

Algorithmic Acceleration of Parallel ALS for Collaborative Filtering: Speeding up Distributed Big Data Recommendation in Spark

Manda Winlaw*, Michael B Hynes[†], Anthony Caterini[‡], Hans De Sterck^{§,1}

Department of Applied Mathematics

University of Waterloo, Canada

Email: *mwinlaw@uwaterloo.ca, [†]mbhynes@uwaterloo.ca, [‡]alcaterini@uwaterloo.ca, [§]hdesterck@uwaterloo.ca

Abstract—Collaborative filtering algorithms are important building blocks in many practical recommendation systems. For example, many large-scale data processing environments include collaborative filtering models for which the Alternating Least Squares (ALS) algorithm is used to compute latent factor matrix decompositions. In this paper, we propose an approach to accelerate the convergence of parallel ALS-based optimization methods for collaborative filtering using a nonlinear conjugate gradient (NCG) wrapper around the ALS iterations. We also provide a parallel implementation of the accelerated ALS-NCG algorithm in the Apache Spark distributed data processing environment, and an efficient line search technique as part of the ALS-NCG implementation that requires only one pass over the data on distributed datasets. In serial numerical experiments on a linux workstation and parallel numerical experiments on a 16 node cluster with 256 computing cores, we demonstrate that the combined ALS-NCG method requires many fewer iterations and less time than standalone ALS to reach movie rankings with high accuracy on the MovieLens 20M dataset. In parallel, ALS-NCG can achieve an acceleration factor of 4 or greater in clock time when an accurate solution is desired; furthermore, the acceleration factor increases as greater numerical precision is required in the solution. In addition, the NCG acceleration mechanism is efficient in parallel and scales linearly with problem size on synthetic datasets with up to nearly 1 billion ratings. The acceleration mechanism is general and may also be applicable to other optimization methods for collaborative filtering.

Keywords—Recommendation systems, collaborative filtering, parallel optimization algorithms, matrix factorization, Apache Spark, Big Data, scalable methods.

I. INTRODUCTION AND BACKGROUND

Recommendation systems are designed to analyze available user data to recommend items such as movies, music, or other goods to consumers, and have become an increasingly important part of most successful online businesses. One strategy for building recommendation systems is known as collaborative filtering, whereby items are recommended to users by collecting preferences or taste information from many users (see, e.g., [1], [2]). Collaborative filtering methods provide the basis for many recommendation systems [3] and have been used by online businesses such as Amazon [4], Netflix [5], and Spotify [6].

¹Currently at Monash University, School of Mathematical Sciences, Melbourne, Australia

An important class of collaborative filtering methods are latent factor models, for which low-rank matrix factorizations are often used and have repeatedly demonstrated better accuracy than other methods such as nearest neighbor models and restricted Boltzmann machines [5], [7]. A low-rank latent factor model associates with each user and with each item a vector of rank n_f , for which each component measures the tendency of the user or item towards a certain *factor* or *feature*. In the context of movies, a latent feature may represent the style or genre (i.e. drama, comedy, or romance), and the magnitude of a given component of a user's feature vector is proportional to the user's proclivity for that feature (or, for an item's feature vector, to the degree to which that feature is manifested by the item). A low-rank matrix factorization procedure takes as its input the user-item ratings matrix \mathbf{R} , in which each entry is a numerical rating of an item by a user and for which typically very few entries are known. The procedure then determines the low-rank user (\mathbf{U}) and item (\mathbf{M}) matrices, where a column in these matrices represents a latent feature vector for a single user or item, respectively, and $\mathbf{R} \approx \mathbf{U}^T \mathbf{M}$ for the known values in \mathbf{R} . Once the user and item matrices are computed, they are used to build the recommendation system and predict the unknown ratings. Computing user and item matrices is the first step in building a variety of recommendation systems, so it is important to compute the factorization of \mathbf{R} quickly.

The matrix factorization problem is closely related to the singular value decomposition (SVD), but the SVD of a matrix with missing values is undefined. However, since $\mathbf{R} \approx \mathbf{U}^T \mathbf{M}$, one way to find the user and item matrices is by minimizing the squared difference between the approximated and actual value of the known ratings in \mathbf{R} . Minimizing this difference is typically done by one of two algorithms: stochastic gradient descent (SGD) or alternating least squares (ALS) [8], [9]. ALS can be easily parallelized and can efficiently handle models that incorporate implicit data (e.g. the frequency of a user's mouse-clicks or time spent on a website) [10], but it is well-known that ALS can require a large number of iterations to converge. Thus, in this paper, we propose an approach to significantly accelerate the convergence of the ALS algorithm for computing the user and item matrices. We use a nonlinear optimization algorithm, specifically the nonlinear conjugate

gradient (NCG) algorithm [11], as a wrapper around ALS to significantly accelerate the convergence of ALS, and thus refer to this combined algorithm as ALS-NCG. Alternatively, the algorithm can be viewed as a nonlinearly preconditioned NCG method with ALS as the nonlinear preconditioner [12]. Our approach for accelerating the ALS algorithm using the NCG algorithm can be situated in the context of recent research activity on nonlinear preconditioning for nonlinear iterative solvers [13], [14], [12], [15], [16]. Some of the ideas date back as far as the 1960s [17], [18], but they are not well-known and remain under-explored experimentally and theoretically [13].

Parallel versions of collaborative filtering and recommendation are of great interest in the era of big data [19], [20], [21]. For example, the Apache Spark data processing environment [22] contains a parallel implementation of ALS for the collaborative filtering model of [23], [8]. In [19] an advanced distributed SGD method is described in Hadoop environments, followed by work in [20] that considers algorithms based on ALS and SGD in environments that use the Message Passing Interface (MPI). Scalable coordinate descent approaches for collaborative filtering are proposed in [21], also using the MPI framework. In addition to producing algorithmic advances, these papers have shown that the relative performance of the methods considered is often strongly influenced by the performance characteristics of the parallel computing paradigm that is used to implement the algorithms (e.g., Hadoop or MPI). In this paper we implement our proposed ALS-NCG algorithm in parallel using the Apache Spark framework, which is a large-scale distributed data processing environment that builds on the principles of scalability and fault tolerance that are instrumental in the success of Hadoop and MapReduce. However, Spark adds crucial new capabilities in terms of in-memory computing and data persistence for iterative methods, and makes it possible to implement more elaborate algorithms with significantly better performance than is feasible in Hadoop. As such, Spark is already being used extensively for advanced big data analytics in the commercial setting [24]. The parallel Spark implementation of ALS for the collaborative filtering model of [23], [8] forms the starting point for applying the parallel acceleration methods proposed in this paper.

Our contributions in this paper are as follows. The specific optimization problem is formulated in Section II, and the accelerated ALS-NCG algorithm is developed in Section III. In Section IV, we study the convergence enhancements of ALS-NCG compared to standalone ALS in small serial tests using subsets of the MovieLens 20M dataset [25]. Section V describes our parallel implementation in Spark¹, and Section VI contains results from parallel performance tests on a high-end computing cluster with 16 nodes and 256 cores, using both the full MovieLens 20M dataset and a large synthetic dataset sampled from the MovieLens 20M data with up to 6 million users and 800 million ratings. We find that ALS-NCG converges significantly faster than ALS in both the serial and distributed Spark settings, demonstrating the overall speedup provided by our algorithmic acceleration.

¹Source code available: <https://github.com/mbhynes/als-ncg>

II. PROBLEM DESCRIPTION

The acceleration approach we propose in this paper is applicable to a broad class of optimization methods and collaborative filtering models. For definiteness, we choose a specific latent factor model, the matrix factorization model from [8] and [23], and a specific optimization method, ALS [23]. Given the data we use, the model is presented in terms of users and movies instead of the more generic users and items framework.

Let the matrix of user-movie rankings be represented by $\mathbf{R} = \{r_{ij}\}_{n_u \times n_m}$ where r_{ij} is the rating given to movie j by user i , n_u is the number of users, and n_m is the number of items. Note that for any user i and movie j , the value of r_{ij} is either a real number or is missing, and in practice very few values are known. For example, the MovieLens 20M dataset [25] with 138,493 users and 27,278 movies contains only 20 million rankings, accounting for less than 1% of the total possible rankings. In the low-rank factorization of \mathbf{R} with n_f factors, each user i is associated with a vector $\mathbf{u}_i \in \mathbb{R}^{n_f}$ ($i = 1, \dots, n_u$), and each movie j is associated with a vector $\mathbf{m}_j \in \mathbb{R}^{n_f}$ ($j = 1, \dots, n_m$). The elements of \mathbf{m}_j measure the degree that movie j possesses each factor or feature, and the elements of \mathbf{u}_i similarly measures the affinity of user i for each factor or feature. The dot product $\mathbf{u}_i^T \mathbf{m}_j$ thus captures the interaction between user i and movie j , approximating user i 's rating of movie j as $r_{ij} \approx \mathbf{u}_i^T \mathbf{m}_j$. Denoting $\mathbf{U} = [\mathbf{u}_i] \in \mathbb{R}^{n_f \times n_u}$ as the user feature matrix and $\mathbf{M} = [\mathbf{m}_j] \in \mathbb{R}^{n_f \times n_m}$ as the movie feature matrix, our goal is to determine \mathbf{U} and \mathbf{M} such that $\mathbf{R} \approx \mathbf{U}^T \mathbf{M}$ by minimizing the following squared loss function:

$$\mathcal{L}_\lambda(\mathbf{R}, \mathbf{U}, \mathbf{M}) = \sum_{(i,j) \in \mathcal{I}} (r_{ij} - \mathbf{u}_i^T \mathbf{m}_j)^2 + \lambda \left(\sum_i n_{u_i} \|\mathbf{u}_i\|^2 + \sum_j n_{m_j} \|\mathbf{m}_j\|^2 \right), \quad (1)$$

where \mathcal{I} is the index set of known r_{ij} in \mathbf{R} , n_{u_i} denotes the number of ratings by user i , and n_{m_j} is the number of ratings of movie j . The term $\lambda(\sum_i n_{u_i} \|\mathbf{u}_i\|^2 + \sum_j n_{m_j} \|\mathbf{m}_j\|^2)$ is a Tikhonov regularization [26] term commonly included in the loss function to prevent overfitting. The full optimization problem can be stated as

$$\min_{\mathbf{U}, \mathbf{M}} \mathcal{L}_\lambda(\mathbf{R}, \mathbf{U}, \mathbf{M}). \quad (2)$$

III. ACCELERATING ALS CONVERGENCE BY NCG

A. Alternating Least Squares Algorithm

The optimization problem in (2) is not convex. However, if we fix one of the unknowns, either \mathbf{U} or \mathbf{M} , then the optimization problem becomes quadratic and we can solve for the remaining unknown as a least squares problem. Doing this in an alternating fashion is the central idea behind ALS.

Consider the first step of the ALS algorithm in which \mathbf{M} is fixed. We can determine the least squares solution to (1) for each \mathbf{u}_i by setting all the components of the gradient of (1)

related to \mathbf{u}_i to zero: $\frac{\partial \mathcal{L}_\lambda}{\partial u_{ki}} = 0 \quad \forall i, k$ where u_{ki} is an element of \mathbf{U} . Expanding the terms in the derivative of (1), we have

$$\begin{aligned} \sum_{j \in \mathcal{I}_i} 2(\mathbf{u}_i^T \mathbf{m}_j - r_{ij}) \mathbf{m}_{kj} + 2\lambda n_{u_i} u_{ki} &= 0 \quad \forall i, k \\ \Rightarrow \sum_{j \in \mathcal{I}_i} m_{kj} \mathbf{u}_i^T \mathbf{m}_j + \lambda n_{u_i} u_{ki} &= \sum_{j \in \mathcal{I}_i} m_{kj} r_{ij} \quad \forall i, k, \end{aligned}$$

where m_{kj} is an element of \mathbf{M} . In vector form, the resultant linear system for any \mathbf{u}_i is

$$(\mathbf{M}_{\mathcal{I}_i} \mathbf{M}_{\mathcal{I}_i}^T + \lambda n_{u_i} \mathbf{I}) \mathbf{u}_i = \mathbf{M}_{\mathcal{I}_i} \mathbf{R}^T(i, \mathcal{I}_i) \quad \forall i,$$

where \mathbf{I} is the $n_f \times n_f$ identity matrix, \mathcal{I}_i is the index set of movies user i has rated, and $\mathbf{M}_{\mathcal{I}_i}$ represents the sub-matrix of \mathbf{M} where columns $j \in \mathcal{I}_i$ are selected. Similarly, $\mathbf{R}(i, \mathcal{I}_i)$ is a row vector that represents the i th row of \mathbf{R} with only the columns in \mathcal{I}_i included. The explicit solution for \mathbf{u}_i is then given by

$$\mathbf{u}_i = \mathbf{A}_i^{-1} \mathbf{v}_i \quad \forall i, \quad (3)$$

where $\mathbf{A}_i = \mathbf{M}_{\mathcal{I}_i} \mathbf{M}_{\mathcal{I}_i}^T + \lambda n_{u_i} \mathbf{I}$, and $\mathbf{v}_i = \mathbf{M}_{\mathcal{I}_i} \mathbf{R}^T(i, \mathcal{I}_i)$. The analogous solution for the columns of \mathbf{M} is found by fixing \mathbf{U} , where each \mathbf{m}_j is given by

$$\mathbf{m}_j = \mathbf{A}_j^{-1} \mathbf{v}_j \quad \forall j, \quad (4)$$

where $\mathbf{A}_j = \mathbf{U}_{\mathcal{I}_j} \mathbf{U}_{\mathcal{I}_j}^T + \lambda n_{m_j} \mathbf{I}$, $\mathbf{v}_j = \mathbf{U}_{\mathcal{I}_j} \mathbf{R}(\mathcal{I}_j, j)$. Here, \mathcal{I}_j is the index set of users that have rated movie j , $\mathbf{U}_{\mathcal{I}_j}$ represents the sub-matrix of $\mathbf{U} \in \mathbb{R}^{n_f \times n_u}$ where columns $i \in \mathcal{I}_j$ are selected, and $\mathbf{R}(\mathcal{I}_j, j)$ is a column vector that represents the j th column of \mathbf{R} with only the rows in \mathcal{I}_j included.

Algorithm 1 summarizes the ALS algorithm used to solve the optimization problem given in (2). From (3) and (4), we note that each of the columns of \mathbf{U} and \mathbf{M} may be computed independently; thus ALS may be easily implemented in parallel, as in [23]. However, the ALS algorithm can require many iterations for convergence, and we now propose an acceleration method for ALS that can be applied to both the parallel and serial versions.

B. Accelerated ALS Algorithm

In this section, we develop the accelerated ALS-NCG algorithm to solve the collaborative filtering optimization problem

Algorithm 1: Alternating Least Squares (ALS)

Output: \mathbf{U}, \mathbf{M}
Initialize \mathbf{M} with random values;
while *Stopping criteria have not been satisfied* **do**
 for $i = 1, \dots, n_u$ **do**
 $\mathbf{u}_i \leftarrow \mathbf{A}_i^{-1} \mathbf{v}_i$;
 end
 for $j = 1, \dots, n_m$ **do**
 $\mathbf{m}_j \leftarrow \mathbf{A}_j^{-1} \mathbf{v}_j$;
 end
end

minimizing (1). In practice we found that the nonlinear conjugate gradient algorithm by itself was very slow to converge to a solution of (2), and thus do not consider it as an alternative to the ALS algorithm. Instead we propose using NCG to accelerate ALS, or, said differently, the combined algorithm uses ALS as a nonlinear preconditioner for NCG [12].

The standard NCG algorithm is a line search algorithm in continuous optimization that is an extension of the CG algorithm for linear systems. While the linear CG algorithm is specifically designed to minimize the convex quadratic function $\phi(\mathbf{x}) = \frac{1}{2} \mathbf{x}^T \mathbf{A} \mathbf{x} - \mathbf{b}^T \mathbf{x}$, where $\mathbf{A} \in \mathbb{R}^{n \times n}$ is a symmetric positive definite matrix, the NCG algorithm can be applied to general constrained optimization problems of the form $\min_{\mathbf{x} \in \mathbb{R}^n} f(\mathbf{x})$. Here, the minimization problem is (2), where the matrices \mathbf{U} and \mathbf{M} are found by defining the vector $\mathbf{x} \in \mathbb{R}^{n_f \times (n_u + n_m)}$ as

$$\mathbf{x}^T = [\mathbf{u}_1^T \mathbf{u}_2^T \dots \mathbf{u}_{n_u}^T \mathbf{m}_1^T \mathbf{m}_2^T \dots \mathbf{m}_{n_m}^T] \quad (5)$$

with function $f(\mathbf{x}) = \mathcal{L}_\lambda$ as in (1).

The NCG algorithm generates a sequence of iterates \mathbf{x}_i , $i \geq 1$, from the initial guess \mathbf{x}_0 using the recurrence relation

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k.$$

The parameter $\alpha_k > 0$ is the step length determined by a line search along search direction \mathbf{p}_k , which is generated by the following rule:

$$\mathbf{p}_{k+1} = -\mathbf{g}_{k+1} + \beta_{k+1} \mathbf{p}_k, \quad \mathbf{p}_0 = -\mathbf{g}_0, \quad (6)$$

where β_{k+1} is the update parameter and $\mathbf{g}_k = \nabla f(\mathbf{x}_k)$ is the gradient of $f(\mathbf{x})$ evaluated at \mathbf{x}_k . The update parameter β_{k+1} can take on various different forms. In this paper, we use the variant of β_{k+1} developed by Polak and Ribière [27]:

$$\beta_{k+1} = \frac{\mathbf{g}_{k+1}^T (\mathbf{g}_{k+1} - \mathbf{g}_k)}{\mathbf{g}_k^T \mathbf{g}_k}. \quad (7)$$

Note that if a convex quadratic function is optimized using the NCG algorithm with an exact line search, then (7) reduces to the same β_{k+1} as in the original CG algorithm for linear systems [11].

The preconditioning of the NCG algorithm by the ALS algorithm modifies the expressions for β_{k+1} and \mathbf{p}_{k+1} as follows, and is summarized in Algorithm 2. Let $\bar{\mathbf{x}}_k$ be the iterate generated from one iteration of the ALS algorithm applied to \mathbf{x}_k , $\bar{\mathbf{x}}_k = P(\mathbf{x}_k)$, where P represents one iteration of ALS. This iterate is incorporated into the NCG algorithm by defining the preconditioned gradient direction generated by ALS as

$$\bar{\mathbf{g}}_k = \mathbf{x}_k - \bar{\mathbf{x}}_k = \mathbf{x}_k - P(\mathbf{x}_k),$$

and replacing \mathbf{g}_k with $\bar{\mathbf{g}}_k$ in (6). Note that $-\bar{\mathbf{g}}_k$ is expected to be a descent direction that is an improvement compared to the steepest descent direction $-\mathbf{g}_k$. The update parameter β_{k+1} is redefined as $\bar{\beta}_{k+1}$ with the form

$$\bar{\beta}_{k+1} = \frac{\bar{\mathbf{g}}_{k+1}^T (\mathbf{g}_{k+1} - \mathbf{g}_k)}{\bar{\mathbf{g}}_k^T \mathbf{g}_k}, \quad (8)$$

Algorithm 2: Accelerated ALS (ALS-NCG)

Input: \mathbf{x}_0
Output: \mathbf{x}_k
 $\bar{\mathbf{g}}_0 \leftarrow \mathbf{x}_0 - P(\mathbf{x}_0);$
 $\mathbf{p}_0 \leftarrow -\bar{\mathbf{g}}_0;$
 $k \leftarrow 0;$
while $\mathbf{g}_k \neq 0$ **do**
 Compute $\alpha_k;$
 $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + \alpha_k \mathbf{p}_k;$
 $\bar{\mathbf{g}}_{k+1} \leftarrow \mathbf{x}_{k+1} - P(\mathbf{x}_{k+1});$
 Compute $\bar{\beta}_{k+1};$
 $\mathbf{p}_{k+1} \leftarrow -\bar{\mathbf{g}}_{k+1} + \bar{\beta}_{k+1} \mathbf{p}_k;$
 $k \leftarrow k + 1;$
end

where (8) is similar to (7), however not every instance of \mathbf{g}_k has been replaced with $\bar{\mathbf{g}}_k$. The specific form for $\bar{\beta}_{k+1}$ is chosen because if Algorithm 2 were applied to the convex quadratic problem with an exact line search and P were represented by a preconditioning matrix \mathbf{P} , then Algorithm 2 would be equivalent to preconditioned CG with preconditioner \mathbf{P} . In [12], an overview and an in-depth analysis are given of the different possible forms for $\bar{\beta}_{k+1}$, however (8) performed best in our numerical experiments. Note that the primary computational cost in Algorithm 2 comes from computing both the ALS iteration, $P(\mathbf{x}_k)$, and α_k , the step length parameter using a line search.

IV. SERIAL PERFORMANCE OF ALS-NCG

The ALS and ALS-NCG algorithms were implemented in serial in MATLAB, and evaluated using the MovieLens 20M dataset [25]. The entire MovieLens 20M dataset has 138,493 users, 27,278 movies, and just over 20 million ratings, where each user has rated at least 20 movies. To investigate the algorithmic performance as a function of problem size, both the ALS and ALS-NCG algorithms were run on subsets of the MovieLens 20M dataset. In creating subsets, we excluded outlier users with either very many or very few movie ratings relative to the median number of ratings per user. To construct a subset with n_u users, the users from the full dataset were sorted in descending order by the values of n_{u_i} for each user. Denoting the index of the user with the median number of ratings by c , the set of users from index $c - \lfloor \frac{n_u}{2} \rfloor$ to $c + (\lceil \frac{n_u}{2} \rceil - 1)$ were included. Once the users were determined, the same process was used to select the movies, where the ratings per movie were computed only for the chosen users.

All serial experiments were performed on a linux workstation with a quad-core 3.16 GHz processor (Xeon X5460) and 8 GB of RAM. For the ALS-NCG algorithm, the Moré-Thuente line search algorithm from the Poblano toolbox [28] was used to compute α_k . The line search parameters were as follows: 10^{-4} for the sufficient decrease condition tolerance, 10^{-2} for the curvature condition tolerance, and an initial step length of 1 and a maximum of 20 iterations. The stopping criteria for both ALS and ALS-NCG were the maximum number of

iterations as well as a desired tolerance value in the gradient norm normalized by the number of variables, $\frac{1}{N} \|\mathbf{g}_k\|$ for $N = n_f \times (n_u + n_m)$. For both algorithms, a normalized gradient norm of less than 10^{-6} was required within at most 10^4 iterations. In addition, ALS-NCG had a maximum number of allowed function evaluations in the line search equal to 10^7 .

The serial tests were performed on ratings matrices of 4 different sizes: $n_u \times n_m = 400 \times 80, 800 \times 160, 1600 \times 320$ and 3200×640 , where n_u is the number of users and n_m is the number of movies. For each ratings matrix, ALS and ALS-NCG were used to solve the optimization problem in (2) with $\lambda = 0.1$ and $n_f = 10$. The algorithms were each run using 20 different random starting iterates (which were the same for both algorithms) until one of the stopping criteria was reached. Table I summarizes the timing results for different problem sizes, where the given times are written in the form $a \pm b$ where a is the mean time in seconds and b is the standard deviation about the mean. Since computing the gradient is not explicitly required in the ALS algorithm, the computation time for the gradient norm was excluded from the timing results for the ALS algorithm. Runs that did not converge based on the gradient norm tolerance were not included in the mean and standard deviation calculations, however the only run that did not converge to $\frac{1}{N} \|\mathbf{g}_k\| < 10^{-6}$ before reaching the maximum number of iterations was a single 1600×320 ALS run. The large standard deviations in the timing measurements stem from the variation in the number of iterations required to reduce the gradient norm. The fourth column of Table I shows the acceleration factor of ALS-NCG, computed as the mean time for convergence of ALS divided by the mean time for convergence of ALS-NCG. We see from this table that ALS-NCG significantly accelerates the ALS algorithm for all problem sizes. Similarly, Table II summarizes the number of iterations required to reach convergence for each algorithm. Again, the results were calculated based on converged runs only.

From Tables I and II it is clear that ALS-NCG accelerates the convergence of the ALS algorithm, using the gradient norm as the measure of convergence. However, since the factor

TABLE I
TIMING RESULTS OF ALS AND ALS-NCG.

Problem Size $n_u \times n_m$	Time (s)		Acceleration Factor
	ALS	ALS-NCG	
400×80	56.50 ± 38.06	12.22 ± 4.25	4.62
800×160	162.0 ± 89.94	47.57 ± 20.02	3.41
1600×320	30.61 ± 120.3	116.2 ± 31.56	2.84
3200×640	960.8 ± 364.0	303.7 ± 111.3	3.16

TABLE II
ITERATION RESULTS OF ALS AND ALS-NCG.

Problem Size $n_u \times n_m$	Number of Iterations		Acceleration Factor
	ALS	ALS-NCG	
400×80	2181 ± 1466	158.8 ± 54.3	12.74
800×160	3048 ± 1689	290.4 ± 128.1	10.50
1600×320	3014 ± 1098	302.9 ± 86.3	9.95
3200×640	4231 ± 1602	329.6 ± 127.5	12.84

matrices \mathbf{U} and \mathbf{M} are used to make recommendations, we would also like to examine the convergence of the algorithms in terms of the accuracy of the resultant recommendations. In particular, we are interested in the rankings of the top t movies (e.g. top 20 movies) for each user, and want to explore how these rankings change with increasing number of iterations for both ALS and ALS-NCG. If the rankings of the top t movies for each user no longer change, then the algorithm has likely computed an accurate solution.

To measure the relative difference in rankings we use a metric that is based on the number of pairwise swaps required to convert a vector of movie rankings \mathbf{p}_2 into another vector \mathbf{p}_1 , but only for the top t movies. We use a modified Kendall-Tau [29] distance to compute the difference between ranking vectors based only on the rankings of the top t items, normalize the distance to range in $[0, 1]$, and subsequently average the distances over all users. To illustrate the ranking metric, consider the following example, where $\mathbf{p}_1 = [6, 3, 1, 2, 4, 5]$, and $\mathbf{p}_2 = [3, 4, 2, 5, 6, 1]$ are two different rankings of movies for user i , and let $t = 2$. We begin by finding the top t movies in \mathbf{p}_1 . Here, user i ranks movie 6 highest. In \mathbf{p}_2 , movie 6 is ranked 5th, and there are 4 pairwise inversions required to place movie 6 in the first component of \mathbf{p}_2 , producing a new ranking vector for the second iteration, $\tilde{\mathbf{p}}_2 = [6, 3, 4, 2, 5, 1]$. The 2nd highest ranked movie in \mathbf{p}_1 is movie 3. In $\tilde{\mathbf{p}}_2$, movie 3 is already ranked 2nd; as such, no further inversions are required. In this case, the total number of pairwise swaps required to match \mathbf{p}_2 to \mathbf{p}_1 for the top 2 rankings is $s_i = 4$. The distance s_i is normalized to 1 if no inversions were needed (i.e. $\mathbf{p}_1 = \mathbf{p}_2$ in the top t spots) and 0 if the maximum number of inversions are needed. The maximum number of inversions occurs if \mathbf{p}_2 and \mathbf{p}_1 are in opposite order, requiring $n_m - 1$ inversions in the first step, $n_m - 2$ inversions in the second step, and so on until $n_m - t$ inversions are needed in the t -th step. Thus, the maximum number of inversions is $s_{\max} = (n_m - 1) + (n_m - 2) + \dots + (n_m - t) = \frac{t}{2}(2n_m - t - 1)$, yielding the ranking accuracy metric for user i as

$$q_i = 1 - \frac{s_i}{s_{\max}}.$$

The total ranking accuracy for an algorithm is taken as the average value of q_i across all users relative to ranking of the top t movies for each user produced from the solution obtained after the algorithm converged.

The normalized Kendall-Tau ranking metric for the top t rankings described above was used to evaluate the accuracy of ALS and ALS-NCG as a function of running time for $t = 20$. Tables III and IV summarize the time needed for each

TABLE III
RANKING ACCURACY TIMING RESULTS FOR PROBLEM SIZE 400×80

Ranking Accuracy	Time (s)		Acceleration Factor
	ALS	ALS-NCG	
70%	0.37 ± 0.17	0.65 ± 0.16	0.58
80%	7.88 ± 7.03	2.87 ± 1.72	2.74
90%	37.08 ± 37.62	6.92 ± 4.47	5.36
100%	101.29 ± 68.04	14.07 ± 5.25	7.20

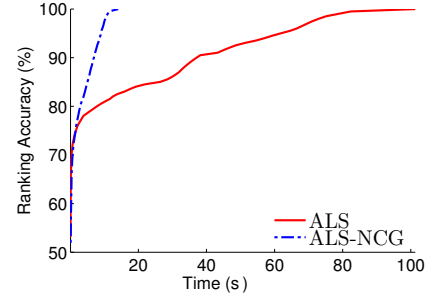


Fig. 1. Average ranking accuracy versus time for problem size 400×80 .

algorithm to reach a specified percentage ranking accuracy for ratings matrices with different sizes. Both algorithms were run with 20 different random starting iterates, and the time to reach a given ranking accuracy is written in the form $a \pm b$ where a is the mean time across all converged runs and b is the standard deviation about the mean. In the fourth column of Tables III and IV, we have computed the acceleration factor of ALS-NCG as the ratio of the mean time for ALS to the mean time for ALS-NCG. To reach 70% accuracy, ALS is faster for both problem sizes, however for accuracies greater than 70%, ALS-NCG converges in significantly less time than ALS. Thus, if an accurate solution is desired, ALS-NCG is much faster in general than ALS. This is further illustrated in Fig. 1, which shows the average time needed for both algorithms to reach a given ranking accuracy for the 400×80 ratings matrix. Here, both ALS and ALS-NCG reach a ranking accuracy of 75% in less than a second, however it then takes ALS approximately 100 s to increase the ranking accuracy from 75% to 100%, while ALS-NCG reaches the final ranking in only 14 s.

V. PARALLEL IMPLEMENTATION OF ALS-NCG IN SPARK

A. Apache Spark

Apache Spark is a fault-tolerant, in-memory cluster computing framework designed to supersede MapReduce by maintaining program data in memory as much as possible between distributed operations. The Spark environment is built upon two components: a data abstraction, termed a resilient distributed dataset (RDD) [22], and the task scheduler, which uses a *delay scheduling* algorithm [30]. We describe the fundamental aspects of RDDs and the scheduler below.

Resilient Distributed Datasets. RDDs are immutable, distributed datasets that are evaluated lazily via their provenance information—that is, their functional relation to other RDDs or datasets in stable storage. To describe an RDD, consider

TABLE IV
RANKING ACCURACY TIMING RESULTS FOR PROBLEM SIZE 800×160 .

Ranking Accuracy	Time (s)		Acceleration Factor
	ALS	ALS-NCG	
70%	0.76 ± 0.34	1.43 ± 0.35	0.53
80%	19.83 ± 13.67	12.61 ± 10.13	1.57
90%	103.91 ± 70.97	33.25 ± 22.34	3.12
100%	310.98 ± 168.67	59.88 ± 28.20	5.19

an immutable distributed dataset D of k records with homogeneous type: $D = \bigcup_i^k d_i$ with $d_i \in \mathcal{D}$. The distribution of D across a computer network of nodes $\{v_\alpha\}$, such that d_i is stored in memory or on disk on node v_α , is termed its *partitioning* according to a partition function $P(d_i) = v_\alpha$. If D is expressible as a finite sequence of deterministic operations on other datasets D_1, \dots, D_l that are either RDDs or persistent records, then its lineage may be written as a directed acyclic graph \mathcal{L} formed with the parent datasets $\{D_i\}$ as the vertices, and the operations along the edges. Thus, an RDD of type \mathcal{D} (written RDD $[D]$) is the tuple (D, P, \mathcal{L}) .

Physically computing the records $\{d_i\}$ of an RDD is termed its *materialization*, and is managed by the Spark scheduler program. To allocate computational tasks to the compute nodes, the scheduler traverses an RDD's lineage graph \mathcal{L} and divides the required operations into stages of local computations on parent RDD partitions. Suppose that $R_0 = (\bigcup_i x_i, P_0, \mathcal{L}_0)$ were an RDD of numeric type RDD $[\mathbb{R}]$, and let $R_1 = (\bigcup_i y_i, P_1, \mathcal{L}_1)$ be the RDD resulting from the application of function $f: \mathbb{R} \rightarrow \mathbb{R}$ to each record of R_0 . To compute $\{y_i\}$, R_1 has only a single parent in the graph \mathcal{L}_1 , and hence the set of tasks to perform is $\{f(x_i)\}$. This type of operation is termed a *map* operation. If $P_1 = P_0$, \mathcal{L}_1 is said to have a *narrow* dependency on R_0 : each y_i may be computed locally from x_i , and the scheduler would allocate the task $f(x_i)$ to a node that stores x_i .

Stages consist only of local map operations, and are bounded by *shuffle* operations that require communication and data transfer between the compute nodes. For example, shuffling is necessary to perform *reduce* operations on RDDs, wherein a scalar value is produced from an associative binary operator applied to each element of the dataset. In implementation, a shuffle is performed by writing the results of the tasks in the preceding stage, $\{f(x_i)\}$, to a local file buffer. These shuffle files may or may not be written to disk, depending on the operating system's page table, and are fetched by remote nodes as needed in the subsequent stage.

Delay Scheduling and Fault Tolerance. The simple delay scheduling algorithm [30] prioritizes data locality when submitting tasks to the available compute nodes. If v_α stores the needed parent partition x_i to compute task $f(x_i)$, but is temporarily unavailable due to faults or stochastic delays, rather than submitting the task on another node, the scheduler will wait until v_α is free. However, if v_α does not become available within a specified maximum delay time (several seconds in practice), the scheduler will resubmit the tasks to a different compute node. However, as x_i is not available in memory on the different node, the lineage \mathcal{L}_0 of the RDD R_0 must be traversed further, and the tasks required to compute x_i from the parent RDDs of R_0 will be submitted for computation in addition to $f(x_i)$. Thus, fault tolerance is achieved in the system through recomputation.

B. ALS Implementation in Spark

The Apache Spark codebase contains a parallel implementation of ALS for the collaborative filtering model of [23],

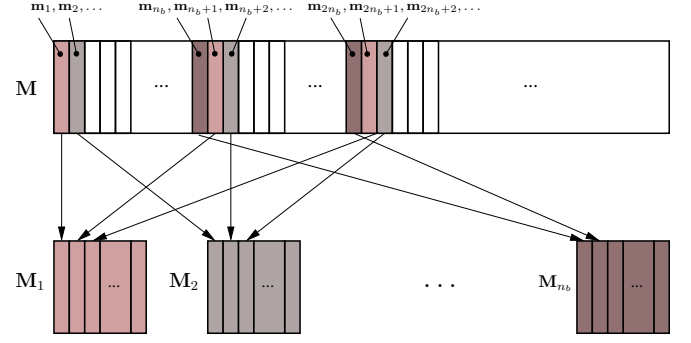


Fig. 2. Hash partitioning of the columns of M , depicted as rectangles, into n_b blocks M_1, \dots, M_{n_b} . Each block M_{j_b} and its index j_b forms a partition of the M RDD of type RDD $[(j_b, M_{j_b})]$.

[8]; see [24]. We briefly outline the main execution and relevant optimizations of the existing ALS implementation. The implementation stores the factor matrices U and M as single precision column block matrices, with each block as an RDD partition. A U column block stores factor vectors for a subset of users, and an M column block stores factor vectors for a subset of movies. The ratings matrix is stored twice: both R and R^T are stored in separate RDDs, partitioned in row blocks (i.e., R is partitioned by users and R^T by movies). Row blocks of R and R^T are stored in a compressed sparse matrix format. The U and R RDDs have the same partitioning, such that a node that stores a partition of U also stores the partition of R such that the ratings for each user in its partitions of U are locally available. When updating a U block according to (3), the required ratings are available in a local R block, but the movie factor vectors in M corresponding to the movies rated by the users in a local U block must be shuffled across the network. These movie factors are fetched from different nodes, and, as explained below, an optimized routing table strategy is used from [24] that avoids sending duplicate information. Similarly, updating a block of M according to (4) uses ratings data stored in a local R^T block, but requires shuffling of U factor vectors using a second routing table.

Block Partitioning. All RDDs are partitioned into n_b partitions, where n_b is an integer multiple of the number of available compute cores in practice. For example, M is divided into column blocks M_{j_b} with block (movie) index $j_b \in \{0, \dots, n_b - 1\}$ by hash partitioning the movie factor vectors such that $m_j \in R_{j_b}$ if $j \equiv j_b \pmod{n_b}$ as in Fig. 2. Similarly, U is hash partitioned into column blocks U_{i_b} with block (user) index $i_b \in \{0, \dots, n_b - 1\}$. The RDDs for M and U can be taken as type RDD $[(j_b, M_{j_b})]$ and RDD $[(i_b, U_{i_b})]$, where the blocks are tracked by the indices j_b and i_b . R is partitioned by rows (users) into blocks with type RDD $[(i_b, R_{i_b})]$ with the same partitioning as the RDD representing U (and similarly for the R^T and M RDDs). By sharing the same user-based partitioning scheme, the blocks R_{i_b} and U_{i_b} are normally located on the same compute node, except when faults occur. The same applies to $R_{j_b}^T$ and M_{j_b} due to the movie-based partitioning scheme.

Routing Table. Fig. 3 shows how a routing table optimizes

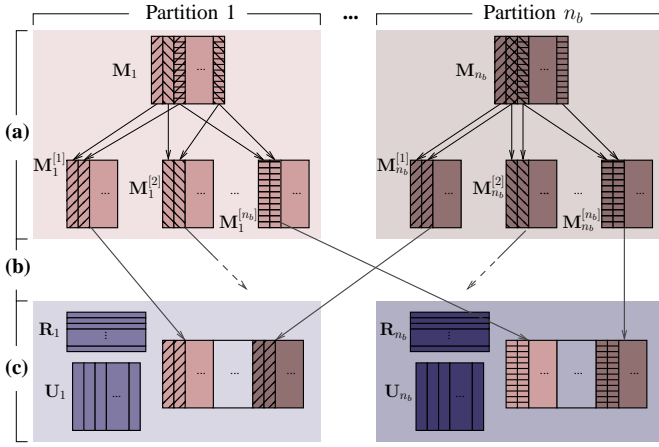


Fig. 3. Schematic of the use of the routing table $T_m(m_j)$ in the Spark ALS shuffle. In (a) the blocks $\{M_{j_b}\}$ are filtered using $T_m(m_j)$ for each destination R_{i_b} , and shuffled to the respective blocks in (b), where arrows between the shaded backgrounds represent network data transfer between different compute nodes. In (c), when updating block U_{i_b} , the ratings information is locally available in R_{i_b} .

data shuffling in ALS. Suppose we want to update the user factor block U_{i_b} according to (3). The required ratings data, R_{i_b} , is stored locally, but a global shuffle is required to obtain all movie factor vectors in M that correspond to the movies rated by the users in U_{i_b} . To optimize the data transfer, a routing table $T_m(m_j)$ is constructed by determining, for each of the movie factor blocks M_{j_b} , which factor vectors have to be sent to each partition of R_{i_b} (that may reside on other nodes). In Fig. 3 (a), the blocks M_{j_b} are filtered using $T_m(m_j)$ such that a given $m_j \in M_{j_b}$ is written to the buffer destined for R_{i_b} , $M_{j_b}^{[i_b]}$, only once regardless of how many u_i in U_{i_b} have ratings for movie j , and only if there is at least one u_i in U_{i_b} that has rated movie j . This is shown by the hatching of each m_j vector in Fig. 3 (a); for instance, the first column in M_1 is written only to $M_1^{[1]}$ and has one set of hatching lines, but the last column is written to both $M_1^{[2]}$ and $M_1^{[n_b]}$ and correspondingly has two sets of hatching lines. Once the buffers are constructed, they are shuffled to the partitions of R , as in Fig. 3 (b) such that both the movie factors and ratings are locally available to compute the new U_{i_b} block, as in Fig. 3 (c). The routing table formulation for shuffling U , with mapping $T_u(u_i)$, is analogous. Note that the routing tables are constructed before the while loop in Algorithm 1, and hence do not need recomputation in each iteration.

Evaluation of u_i and m_j via (3) and (4). As in Fig. 3 (c), a compute node that stores R_{i_b} , will obtain n_b buffered arrays of filtered movie factors. Once the factors have been shuffled, A_j is computed for each u_i as $\sum_{j \in \mathcal{I}_i} m_j m_j^T + n_{u_i} I$, and v_j as $\sum_{j \in \mathcal{I}_i} r_{ij} m_j$ using the Basic Linear Algebra Subprograms (BLAS) library [31]. The resulting linear system for u_i is then solved via the Cholesky decomposition using LAPACK routines [32], giving the computation to update U an asymptotic complexity of $\mathcal{O}(n_u n_f^3)$ since n_u linear systems must be solved. Solving for m_j is an identical operation with the appropriate routing table and has $\mathcal{O}(n_m n_f^3)$ complexity.

C. ALS-NCG Implementation in Spark

We now discuss our contributions in parallelizing ALS-NCG for Spark. Since the calculation of α_k in Algorithm 2 requires a line search, the main technical challenges we address are how to compute the loss function in (1) and its gradient in an efficient way in Spark, obtaining good parallel performance. To this end, we formulate a backtracking line search procedure that dramatically reduces the cost of multiple function evaluations, and we take advantage of the routing table mechanism to obtain fast communication. We also extend the ALS implementation in Spark to support the additional NCG vector operations required in Algorithm 2 using BLAS routines.

Vector Storage. The additional vectors \bar{x} , \bar{g} , and \bar{p} were each split into two separate RDDs, such that blocks corresponding to the components of u_i (see (5)) were stored in one RDD and partitioned in the same way as U with block index i_b . Analogously, blocks corresponding to components of m_j were stored in another RDD, partitioned in the same way as M with block index j_b . This ensured that all vector blocks were aligned component-wise for vector operations; furthermore, the \bar{p} blocks could also be shuffled efficiently using the routing tables in the line search (see below).

Vector Operations. RDDs have a standard operation termed a *join*, in which two RDDs R_1 and R_2 representing the datasets of tuples $\bigcup_i (k_i, a_i)$ and $\bigcup_i (k_i, b_i)$ with type $(\mathcal{K}, \mathcal{A})$ and $(\mathcal{K}, \mathcal{B})$, respectively, are combined to produce an RDD R_3 of type RDD $[(\mathcal{K}, (\mathcal{A}, \mathcal{B}))]$, where $R_3 = R_1 \text{.join}(R_2)$ represents the dataset $\bigcup_i (k_i, (a_i, b_i))$ of combined tuples and k_i is a key common to both R_1 and R_2 . Parallel vector operations between RDD representations of blocked vectors x and y were implemented by joining the RDDs by their block index i_b and calling BLAS level 1 interfaces on the vectors within the resultant tuples of aligned vector blocks, $\{(x_{i_b}, y_{i_b})\}$. Since the RDD implementations of vectors had the same partitioning schemes, this operation was local to a compute node, and hence very inexpensive. One caveat, however, is that BLAS subprograms generally perform modifications to vectors *in place*, overwriting their previous components. For fault tolerance, RDDs must be immutable; as such, whenever BLAS operations were required on the records of an RDD, an entirely new vector was allocated in memory and the RDD's contents copied to it. Algorithm 3 shows the operations required for the vector addition $ax + by$ using the BLAS `axpyby` routine (note that the result overwrites y in place). The inner product of two block vector RDDs and norm of a single block vector RDD were implemented in a similar manner to vector addition, with an additional reduce operation to sum the scalar component-wise dot products. These vector operations were used to compute β_{k+1} in (8).

Line Search & Loss Function Evaluation. We present a computationally cheap way to implement a backtracking line search in Spark for minimizing (1). A backtracking line search minimizes a function $f(x)$ along a descent direction \bar{p} by decreasing the step size in each iteration by a factor of

Algorithm 3: RDD Block Vector BLAS .axpby

Input: $\mathbf{x} = \text{RDD}[(i_b, \mathbf{x}_{i_b})]$; $\mathbf{y} = \text{RDD}[(i_b, \mathbf{y}_{i_b})]$; $a, b \in \mathbb{R}$
Output: $\mathbf{z} = a\mathbf{x} + b\mathbf{y}$
 $\mathbf{z} \leftarrow \mathbf{x}.\text{join}(\mathbf{y}).\text{map}\{$
 Allocate \mathbf{z}_{i_b} ;
 $\mathbf{z}_{i_b} \leftarrow \mathbf{y}_{i_b}$;
 Call BLAS .axpby($a, \mathbf{x}_{i_b}, b, \mathbf{z}_{i_b}$);
 Yield (i_b, \mathbf{z}_{i_b}) ;
}

$\tau \in (0, 1)$, and terminates once a sufficient decrease in $f(\mathbf{x})$ is achieved. This simple procedure is summarized in Algorithm 4, where it is important to note that a line search requires computing \mathbf{g} once, as well as $f(\mathbf{x})$ in each iteration of the line search. To avoid multiple shuffles in each line search iteration, instead of performing multiple evaluations of (1) directly, we constructed a polynomial with degree 4 in the step size α by expanding (1) with $\mathbf{u}_i = \mathbf{x}_{u_i} + \alpha \mathbf{p}_{u_i}$ and $\mathbf{m}_j = \mathbf{x}_{m_j} + \alpha \mathbf{p}_{m_j}$, where \mathbf{x}_{u_i} refers to the components of \mathbf{x} related to user i and mutatis mutandis for the vectors \mathbf{p}_{u_i} , \mathbf{x}_{m_j} , and \mathbf{p}_{m_j} . From the bilinearity of the inner product, this polynomial has the form $Q(\alpha) = \sum_{n=0}^4 \left(\sum_{(i,j) \in \mathcal{I}} C_{ij}^{[n]} \right) \alpha^n$, where the terms $C_{ij}^{[n]}$ in the summation for each coefficient only require level 1 BLAS operations between the block vectors \mathbf{x}_{u_i} , \mathbf{p}_{u_i} , \mathbf{x}_{m_j} , and \mathbf{p}_{m_j} for known $(i, j) \in \mathcal{I}$. Thus, coefficients of $Q(\alpha)$ were computed at the beginning of the line search with a *single* shuffle operation using the routing table $T_m(\mathbf{m}_j)$ to match vector pairs with dot products contributing to the coefficients. Here, $T_m(\mathbf{m}_j)$ was chosen since there is far less communication required to shuffle $\{\mathbf{m}_j\}$, as $n_u \gg n_m$. Since each iteration of the line search was very fast after computing the coefficients of $Q(\alpha)$, we used relatively large values of $\tau = 0.9$, $c = 0.5$, and $\alpha_0 = 10$ in Algorithm 4 that searched intensively along direction \mathbf{p} .

Gradient Evaluation. We computed \mathbf{g}_k with respect to a block for \mathbf{u}_i using only BLAS level 1 operations as $2\lambda n_{u_i} \mathbf{u}_i + 2 \sum_{j \in \mathcal{I}_i} \mathbf{m}_j (\mathbf{u}_i^T \mathbf{m}_j - r_{ij})$, with an analogous operation with respect to each \mathbf{m}_j . As this computation requires matching the \mathbf{u}_i and \mathbf{m}_j factors, the routing tables $T_u(\mathbf{u}_i)$ and $T_m(\mathbf{m}_j)$ were used to shuffle \mathbf{u}_i and \mathbf{m}_j consecutively. Evaluating \mathbf{g}_k can be performed with $\mathcal{O}(n_f(n_u \sum_i n_{u_i} + n_m \sum_j n_{m_j}))$ operations, but requires two shuffles. As such, the gradient computation required as much communication as a single iteration of ALS in which both \mathbf{U} and \mathbf{M} are updated.

Algorithm 4: Backtracking Line Search

Input: $\mathbf{x}, \mathbf{p}, \mathbf{g}, \alpha_0, c \in (0, 1), \tau \in (0, 1)$
Output: α_k
 $k \leftarrow 0$;
while $f(\mathbf{x} + \alpha_k \mathbf{p}) - f(\mathbf{x}) > \alpha_k c \mathbf{g}^T \mathbf{p}$ **do**
 $\alpha_{k+1} \leftarrow \tau \alpha_k$;
 $k \leftarrow k + 1$;
end

VI. PARALLEL PERFORMANCE OF ALS-NCG

Our comparison tests of the ALS and ALS-NCG algorithms in Spark were performed on a computing cluster composed of 16 homogeneous compute nodes, 1 storage node hosting a network filesystem, and 1 head node. The nodes were interconnected by a 10 Gb ethernet managed switch (PowerConnect 8164). Each compute node was a 64 bit rack server (PowerEdge R620) running Ubuntu 14.04, with linux kernel 3.13. The compute nodes all had two 8-core 2.60 GHz chips (Xeon E5-2670) and 256 GB of SDRAM. The head node had the same processors as the compute nodes, but had 512 GB of RAM. The single storage node (PowerEdge R720) contained two 2 GHz processors, each with 6 cores (Xeon E5-2620), 64 GB of memory, and 12 hard disk drives of 4 TB capacity and 7200 RPM nominal speed. Finally, compute nodes were equipped with 6 ext4-formatted local SCSI 10k RPM hard disk drives, each with a 600 GB capacity.

Our Apache Spark assembly was built from a snapshot of the 1.3 release using Oracle’s Java 7 distribution, Scala 2.10, and Hadoop 1.0.4. Input files to Spark programs were stored on the storage node in plain text. The SCSI hard drives on the compute nodes’ local filesystems were used as Spark spilling and scratch directories, and the Spark checkpoint directory for persisting RDDs was specified in the network filesystem hosted by the storage node, and accessible to all nodes in the cluster. Shuffle files were consolidated into larger files, as recommended for ext4 filesystems [33]. In our experiments, the Spark master was executed on the head node, and a single instance of a Spark executor was created on each compute node. It was empirically found that the ideal number of cores to make available to the Spark driver was 16 per node, or the number of *physical* cores for a total of 256 available cores. The value of n_b was set to the number of cores in all experiments.

To compare the performance of ALS and ALS-NCG in Spark, the two implementations were tested on the MovieLens 20M dataset with $\lambda = 0.01$ and $n_f = 100$. In both algorithms, the RDDs were checkpointed to persistent storage every 10 iterations, since it is a widely known issue in the Spark community that RDDs with very long lineage graphs cause stack overflow errors when the scheduler recursively traverses their lineage [34]. Since RDDs are materialized via lazy evaluation, to obtain timing measurements, actions were triggered to physically compute the partitions of each block vector RDD at the end of each iteration in Algorithm 1 and 2. For each experimental run of ALS and ALS-NCG, two experiments with the same initial user and movie factors were performed: in one, the gradient norm in each iteration was computed and printed (incurring additional operations); in the other experiment, no additional computations were performed such that the elapsed times for each iteration were correctly measured.

Fig. 4 shows the convergence in gradient norm, normalized by the degrees of freedom $N = n_f \times (n_u + n_m)$, for six separate runs of ALS and ALS-NCG on 8 compute nodes for the MovieLens 20M dataset. The subplots (a) and (b) show

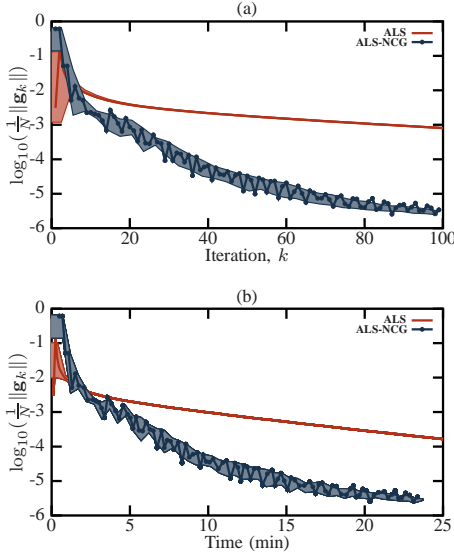


Fig. 4. Convergence in normalized gradient norm $\frac{1}{N} \|g_k\|$ for 6 instances of ALS and ALS-NCG with different starting values in both (a) iteration and (b) clock time, for the MovieLens 20M dataset. The two solid lines in each panel show actual convergence traces for one of the instances, while the shaded regions show the standard deviation about the mean value over all instances, computed for non-overlapping windows of 3 iterations for (a), and 30 s for (b). The experiments were conducted on 8 nodes (128 cores).

$\frac{1}{N} \|g_k\|$ over 100 iterations and 25 minutes, respectively. This time frame was chosen since it took just over 20 minutes for ALS-NCG to complete 100 iterations; note that in Fig. 4 (b), 200 iterations of ALS are shown, since with this problem size it took approximately twice as long to run a single iteration of ALS-NCG. The shaded regions in Fig. 4 show the standard deviation about the mean value for $\frac{1}{N} \|g_k\|$ across all runs, computed for non-overlapping windows of 3 iterations for subplot (a) and 30 s for subplot (b). Even within the uncertainty bounds, ALS-NCG requires much less time and many fewer iterations than ALS to reach accurate values of $\frac{1}{N} \|g_k\|$ (e.g. below 10^{-3}).

The operations that we have implemented in ALS-NCG that are additional to standard ALS in Spark have computational complexity that is linear in problem size. To verify the expected linear scaling, experiments with a constant number of nodes and increasing problem size up to 800 million ratings were performed on 16 compute nodes. Synthetic ratings matrices were constructed by sampling the MovieLens 20M dataset such that the synthetic dataset had the same statistical sparsity, realistically simulating the data transfer patterns in each iteration. To do this, first the dimension n_u of the sampled dataset was fixed, and for each user n_{u_i} was sampled from the empirical probability distribution $p(n_{u_i} | \mathbf{R})$ of how many movies each user ranked, computed empirically from the MovieLens 20M dataset. The n_{u_i} movies were then sampled from the empirical likelihood of sampling the j th movie, $p(\mathbf{m}_j | \mathbf{R})$, and the resultant rating value was sampled from the distribution of numerical values for all ratings. The choice of scaling up the users (for a fixed set of movies) was made

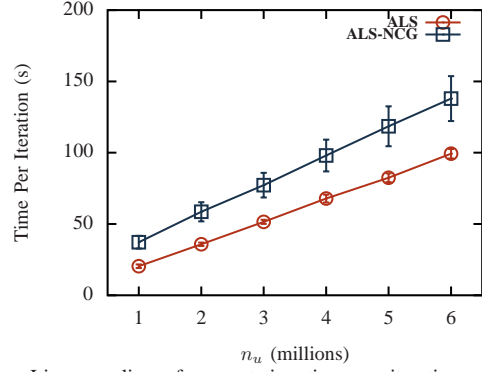


Fig. 5. Linear scaling of computation time per iteration with increasing n_u on a synthetic dataset for ALS and ALS-NCG on 16 compute nodes (256 cores) for up to 6M users, corresponding to 800M ratings.

to model the situation in which an industry's user base grows far more rapidly than its items.

Fig. 5 shows the linear scaling in computation time for both ALS and ALS-NCG. The values shown are average times per iteration over 50 iterations, for n_u from 1 to 6 million, corresponding to the range from 133 to 800 million ratings. The error bars show the uncertainty in this measurement, where the standard deviation takes into account that iterations with and without checkpointing come from two different populations with different average times. The uncertainty in the time per iteration for ALS-NCG is larger due to the greater overhead of memory management and garbage collection by the Java Virtual Machine required with more RDDs. While each iteration of ALS-NCG takes longer due to the additional line search and gradient computations, we note that many fewer iterations are required to converge.

Finally, we compute the relative speedup that was attainable on the large synthetic datasets. For the value of $\frac{1}{N} \|g_k\|$ in each iteration of ALS-NCG, we determined how many iterations of regular ALS were required to achieve an equal or lesser value gradient norm. Due to the local variation in $\frac{1}{N} \|g_k\|$ (as in Fig. 4), a moving average filter over every two iterations was applied to the ALS-NCG gradient norm values. The total time required for ALS and ALS-NCG to reach a given gradient norm was then estimated from the average times per iteration in Fig. 5. The ratios of these total times for ALS and ALS-NCG are shown in Fig. 6 as the relative speedup factor for the 1M, 3M, and 6M users ratings matrices. When an accurate solution is desired, ALS-NCG often achieves faster convergence by a factor of 3 to 5, with the acceleration factor increasing with greater desired accuracy in the solution.

VII. CONCLUSION

In this paper, we have demonstrated how the addition of a nonlinear conjugate gradient wrapper can accelerate the convergence of ALS-based collaborative filtering algorithms when accurate solutions are desired. Furthermore, our parallel ALS-NCG implementation can significantly speed up big data recommendation in the Apache Spark distributed computing environment, and the proposed NCG acceleration can be naturally extended to Hadoop or MPI environments. It may also

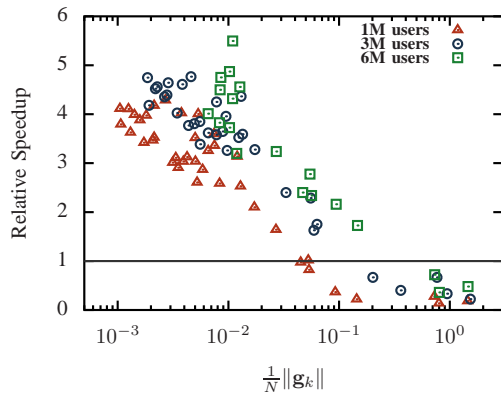


Fig. 6. Speedup of ALS-NCG over ALS as a function of normalized gradient norm on 16 compute nodes (256 cores), for a synthetic problem with up to 6M users and 800M ratings. ALS-NCG can easily outperform ALS by a factor of 4 or more, especially when accurate solutions (small normalized gradients) are required or problem sizes are large.

be applicable to other optimization methods for collaborative filtering. Though we have focused on a simple latent factor model, our acceleration can be used with any collaborative filtering model that uses ALS. We expect that our acceleration approach will be especially useful for advanced collaborative filtering models that achieve low root mean square error (RMSE), since these models require solving the optimization problem accurately, and that is precisely where accelerated ALS-NCG shows the most benefit over standalone ALS.

ACKNOWLEDGEMENTS

This work was supported in part by NSERC of Canada, and by the Scalable Graph Factorization LDRD Project, 13-ERD-072, under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

REFERENCES

- [1] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl, "Item-based collaborative filtering recommendation algorithms," in *Proceedings of the 10th International Conference on World Wide Web*, 2001, pp. 285–295.
- [2] Y. Koren, "Factorization meets the neighborhood: a multifaceted collaborative filtering model," in *14th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2008, pp. 426–434.
- [3] J. Bobadilla, F. Ortega, A. Hernando, and A. Gutiérrez, "Recommender systems survey," *Knowledge-Based Systems*, vol. 46, pp. 109–132, 2013.
- [4] G. Linden, B. Smith, and J. York, "Amazon.com recommendations: Item-to-item collaborative filtering," *IEEE Internet Comput.*, vol. 7, no. 1, pp. 76–80, 2003.
- [5] R. M. Bell and Y. Koren, "Lessons from the Netflix prize challenge," *SIGKDD Explor. Newsl.*, vol. 9, no. 2, pp. 75–79, 2007.
- [6] C. C. Johnson, "Logistic matrix factorization for implicit feedback data," in *NIPS Workshop on Distributed Machine Learning and Matrix Computations*, 2014.
- [7] G. Dror, N. Koenigstein, Y. Koren, and M. Weimer, "The Yahoo! music dataset and KDD-cup '11," *JMLR: Workshop and Conference Proceedings*, vol. 18, pp. 3–18, 2012.
- [8] Y. Koren, R. Bell, and C. Volinsky, "Matrix factorization techniques for recommender systems," *Computer*, vol. 42, no. 8, pp. 30–37, 2009.
- [9] S. Funk, "Netflix update: Try this at home," <http://sifter.org/~simon/journal/20061211.html>, 2006.
- [10] Y. Hu, Y. Koren, and C. Volinsky, "Collaborative filtering for implicit feedback datasets," in *Proceedings of the 2008 Eighth IEEE International Conference on Data Mining*, 2008, pp. 263–272.
- [11] J. Nocedal and S. J. Wright, *Numerical Optimization*, 2nd ed. New York: Springer, 2006.
- [12] H. De Sterck and M. Winlaw, "A nonlinearly preconditioned conjugate gradient algorithm for rank- R canonical tensor approximation," *Numer. Linear Algebra Appl.*, vol. 22, pp. 410–432, 2015.
- [13] P. Brune, M. G. Knepley, B. F. Smith, and X. Tu, "Composing scalable nonlinear algebraic solvers," *SIAM Review*, forthcoming.
- [14] H. De Sterck, "A nonlinear GMRES optimization algorithm for canonical tensor decomposition," *SIAM J. Sci. Comput.*, vol. 34, pp. A1351–A1379, 2012.
- [15] H. Fang and Y. Saad, "Two classes of multisection methods for nonlinear acceleration," *Numer. Linear Algebra Appl.*, vol. 16, pp. 197–221, 2009.
- [16] H. Walker and P. Ni, "Anderson acceleration for fixed-point iterations," *SIAM J. Numer. Anal.*, vol. 49, pp. 1715–1735, 2011.
- [17] D. Anderson, "Iterative procedures for nonlinear integral equations," *J. ACM*, vol. 12, pp. 547–560, 1965.
- [18] P. Concus, G. H. Golub, and D. P. O'Leary, "Numerical solution of nonlinear elliptical partial differential equations by a generalized conjugate gradient method," *Computing*, vol. 19, pp. 321–339, 1977.
- [19] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis, "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, 2011, pp. 69–77.
- [20] C. Teflioudi, F. Makari, and R. Gemulla, "Distributed matrix completion," in *12th International Conference on Data Mining (ICDM)*. IEEE, 2012, pp. 655–664.
- [21] H.-F. Yu, C.-J. Hsieh, I. Dhillon *et al.*, "Scalable coordinate descent approaches to parallel matrix factorization for recommender systems," in *12th International Conference on Data Mining (ICDM)*. IEEE, 2012, pp. 765–774.
- [22] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, 2012, pp. 15–28.
- [23] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan, "Large-scale parallel collaborative filtering for the Netflix prize," in *Proceedings of the 4th International Conference on Algorithmic Aspects in Information and Management*, 2008, pp. 337–348.
- [24] C. Johnson, "Music recommendations at scale with Spark," 2014, Spark Summit. [Online]. Available: <https://spark-summit.org/2014/talk/music-recommendations-at-scale-with-spark>
- [25] "MovieLens 20M dataset," March 2015, GroupLens. [Online]. Available: <http://grouplens.org/datasets/movielens/20m/>
- [26] A. N. Tikhonov and V. Y. Arsenin, *Solutions of ill-posed problems*. New York: John Wiley, 1977.
- [27] E. Polak and G. Ribière, "Note sur la convergence de méthodes de directions conjuguées," *Revue Française d'Informatique et de Recherche Opérationnelle*, vol. 16, pp. 35–43, 1969.
- [28] D. M. Dunlavy, T. G. Kolda, and E. Acar, "Poblano v1.0: A MATLAB toolbox for gradient-based optimization," Sandia National Laboratories, Albuquerque, NM and Livermore, CA, Tech. Rep. SAND2010-1422, March 2010.
- [29] M. G. Kendall, "A new measure of rank correlation," *Biometrika*, vol. 30, no. 1/2, pp. 81–93, 1938.
- [30] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling," in *Proceedings of the 5th European Conference on Computer systems*, 2010, pp. 265–278.
- [31] L. S. Blackford, A. Petit, R. Pozo, K. Remington, R. C. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry *et al.*, "An updated set of basic linear algebra subprograms (BLAS)," *ACM Trans. Math. Software*, vol. 28, no. 2, pp. 135–151, 2002.
- [32] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammerling, A. McKenney *et al.*, *LAPACK Users' Guide*. SIAM, 1999, vol. 9.
- [33] A. Davidson and A. Or, "Optimizing shuffle performance in Spark," *University of California, Berkeley-Department of Electrical Engineering and Computer Sciences, Tech. Rep.*, 2013.
- [34] J. Dai, "Experience and lessons learned for large-scale graph analysis using GraphX," 2015, Spark Summit East. [Online]. Available: <https://spark-summit.org/east-2015/talk/experience-and-lessons-learned-for-large-scale-graph-analysis-using-graphx>