Chapter 11

Scalability and Distribution of Collaborative Recommenders

Evangelia Christakopoulou
Computer Science & Engineering
University of Minnesota, Minneapolis, MN
evangel@cs.umn.edu

Shaden Smith
Computer Science & Engineering
University of Minnesota, Minneapolis, MN
shaden@cs.umn.edu

Mohit Sharma
Computer Science & Engineering
University of Minnesota, Minneapolis, MN
mohit@cs.umn.edu

Alex Richards
Department of Computer Engineering
San José State University
alexander.richards@sjsu.edu

David Anastasiu
Department of Computer Engineering
San José State University
david.anastasiu@sjsu.edu

George Karypis
Computer Science & Engineering
University of Minnesota, Minneapolis, MN
karypis@cs.umn.edu

Recommender systems are ubiquitous; they are foundational to a wide variety of industries ranging from media companies such as Netflix to e-commerce companies such as Amazon. As recommender systems continue to permeate the marketplace, we observe two major shifts which must be addressed. First, the amount of data used to provide quality recommendations grows at an unprecedented rate. Secondly, modern computer architectures display great processing capabilities that significantly outpace memory speeds. These two trend shifts must be taken into account in order to design recommendation systems that can efficiently handle the amount of available data by distributing computations in order to take advantage of modern parallel architectures. In this chapter, we present ways to scale popular collaborative recommendation methods via parallel computing.

11.1. Introduction

Recommender systems are ubiquitous; they are foundational to a wide variety of industries ranging from media companies such as Netflix to e-commerce companies such as Amazon. Their popularity is attributed to their ability to effectively navigate users through a plethora of product options which would otherwise go unexplored. As recommender systems continue to permeate the marketplace, we observe two major shifts which must be addressed.

First, the amount of data used to provide quality recommendations grows at an unprecedented rate. For example, companies such as Netflix stream millions of movies each day. Secondly, modern computer architectures forego great changes. The last two decades have seen available processing capabilities significantly outpace memory speeds. This disparity has shifted the cost of many computations from mathematical operations to data movements. In consequence, algorithm designers must now expose large amounts of parallelism while reducing data movement costs. These two trend shifts must be taken into account in order to design recommendation systems that can efficiently handle the amount of available data by distributing computations in order to take advantage of modern parallel architectures.

Unfortunately, designing successful recommendation systems that can effectively utilize modern parallel architectures is not always straightforward. Most popular recommendation algorithms are inherently unstructured, meaning that data accesses are not known apriori because they are determined by the structure of the input data. The unstructured nature of

the underlying computations is further complicated by non-uniform distributions exhibited by real-world ratings datasets. A small number of popular items and prolific users cause the data to follow a power-law distribution, which presents challenges when partitioning work to processing cores in a balanced manner.

In this chapter, we present ways to scale popular collaborative recommendation methods via parallel computing. The methods we present span both of the primary recommendation tasks: the top-N recommendation task, where we are interested in whether a user will likely purchase an item, and the rating prediction task, which focuses on determining how much a user would enjoy a product. The nearest neighbor methods presented in Section 11.2 are used for both tasks. The sparse linear methods presented in Section 11.3 are oriented towards the top-N recommendation task. Finally, the matrix and tensor factorization methods presented in Section 11.4 can be used for both tasks, but for the purposes of this chapter, they are primarily viewed in the context of rating prediction.

The presented methods are very different from each other and exhibit different parallelization opportunities. For each method, an overview is presented, along with a discussion of how it can be scaled and experimental results showing the runtimes and speedup achieved on the MovieLens 10M (ML10M) dataset.

The rest of the chapter has the following structure: Subsections 11.1.1 and 11.1.2 present the notation and evaluation methodology used in the rest of the chapter. Section 11.2 presents methods to efficiently identify neighbors in the nearest-neighbor approaches. Section 11.3 describes sparse linear methods, where both the neighbors to a target item and their similarities are estimated through solving an optimization problem and discusses their scalability. Finally, Section 11.4 gives an overview and discusses the efficiency of matrix and tensor factorization approaches.

11.1.1. *Notation*

In the rest of the chapter, we will use bold capital letters to refer to matrices (e.g., $\mathbf{A}$) and bold lower case letters to refer to vectors (e.g., $\boldsymbol{p}$). The vectors are assumed to be column vectors. Thus, $\boldsymbol{a}_i$ refers to the ith column of matrix $\mathbf{A}$ and we will use the transpose ($\boldsymbol{a}_i^T$) to describe row vectors.

We note the rating matrix, which contains the feedback given by n users to m items as $\mathbf{R}$. The dimensions of $\mathbf{R}$ are $n \times m$. We will use the term u to note a user, and i to note an item. The rating given by a user u to

an item i will be noted by r_{ui} and with a slight abuse of notation will be used to show both the explicit rating that user u gave to item i and/or the implicit feedback (purchase/like) from u to i. We will use the notation $\hat{r}_{ui}$ to refer to the predicted rating. Finally, we use the notation $|\cdot|$ to refer to the number of non-zeros in the corresponding matrix or vector.

11.1.2. *Evaluation*

Dataset Throughout this chapter, we will demonstrate the efficiency and effectiveness of different recommender methods using the MovieLens 10 Million (ML10M) [Harper and Konstan (2015)] ratings dataset. ML10M consists of 10 million ratings provided to 10677 movies by 69878 users. Each rating is accompanied by a timestamp. The timestamps span 158 months and are used in tensor factorization approaches in Section 11.4.

Evaluation In order to evaluate the performance of the methods, we employ a leave-one-out validation scheme. For every user, we leave out one rating and we train the model on the rest of the user' ratings. All the left-out ratings comprise the test set. We repeat this procedure three times, thus resulting in three folds (three train sets and three associated test sets). In the end, for every method, we report the average time taken and the average performance of the folds.

As the exact rating is not used in top-N methods, the implicit feedback of the ML10M dataset is used instead in Section 11.3. In this scenario, non-zero rating values in $\mathbf{R}$ are replaced with 1s, signifying the existence of a rating given by a user to an item in the original data. To evaluate top-N recommendation methods, we need to investigate whether the item of the user that belongs to the test set is included in the list of top-N recommendations to the user, and in which position. Therefore, we use two performance metrics: HR and ARHR, where:

$$HR = \frac{\#hits}{\#users}, \tag{11.1}$$

and

$$ARHR = \sum_{i=1}^{\#hits} \frac{1}{p_i}. \tag{11.2}$$

The $\#hits$ denotes the number of times that the test items were included in the top-N recommendation list and p_i is the position of the test item in the recommended list, with $p_i = 1$ being the top position. In this chapter,

we have evaluated the top-N recommendation methods by computing HR and ARHR for $N = 10$.

In the rating prediction methods, the explicit ratings are used. Also, when presenting the tensor factorization methods in Section 11.4, the associated timestamps are used beyond the ratings. To evaluate the rating prediction methods, we need to see how similar or different the predicted values of the ratings are with the actual ratings. We employ the Root Mean Squared Error (RMSE) to do that:

$$RMSE = \sqrt{\frac{\sum_u \sum_i (r_{ui} - \hat{r}_{ui})^2}{|\mathbf{R}|}}. \tag{11.3}$$

More details on RMSE, HR, ARHR and other evaluation measures can be found in Chapter 9. A thorough explanation of the difference between explicit and implicit feedback can be found in Chapter 7.

System characteristics For consistency and comparison purposes, all experiments are executed on the same system[1] consisting of identical nodes equipped with 64 GB RAM and two twelve-core 2.5 GHz Intel Xeon E5-2680v3 (Haswell) processors. Each core is equipped with 32 KB L1 cache and 256 KB of L2 cache, and the 12 cores on each socket share 30 MB of L3 cache.

11.2. Scaling up nearest-neighbor approaches

A number of recommender systems rely on finding nearest neighbors among users or items as an integral part of deriving recommendations or training a predictive recommendation model. More details on nearest-neighbor approaches can be found in Chapter 1. A neighbor is defined as a user/item who is similar to the target user/item, based on a similarity notion (e.g. the target user and the neighbor user have co-rated a lot of items). The symbol $\mathcal{N}_i(u)$ represents the set of neighbors of the item i rated by the user u. Similarly, the symbol $\mathcal{N}_u(i)$ represents the set of neighbors of user u, who have rated item i. In this section, we discuss approaches to speed up neighborhood identification, which directly affects recommendation efficiency.

[1] https://www.msi.umn.edu/content/mesabi

11.2.1. *Use of neighborhoods in Recommender Systems*

Collaborative filtering based recommenders, such as the user-based neighborhood method [Konstan *et al.* (1997)] or item-based neighborhood methods [Sarwar *et al.* (2001); Deshpande and Karypis (2004)], first identify a set of neighbors and then use the ratings associated with those neighbors to derive the predicted rating for the user in question. User-based methods identify a set of users similar to target user u and predict the rating of user u on item i as

$$\hat{r}_{ui} = \mu_u + \frac{\sum_{v \in \mathcal{N}_u(i)} \text{sim}(u,v)(r_{vi} - \mu_v)}{\sum_{v \in \mathcal{N}_u(i)} \text{sim}(u,v)},$$

where $\text{sim}(u,v)$ denotes the similarity between the users u and v, and μ_u and μ_v are the means of the ratings provided by users u and v, respectively.

Item-based neighborhood methods, however, derive the rating of user u on target item i by considering other items that have been rated (generally high) by the user u. The predicted rating of u on i is given by

$$\hat{r}_{ui} = \mu_i + \frac{\sum_{j \in \mathcal{N}_i(u)} \text{sim}(i,j)(r_{uj} - \mu_j)}{\sum_{j \in \mathcal{N}_i(u)} \text{sim}(i,j)}.$$

where $\text{sim}(i,j)$ denotes the similarity between the items i and j, and μ_i and μ_j are the means of the ratings received by items i and j, respectively.

A number of different recommenders can be designed given different choices in applying the formulas above with respect to user and item representations, similarity function, or neighborhood construction. The standard approach is to represent user u as the sparse vector of ratings for items rated by the user (row u in the ratings matrix $\mathbf{R}$) and item i as the vector of all ratings given to item i by users (column i in $\mathbf{R}$). Given a vector representation of users and items, the similarity between users or between items is most often computed as the cosine similarity or Pearson correlation coefficient between their respective vectors. Finally, the neighborhoods $\mathcal{N}_u(i)$ and $\mathcal{N}_i(u)$ can be constructed by finding all neighbors with a similarity above some minimum threshold ϵ, or one may consider only the k closest neighbors to the target user or item.

Beyond deriving recommendations by relying on similar users or items, some optimization-based recommenders learn a recommendation model by focusing only on the ratings of similar users or items during the learning process (e.g., the fs-SLIM model [Ning and Karypis (2011)]).

Naïve approaches will compare each user to every other user, thus leading to quadratic complexity in the number of computed similarities. In the remainder of this section, we will discuss efficient methods that identify nearest neighbors given user-item ratings. The methods rely on aggressive pruning of the search space by identifying pairs of users or items that cannot be similar enough based on theoretic upper bounds on their computed similarity. Additional performance gains are achieved via efficient use of the memory hierarchy of modern computing systems and shared-memory parallelism.

We focus our discussion on two nearest neighbor problems useful in the recommender systems context. The ϵ-nearest neighbor graph (ϵNNG) construction problem, also known as *all-pairs similarity search* (APSS) or *similarity join*, identifies, for each user/item in a set, all other users/items with a similarity of at least ϵ. On the other hand, the k-nearest neighbor graph (kNNG) construction constrains each identified neighborhood to the k users/items closest to the target user/item. To simplify the discussion, we will describe the methods in the context of constructing item neighborhoods. The same methods can be applied to find user-based neighbors.

11.2.2. *ϵ-nearest neighbor graph construction*

Recently, several methods have been proposed that efficiently construct the ϵNNG by filtering (or ignoring) pairs of items that cannot be neighbors [Bayardo *et al.* (2007); Anastasiu and Karypis (2014, 2015b); Anastasiu (2017)]. Item rating profile vectors are inherently sparse, as few users may consume and rate each item. The proposed methods take advantage of this sparsity and use data structures and processing strategies designed to eliminate unnecessary memory accesses and multiplication operations. The L_2-norm All Pairs (L2AP) [Anastasiu and Karypis (2014)] and Parallel L_2-norm All Pairs (pL2AP) [Anastasiu and Karypis (2015b)] methods construct an *exact* neighborhood graph, finding the same neighbors as those found by a brute-force method that compares each user/item against all other users/items. Cosine Approximate Nearest Neighbors (CANN) [Anastasiu (2017)], on the other hand, finds most but not necessarily all of the neighbors with a similarity of at least ϵ.

These methods use an *inverted index* data structure to eliminate unnecessary comparisons. The inverted index is represented by the sparse user rating profiles. It consists of a set of lists, one for each user, such that the uth list contains pairs (i, r_{ui}) for all items i that have a non-zero r_{ui}

rating. Many unnecessary memory accesses and similarity computations can be avoided by only comparing an item against the set of items found in the inverted index lists for the users that rated it. In this way, two items that have not been rated by any common user will never have their similarity computed.

Additional savings can be achieved by taking advantage of the input similarity threshold ϵ. Note that cosine similarity measures the cosine of the angle between the two vectors and is thus independent of the vector lengths. A standard preprocessing step in computing cosine similarity is to normalize the vectors with respect to their L_2-norm, which reduces computing the cosine similarity of two vectors to finding their dot-product. The methods compute only part of the dot-product of profile vectors for most pairs of items, e.g., using only the tail-end features in the profile vector. Several theoretic upper bounds of vector dot-products are used to estimate the portion of the dot-product for the leading features. If the sum of the estimate and computed portions of the dot-product is below ϵ, the items cannot be similar enough and are pruned.

Many item comparisons are completely avoided through a *partial indexing* strategy. Only a few of the leading features of the profile of an item i are indexed, enough to ensure that any item j with a similarity of at least ϵ will be found by traversing the partial index. This strategy leads to a two phase process for constructing the exact ϵNNG. First, partial similarities (dot-products) are computed using values stored in the inverted index lists for users that rated item i, which are called *candidates*. In the second phase, the un-indexed portion of each of the candidate profile vectors is used to finish computing similarities only for those items with non-zero similarity after the first phase. In both phases, additional similarity upper-bounds are used to eliminate candidates that cannot be similar enough.

Parallelization of pL2AP focuses on a cache-tiling strategy that aims to fit critical data structures used during similarity search in the high-speed yet limited cache memory of the system. The method splits the set of items such that each subset has a partial inverted index that can fit in cache memory. Each core is then assigned small sets of neighborhood searches for 20 consecutive items, which could be independently executed. Additionally, a small-memory footprint hash table data structure is proposed which is uniquely suited to the memory access patterns in pL2AP and provides fast access to profile vector values and meta-data necessary for computing similarity upper bounds. Algorithm 1 provides a sketch of the pL2AP processing pipeline. Additional details for the algorithm and the different

Algorithm 1 pL2AP

```
1: Normalize profiles for every item i.
2: for all items i in parallel do
3:    Identify prefix to be indexed.
4: end for
5: Split items to index into tiles based on cumulative prefix size.
6: for all index tiles in parallel do
7:    Create partial inverted index for assigned tile items.
8: end for
9: for all index tiles do
10:   for all items not in already processed index tiles in parallel do
11:      Use partial inverted index to identify candidate neighbors.
12:      Filter some candidates based on similarity upper-bound estimates.
13:      Use un-indexed portion of candidate vectors to finish computing
         their similarity, while continuing to filter those with estimates be-
         low ϵ.
14:      Output un-filtered candidates with similarity ≥ ϵ.
15:   end for
16: end for
```

similarity upper-bounds used in the filtering process in pL2AP can be found in [Anastasiu and Karypis (2014, 2015b); Anastasiu (2017)].

Table 11.1 and Figure 11.1 show the runtime and parallel speedups of pL2AP, when building the ϵNNG for items in the ML10M training datasets, given ϵ ranging between 0.1 and 0.9. The method pL2AP is compared against pij, a baseline that uses similar cache-tiling strategies as pL2AP but does not prune the search space. Instead, it computes similarities for all items co-rated by at at least one user. The left graph of Figure 11.1 shows execution times in seconds, averaged over all three training folds, while the right one shows strong scaling results for the two methods, measuring the speedup of each method against their own serial execution. Strong scaling is when the problem size remains the same but the amount of parallelism increases. By effectively eliminating unnecessary similarity computations, pL2AP is able to achieve 4.24x–29.27x speedup over pij for different similarity thresholds ϵ.

Table 11.1. Runtime and speedup of pL2AP over pij on the ML10M dataset, when executed using 24 cores.

Method	$\epsilon = 0.1$	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
	time (s)								
pij	3.22	3.22	3.22	3.22	3.22	3.22	3.22	3.23	3.22
pL2AP	0.76	0.55	0.42	0.34	0.26	0.21	0.15	0.13	0.11
	speedup								
pL2AP	4.24	5.85	7.67	9.38	12.38	15.60	21.47	24.82	29.27

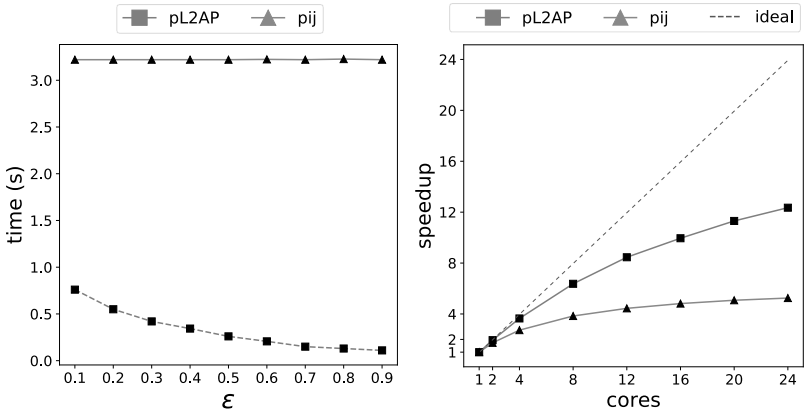


Fig. 11.1. Runtime (left) and strong scaling (right) of pL2AP and the naïve baseline pij when executed on the ML10M dataset.

11.2.3. *k-nearest neighbor graph construction*

One potential problem with using the ϵNNG to derive recommendations is that, given a high enough value for ϵ, some neighborhoods may not contain any neighbors. The kNNG provides a guaranteed estimate of local preferences by retrieving the k nearest neighbors for each item in the set. L2Knng [Anastasiu and Karypis (2015a)] and pL2Knng [Anastasiu and Karypis (2016)] have been proposed for the purpose of efficiently constructing the *exact* kNNG. The main idea in L2Knng is to bootstrap the similarity search with a quickly constructed approximate graph. The minimum similarities in the approximate neighborhoods can then be used as filtering criteria in a search framework similar to the one in L2AP.

In the first phase of constructing the kNNG, L2Knng efficiently finds most, but not necessarily all of the k items closest to each target item, heuristically choosing a finite set of comparison items that are likely to be in the exact neighborhood. First, L2Knng identifies items that have high-value ratings in common with the target item, building an initial approximate kNNG. This graph is then iteratively improved by looking for neighbors among the neighbors of current neighbors. Finally, a filtering framework similar to the one described in Section 11.2.2 is employed to construct the exact kNNG. Unlike L2AP, L2Knng does not have an input threshold ϵ that could be used for pruning. Instead, it relies on the idea that any item that has the potential to be in the exact neighborhood of the target must have a similarity greater than the minimum similarity of the target with any item in its current approximate neighborhood. These minimum neighborhood similarities are used to forgo most of the item pair comparisons.

Similar to pL2AP, parallelization in pL2Knng focuses on cache-tiling and strategies for maximizing load balance among the cores. Unlike pL2AP, neighborhood searches are not independent. Given that cosine similarity is commutative, a neighbor j that the method finds for an item i could also benefit from the search if item i is not yet in j's neighborhood and the similarity between i and j is greater than the minimum neighborhood similarity in j's neighborhood. A lock-free update strategy is used for the in-memory shared neighborhood graph to address the potential resource contention encountered when items i and j are being processed by different cores. Algorithm 2 provides a high-level sketch of the pL2Knng method. Additional details regarding the initial approximate graph construction and filtering used to build the exact kNNG solution can be found in [Anastasiu and Karypis (2015a, 2016); Anastasiu (2017)].

Table 11.2 and Figure 11.2 show the efficiency of the parallel method, pL2Knng, when building the kNNG for items in the ML10M training datasets, given k ranging between 5 and 50. Our method, pL2Knng, is compared against pkij, a similar baseline to pij that uses similar cache-tiling strategies but does not prune the search space. The left graph of Figure 11.2 shows execution times in seconds, averaged over all three training folds, while the right one shows strong scaling results for the two methods, measuring the speedup of each method against their own serial execution. By effectively eliminating unnecessary similarity computations, pL2Knng is able to achieve 2.2x–2.97x speedup over pkij for different k values. Given larger datasets, such as one containing 1M pages from the English Wikipedia Web site, containing almost half a billion non-zero

Algorithm 2 pL2Knng

1: Normalize profiles for every item i.
2: **for all** items i **in parallel do**
3: Choose $\mu \geq k$ candidates highly co-rated with i by some user.
4: Compute candidate similarities and keep top-k candidates.
5: **end for**
6: **for all** items i **in parallel do**
7: Choose μ candidates from the neighborhoods of the current k neighbors.
8: Compute candidate similarities and update ith neighborhood as necessary.
9: **end for**
10: Use minimum neighborhood similarities to define partial inverted index tiles.
11: **for all** index tiles **do**
12: **for all** items i not in already processed index tiles **in parallel do**
13: Use partial inverted index to identify candidate neighbors.
14: Filter some candidates based on similarity upper-bound estimates.
15: Use un-indexed portion of candidate vectors to finish computing their similarity, while continuing to filter those with estimates below minimum similarities in the candidate or i's neighborhoods.
16: Update neighborhoods of i and un-filtered candidates as necessary.
17: **end for**
18: **end for**

values, pL2Knng has been shown to outperform pkij by 7.3x–11.5x for $k \leq 50$ [Anastasiu and Karypis (2015b)]. The results show the value of pruning the search space as a means to improve the efficiency of nearest neighbor identification.

Table 11.2. Execution times and speedup of pL2Knng over pkij on the ML10M dataset, when executed using 24 cores.

Method	k=5	10	20	30	40	50
	time (s)					
pKij	3.23	3.235	3.237	3.198	3.24	3.247
pL2Knng	1.089	1.136	1.228	1.324	1.408	1.479
	speedup					
pL2Knng	2.97	2.85	2.64	2.42	2.3	2.2

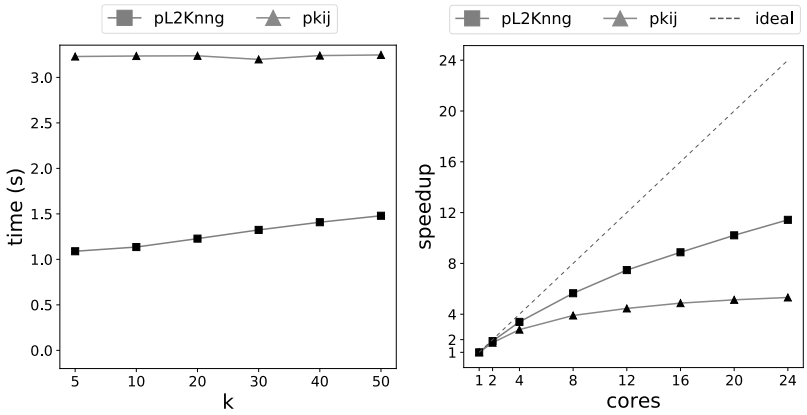


Fig. 11.2. Runtime (left) and strong scaling (right) of pL2Knng and the naïve baseline pkij when executed on the ML10M dataset.

11.3. Efficiently Estimating Item-Item Similarities by solving an Optimization Problem

11.3.1. *Sparse LInear Methods for Top-N Recommendation (SLIM)*

In contrast to the standard item-based methods which use a predefined similarity measure like cosine or Pearson correlation, Sparse LInear Methods (SLIM) [Ning and Karypis (2011)] learn the item-item relationships from the user-item feedback matrix $\mathbf{R}$ instead. SLIM is a popular method for top-N recommendation, as it has been shown to provide high-quality recommendations [Ning and Karypis (2011)]. In SLIM, the rating for an item is predicted as a sparse aggregation of the existing ratings provided

by the user:

$$\hat{r}_{ui} = \boldsymbol{r}_u^T \boldsymbol{s}_i,$$

where $\boldsymbol{r}_u^T$ is the uth row of the rating matrix $\mathbf{R}$ and $\boldsymbol{s}_i$ is a sparse vector containing non-zero aggregation coefficients over all items. The sparse aggregation coefficient matrix $\mathbf{S}$ of size $m \times m$, capturing the item-item relationships is estimated by solving the following optimization problem:

$$\begin{aligned} \underset{\mathbf{S}}{\text{minimize}} \quad & \frac{1}{2}||\mathbf{R} - \mathbf{RS}||_F^2 + \frac{\beta}{2}||\mathbf{S}||_F^2 + \lambda||\mathbf{S}||_1 \\ \text{subject to} \quad & \mathbf{S} \geq 0 \\ & diag(\mathbf{S}) = 0. \end{aligned} \tag{11.4}$$

The optimization problem of Equation 11.4 tries to minimize the training error, denoted by $||\mathbf{R} - \mathbf{RS}||_F^2$, while also regularizing the matrix $\mathbf{S}$. The problem uses two regularizers. The first one is the Frobenius norm of the matrix $\mathbf{S}$ (noted by $||\mathbf{S}||_F^2$), which is controlled by the parameter β, in order to prevent overfitting. The other regularizer is the l_1 norm of the matrix $\mathbf{S}$ (noted by $||\mathbf{S}||_1$), which is controlled by the parameter λ, in order to promote sparsity [Tibshirani (1996)]. Larger values of β and λ leads to more severe regularization. The use of both l_F and l_1 regularization makes the optimization problem of Equation 11.4 an elastic net problem [Zou and Hastie (2005)].

The non-negativity constraint on $\mathbf{S}$ imposes the item-item relations to be positive. The constraint diag($\mathbf{S}$) = 0 is added to avoid trivial solutions (e.g., $\mathbf{S}$ corresponding to the identity matrix) and ensure that r_{ui} is not used to compute $\hat{r}_{ui}$.

11.3.1.1. *Parallelizing SLIM*

Equation 11.4 can be accelerated by learning similarities in parallel for every target item i, as every column of $\mathbf{S}$ can be learned independently from the other columns. Then the optimization problem of Equation 11.4 changes to a set of optimization problems of the form:

$$\begin{aligned} \underset{\boldsymbol{s}_i}{\text{minimize}} \quad & \frac{1}{2}||\boldsymbol{r}_i - \mathbf{R}\boldsymbol{s}_i||_2^2 + \frac{\beta}{2}||\boldsymbol{s}_i||_2^2 + \lambda||\boldsymbol{s}_i||_1 \\ \text{subject to} \quad & \boldsymbol{s}_i \geq 0 \\ & s_{ii} = 0, \end{aligned}$$

which allows us to estimate the ith column of $\mathbf{S}$, noted by $\boldsymbol{s}_i$. The term $\boldsymbol{r}_i$ refers to the ith column of the training matrix $\mathbf{R}$. The problem is solved with the use of coordinate descent and soft thresholding [Friedman *et al.* (2010)].

The software implementation of SLIM provided by the author Xia Ning[2] utilizes the property that different columns of the sparse aggregation coefficient matrix can be solved independently and allows the users to specify which columns of the sparse aggregation coefficient matrix they would like to estimate. The software is implemented with the use of the Bound Constrained Least Squares (BCLS) library[3].

In order to fully utilize the benefits from the parallel estimation of different columns of $\mathbf{S}$, we use a multithreaded implementation of SLIM which relies on OpenMP. This allows us to have parallelism within a multi-core node. Each thread is assigned a set of columns i and estimates the associated sparse aggregation coefficient vectors $\boldsymbol{s}_i$. After all the threads have estimated the set of $\boldsymbol{s}_i$ vectors, the vectors are combined into the overall sparse aggregation coefficient matrix $\mathbf{S}$. We will refer to the multithreaded implementation of SLIM, as mt-SLIM. Figure 11.3 shows the speedup achieved by mt-SLIM on the ML10M dataset, with respect to the serial runtime (cores = 1). The results shown correspond to the time taken for model estimation and they correspond to the average of three folds.

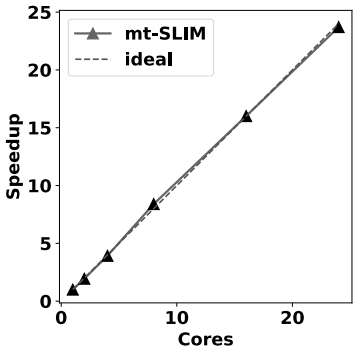


Fig. 11.3. The speedup achieved by mt-SLIM on the ML10M dataset, while increasing the number of cores (strong scaling).

[2]http://www-users.cs.umn.edu/~xning/slim/html/
[3]http://www.cs.ubc.ca/~mpf/bcls/index.html

As both the rating matrix and the estimated sparse aggregation coefficient matrix are sparse, they are stored in CSR (Compressed Sparse Row) format, in which three one-dimensional arrays are stored, that contain the non-zero values, with their associated row and column indices.

11.3.1.2. *Accelerating the training time of SLIM during parameter search*

In order to find the pair of regularization parameters β and λ that give the best results, a parameter search needs to be conducted. However, the number of models to estimate increases quadratically with the number of values of the regularization parameters β and λ explored. In order to be able to estimate the models more efficiently, mt-SLIM utilizes 'warm-start'. This means that with the exception of the model estimated with the very first choice of parameters, every subsequent model is initialized with the previous model estimated with a different choice of regularization parameters, instead of being initialized with zero values.

Figure 11.4 compares the time spent by mt-SLIM without initialization and mt-SLIM with warm start, for the same number of cores and and for the same choice of regularization parameters ($\beta = 10$, $\lambda = 1$). We can see that mt-SLIM with warm start is on average 15 times faster than mt-SLIM with no initialization.

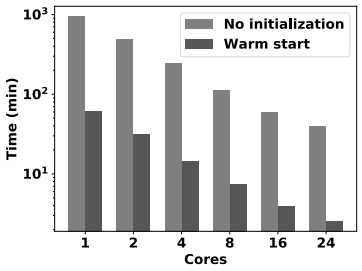


Fig. 11.4. The time in minutes achieved by mt-SLIM with and without warm start on the ML10M dataset for $\beta = 10$ and $\lambda = 1$, while increasing the number of cores (strong scaling).

By evaluating the performance of mt-SLIM with no initialization and with warm start, we get the same performance results, which shows that with warm start, we gain in estimation times, without compromising the quality of the performance.

11.3.2. *Global and Local Sparse LInear Methods for Top-N Recommendation (GLSLIM)*

A limitation of SLIM is that it estimates only a single model for all the users. In many cases, there are differences in users' behavior, who can have diverse preferences. These cannot be captured by a single model. Recently, GLSLIM [Christakopoulou and Karypis (2016)] was proposed, which utilizes both user-subset specific models and a global model, and was shown to improve the top-N recommendation quality. The models, (which are estimated with SLIM) are jointly optimized and combined in a personalized way. Also, GLSLIM automatically identifies the appropriate user subsets. If we note the global model as $\mathbf{S}$ and the local user-subset specific models as $\mathbf{S}^{p_u}$, where $p_u \in \{1..k\}$ denotes the user subset, then the predicted rating of user u, who belongs to subset p_u for item i, will be estimated as:

$$\hat{r}_{ui} = \sum_{l \in R_u} g_u s_{li} + (1 - g_u) s_{li}^{p_u}, \tag{11.5}$$

where s_{li} shows the global item-item similarity between the lth item rated by the user u and the target item i and $s_{li}^{p_u}$ shows the p_u user-subset specific similarity between the lth rated item by u and the target item i. The term g_u is the personalized weight which controls the interplay between the global and the local model and ranges between 0 and 1.

GLSLIM is an iterative algorithm which jointly optimizes the global and local models, the user assignment and the personalized weights. The global and local models are estimated by solving an elastic net optimization problem. Following SLIM, GLSLIM can estimate the columns of its global and local models independent of the other columns. Separate regularization is enforced on the global and on the local models, in order to allow more flexibility in model estimation: we thus have the global l_2 regularization parameter β_g, the global l_1 regularization parameter λ_g, the local l_2 regularization parameter β_l and the local l_1 regularization parameter λ_l. Initially, the users are assigned to clusters. In each iteration, every user is assigned to the subset that resulted in the smallest training error, and his personalized weight is updated accordingly. The models and the user assignment with the personalized weights are updated iteratively, until convergence (the algorithm converges when the users switching subsets are less than one percent). An overview of the algorithm is shown in Algorithm 3.

Algorithm 3 GLSLIM

1: Assign $g_u = 0.5$, to every user u.
2: Compute the initial clustering of users with CLUTO[4].
3: **while** number of users who switched clusters $> 1\%$ of the total number of users **do**
4: Estimate $\mathbf{S}$ and $\mathbf{S}^{p_u}$, $\forall p_u \in \{1, \ldots, k\}$. The estimation is initialized in all iterations but the first one with the corresponding matrices $\mathbf{S}$ and $\mathbf{S}^{p_u}$, $\forall p_u \in \{1, \ldots, k\}$ computed in the previous iteration.
5: **for all** user u **do**
6: **for all** cluster p_u **do**
7: Compute g_u for cluster p_u by minimizing the squared error.
8: Compute the training error.
9: **end for**
10: Assign user u to the cluster p_u that has the smallest training error and update g_u to the corresponding one for cluster p_u.
11: **end for**
12: **end while**

After having completed the training, the top-N recommendation is performed in the following way: for user u, the ratings of all the unrated items i are estimated with Equation 11.5, and the items with the N highest ratings are recommended to the user.

11.3.2.1. *Parallelizing GLSLIM*

We can see from Algorithm 3 that every iteration has two parts: estimating the global and local models (line 4) and user refinement (lines $5-11$). Both parts allow for parallelization, each in its own way. The model estimation part can be parallelized with respect to the items, since every column of the models can be estimated independently of the others. The user refinement part can be parallelized with respect to the users, as provided the models are fixed, the assignment and personalized weight of each user does not depend on the other users.

Taking advantage of these parallelization opportunities, there is an MPI-based GLSLIM software[5], which we use for our subsequent experiments. GLSLIM relies on MPI, instead of OpenMP which was used for mt-SLIM, as it requires more computations than SLIM. SLIM solves one elastic net

[4] http://glaros.dtc.umn.edu/gkhome/cluto/cluto/overview
[5] http://www-users.cs.umn.edu/~evangel/code.html

problem for the whole training matrix $\mathbf{R}$, while GLSLIM is iterative and in each iteration, a new elastic net problem is solved for the global matrix and for all user subsets. Thus, the distributed framework MPI is employed, which allows model estimation and user refinement to be done in a distributed way, thus taking advantage of multiple nodes (where each node consists of cores).

Figure 11.5 shows the speedup achieved by GLSLIM on different nodes, with respect to the time taken by GLSLIM on one node (which consists of 24 cores in our shown results), for the ML10M dataset.

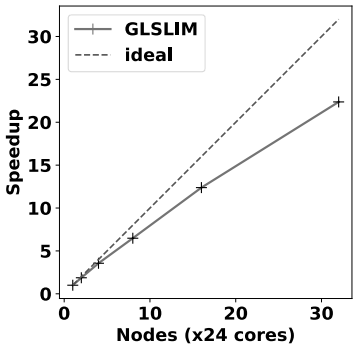


Fig. 11.5. The speedup achieved by GLSLIM on the ML10M dataset, while increasing the number of nodes. The speedup is computed with respect to the time of running GLSLIM on one node.

11.3.2.2. *Accelerating the training time of GLSLIM during parameter search*

GLSLIM has many parameters, for which a parameter search needs to be conducted in order to find the set of them that gives the best performance: the regularization parameters β_g, λ_g, β_l, λ_l, and the number of user subsets k. We can see that the cost of finding the best possible set of parameters increases exponentially. It is thus crucial to be able to run GLSLIM as efficiently as possible.

In order to do so, we again employ warm start. This is done in the following way: When estimating a model with a new choice of parameters, we use another model learned with a different choice of parameters as its initialization. Thus, the only time it is needed to estimate a model with

no initialization is when estimating the very first model for this dataset (model of the first iteration with the first choice of parameters). After it is estimated, the models of the subsequent iterations get initialized with the models of the previous iterations. Then, when moving on to a new choice of parameters, the model of the first iteration is initialized with the model estimated with the previous choice of parameters and so on.

Figure 11.6 shows the time taken in minutes to run GLSLIM on ML10M with 'warm start' and with 'no initialization'. Figure 11.6 shows the total time for all iterations when run with $k = 5$ user subsets and with l_2 regularizations parameters $\beta_g = \beta_l = 10$ and l_1 regularization parameters $\lambda_g = \lambda_l = 1$. Note that four iterations were needed until convergence. Also note that the greatest part of the time shown corresponds to the model estimations, as the user refinement does not take more than a couple of seconds (in this example, the user refinement part took fourteen seconds when run on one node). A speedup of $4\times$ is achieved by employing warm start.

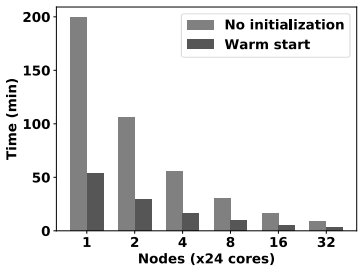


Fig. 11.6. The total time in minutes achieved by GLSLIM with and without warm start on the ML10M dataset, while increasing the number of nodes.

Table 11.3 shows the top-N recommendation performance and training times of SLIM and GLSLIM with warm start, when run with the same parameters $\beta = \beta_g = \beta_l = 10$ and $\lambda = \lambda_g = \lambda_l = 1$. Five user subsets were used for GLSLIM. The top-N performance is measured in terms of HR (Equation 11.1) and ARHR (Equation 11.2). The reported time corresponds to running SLIM and GLSLIM on one node (24 cores). This is done for fairness of comparison between the two methods. The shown times correspond to the warm-start right-most column of Figure 11.4 and the warm-start left-most column of Figure 11.6. We can see that GLSLIM has an average performance gain of 9.5% over SLIM, while requiring more

Table 11.3. Comparison of SLIM with GLSLIM in terms of top-N performance and training time.

Method	HR	ARHR	Time (min)
SLIM	0.310	0.152	2.56
GLSLIM	0.336	0.167	51.72

Time corresponds to 'warm start' time in minutes, and corresponds to the time taken on one node (24 cores). GLSLIM time corresponds to total time of all iterations until convergence.

time-consuming training; although higher number of nodes used allows for great decrease in running time.

11.4. Scaling up latent factor approaches

Latent factor approaches are a class of methods that map users and items to vectors in a common low-rank space known as the *latent space*. A detailed overview of latent space approaches can be found in Chapter 2. Latent factor approaches are perhaps the most popular techniques used for rating prediction. The success of these approaches has led to a wealth of research on developing algorithms to facilitate high-quality recommendations from massive training datasets. These algorithms exhibit complex tradeoffs in terms of computational characteristics, convergence rate, and available parallelism.

11.4.1. *Overview of matrix and tensor factorization*

Matrix factorization approaches [Koren (2008)] are state-of-the-art collaborative filtering methods and have gained high popularity since the Netflix Prize [Koren (2009); Takács *et al.* (2009)]. They assume that the user-item rating matrix $\mathbf{R}$ is low rank and can be computed as a product of two matrices known as the user and the item latent factors (denoted $\mathbf{P}$ and $\mathbf{Q}$, respectively). Rows of $\mathbf{P}$ and $\mathbf{Q}$ are F-dimensional vectors which represent the corresponding user or item. The value F is referred to as the *rank* of the factorization.

An item's latent factor, denoted $\mathbf{q}_i$, represents a few characteristics of the item, and a user's latent factor, denoted $\mathbf{p}_u$, signifies how much a user weights these characteristics. The predicted rating for the user u on the item i is given by

$$\hat{r}_{ui} = \mathbf{p}_u^T \mathbf{q}_i.$$

The completed matrix $\hat{\mathbf{R}} = \mathbf{P}\mathbf{Q}^T$ is used to serve the recommendation to the user for the items for which their preferences were unknown in the original matrix $\mathbf{R}$.

The user and the item latent factors are estimated by minimizing a regularized squared loss

$$\underset{\mathbf{P},\mathbf{Q}}{\text{minimize}} \quad \frac{1}{2} \sum_{r_{ui} \in \mathbf{R}} \left(r_{ui} - \mathbf{p}_u^T \mathbf{q}_i\right)^2 + \frac{\beta}{2} \left(||\mathbf{P}||_F^2 + ||\mathbf{Q}||_F^2\right), \tag{11.6}$$

where the parameter β controls the Frobenius norm regularization to prevent overfitting.

Additionally, instead of optimizing for rating predictions, one can optimize for ranking performance by substituting a ranking loss function instead of the squared error loss function. For example, Bayesian Personalized Ranking (BPR) [Rendle and Schmidt-Thieme (2010)], Collaborative Less-is-More Filtering (CLiMF) [Shi *et al.* (2012)] and CofiRank [Weimer *et al.* (2008)] optimize approximation of different ranking metrics to estimate the user and the item latent factors for better ranking performance.

Ratings are often accompanied by contextual information associated with the ratings. For example, the ML10M dataset provides both *timestamps* and *tags* which can be used to improve recommendation quality.

The traditional ratings matrix can be extended to include contextual information in the form of a *tensor*, which is the generalization of a matrix to higher orders. For example, associating each rating with a timestamp would result in a third-order tensor whose modes represent users, items, and time. Latent factor approaches can be extended to include higher-order data provided by tensors. The canonical polyadic decomposition (CPD) is a popular model for tensor factorization which has be used successfully for rating prediction. The CPD seeks to model a ratings tensor $\boldsymbol{\mathcal{R}}$ as the combination of user factor $\mathbf{P}$, item factor $\mathbf{Q}$, and context factor $\mathbf{C}$. The resulting optimization problem closely resembles that of matrix factorization:

$$\underset{\mathbf{P},\mathbf{Q},\mathbf{C}}{\text{minimize}} \quad \frac{1}{2} \sum_{r_{uik} \in \boldsymbol{\mathcal{R}}} \left(r_{uik} - \sum_{f=1}^{F} p_{uf} q_{if} c_{kf}\right)^2 + \frac{\beta}{2} \left(||\mathbf{P}||_F^2 + ||\mathbf{Q}||_F^2 + ||\mathbf{C}||_F^2\right).$$

The estimation of user and item latent factors by solving Equation 11.6 is one method of solving a problem referred to as *matrix completion*. It is a non-convex and computationally expensive problem. Several optimization algorithms have been successfully applied for matrix completion on large scale datasets.

Experimental environment. In the remaining discussion, we evaluate three latent factor approaches that solves matrix completion problem. Each algorithm is iterative in nature, though by convention we refer to these iterations as *epochs*. We define an epoch as the work required to update the latent factors one time using all available rating data. Convergence is detected when the RMSE does not improve for twenty epochs. We fix F, the rank of the factorization, to 40. All presented results are collected using SPLATT [Smith and Karypis (2015)], a publicly available[6] toolkit for high-performance sparse tensor factorizations. While optimized for tensors, SPLATT supports matrix factorizations because a matrix is equivalent to a two-mode tensor. SPLATT has also been integrated into the Spark+MPI framework [Anderson *et al.* (2017)], achieving over 10× speedup over pure Spark solutions.

11.4.2. *Alternating Least Squares (ALS)*

ALS was one of the first matrix completion algorithms applied to large scale data [Zhou *et al.* (2008)]. ALS is based on the observation that if we solve Equation 11.6 for one latent factor at a time, the solution has a linear least squares solution. ALS is an iterative algorithm which first minimizes with respect to $\mathbf{P}$ and then $\mathbf{Q}$. The process is repeated until convergence.

Let $\mathbf{r}_u$ be the vector of all ratings supplied by user u. $\mathbf{H}_u$ is an $|\mathbf{r}_u|\times F$ matrix whose rows are the feature vectors $\mathbf{q}_i$, for each item i rated in $\mathbf{r}_u$. Similarly, $\mathbf{r}_i$ is the vector of all ratings supplied for item i, and $\mathbf{H}_i$ is an $|\mathbf{r}_i|\times F$ matrix. ALS proceeds by updating all $\mathbf{p}_u$ followed by all $\mathbf{q}_i$:

$$\begin{aligned} \mathbf{p}_u &\leftarrow \left(\mathbf{H}_u^T\mathbf{H}_u + \beta\mathbf{I}\right)^{-1}\mathbf{H}_u^T\mathbf{r}_u, \quad \forall u \in 1,\ldots,m \\ \mathbf{q}_i &\leftarrow \left(\mathbf{H}_i^T\mathbf{H}_i + \beta\mathbf{I}\right)^{-1}\mathbf{H}_i^T\mathbf{r}_i, \quad \forall i \in 1,\ldots,n. \end{aligned} \tag{11.7}$$

The full procedure is outlined in Algorithm 4. Extending Equation 11.7 to tensors changes the construction of the $\mathbf{H}_u$ matrices [Shao (2012)]. For example, the row of $\mathbf{H}_u$ associated with rating r_{uik} is the elementwise multiplication of the corresponding feature vectors $\mathbf{q}_i$ and $\mathbf{c}_k$. The $\mathbf{H}_i$ and $\mathbf{H}_k$ matrices are constructed similarly.

Each row in Equation 11.7 is independent and thus can be computed in parallel [Zhou *et al.* (2008)]. The simplicity of this approach has led ALS to be optimized for high-performance shared- and distributed-memory systems[Karlsson *et al.* (2015); Smith *et al.* (2017)], GPUs [Gates *et al.* (2015);

[6] http://cs.umn.edu/~splatt/

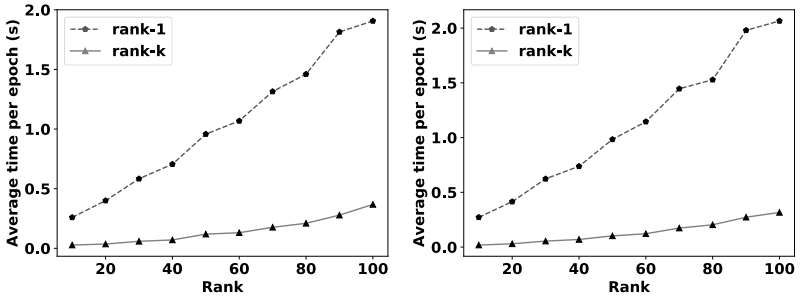
Algorithm 4 Matrix factorization via alternating least squares (ALS)

1: Initialize $\mathbf{P}$ and $\mathbf{Q}$ randomly.
2: **while** $\mathbf{P}$ and $\mathbf{Q}$ have not converged **do**
3: **for all** user u in parallel **do**
4: $\mathbf{H}_u \leftarrow \mathbf{0}$.
5: For each rating r_{ui} in $\mathbf{r}_u$, append row $\mathbf{q}_i$ to $\mathbf{H}_u$.
6: $\mathbf{p}_u \leftarrow \left(\mathbf{H}_u^T\mathbf{H}_u + \beta\mathbf{I}\right)^{-1}\mathbf{H}_u^T\mathbf{r}_u$.
7: **end for**
8: **for all** item i in parallel **do**
9: $\mathbf{H}_i \leftarrow \mathbf{0}$.
10: For each rating r_{ui} in $\mathbf{r}_i$, append row $\mathbf{p}_u$ to $\mathbf{H}_i$.
11: $\mathbf{q}_i \leftarrow \left(\mathbf{H}_i^T\mathbf{H}_i + \beta\mathbf{I}\right)^{-1}\mathbf{H}_i^T\mathbf{r}_i$.
12: **end for**
13: **end while**

Tan *et al.* (2016)], Hadoop [Shin and Kang (2014)], and is implemented in Spark's MLlib[7]. Successful approaches distribute the ratings data in a one-dimensional fashion such that all of the ratings required to construct an $\mathbf{H}$ matrix are located on the same node. By distributing the data in this fashion, none of the partially-constructed $\mathbf{H}$ matrices need to be communicated or aggregated. However, this distribution requires that each node stores potentially the entire latent factors. Fortunately, in practice this is not prohibitive on most modern systems.

The computational complexity of ALS is $\mathcal{O}\left(F^2|\mathbf{R}| + F^3(m+n)\right)$ per epoch. In practice, the $\mathcal{O}(F^2)$ computation per rating that comes from constructing the various $\mathbf{H}$ matrices dominates the computation. A common implementation strategy is to process one rating at a time and accumulate directly into $\mathbf{H}_u^T\mathbf{H}_u$ and $\mathbf{H}_u\mathbf{r}_u$ instead of explicitly constructing $\mathbf{H}_u$. However, this strategy ignores the details of modern hardware architectures in which memory movement is more expensive than floating-point operations. Each rating produces an accumulation that is a rank-1 update performing $\mathcal{O}(F^2)$ work on F^2 data. Alternatively, performing a single rank-k update by explicitly forming $\mathbf{H}_u$ instead performs $\mathcal{O}(|\mathbf{r}_u|F^2)$ work on $(|\mathbf{r}_u|F + F^2)$ data [Gates *et al.* (2015); Smith *et al.* (2017)]. While the final amount of work is the same, the rank-k update fetches less data from memory and is thus better suited for modern processors. We explore this phenomenon

[7] https://spark.apache.org/mllib/



(a) Matrix (b) Tensor

Fig. 11.7. Average time per epoch when using rank-1 and rank-k updates during ALS. Execution is on 24 cores and an epoch is counted as updating each factor matrix once.

in Figure 11.7, which illustrates runtime per epoch as F is increased. Using rank-k updates can be over 10× faster than the more common rank-1 updates.

11.4.3. *Stochastic Gradient Descent (SGD)*

SGD is an optimization algorithm that trades a large number of epochs for a low computational complexity. An epoch consists of processing all ratings one-at-a-time in random order and updating the factorization based on the local gradient. Updates are of the form:

$$\begin{aligned} e_{ui} &\leftarrow r_{ui} - \mathbf{p}_u^T \mathbf{q}_i, \\ \mathbf{p}_u &\leftarrow \mathbf{p}_u + \eta \left(e_{ui} \mathbf{q}_i - \beta \mathbf{p}_u \right), \\ \mathbf{q}_i &\leftarrow \mathbf{q}_i + \eta \left(e_{ui} \mathbf{p}_u - \beta \mathbf{q}_i \right), \end{aligned} \tag{11.8}$$

where η is a hyperparameter representing the learning rate. The complexity of Equation 11.8 is linear in F, resulting in a total complexity of $\mathcal{O}(F|\mathbf{R}|)$ per epoch. The low complexity and simple implementation of SGD has led to it being widely adopted by researchers and industry alike. The details of SGD are outlined in Algorithm 5.

SGD is less straightforward than ALS to parallelize. Since processing a rating updates rows of both $\mathbf{P}$ and $\mathbf{Q}$, special care must be taken to prevent the same rows from being modified at the same time (called a *race condition*). There are two broad approaches for parallelizing SGD.

Stratified methods are based on the observation that if two ratings do not overlap (i.e., they have neither a row nor a column in common) then they can be updated with Equation 11.8 in parallel. This strategy was

Algorithm 5 Matrix factorization via stochastic gradient descent (SGD)

```
Initialize P and Q randomly.
while P and Q have not converged do
    Shuffle the permutation of ratings.
    for all rating r_ui do
        e_ui ← r_ui − p_u^T q_i
        p_u ← p_u + η (e_ui q_i − β p_u).
        q_i ← q_i + η (e_ui p_u − β q_i).
    end for
end while
```

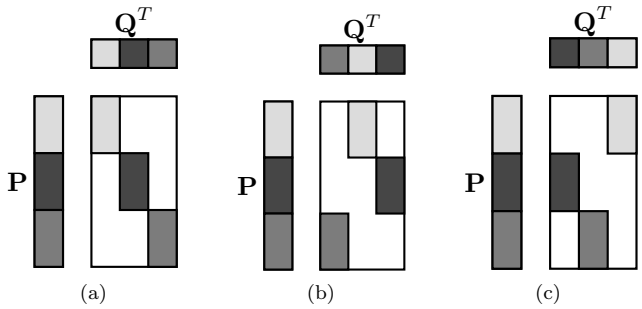


Fig. 11.8. Stratified SGD with three workers. Colored blocks represent independent sets of ratings and the rows of **P** and **Q** which model them. Each colored block of ratings and their corresponding rows can be processed in parallel.

introduced by DSGD [Gemulla *et al.* (2011)], which imposes a grid on **R** to identify blocks that can be processed in parallel. Stratification is illustrated in Figure 11.8. Stratification has proven to be an effective strategy for parallelizing SGD and has been extended in works on multithreaded environments, distributed systems, and GPUs [Zhuang *et al.* (2013); Yun *et al.* (2014); Xie *et al.* (2017)].

Asynchronous methods rely on the stochastic nature of SGD to allow overlapping updates. This technique was popularized by Hogwild [Recht *et al.* (2011)] on shared-memory systems. The key idea is to simply allow race conditions to occur without attempting to avoid them. Convergence is still achieved due to the iterative nature of SGD and infrequent overlaps from the high level of sparsity in **R**. Teflioudi *et al.* later introduced asynchronous SGD (ASGD) [Teflioudi *et al.* (2012)] for distributed computing environments. During ASGD, nodes maintain locally modified copies of **P**

and **Q** and updates are asynchronously communicated several times per epoch. Overlapping updates are averaged with the master copy and sent to workers.

Extending the formulation of SGD to tensors is straightforward and again only requires additional elementwise multiplications [Shao (2012)]. However, parallelization becomes a significant challenge when a contextual mode is added to the data. The number of blocks in a stratified SGD algorithm increases *exponentially* with the number of tensor modes despite the work per rating only increasing linearly, and thus the time of synchronization and communication quickly dominate the factorization time. Asynchronous methods also suffer because the number of unique contexts is typically much smaller than the number of users or items, resulting in more frequent update conflicts. A hybrid of stratification and asynchronous SGD addresses these challenges, but the hybrid is still bested by ALS at large numbers of cores [Smith *et al.* (2017)].

11.4.4. *Coordinate Descent (CCD++)*

Coordinate descent is a class of optimization algorithms which update one parameter of the output at a time. CCD++ is a column-oriented descent algorithms for matrix completion [Yu *et al.* (2012)]. CCD++ updates columns of **P** and **Q** in sequence, with a single parameter update taking the form

$$p_{uf} \leftarrow \frac{\sum_{r_{ui} \in \mathbf{R}} (r_{ui} - \mathbf{p}_u^T \mathbf{q}_i + p_{uf} q_{if}) q_{if}}{\beta + \sum_{r_{ui} \in \mathbf{R}} q_{if}^2}. \tag{11.9}$$

The full procedure is outlined in Algorithm 4. If all $(r_{ui} - \mathbf{p}_u^T \mathbf{q}_i)$ are pre-computed, CCD++ has a complexity of $\mathcal{O}(F|\mathbf{R}|)$ per epoch, matching SGD. The extension of CCD++ to tensors follows that of ALS and SGD, in which additional elementwise multiplications are introduced to the formulation [Karlsson *et al.* (2015)].

Similar to ALS, each column entry is independent and can thus be computed in parallel. CCD++ has accordingly been parallelized on shared- and distributed-memory systems [Yu *et al.* (2012); Karlsson *et al.* (2015); Smith *et al.* (2017)] and GPUs [Nisa *et al.* (2017)]. However, unlike ALS, the communication cost from aggregating partial computations is only of constant size per column as opposed to the larger **H** matrices of ALS. The lower communication volume affords more flexible partitionings of the ratings. Recent work has shown that a Cartesian (i.e., grid) distribution

Algorithm 6 Matrix factorization via coordinate descent (CCD++)

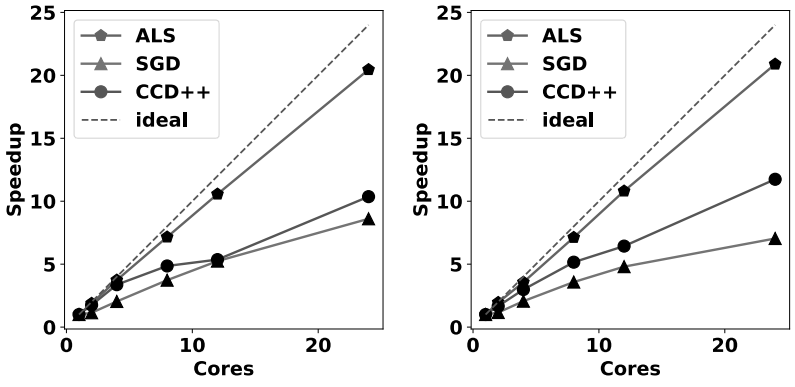
```
Initialize P and Q randomly.
while P and Q have not converged do
  for all column f do
    Pre-compute all error terms: (r_ui − p_u^T q_i).
    for all user u in parallel do
      Update p_uf following (11.9).
    end for
    for all item i in parallel do
      Update q_if following (11.9).
    end for
  end for
end while
```

of the data is an effective formulation and has been scaled to over sixteen thousand cores [Smith *et al.* (2017)].

11.4.5. *Evaluation of optimization algorithms*

Parallel scalability. We examine the parallel scalability of ALS, SGD, and CCD++ for matrix and tensor completion in Figure 11.9. ALS scales notably better than the competing methods. The scalability of ALS comes from being rich in dense linear algebra kernels which effectively use the floating-point hardware found in each core, instead of being bound by memory bandwidth which is a shared resource. In contrast, SGD and CCD++ perform a factor of F fewer floating-point operations per rating processed, resulting in heavy reliance on available memory bandwidth. Interestingly, the scalability of CCD++ and SGD is also more dependent on the size and characteristics of the ratings dataset. Figure 11.10 shows parallel scalability on a tensor of 210 million Yahoo! music ratings with timestamps from the 2011 KDD cup [Dror *et al.* (2012)]. CCD++ achieves perfect speedup on this significantly larger and more sparse dataset.

Time to solution. Finally, we compare solution qualities and convergence times for the latent factor approaches in Table 11.4. In the matrix case, SGD arrives at the lowest RMSE while being competitive in runtime to ALS. CCD++ obtains the lowest RMSE in the tensor case, but at 5× the runtime of the similar-quality ALS. Utilizing the timestamp for tensor completion notably improves the RMSE for ALS and CCD++, but not SGD.



(a) Matrix (b) Tensor

Fig. 11.9. Speedup on ML10M dataset scaling from 1 to 24 cores. SGD is parallelized using Hogwild [Recht *et al.* (2011)].

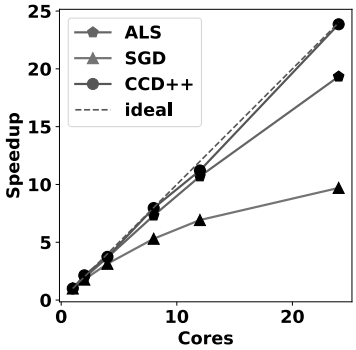


Fig. 11.10. Parallel speedup on a three-mode tensor made from 210 million Yahoo! music ratings.

Since timestamps are grouped by month (133 months in total), the number of independent months is significantly more limited than the number of independent users or items. Thus, there are frequent overlapping updates to the $\mathbf{C}$ latent factor. Lastly, we note that the time-to-solution for CCD++ is longer in the matrix case than the tensor case, despite performing less work and arriving at a higher RMSE. While the tensor completion algorithm performs more work than the matrix equivalent, in practice we find that it converges in fewer epochs.

Table 11.4. Comparison of solution quality and time.

	Matrix		Tensor	
Algorithm	**RMSE**	**Time (s)**	**RMSE**	**Time (s)**
ALS	0.8805	4.82	0.8662	50.26
SGD	0.8747	11.54	0.9107	17.19
CCD++	0.8955	435.61	0.8622	256.25

RMSE is evaluated on the test dataset, averaged over three folds.
Time measures the time to convergence.

11.4.6. *Singular Value Decomposition (SVD)*

The key idea of SVD-based models is to factorize the user-item rating matrix to a product of two lower rank matrices, one containing the user factors and the other containing the item factors. Since conventional SVD is undefined in the presence of missing values, *PureSVD* [Cremonesi *et al.* (2010)] treats all the missing values as zeros prior to the application of the standard SVD method. PureSVD is shown to be suitable for the top-N recommendations task. The better top-N recommendation performance of PureSVD in comparison to the standard matrix completion-based approaches that are optimized for rating predictions, can be attributed to the fact that it considers all the items present in the catalog rather than considering only the items rated by the user. Additionally, for ranking purposes, it does not need to predict the exact ratings but only requires to achieve a correct relative ordering of the predictions for a user.

PureSVD estimates the rating matrix $\mathbf{R}$ as

$$\hat{\mathbf{R}} = \mathbf{U}\mathbf{\Sigma}\mathbf{Q}^T,$$

where $\mathbf{U}$ is a $n \times F$ orthonormal matrix, $\mathbf{Q}$ is an $m \times F$ orthonormal matrix and $\mathbf{\Sigma}$ is an $F \times F$ diagonal matrix, containing the F largest singular values. It can be noted that the matrix $\mathbf{P}$ representing the user factors can be derived by

$$\mathbf{P} = \mathbf{U}\mathbf{\Sigma}.$$

The matrices $\mathbf{U}$, $\mathbf{\Sigma}$ and $\mathbf{Q}$ can be estimated by solving the following optimization problem with orthonormal constraints

$$\begin{aligned} \underset{\mathbf{U},\mathbf{Q},\mathbf{\Sigma}}{\text{minimize}} \quad & \frac{1}{2}||\mathbf{R} - \sum_{i=1}^{F} \sigma_i u_i q_i^T||_F^2 \\ \text{subject to} \quad & \mathbf{U}^T\mathbf{U} = \mathbf{I} \\ & \mathbf{Q}^T\mathbf{Q} = \mathbf{I}, \end{aligned} \tag{11.10}$$

where $\mathbf{I}$ is the identity matrix, σ_i denotes the ith largest singular value, u_i represents the ith column vector of $\mathbf{U}$ and q_i denotes the ith column vector of $\mathbf{Q}$. The application of PureSVD on large scale sparse matrices can be optimized with the *Golub-Kahan-Lanczos Bidiagonalization* [Golub and Kahan (1965); Lanczos (1950)] approach. It computes the SVD of given matrix $\mathbf{R}$ in two steps. First, it bidiagonalizes $\mathbf{R}$ using Lanczos procedure as,

$$\mathbf{R} = \mathbf{P}\mathbf{B}\mathbf{Q}^T, \tag{11.11}$$

where $\mathbf{P}$ and $\mathbf{Q}$ are unitary matrices, and $\mathbf{B}$ is an upper bidiagonal matrix. The Lanczos procedure can take advantage of optimized sparse matrix-vector multiplications and efficient orthogonalization. Then, it uses an efficient method [Demmel and Kahan (1990)] to compute the singular values of $\mathbf{B}$ without computing $\mathbf{B}^T\mathbf{B}$ as,

$$\mathbf{B} = \mathbf{X}\mathbf{\Sigma}\mathbf{Y}^T. \tag{11.12}$$

Now, using Equations 11.11 and 11.12 we can compute left singular vectors, i.e., $\mathbf{U} = \mathbf{P}\mathbf{X}$, and right singular vectors, i.e., $\mathbf{V} = \mathbf{Q}\mathbf{Y}$, of matrix $\mathbf{R}$.

Modern randomized matrix approximation techniques [Halko *et al.* (2011)] can be used to compute a faster but approximate SVD of the rating matrix. We will refer to these approximation techniques as *Randomized SVD*. Essentially, Randomized SVD technique is carried out in two steps. First, it tries to find $\mathbf{Q}$ with F orthonormal columns such that,

$$\mathbf{R} \approx \mathbf{Q}\mathbf{Q}^T\mathbf{A}. \tag{11.13}$$

Next, it constructs $\mathbf{B} = \mathbf{Q}^T\mathbf{A}$, and since $\mathbf{B}$ has relatively smaller number of rows, i.e., F, we can employ standard methods to efficiently compute SVD of $\mathbf{B}$ as,

$$\mathbf{B} = \tilde{\mathbf{U}}\mathbf{\Sigma}\mathbf{V}^T. \tag{11.14}$$

Thus, left singular vector of $\mathbf{R}$ can be approximated by, $\mathbf{U} = \mathbf{Q}\tilde{\mathbf{U}}$, and right singular vectors can be approximated by $\mathbf{V}$. Randomized SVD can take advantage of efficient sparse matrix-matrix multiplication to find $\mathbf{Q}$ and to compute $\mathbf{B}$.

For our experiments, we used the optimized and parallel implementation of Golub-Kahan-Lanczos Bidiagonalization approach available in SLEPc[8] [Hernandez *et al.* (2007)] for PureSVD, and utilized the implementation of Randomized SVD available in RedSVD[9]. These implementations rely

[8] slepc.upv.es

[9] https://github.com/ntessore/redsvd-h

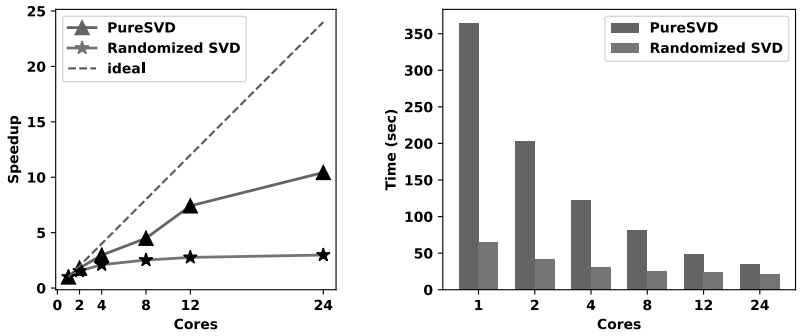


Fig. 11.11. Speedup (left) and total time in seconds (right) achieved by PureSVD and Randomized SVD on the ML10M dataset ($F = 500$).

on well-studied sparse and dense linear algebra operations, that are further optimized for efficient usage on high-performance computers [Anderson *et al.* (1990)]. Figure 11.11 shows the speedup and total time achieved by PureSVD and Randomized SVD on ML10M dataset with increasing number of cores. As can be seen in the figure, the parallel implementation of PureSVD achieves better speedup than Randomized SVD with increase in the number of cores. Also, the time taken by Randomized SVD is lower in comparison to that of PureSVD on a single core. Table 11.5 presents the results for the best ranking performance achieved by both the methods on ML10M dataset. As can be seen in the table, PureSVD and Randomized SVD do not outperform SLIM for top-N recommendation but the time taken by PureSVD and Randomized SVD is lower than that of SLIM. Furthermore, PureSVD outperforms Randomized SVD for top-N recommendation performance. We should note though that the performance of Randomized SVD is comparable to that of PureSVD, and therefore Randomized SVD can serve as an alternative to PureSVD under time and compute resource constraints. Also, Randomized SVD needs higher rank to achieve its best performance.

11.5. Conclusion

In this chapter, we presented different methods which speed up popular collaborative recommenders, by taking advantage of modern parallel multicore architectures. We discussed ways to efficiently identify neighbors in

Table 11.5. Comparison of PureSVD with Randomized SVD in terms of top-N performance and training time.

Method	HR	ARHR	Rank (F)	Time (sec)
PureSVD	0.292	0.139	60	9.24
Randomized SVD	0.247	0.112	400	15.46

Time corresponds to the time taken on one node (24 cores).

k-nearest neighbor approaches in Section 11.2. We investigated how to parallelize the sparse linear methods well-suited for the top-N recommendation task, presented in Section 11.3 and how to speed up their parameter search. Finally, in Section 11.4, we showed ways to scale up the latent factor approaches, which could extend to tensor factorization approaches. In each section, we also presented experimental results on the popular ML10M dataset, illustrating the runtimes and speedup achieved in comparison to serial core implementations. Overall, the goal of this chapter is to illustrate that modern popular collaborative recommenders, although of very different nature, are able to be parallelized. We believe that research that focuses on ways to distribute and scale popular collaborative recommenders is crucial, as it leads to faster training times without sacrificing recommendation quality.

References

Anastasiu, D. C. (2017). Cosine approximate nearest neighbors, in P. Haber, T. Lampoltshammer and M. Mayr (eds.), *Data Science – Analytics and Applications*, iDSC 2017 (Springer Fachmedien Wiesbaden, Wiesbaden), ISBN 978-3-658-19287-7, pp. 45–50.

Anastasiu, D. C. and Karypis, G. (2014). L2ap: Fast cosine similarity search with prefix l-2 norm bounds, in *30th IEEE International Conference on Data Engineering*, ICDE '14, pp. 784–795, doi:10.1109/ICDE.2014.6816700.

Anastasiu, D. C. and Karypis, G. (2015a). L2knng: Fast exact k-nearest neighbor graph construction with l2-norm pruning, in *24th ACM International Conference on Information and Knowledge Management*, CIKM '15 (ACM, New York, NY, USA), ISBN 978-1-4503-3794-6, pp. 791–800, doi: 10.1145/2806416.2806534, http://doi.acm.org/10.1145/2806416.2806534.

Anastasiu, D. C. and Karypis, G. (2015b). Pl2ap: Fast parallel cosine similarity search, in *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, IA3 '15 (ACM, New York, NY, USA), pp. 8:1–8:8.

Anastasiu, D. C. and Karypis, G. (2016). Fast parallel cosine k-nearest neighbor graph construction, in *2016 6th Workshop on Irregular Applications: Architecture and Algorithms (IA3)*, pp. 50–53, doi:10.1109/IA3.2016.013.

Anderson, E., Bai, Z., Dongarra, J., Greenbaum, A., McKenney, A., Du Croz, J., Hammarling, S., Demmel, J., Bischof, C. and Sorensen, D. (1990). Lapack: A portable linear algebra library for high-performance computers, in *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, Supercomputing '90 (IEEE Computer Society Press, Los Alamitos, CA, USA), ISBN 0-89791-412-0, pp. 2–11, http://dl.acm.org/citation.cfm?id=110382.110385.

Anderson, M., Smith, S., Sundaram, N., Capotă, M., Zhao, Z., Dulloor, S., Satish, N. and Willke, T. L. (2017). Bridging the gap between HPC and Big Data frameworks, *Proceedings of the VLDB Endowment (PVLDB '17)*.

Bayardo, R. J., Ma, Y. and Srikant, R. (2007). Scaling up all pairs similarity search, in *Proceedings of the 16th International Conference on World Wide Web*, WWW '07 (ACM, New York, NY, USA), pp. 131–140.

Christakopoulou, E. and Karypis, G. (2016). Local item-item models for top-n recommendation, in *Proceedings of the 10th ACM Conference on Recommender Systems* (ACM), pp. 67–74.

Cremonesi, P., Koren, Y. and Turrin, R. (2010). Performance of recommender algorithms on top-n recommendation tasks, in *Proceedings of the Fourth ACM Conference on Recommender Systems*, RecSys '10 (ACM, New York, NY, USA), ISBN 978-1-60558-906-0, pp. 39–46, doi:10.1145/1864708.1864721, http://doi.acm.org/10.1145/1864708.1864721.

Demmel, J. and Kahan, W. (1990). Accurate singular values of bidiagonal matrices, *SIAM Journal on Scientific and Statistical Computing* **11**, 5, pp. 873–912.

Deshpande, M. and Karypis, G. (2004). Item-based top-n recommendation algorithms, *ACM Transactions on Information Systems (TOIS)* **22**, 1, pp. 143–177.

Dror, G., Koenigstein, N., Koren, Y. and Weimer, M. (2012). The yahoo! music dataset and kdd-cup'11. in *KDD Cup*, pp. 8–18.

Friedman, J., Hastie, T. and Tibshirani, R. (2010). Regularization paths for generalized linear models via coordinate descent, *Journal of statistical software* **33**, 1, p. 1.

Gates, M., Anzt, H., Kurzak, J. and Dongarra, J. (2015). Accelerating collaborative filtering using concepts from high performance computing, in *Big Data (Big Data), 2015 IEEE International Conference on* (IEEE), pp. 667–676.

Gemulla, R., Nijkamp, E., Haas, P. J. and Sismanis, Y. (2011). Large-scale matrix factorization with distributed stochastic gradient descent, in *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining* (ACM), pp. 69–77.

Golub, G. and Kahan, W. (1965). Calculating the singular values and pseudo-inverse of a matrix, *Journal of the Society for Industrial and Applied Mathematics, Series B: Numerical Analysis* **2**, 2, pp. 205–224.

Halko, N., Martinsson, P.-G. and Tropp, J. A. (2011). Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions, *SIAM review* **53**, 2, pp. 217–288.

Harper, F. M. and Konstan, J. A. (2015). The movielens datasets: History and context, *ACM Trans. Interact. Intell. Syst.* **5**, 4, pp. 19:1–19:19.

Hernandez, V., Roman, J., Tomas, A. and Vidal, V. (2007). Restarted lanczos bidiagonalization for the svd in slepc, *STR-8, Tech. Rep.*

Karlsson, L., Kressner, D. and Uschmajew, A. (2015). Parallel algorithms for tensor completion in the cp format, *Parallel Computing*.

Konstan, J. A., Miller, B. N., Maltz, D., Herlocker, J. L., Gordon, L. R. and Riedl, J. (1997). Grouplens: Applying collaborative filtering to usenet news, *Commun. ACM* **40**, 3, pp. 77–87, doi:10.1145/245108.245126, http://doi.acm.org/10.1145/245108.245126.

Koren, Y. (2008). Factorization meets the neighborhood: a multifaceted collaborative filtering model, in *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining* (ACM), pp. 426–434.

Koren, Y. (2009). The bellkor solution to the netflix grand prize, *Netflix prize documentation* **81**, pp. 1–10.

Lanczos, C. (1950). *An iteration method for the solution of the eigenvalue problem of linear differential and integral operators* (United States Governm. Press Office Los Angeles, CA).

Ning, X. and Karypis, G. (2011). Slim: Sparse linear methods for top-n recommender systems, in *2011 IEEE 11th International Conference on Data Mining* (IEEE), pp. 497–506.

Nisa, I., Sukumaran-Rajam, A., Kunchum, R. and Sadayappan, P. (2017). Parallel CCD++ on GPU for matrix factorization, in *Proceedings of the General Purpose GPUs (GPGPU)* (ACM), pp. 73–83.

Recht, B., Re, C., Wright, S. and Niu, F. (2011). Hogwild: A lock-free approach to parallelizing stochastic gradient descent, in *Advances in Neural Information Processing Systems*, pp. 693–701.

Rendle, S. and Schmidt-Thieme, L. (2010). Pairwise interaction tensor factorization for personalized tag recommendation, in *Proceedings of the third ACM international conference on Web search and data mining* (ACM), pp. 81–90.

Sarwar, B., Karypis, G., Konstan, J. and Riedl, J. (2001). Item-based collaborative filtering recommendation algorithms, in *Proceedings of the 10th international conference on World Wide Web* (ACM), pp. 285–295.

Shao, W. (2012). *Tensor Completion*, Master's thesis, Universität des Saarlandes Saarbrücken.

Shi, Y., Karatzoglou, A., Baltrunas, L., Larson, M., Oliver, N. and Hanjalic, A. (2012). Climf: learning to maximize reciprocal rank with collaborative less-is-more filtering, in *Proceedings of the sixth ACM conference on Recommender systems* (ACM), pp. 139–146.

Shin, K. and Kang, U. (2014). Distributed methods for high-dimensional and large-scale tensor factorization, in *Data Mining (ICDM), 2014 IEEE International Conference on*, pp. 989–994.

Smith, S. and Karypis, G. (2015). SPLATT: the Surprisingly Parallel spArse Tensor Toolkit, http://cs.umn.edu/~splatt/.

Smith, S., Park, J. and Karypis, G. (2017). Hpc formulations of optimization algorithms for tensor completion, *Parallel Computing*.

Takács, G., Pilászy, I., Németh, B. and Tikk, D. (2009). Scalable collaborative filtering approaches for large recommender systems, *Journal of machine learning research* **10**, Mar, pp. 623–656.

Tan, W., Cao, L. and Fong, L. (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* (ACM), pp. 219–230.

Teflioudi, C., Makari, F. and Gemulla, R. (2012). Distributed matrix completion, in *Data Mining (ICDM), 2012 IEEE 12th International Conference on* (IEEE), pp. 655–664.

Tibshirani, R. (1996). Regression shrinkage and selection via the lasso, *Journal of the Royal Statistical Society. Series B (Methodological)*, pp. 267–288.

Weimer, M., Karatzoglou, A., Le, Q. V. and Smola, A. J. (2008). Cofi rank-maximum margin matrix factorization for collaborative ranking, in *Advances in neural information processing systems*, pp. 1593–1600.

Xie, X., Tan, W., Fong, L. L. and Liang, Y. (2017). CuMF_SGD: Parallelized stochastic gradient descent for matrix factorization on gpus, in *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, pp. 79–92.

Yu, H.-F., Hsieh, C.-J., Dhillon, I. *et al.* (2012). Scalable coordinate descent approaches to parallel matrix factorization for recommender systems, in *Data Mining (ICDM), 2012 IEEE 12th International Conference on* (IEEE), pp. 765–774.

Yun, H., Yu, H.-F., Hsieh, C.-J., Vishwanathan, S. V. N. and Dhillon, I. (2014). Nomad: Non-locking, stochastic multi-machine algorithm for asynchronous and decentralized matrix completion, *Proc. VLDB Endow.* **7**, 11, pp. 975–986, doi:10.14778/2732967.2732973, http://dx.doi.org/10.14778/2732967.2732973.

Zhou, Y., Wilkinson, D., Schreiber, R. and Pan, R. (2008). Large-scale parallel collaborative filtering for the netflix prize, in *Algorithmic Aspects in Information and Management* (Springer), pp. 337–348.

Zhuang, Y., Chin, W.-S., Juan, Y.-C. and Lin, C.-J. (2013). A fast parallel sgd for matrix factorization in shared memory systems, in *Proceedings of the 7th ACM conference on Recommender systems* (ACM), pp. 249–256.

Zou, H. and Hastie, T. (2005). Regularization and variable selection via the elastic net, *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* **67**, 2, pp. 301–320.

Index