam Das
University of California, Los Angeles
ariyam@cs.ucla.edu

Ishan Upadhyaya
University of California, Los Angeles
ishan793@ucla.edu

Xiangrui Meng
Databricks
meng@databricks.com

Ameet Talwalkar
University of California, Los Angeles
ameet@cs.ucla.edu

ABSTRACT

Industrial-scale machine learning applications often train and maintain massive models that can be on the order of hundreds of millions to billions of parameters. Model parallelism thus plays a significant role to support these machine learning tasks. Recent work in this area has been dominated by parameter server architectures that follow an asynchronous computation model, introducing added complexity and approximation in order to scale to massive workloads. In this work, we explore model parallelism in the distributed bulk-synchronous parallel (BSP) setting, leveraging some recent progress made in the area of high performance computing, in order to address these complexity and approximation issues. Using collaborative filtering as a case-study, we introduce an efficient model parallel industrial scale algorithm for alternating least squares (ALS), along with a highly optimized implementation of ALS that serves as the default implementation in MLlib, Apache Spark's machine learning library. Our extensive empirical evaluation demonstrates that our implementation in MLlib compares favorably to the leading open-source parameter server framework, and our implementation scales to massive problems on the order of 50 billion ratings and close to 1 billion parameters.

CCS CONCEPTS

•Information systems → Collaborative filtering; •Computing methodologies → Factorization methods; Distributed algorithms;

KEYWORDS

Collaborative filtering; Bulk synchronous parallel (BSP) systems; Alternating least squares (ALS); Model parallelism; Parameter servers; Apache Spark

1 INTRODUCTION

The emergence of massive datasets has motivated the data parallel paradigm [5] in the distributed setting. Indeed, modern industrial learning applications like collaborative filtering for recommendation systems or click-through rate prediction for display advertisements [14] require such distributed storage and processing capabilities. However, these learning applications also generate massive machine learning models, often on the order of hundreds of millions to billions of parameters in size. These models are typically too large to store and update on a single machine, thus motivating model parallel industrial scale approaches for efficient storage and updating of massive models.

Bulk synchronous parallel (BSP) systems such as Apache Hadoop and Apache Spark are arguably the most commonly used systems for data parallel tasks, given their elegant and serial-equivalent computational model. However, BSP systems have received much less attention for model parallel learning tasks, as the iterative nature of most distributed learning algorithms is seemingly a mismatch with BSP systems' synchronization requirements. This mismatch has motivated the development of asynchronous parameter server architectures [7, 37], which handle model parallelism by using a stale synchronous update mechanism that relieves the bottleneck of straggler nodes and thus reduces idle computation resources. Recent parameter server frameworks like [18, 33], built around the flexible asynchronous communication model, have shown to consistently outperform BSP systems like Apache Mahout [27] and graph-centric platforms such as GraphLab [20].

However, these asynchronous mechanisms [11, 16] introduce added complexities in machine learning applications. For instance, in order to ensure convergence of these approximate solutions, parameter servers typically use a dedicated scheduler [18, 33] which identifies the parameters that can be updated at any given time. The scheduler also allots higher priority to parameters with slow convergence rates to speed up convergence [33]. Parameter servers thus incur additional computational expenses in the form of scheduling overhead and dependency structure resolution [33], and require significant resources to develop and maintain their complex software implementations.

In light of the complexities associated with parameter servers, in this work we revisit the efficacy of model parallelism in BSP systems. We focus on matrix factorization approaches [15, 28] for collaborative filtering as a case-study to compare BSP and parameter servers systems. While naive BSP algorithms for matrix factorization may not be scalable, we demonstrate that an industrial scale algorithmic approach, along with an optimized implementation in Spark can compete with a state-of-the-art open source

CIKM'17, November 6–10, 2017, Singapore.
© 2016 ACM. ISBN 978-1-4503-4918-5/17/11…$15.00
DOI: http://dx.doi.org/10.1145/3132847.3132862

parameter server implementation, and can even outperform it. Our two main contributions can be summarized as follows:

- We introduce a industrial scale distributed algorithm for the classic Alternating Least Squares (ALS) method for matrix factorization in the BSP setting. Our algorithm currently serves as the default collaborative filtering method in a leading open source machine learning library – Spark MLlib [24]. In particular, we describe a novel *block-to-block* join technique along with a series of rigorous engineering efforts, the latter of which translates many abstract concepts originating from recent progress in high performance computing [6] into highly-efficient communication, storage, and computational optimizations. Together, our novel algorithm and implementation optimizations result in significant speedups and scalability for large-scale model parallel machine learning applications.
- We present comprehensive experimental results on industrial-scale datasets comparing our optimized ALS algorithm with an optimized open-source parameter server implementation (Petuum [33]). Our results demonstrate that our BSP approach is comparable in terms of accuracy and runtime with Petuum for models with relatively small numbers of parameters, while we significantly outperform Petuum as the number of model parameters increases. We perform an extensive empirical evaluation with three industrial datasets (one of which is from an actual industrial deployment) that contain as many as 50 billion ratings. We learn matrix factorization models with up to a billion parameters, thereby showing the robustness, efficiency and scalability of our implementation.

The rest of the paper is organized as follows. In section 2, we briefly review the recent literature on collaborative filtering problem. Section 3 reviews the state-of-the-art distributed systems used for large-scale collaborative filtering problems. Section 4 first discusses the limitations of a naive BSP ALS implementation, and then gradually introduces our various optimizations in detail that result in an efficient ALS algorithm, along with the lessons learnt on the way. The experimental results on industry-scale datasets are presented in Section 5.

2 BACKGROUND

The leading approach to solving large-scale collaborative filtering problems for recommendation engines is via matrix factorization [1, 15]. In this model, we want to learn the hidden user and item factors from an incomplete ratings matrix. More formally, let $A \in \mathbb{R}^{n_u \times n_v}$ be the incomplete ratings data, where n_u and n_v denote the total number of users and items, respectively. Then, the matrix factorization problem can be formulated as:

$$\min_{\substack{U \in \mathbb{R}^{n_u \times k} \\ V \in \mathbb{R}^{n_v \times k}}} \underbrace{\sum_{(i,j)\in\Omega} (A_{ij} - u_i^T v_j)^2 + \lambda(||U||_F^2| + ||V||_F^2)}_{g(U,V;A)} \qquad (1)$$

where Ω represents the indices for observed ratings, λ is the regularization parameter, and u_i^T and v_j^T denote the i^{th} and j^{th} row vectors of the matrices U and V representing the k-dimensional feature vectors of user i and item j, respectively. Minimizing the above objective function allows us to approximate the incomplete ratings matrix as the product of two rank-k matrices.

While the input ratings matrix A is typically sparse, the user and item feature vectors are usually dense. Thus this model requires $(n_u + n_v) * k$ parameters. In industrial-scale datasets, the number of input ratings can be on the order of billions, while n_u can range anywhere from $10^6 - 10^8$. Thus, even when considering low rank models, e.g., $k = 10$, the number of model parameters can be on the order of billions. The scale of both the input ratings and the models for such large-scale problems often necessitates storage across multiple machines, thus requiring efficient communication mechanisms for model training.

Classically, the Alternating Least Squares (ALS) [29] method solves the the objective function $g(U, V; A)$ by alternatively updating U and V, using the following closed form expression, shown here for a user factor:

$$u_i^* = (\hat{V_i}^T \hat{V_i} + \lambda I_k)^{-1} \hat{V_i}^T A_i \qquad (2)$$

where $\hat{V_i}$ is the matrix containing the vectors of items rated by user i and A_i is row vector containing ratings by user i. Previous work [38] has shown that the subproblems for each user or item can trivially be solved in parallel, thus suggesting a natural parallelization strategy which has been adopted in systems like Apache Mahout [27]. However, as we will discuss, communication bottlenecks limit the scalability of these straightforward parallelization schemes for industrial-scale applications.

Alternatively, $g(U, V; A)$ can also be solved using Stochastic Gradient Descent (SGD) [25] or coordinate descent [35]. SGD enjoys strong convergence guarantees [9, 17], and asynchronous update schemes have been proposed to scale SGD to large-scale collaborative filtering problems [16, 25]. However, as pointed out in various works [8, 35, 36], the convergence of SGD can be sensitive to the learning rate and SGD can scale poorly in distributed settings when the number of workers is large (on the order of hundreds). Thus, many next generation parameter server frameworks (e.g. Petuum-Strads, DiFacto) have adopted the coordinate descent approach.

3 COMPETING SYSTEMS

BSP systems like Apache Mahout, Spark and parameter server frameworks like Petuum [33], DiFacto [19] have adopted different approaches for the collaborative filtering problem, *each of which is particularly well-suited for its individual framework.* The underlying implementation on these frameworks are highly optimized and tuned in accordance to each of these systems.

One of the earliest open-source industrial scale solution for collaborative filtering on BSP systems was the ALS implementation on Apache Mahout [27]. However, recently an implementation of ALS on Spark (MLlib version 1.1) has been shown to outperform Mahout by an order of magnitude in the benchmark study in [24], mainly due to Spark's in-memory capabilities. However, this Spark implementation still suffered from communication bottlenecks, high storage requirements, and significant computational overheads, making it inefficient for large-scale datasets and models. Likewise, graph based platforms like GraphLab [20], with built-in scheduling mechanisms, follow a vertex-centric computing model [34] to ensure strong consistency among its model variables (the Gather-Apply-Scatter abstraction), thereby making its communication cost comparable to the synchronous setting on the BSP frameworks.

The recently proposed parameter server systems [18, 19, 33] leverage the idea of error-tolerance in machine learning algorithms and follow an asynchronous/stale synchronous update mechanism, where model parameters are not synchronized at every epoch. The first generation parameter server framework PMLS Bosen [10] mainly provided a data parallel machine learning framework which was unable to scale to billions of model parameters. It was recently replaced by the state-of-the-art model parallel parameter server system Petuum (or PMLS Strads), which have been shown to outperform BSP systems and graph-central frameworks like GraphLab [33].

In addition to a stale-synchronous model, Petuum also offers a dynamic scheduler that prioritizes computation towards non-converged model parameters, hoping for a faster convergence. However, the asynchrony in the updates can lead to the divergence of strongly correlated model parameters. This observation introduces additional overhead for the scheduler, as it must decide which model parameters can be updated independently. With respect to the matrix factorization problem, Petuum solves the objective function in equation (1) using a highly optimized distributed implementation of coordinate descent [13].

Another recent parameter server system, DiFacto [19] supports large-scale distributed factorization machines to solve the recommendation problem. However, this system is optimized to incorporate user features. Unlike all the other aforementioned competing systems, DiFacto solves a more complex objective than the one described in equation (1).

4 DISTRIBUTED ALS IMPLEMENTATION

In this section, we first examine a trivial distributed ALS algorithm, which *broadcasts everything* as shown in Figure 1a. In this simple algorithm, the ratings matrix is loaded into the memory of the master which then initializes the model and broadcasts both data and initial model parameters. At the end of each ALS iteration, the updated parameters are then broadcasted back. Consequently, this approach involves a lot of communication overhead and can only work for small data and model sizes.

A better alternative would be to use a *data parallel* approach, where the ratings matrix is partitioned across the workers and the master initializes the models and broadcasts only the parameters. This case is shown in Figure 1b. The updated model at the end of each iteration is then broadcasted again. This algorithm works reasonably well on large datasets, provided the number of model parameters remains small.

However this approach also does not scale to large model sizes that do not fit in the memory of a single machine. In fact, for training these large models, we need to leverage the idea of model parallelism, where models (user and item factors) are initialized at the workers and shared at each iteration via joins between the workers as shown in Figure 1c. However, as we show next in Section 4.1.1, a naive model parallel algorithm may not scale well due to the overall communication cost. After discussing the inadequacies of these naive algorithms, we will then gradually introduce our optimization strategies to solve these limitations.

4.1 Reducing Communication Cost

4.1.1 All-to-All Join. In a naive model parallel ALS algorithm, the ratings matrix, the user and item factors are distributed across the workers in a cluster. Let us assume in an ALS iteration, we need to calculate the item factors from the previously computed user factors. This means the appropriate user factors need to be sent to the machines containing the relevant item factors [26]. To build an intuitive understanding of this approach, consider the bipartite graph presented in Figure 2. In this figure, the users are represented by the nodes on the left and the items by nodes on the right. Each edge here represents a rating. Now, for computing an item feature vector, say for v_1, we need to send the user factors for users u_1 and u_2 to the node storing the factors for v_1. Similarly, while computing factors for v_2, we need to send the factors of u_1 and u_2 along with u_3 to the node, which has the factors of item v_2. This is similar to the message passing mechanism of vertex programming models executed on platforms like GraphLab [20] and Pregel [21], irrespective of mirroring technique [34]. It is also important to note that, even when the factors for v_1 and v_2 are hosted on the same machine, this algorithm will still end up sending the factors of u_2 twice. In fact, the communication cost in this algorithm is governed by the number of edges times the size of the factors we are computing. Since, the number of edges is equal to the number of ratings, the communication cost is given by $O(nk)$, where n is the number of ratings under consideration and k is the rank.

Solving a least squares convex problem, using the closed form expression shown in (2), has a computational time complexity of $O(nk^2)$. Although, the communication cost is less than the computation cost of $O(nk^2)$, but the computation is done in CPU. Thus for large-scale models, the network bandwidth and the communication cost form the main bottleneck in BSP systems like Spark (or graph-based platforms like GraphLab) as naive algorithms end up shuffling huge amount of data over the network for each model update in every iteration. Therefore to mitigate this issue, we propose the following novel block-to-block strategy.

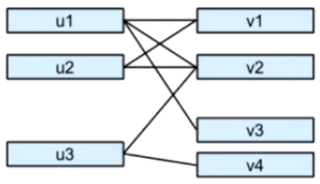
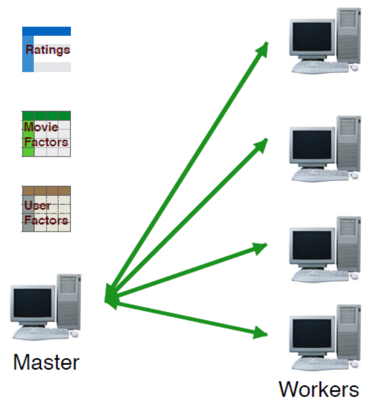
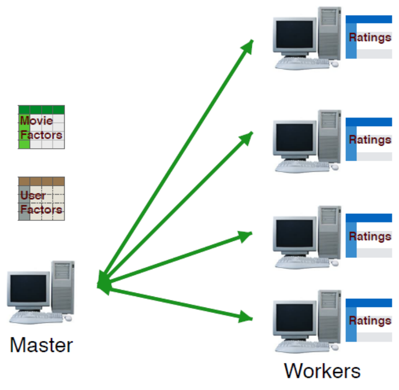


Figure 2: All to all join adversely impacts communication cost thereby making naive model parallel ALS unscalable.

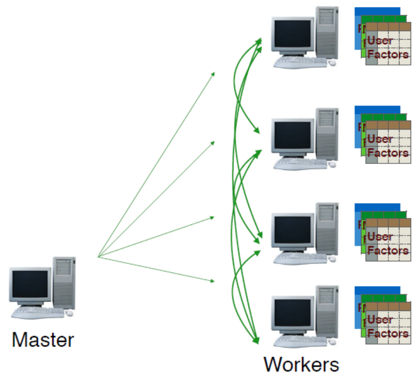
4.1.2 Block-to-Block Join. In a block-to-block join algorithm, we use our own data structure to reduce the shuffling overhead by partitioning the ratings, user factors and item factors into blocks distributed over the workers in the cluster. In this algorithm, we create two sets of blocks – `InBlocks` and `OutBlocks`. Formally, the `OutBlocks` *store the factors which are needed to be sent out*, while



(a) Broadcast everything



(b) Data parallel architecture



(c) Model parallel architecture

Figure 1: Different architectures for distributed ALS implementation.

`InBlocks` *store the factors which are being computed during the current iteration.* Continuing with the same example introduced earlier, the corresponding `InBlocks` and `OutBlocks` are shown in Figure 3. Here, we store the factors for users u_1 and u_2 on the `OutBlock` P_1 and the factors for user u_3 on `OutBlock` P_2. Similarly, we partition the item factors for items v_1, v_2, v_3 and v_4 evenly into `Inblocks` Q_1 and Q_2. If we assume that each block is stored on a single node, then the user factors in `OutBlock` P_1 only need to be sent once over the network to update both the factors for v_1 and v_2, thereby reducing the communication cost over all-to-all join.

In addition to the `InBlocks` and `OutBlocks`, we also need two auxiliary data structures – `OutLink` mapping and `InLink` mapping. The `OutLink` mapping *stores the user/item ids and feature vectors for a given* `OutBlock`*, along with a list of destination* `InBlocks` *that indicates where to send these vectors.* The second auxiliary data structure is called `InLink` mapping and is stored at each `InBlock`. It *represents a sub-graph that indicates which user/item factors to use for computing a given item/user factor* respectively, once the required data has been sent using the `OutLink` mapping.

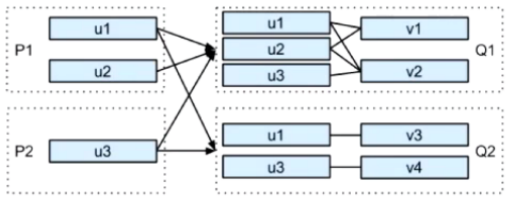


Figure 3: Reducing communication cost using block to block join.

4.1.3 Ratings Matrix Distribution. The distribution of the input ratings matrix can also play a significant role in minimizing the communication overhead. For example, consider the `1D` partitioning shown in Figure 4, where the ratings are distributed in a row-wise fashion across the workers. Unfortunately, this partitioning would create a significant amount of data shuffling across the cluster while computing the item factors (similar argument applies for user factors if we use a column-wise partitioning).

Another alternative is to use `2D` partitioning shown in Figure 4, where disjoint blocks of the ratings matrix are distributed across the workers. However, in the worst case, a worker can end up having just a single rating due to this `2D` partitioning. As a result, computing the user and item factors corresponding to that worker would involve a lot of data shuffling.

In order to resolve the above issues, we leverage the concept of hybrid `1.5D` partitioning [6] as shown in Figure 4. In this strategy, two copies of the sub ratings matrix are cached with the corresponding `InLink` mapping at the `InBlocks` to reduce the cost of shuffling at each iteration. One copy is used the for `InBlocks` for users and another for `InBlocks` for items. These algorithmic optimizations allow us to significantly reduce the communication cost. However, with `1.5D` partitioning, we are now storing two copies of the ratings. Therefore, we also need to store the `InBlocks` efficiently, as otherwise a storage overhead would lead to a larger number of machines being required in the cluster, thus adversely impacting the cost and scalability.

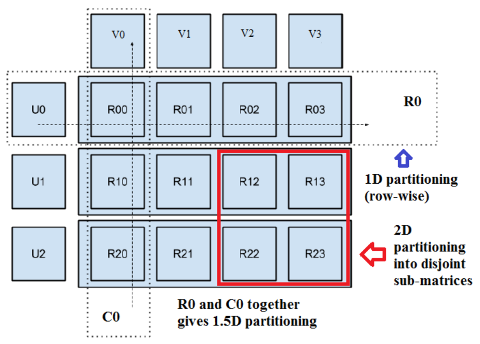

V0
V1
V2
V3
U0
R00
R01
R02
R03
R0
U1
R10
R11
R12
R13
1D partitioning (row-wise)
U2
R20
R21
R22
R23
2D partitioning into disjoint sub-matrices
C0
R0 and C0 together gives 1.5D partitioning

Figure 4: Three different strategies for distributing the ratings matrix.

4.2 Reducing Storage Overhead

4.2.1 Naive Implementation. As discussed earlier, the `InBlocks` for items contain user IDs (with corresponding feature vectors) along with the ratings for each corresponding user. The same logic follows for `InBlocks` for users as well. The naive way of arranging the `InBlocks` for items would be to use an array of tuples in the format `<itemid:int, userid:int, rating:double>`. For instance, the *InBlock* Q_1 in Figure 3 can be stored naively as: $[(v_1,u_1,r_{11}),(v_2,u_1,r_{12}),(v_1,u_2,r_{21}),(v_2,u_2,r_{22}),(v_2,u_3,r_{32})]$ However, there are two significant problems with this naive storage scheme – first, there will be a huge storage overhead as each tuple needs to maintain additional pointer, where the pointers are usually of storage type long (i.e. additional bytes for each tuple). Secondly, having a tuple for each rating would create billions of objects across all the Java virtual machines spawned on the different nodes in a cluster. This would create a high garbage collection (GC) pressure and increase the computational overhead.

4.2.2 Semi-naive Implementation. A slightly better approach would be to use three primitive arrays instead of an array of tuples, i.e.,: $([v_1,v_2,v_1,v_2,v_2],[u_1,u_1,u_2,u_2,u_3],[r_{11},r_{12},r_{21},r_{22},r_{32}])$. This would reduce the GC pressure as well as the storage overhead compared to the naive strategy. It is worth mentioning that instead of using array of type `double` to store the ratings, we use the primitive type `float`, as this slight loss of precision does not impact the accuracy of the ALS solution, but it reduces computational and storage overhead making the implementation more scalable. However note that the array of items is not sorted in this storage scheme. Thus, we have to start solving the least squares problems for v_1 and v_2 together to compute their corresponding item factors. In other words, we have to construct all the sub-problems together for the different items and store them in memory. This will give a worst case space complexity of $O(n_j k^2)$, where n_j is the number of items in the given `InBlock`.

4.2.3 Pre-sorted Array based Implementation. One way to reduce this storage complexity of the semi-naive implementation is to sort the primitive arrays accordingly. For example, the following instance shows the items are sorted in ascending order and the users and the ratings are sorted in accordance to the items: $([v_1,v_1,v_2,v_2,v_2],[u_1,u_2,u_1,u_2,u_3],[r_{11},r_{21},r_{12},r_{22},r_{32}])$. With the primitive arrays sorted, we can solve the sub-problems one after another in sequence, i.e., we need space only to store the sub-problem for one item vector. Thus, the space complexity can be reduced from $O(n_j k^2)$ to $O(k^2)$. However, it is worth noting that sorting the three primitive arrays must be carried out efficiently as otherwise it can significantly impact the computational overhead. This is discussed further in the next section.

4.2.4 Index based Implementation. With the pre-sorted array based implementation, we observed that there are duplicated items in the array adding to the overall storage overhead. Thus, we compressed this item array further by removing the duplicates and instead keeping a local index to point which user vectors are required by which items. Consider the following example:

$$([v_1,v_2],[0,2,5],[u_1,u_2,u_1,u_2,u_3],[r_{11},r_{21},r_{12},r_{22},r_{32}])$$

This example implies that for calculating the item vector for v_1, we would need the vectors of the users stored in the user array from index 0 to 2 (2 not included). Similarly, we would need the vectors of the users stored in the user array at indices 2, 3 and 4 to compute the corresponding item vector for v_2.

Although index based implementations is quite popular in webgraph compressions [3], following this strategy in our setting introduces novel challenges. Specifically, we would still need to perform lookup operations to identify where the user factor is located in the *OutBlock* for a given a user id. This implies that we need to create a hash map with user ids as keys and the corresponding addresses of the user factors as values. For example, while computing v_1, we learned that we need to use the user factor for u_1 and hence we need to perform a map lookup to identify the location of the corresponding user factor in the *OutBlock* P_1. Unfortunately, a hash map for millions of objects increases the storage burden as well as the computational overhead, since the lookup is not efficient for large number of objects. Some of the earlier distributed ALS implementations like in Mahout [27], Spark version 1.2 have been adversely impacted by use of hash map or hash table.

4.2.5 Block-encoding based Implementation. In the index based implementation, note that the user id is not directly required in the computation of the item vectors. Instead user id is only used to lookup the corresponding user factor from the `OutBlock`. This observation allows us to use traditional encoding techniques, where we encode this address information and store them into the user array directly, instead of the user ids. We perform this by encoding the block id of the `OutBlock` into the higher bits of an integer and using its lower bits to pack the local index of the user vectors.

Consider the same running example with block-encoding based implementation, as shown below:

$$([v_1,v_2],[0,2,5],[0|0,0|1,0|0,0|1,1|0],[r_{11},r_{21},r_{12},r_{22},r_{32}])$$

P_1 and P_2 are assigned block ids 0 and 1 respectively. Thus, we replace the user id u_2 in the user array with $0|1$, indicating that the user factor can be found in block 0 (i.e. P_1) at local index 1 (i.e., the second vector). This encoding using the bits of an integer easily allows us to deal up to 4 billions of users/items.

In addition to the above optimizations, also recall that in an ALS iteration we compute $\hat{V_i}^T \hat{V_i}$ as shown in equation 2. However, this being a symmetric matrix, we only compute and store the lower half, thus further reducing the computation and storage overhead by a factor of 2.

4.3 Reducing Computational Overhead

In this section, we briefly summarize three engineering aspects which play a significant role in the overall computational overhead – how to reduce the GC pressure, how to efficiently sort three large primitive arrays and how to perform fast linear algebra operations.

- *Reducing GC pressure:* This plays a significant role in many BSP systems like Apache Hadoop and Apache Spark that uses Java. The presence of a large number of objects in memory usually causes the JVM to pause frequently and allow the garbage collector to free up heap space. This GC action is random making different worker nodes straggle in different ALS iterations, thereby significantly increasing the overall computational time. Thus, avoiding tuple storage structure as mentioned in the naive storage technique or not using hash maps as required in the index based implementation significantly mitigates the GC pressure by creating fewer objects. In addition, we also use specialized code during initial partitioning of the ratings to prevent generating millions of tiny temporary objects.

- *Sorting primitive arrays:* As mentioned in the pre-sorted array based implementation, we need to arrange three primitive arrays. Quicksort [12], which is the default sort in Java 6, creates lot of swaps and objects. As such, we use an alternative, TimSort[1], which is a hybrid of insertion and merge sort and is extremely efficient in sorting large sequence of numbers.
- *Efficient linear algebra operations:* As presented in equation (2), the computation time of an ALS iteration depends on performing fast matrix multiplications and inversions. Hence we call standard and efficient native binaries of LAPACK and BLAS [31] via the Java Native Interface (JNI) to perform these operations.

In addition to the above strategies, avoiding map lookups (in block-encoding based implementation) and using `float` type to store ratings (at a marginal loss of precision) reduces the overall computational runtime.

5 EXPERIMENTS

5.1 Setup

We used two setups for our experiments. In the first setup, we used a 15-node `Microsoft Azure` cluster with standard `D12v2` instances, each of which has 4 cores, 28 GB RAM and runs on Ubuntu 14.04 LTS operating system. We used this as our default setup for most of our experiments. In this setup, 16 threads were spawn per worker node. For each of the experiments conducted on this `Azure` setup, we used the following settings:

- *Spark*: We used the default Spark `HDInsight` cluster on `Azure` which follows a 2+13 configuration, where 2 nodes are assigned as masters (primary and secondary masters) and 13 nodes as slaves. For all our experiments, we used Spark 2.0.
- *Petuum (Strads)*: Each of the 15 nodes was assigned the role of a worker. In addition, one of the nodes was also given the roles of the co-ordinator and scheduler, as required by Petuum[2]. In all our experiments, we disabled the fault tolerance for Petuum, as capturing periodic snapshots can deteriorate its performance. For all our experiments, we used Petuum version 1.1. The staleness for the model was not a user-defined hyper-parameter, but was instead selected automatically by the Strads scheduler.

In our second setup, we used a 16-node `m3.2xlarge` cluster on `AWS` mainly for our ablation study. In this setup, we used a 1+14 configuration for Spark, with one node being assigned as the master. We used the same configuration for Petuum. In all experiments, we omit Mahout, GraphLab as it is well-documented that Petuum significantly outperforms them for matrix factorization [33].

5.2 Datasets

We worked with the following three industrial scale datasets, the first two of which are available publicly.

1. *R2-Webscope Yahoo Music Ratings dataset*: This is the largest industrial ratings dataset made available for research[3]. This dataset represents a snapshot of the Yahoo users' preference for various songs between 2002 to 2006. It contains around 700 million ratings of 136 thousand songs given by 1.8 million users.
2. *Amazon Review dataset*: This dataset [22] was created based on the ratings of the users across the two most popular product categories. This dataset was preprocessed and stored in a format compatible to both Petuum and Spark. This dataset contained around 11 million users, 3 million items and 30 million ratings.
3. *Industrial Scale dataset*: This dataset is a sample from a real industrial application of collaborative filtering, and consists of approximately 50 billion ratings from 50 million users and 5 million items [23].

5.3 Convergence Criterion

We are interested in comparing the peak performance and scalability of two different industrial scale frameworks (Spark MLlib and Petuum) where each of them have different optimal settings. However, these two systems use different underlying algorithms with *each of them being highly optimized in accordance to their architectures.* For example, Petuum implements coordinate descent which is more suitable for their stale synchronous model while Spark MLlib relies on a synchronous ALS approach for the same collaborative filtering (matrix factorization) application. Notably, comparison of systems implementing different underlying algorithms for the same application have also been reported in [18, 33].

To ensure fair comparison between the two systems (Petuum and Spark), it is important to select a suitable stopping criterion. Since the underlying objective function for both Petuum and Spark are the same, an obvious choice would be to stop at an iteration when the change in the objective value between two consecutive epochs is less than some fixed tolerance ϵ. However, we observed that this strict stopping criterion generally leads to significant overfitting for Petuum on the training set, thereby producing considerably high RMSE on the test data. The authors of [33] suggested to compute the RMSE on a validation dataset at the end of each iteration of coordinate descent in Petuum and save the most optimal model, while running the computation until the strict ϵ stopping criterion is reached[4]. All the computations still needed to be run until the ϵ criterion is reached, as one cannot make any broad assumptions before-hand about the validation RMSE versus iterations curve.

Figure 5a plots the validation RMSE (on a given Yahoo music validation dataset) against the coordinate descent iterations in Petuum, while a model is being trained on a 10% sample of Yahoo music training dataset. As shown in the figure, the validation RMSE

[1]TimSort was used to win the 2014 Gray Sort Benchmark [32].
[2]These configurations were validated by the Petuum team [33].
[3]https://webscope.sandbox.yahoo.com/catalog.php?datatype=r
[4]Based on personal communications with the Petuum team.

curve is monotonically decreasing for rank 10, while it follows a "V" shaped curve for rank 20. Figure 5b shows the corresponding plot for Spark. It is important to mention that we found the validation RMSE in Spark to be mostly monotonically decreasing across the ALS iterations. The experimental results are reported here after hyperparameter tuning. Unless otherwise mentioned, in all our subsequent experiments we used $\epsilon = 0.01$.

5.4 Results

In the first set of experiments, we compare the performance of the systems as the model size increases. We conduct these experiments on the complete yahoo dataset with $\lambda = 0.05$. Figure 6a shows the total computation time for different ranks. As is evident from the figure, the performance of Spark and Petuum are comparable when the rank is small. However, as the number of model parameters increases, Spark becomes almost twice as fast than Petuum. This result is consistent with a recent theoretical result [2] which shows that as the number of learners increases the convergence rate following asynchronous update may greatly differ from the theoretical ideal rate. Figure 6b shows the objective function value on the training dataset for each of the systems when the algorithms terminate. Figure 6c shows the corresponding test RMSE computed on a separate test dataset using the best models as determined from a validation dataset. The baseline method shown in Figure 6c simply uses the average rating of an item as its prediction. Figures 6b and 6c indicate that although Spark has a high training objective value, nevertheless, most importantly, it always has a lower test RMSE compared to Petuum. It is also evident from these figures that in case of Spark, as the model size increases the test RMSE gradually reduces and tapers off at very high ranks. However, surprisingly, in case of Petuum, the test RMSE gradually decreases up to rank 40 and then it starts increasing with the rank. This could be possibly attributed to the fact, that a large number of model parameters could also mean a large number of strongly correlated factors [4], thereby leading to significant scheduling challenges and conflicts for Petuum because of its asynchronous update. This could also explain the sharp increase in computation time for Petuum as the rank increases, despite having the flexibility of stale synchronous model. Figure 7 shows the corresponding plot for computation time on the Amazon Reviews dataset for ranks of 40 and 60.

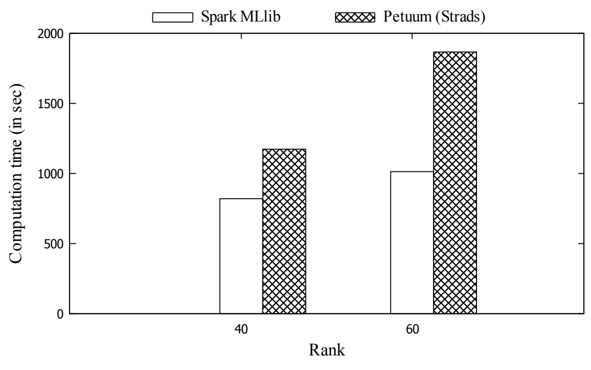


Figure 7: Results comparing computation time on Amazon dataset for ranks 40 and 60 with $\lambda = 0.05$.

In the second set of experiments, we evaluate the scalability of the two systems in terms of the number of ratings. Figure 8 compares the computation time between the two systems for a fixed rank of 60 on different subsets of the complete yahoo data, having varying number of ratings. Here also, Petuum has a comparable performance to Spark for a smaller number of ratings (around 76 million). However, as the number of ratings increases Spark outperforms Petuum significantly becoming almost twice as fast.

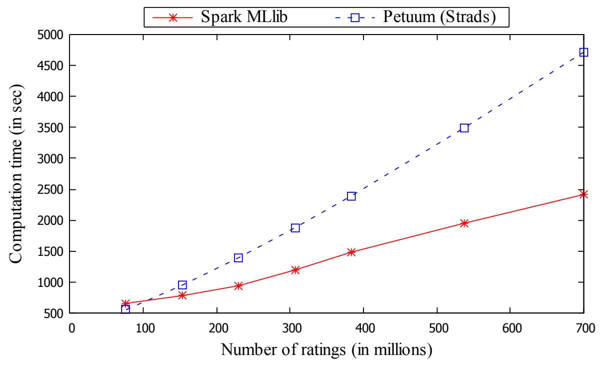


Figure 8: Results comparing computation time on full Yahoo music dataset for different number of ratings with fixed rank of 60 and $\lambda = 0.05$.

Figure 9 plots the objective function value against running time for both the systems. This experiment was conducted on the complete yahoo dataset with a fixed rank of 60. Although Spark has a much higher objective value at the start, it drops sharply and converges faster than Petuum with respect to time as well as in number of iterations. Finally, Figure 10 represents the strong scaling experiments, which were conducted on a 50% sample of the yahoo dataset with a fixed rank of 60. The figure shows that in each of the three setups with 9, 12 and 15 machines Spark outperforms Petuum consistently for the same amount of work.

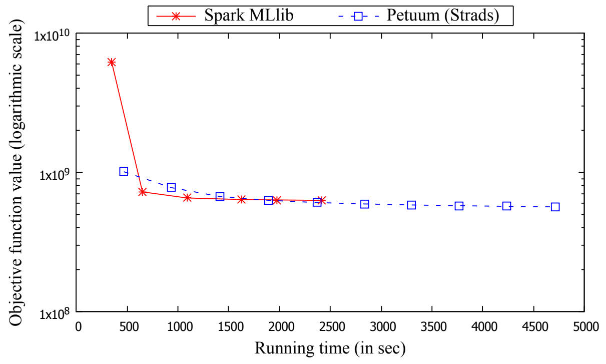
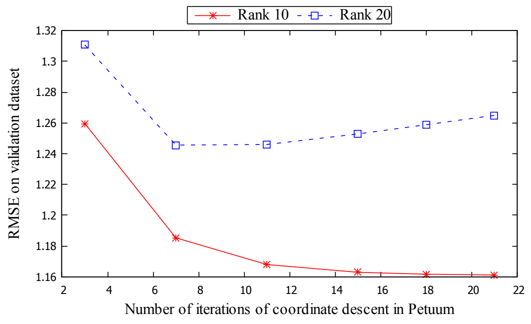
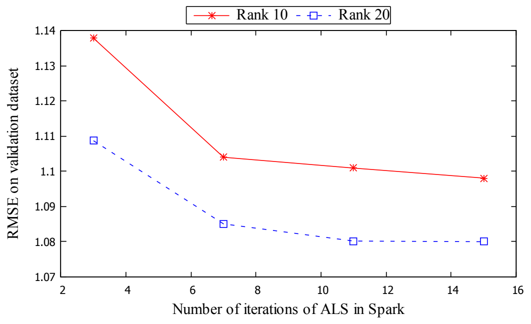


Figure 9: Plot of objective function value with running time on complete Yahoo music dataset for matrix rank 60 and $\lambda = 0.05$**.**

In our final experiment, we present an ablation study on the effectiveness of our storage optimization algorithm. We conducted

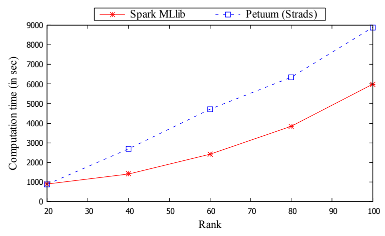


(a) Iteration vs validation RMSE in Petuum

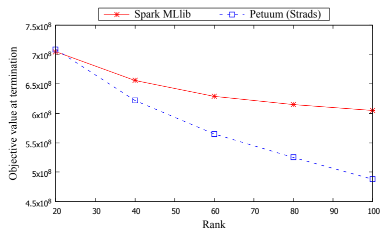


(b) Iteration vs validation RMSE in Spark

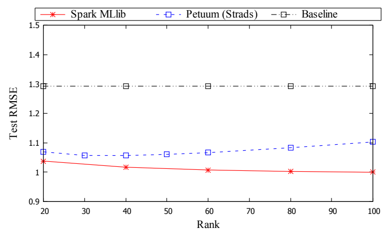
Figure 5: RMSE computed on a separately provided Yahoo music validation dataset for ranks 10 and 20 with $\lambda = 0.05$.



(a) Rank vs computation time



(b) Rank vs objective value at termination



(c) Rank vs test rmse

Figure 6: Performance comparison on full Yahoo music dataset for different ranks with $\lambda = 0.05$.

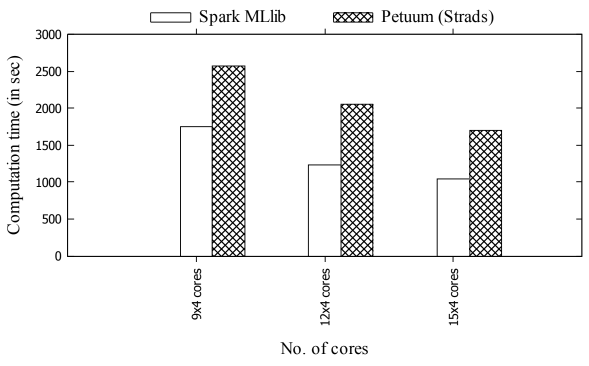


Figure 10: Results comparing computation time on 50% Yahoo music dataset for rank 60 with $\lambda = 0.0001$ and $\epsilon = 0.02$.

this experiment on the Amazon Review dataset using the AWS setup with rank 10. In this study, we compare the storage space required to store the *InBlocks* and *OutBlocks* in the unoptimized version of Spark MLlib (version 1.2) with the block-encoding based implementation, which is available in MLlib from version 1.4 onwards. The corresponding optimization results are reported in Table 1. As observed from the table, block-encoding based implementation can compress *userInBlock* and *itemOutBlock* to nearly 30% of their initial sizes, while *userOutBlock* and *itemInBlock* can be further compressed to 18%.

Table 1: Reduction in Storage Overhead

Block	MLlib 1.2	MLlib 1.4+
userInBlock	941MB	277MB
userOutBlock	355MB	65MB
itemInBlock	1380MB	243MB
itemOutBlock	119MB	37MB

5.5 Real Industrial Deployment

We next present results on an industrial dataset consisting of roughly 50 billion ratings from 50 million users and 5 million items. The matrix factorization implementation on Spark MLlib has been industrially deployed [23] and tested on 32 `r3.8xlarge` nodes (around $10/hr with spot instances). In this setup, it usually takes 1 hour to complete 10 iterations for building a model with rank 10 which consists of 550 million parameters. On average, it usually takes 10 minutes to prepare the *InBlocks* and the *OutBlocks* and then 5 minutes per ALS iteration.

6 CONCLUSION

In this paper, we presented an efficient model parallel algorithm for ALS and discussed its optimized implementation on Apache Spark. This implementation, which is available in the current version of Spark MLlib (version 2.1.0) has been industrially deployed for large-scale collaborative filtering problems. We also benchmarked the performance of our implementation against another optimized implementation developed on the leading open source parameter server framework, Petuum (Strads). Empirical results show that MLlib can actually outperform such asynchronous systems in terms of scalability, accuracy and computation time as the number of parameters increases. In future, we plan to extend the optimization strategies discussed here to develop efficient model parallel algorithms for building linear models on BSP systems and on other architectures like GPUs [30].

ACKNOWLEDGMENT

This work was funded in part via National Science Foundation Award 1555772. We thank our reviewers, Brooke Wenig, Hang Qi and Lisha Li for their thoughtful comments and insights. We also thank the Petuum team for their feedback and assistance with the experimental comparison.

REFERENCES

[1] Kamal Ali and Wijnand van Stam. 2004. TiVo: Making Show Recommendations Using a Distributed Collaborative Filtering Architecture. In *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '04)*. 394–401.

[2] Onkar Bhardwaj and Guojing Cong. 2016. Practical Efficiency of Asynchronous Stochastic Gradient Descent. In *Proceedings of the Workshop on Machine Learning in High Performance Computing Environments (MLHPC '16)*. 56–62.

[3] P. Boldi and S. Vigna. 2004. The Webgraph Framework I: Compression Techniques. In *Proceedings of the 13th International Conference on World Wide Web (WWW '04)*. 595–602.

[4] Joseph K. Bradley, Aapo Kyrola, Danny Bickson, and Carlos Guestrin. 2011. Parallel Coordinate Descent for L1-Regularized Loss Minimization. In *International Conference on Machine Learning (ICML 2011)*.

[5] Cheng-Tao Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary Bradski, Andrew Y. Ng, and Kunle Olukotun. 2006. Map-reduce for Machine Learning on Multicore. In *Proceedings of the 19th International Conference on Neural Information Processing Systems (NIPS'06)*. 281–288.

[6] David Clarke. 2014. *Design and Implementation of Parallel Algorithms for Modern Heterogeneous Platforms Based on the Functional Performance Model.* Ph.D. Dissertation. University College Dublin.

[7] Wei Dai, Abhimanu Kumar, Jinliang Wei, Qirong Ho, Garth Gibson, and Eric P. Xing. 2015. High-performance Distributed ML at Scale Through Parameter Server Consistency Models. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence (AAAI'15)*. 79–87.

[8] Soham De and Tom Goldstein. 2016. Efficient Distributed SGD with Variance Reduction. In *IEEE 16th International Conference on Data Mining (ICDM)*.

[9] Rong Ge, Jason D Lee, and Tengyu Ma. 2016. Matrix Completion has No Spurious Local Minimum. In *Advances in Neural Information Processing Systems 29*, D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett (Eds.). 2973–2981.

[10] Qirong Ho. 2017. Petuum v0.93 Documentation. https://github.com/sailing-pmls/bosen/blob/v0.9.3/Petuumv0.93documentation.pdf/. (2017). [Online; accessed 14-February-2017].

[11] Qirong Ho, James Cipar, Henggang Cui, Jin Kyu Kim, Seunghak Lee, Phillip B. Gibbons, Garth A. Gibson, Gregory R. Ganger, and Eric P. Xing. 2013. More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server. In *Proceedings of the 26th International Conference on Neural Information Processing Systems (NIPS'13)*. 1223–1231.

[12] C. A. R. Hoare. 1961. Algorithm 64: Quicksort. *Commun. ACM* 4, 7 (July 1961).

[13] Petuum Inc. 2017. Petuum Documentation. https://media.readthedocs.org/pdf/petuum/latest/petuum.pdf/. (2017). [Online; accessed 14-February-2017].

[14] Yuchin Juan, Yong Zhuang, Wei-Sheng Chin, and Chih-Jen Lin. 2016. Field-aware Factorization Machines for CTR Prediction. In *Proceedings of the 10th ACM Conference on Recommender Systems (RecSys '16)*. 43–50.

[15] Efthalia Karydi and Konstantinos Margaritis. 2016. Parallel and Distributed Collaborative Filtering: A Survey. *ACM Comput. Surv.* 49, 2 (Aug. 2016), 37:1–37:41.

[16] Janis Keuper and Franz-Josef Pfreundt. 2015. Asynchronous Parallel Stochastic Gradient Descent: A Numeric Core for Scalable Distributed Machine Learning Algorithms. In *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments (MLHPC '15)*. 1:1–1:11.

[17] Jason D Lee, Max Simchowitz, Michael I Jordan, and Benjamin Recht. 2016. Gradient Descent Converges to Minimizers. In *Conference on Learning Theory (COLT 2016)*.

[18] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. 2014. Scaling Distributed Machine Learning with the Parameter Server. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. 583–598.

[19] Mu Li, Ziqi Liu, Alexander J. Smola, and Yu-Xiang Wang. 2016. DiFacto: Distributed Factorization Machines. In *Proceedings of the Ninth ACM International Conference on Web Search and Data Mining (WSDM '16)*. 377–386.

[20] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. 2012. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *Proc. VLDB Endow.* 5, 8 (April 2012), 716–727.

[21] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A System for Large-scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD '10)*. 135–146.

[22] Julian McAuley, Christopher Targett, Qinfeng Shi, and Anton van den Hengel. 2015. Image-Based Recommendations on Styles and Substitutes. In *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '15)*. 43–52.

[23] Xiangrui Meng. 2017. Implementing Large-Scale Matrix Factorization on Apache Spark. In *AAAI 2017 Workshop on Distributed Machine Learning.*

[24] Xiangrui Meng, Joseph K. Bradley, Burak Yavuz, Evan R. Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, D. B. Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. 2015. MLlib: Machine Learning in Apache Spark. *CoRR* abs/1505.06807 (2015).

[25] Jinoh Oh, Wook-Shin Han, Hwanjo Yu, and Xiaoqian Jiang. 2015. Fast and Robust Parallel SGD Matrix Factorization. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '15)*. 865–874.

[26] Anand Rajaraman and Jeffrey David Ullman. 2011. *Mining of Massive Datasets.*

[27] Sebastian Schelter, Christoph Boden, Martin Schenck, Alexander Alexandrov, and Volker Markl. 2013. Distributed Matrix Factorization with Mapreduce Using a Series of Broadcast-joins. In *Proceedings of the 7th ACM Conference on Recommender Systems (RecSys '13)*. 281–284.

[28] Gábor Takács, István Pilászy, Bottyán Németh, and Domonkos Tikk. 2008. Matrix Factorization and Neighbor Based Algorithms for the Netflix Prize Problem. In *Proceedings of the 2008 ACM Conference on Recommender Systems (RecSys '08)*. 267–274.

[29] Gábor Takács and Domonkos Tikk. 2012. Alternating Least Squares for Personalized Ranking. In *Proceedings of the Sixth ACM Conference on Recommender Systems (RecSys '12)*. 83–90.

[30] Wei Tan, Liangliang Cao, and Liana Fong. 2016. Faster and Cheaper: Parallelizing Large-Scale Matrix Factorization on GPUs. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC '16)*. 219–230.

[31] Wikipedia. 2017. Jblas: Linear Algebra for Java. https://en.wikipedia.org/wiki/Jblas:_Linear_Algebra_for_Java. (2017). [Online; accessed 14-February-2017].

[32] Reynold Xin. 2017. Apache Spark the fastest open source engine for sorting a petabyte. https://databricks.com/blog/2014/10/10/spark-petabyte-sort.html. (2017). [Online; accessed 14-February-2017].

[33] Eric P. Xing, Qirong Ho, Wei Dai, Jin-Kyu Kim, Jinliang Wei, Seunghak Lee, Xun Zheng, Pengtao Xie, Abhimanu Kumar, and Yaoliang Yu. 2015. Petuum: A New Platform for Distributed Machine Learning on Big Data. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '15)*. 1335–1344.

[34] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. 2015. Effective Techniques for Message Reduction and Load Balancing in Distributed Graph Computation. In *Proceedings of the 24th International Conference on World Wide Web (WWW '15)*. 1307–1317.

[35] Hsiang-Fu Yu, Cho-Jui Hsieh, Si Si, and Inderjit Dhillon. 2012. Scalable Coordinate Descent Approaches to Parallel Matrix Factorization for Recommender Systems. In *Proceedings of the 2012 IEEE 12th International Conference on Data Mining (ICDM '12)*. 765–774.

[36] Hsiang-Fu Yu, Cho-Jui Hsieh, Si Si, and Inderjit S. Dhillon. 2014. Parallel Matrix Factorization for Recommender Systems. *Knowl. Inf. Syst.* 41, 3 (Dec. 2014), 793–819.

[37] Erheng Zhong, Yue Shi, Nathan Liu, and Suju Rajan. 2016. Scaling Factorization Machines with Parameter Server. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management (CIKM '16)*. 1583–1592.

[38] Yunhong Zhou, Dennis Wilkinson, Robert Schreiber, and Rong Pan. 2008. Large-Scale Parallel Collaborative Filtering for the Netflix Prize. In *Proceedings of the 4th International Conference on Algorithmic Aspects in Information and Management (AAIM '08)*. 337–348.