

Parallel Algorithms for Tensor Completion in the CP Format

Lars Karlsson

Department of Computing Science, Umeå University, 901 87 Umeå, Sweden

Daniel Kressner

MATHICSE-ANCHP SMA, EPF Lausanne, 1015 Lausanne, Switzerland

André Uschmajew

Hausdorff Center for Mathematics & Institute for Numerical Simulation, University of Bonn, 53115 Bonn, Germany

Abstract

Low-rank tensor completion addresses the task of filling in missing entries in multi-dimensional data. It has proven its versatility in numerous applications, including context-aware recommender systems and multivariate function learning. To handle large-scale datasets and applications that feature high dimensions, the development of distributed algorithms is central. In this work, we propose novel, highly scalable algorithms based on a combination of the canonical polyadic (CP) tensor format with block coordinate descent methods. Although similar algorithms have been proposed for the matrix case, the case of higher dimensions gives rise to a number of new challenges and requires a different paradigm for data distribution. The convergence of our algorithms is analyzed and numerical experiments illustrate their performance on distributed-memory architectures for tensors from a range of different applications.

Keywords: low-rank tensor completion, canonical tensor format, parallel tensor completion, parallel cyclic coordinate descent, parallel alternating least squares

1. Introduction

Tensor completion is the task of completing multi-dimensional data with missing values. To make this task well-posed, rank constraints are typically imposed on the tensor. Unlike in the matrix case, the definition of the rank of a tensor is not unique. In this work, we focus on the CP format, in which the tensor is represented as a sum of rank-one tensors.

Email addresses: `larsk@cs.umu.se` (Lars Karlsson), `daniel.kressner@epfl.ch` (Daniel Kressner), `uschmajew@ins.uni-bonn.de` (André Uschmajew)

More specifically, following the notation in [1], the CP format of a tensor $\mathbf{X} \in \mathbb{R}^{I_1 \times I_2 \times \dots \times I_N}$ of order N takes the form

$$\mathbf{X} = \sum_{\ell=1}^L \mathbf{x}_\ell^{(1)} \circ \mathbf{x}_\ell^{(2)} \circ \dots \circ \mathbf{x}_\ell^{(N)}, \quad \mathbf{x}_\ell^{(n)} \in \mathbb{R}^{I_n}, \quad (1)$$

where \circ denotes the vector outer product. The smallest L , for which (1) is possible, is called the tensor rank of \mathbf{X} .

Let \mathcal{A} be a data tensor with known entries in an index set Ω , that is, the entry $a_{i_1 \dots i_N}$ of \mathcal{A} is known if and only if $\mathbf{i} = (i_1, \dots, i_N) \in \Omega$. Then we seek to fit the components $\mathbf{x}_\ell^{(n)}$ of (1) for a tensor \mathbf{X} of tensor rank L to this given data. Letting P_Ω denote the orthogonal projection onto Ω , this can be addressed by solving the optimization problem

$$\min \left\| P_\Omega \left(\mathcal{A} - \sum_{\ell=1}^L \mathbf{x}_\ell^{(1)} \circ \mathbf{x}_\ell^{(2)} \circ \dots \circ \mathbf{x}_\ell^{(N)} \right) \right\|_2^2 + \lambda \sum_{\ell=1}^L \sum_{n=1}^N \|\mathbf{x}_\ell^{(n)}\|_2^2, \quad (2)$$

where $\lambda \geq 0$ is a regularization parameter. The aim of this work is to propose and analyze parallel methods for solving (2). Applications that require the completion of tensors (that is, $N > 2$) include context-aware recommender systems [2], multivariate function learning [3], reconstruction of hyperspectral images [4], and seismic inversion [5].

For $N = 2$ the problem (2) is an instance of the well-known matrix completion problem. Among the many algorithms proposed in the literature, parallel approaches have been proposed based on ALS (alternating least-squares) [6, 7], CCD (cyclic coordinate descent) [8, 9, 10], and SGD (stochastic gradient descent) [6, 11, 12].

For $N > 2$, the problem of tensor factorization into CP format is a well-studied subject for complete data, see [1] and references therein. In contrast, the case of incomplete data (that is, tensor completion in the CP format) has received much less attention. Most existing algorithms [2, 13, 14] use a combination of standard techniques (ALS, nonlinear CG, ...) with weight matrices or imputation in order to deal with the missing data. In [15], an alternating proximal gradient method was applied to (2), but with nonnegativity constraints. To the best of our knowledge, no parallel algorithm has been proposed for (2). For the case of complete data, Phan and Cichoki [16] have studied the parallelization of ALS based on local CP models.

Block coordinate descent (BCD) techniques like ALS are efficient for tensor optimization problems because they exploit the multi-linearity of the outer product. The CP format (1) is linear with respect to each of the factor matrices

$$\mathbf{X}^{(n)} = [\mathbf{x}_1^{(n)}, \mathbf{x}_2^{(n)}, \dots, \mathbf{x}_L^{(n)}] \in \mathbb{R}^{I_n \times L}, \quad (3)$$

which collect all vectors $\mathbf{x}_\ell^{(n)}$ corresponding to a fixed mode $n = 1, \dots, N$. Each step of ALS updates one of the factor matrices, which requires the solution of I_n normal equations of size $L \times L$. To mitigate the cost incurred by these normal equations for larger L and to improve scalability, Yu et al. [9, 10] recently proposed a simpler block coordinate method

for the matrix completion problem ($N = 2$), which they termed CCD++. In this paper, we generalize CCD++ to the tensor case. In contrast to ALS, this method considers a “feature-wise” [9, 10] partition into the block variables $\mathbf{Y}_\ell = [\mathbf{x}_\ell^{(1)}, \mathbf{x}_\ell^{(2)}, \dots, \mathbf{x}_\ell^{(N)}]$ corresponding to the L terms in the sum (1). Each step of CCD++ updates one of these block variables, which amounts to a rank-one approximation and is addressed by performing one or more iterations of ALS. The entries of a single vector $\mathbf{x}_\ell^{(n)}$ can be updated in parallel by an explicit formula without matrix inversion, resulting in a cyclic coordinate descent method operating on scalar variables.

In this work, we develop and analyze novel parallel algorithms for tensor completion, based on ALS and CCD++. As explained in more detail in Section 3, the transition from the matrix case [9, 10] to the tensor cases poses a number of new challenges for load balancing and data distribution.

2. Description of algorithms

In this section, we give a detailed description of ALS and the CCD++ method for solving (2).

2.1. ALS

One step of an ALS iteration consists of optimizing a single factor matrix (3), while keeping the others fixed. The particular form of the objective function (2) implies that this optimization task decouples with respect to the rows of the factor matrix. To derive the corresponding equations, we first recall the CP format (1) in its entry-wise form:

$$x_{i_1 \dots i_N} = \sum_{\ell=1}^L \prod_{n=1}^N [\mathbf{x}_\ell^{(n)}]_{i_n}.$$

Considering a fixed mode $\hat{n} = 1, \dots, n$, the rows of the factor matrix $\mathbf{X}^{(\hat{n})}$ that minimizes (2), while keeping all other factor matrices fixed, are obtained from the solution of the $I_{\hat{n}}$ optimization problems

$$\min_{z_1, z_2, \dots, z_L} \sum_{\substack{\mathbf{i} \in \Omega \\ i_{\hat{n}} = \hat{i}}} \left(a_{\mathbf{i}} - \sum_{\ell=1}^L z_\ell \prod_{\substack{n=1 \\ n \neq \hat{n}}}^N [\mathbf{x}_\ell^{(n)}]_{i_n} \right)^2 + \lambda \sum_{\ell=1}^L z_\ell^2, \quad \hat{i} = 1, \dots, I_{\hat{n}}. \quad (4)$$

Let $|\Omega_{\hat{n}, \hat{i}}|$ be the number of known entries satisfying $i_{\hat{n}} = \hat{i}$. Stacking these entries in a vector $\mathbf{a}_{\hat{n}, \hat{i}}$, and arranging the partial products $\prod_{n \neq \hat{n}} [\mathbf{x}_\ell^{(n)}]_{i_n}$ into an $|\Omega_{\hat{n}, \hat{i}}| \times L$ matrix $\mathbf{H}_{\hat{n}, \hat{i}}$ accordingly, (4) becomes

$$\min_{\mathbf{z}} \|\mathbf{a}_{\hat{n}, \hat{i}} - \mathbf{H}_{\hat{n}, \hat{i}} \mathbf{z}\|_2^2 + \lambda \|\mathbf{z}\|_2^2.$$

When $\lambda > 0$, the solution to this least-squares problem is unique, and given by

$$\mathbf{z}^* = (\mathbf{H}_{\hat{n}, \hat{i}}^\top \mathbf{H}_{\hat{n}, \hat{i}} + \lambda I)^{-1} \mathbf{H}_{\hat{n}, \hat{i}}^\top \mathbf{a}_{\hat{n}, \hat{i}}. \quad (5)$$

Thus, the coordinate update rule for the \hat{i} th row of the \hat{n} th factor matrix consists of replacing the entries $[\mathbf{x}_\ell^{(\hat{n})}]_{\hat{i}}$ for $\ell = 1, 2, \dots, L$ with the entries of \mathbf{z}^* given by (5). As the update does not depend on other entries than those in slice $i_{\hat{n}} = \hat{i}$, all slices can be updated in parallel.

The overall ALS procedure is summarized as Algorithm 1. The choice of starting vectors $\mathbf{x}_\ell^{(n)}$ is random in our experiments. By (4), they should certainly not be initialized with all zeros.

Algorithm 1: ALS

```

1 Initialize all vectors  $\mathbf{x}_\ell^{(n)}$ ;
2 for each outer iteration do
3   for  $\hat{n} = 1, 2, \dots, N$  do
4     for  $\hat{i} = 1, 2, \dots, I_{\hat{n}}$  do in parallel
5       Update  $L$  entries  $[\mathbf{x}_\ell^{(\hat{n})}]_{\hat{i}}$ ,  $\ell = 1, 2, \dots, L$ , using (5);

```

2.2. CCD++

One step of CCD++ consists of optimizing a single term in the sum (1), while keeping the others fixed. To update the $\hat{\ell}$ th term $\mathbf{x}_{\hat{\ell}}^{(1)} \circ \mathbf{x}_{\hat{\ell}}^{(2)} \circ \dots \circ \mathbf{x}_{\hat{\ell}}^{(N)}$, one has to find a best rank-one approximation of

$$\mathcal{A} - \sum_{\ell \neq \hat{\ell}} \mathbf{x}_\ell^{(1)} \circ \mathbf{x}_\ell^{(2)} \circ \dots \circ \mathbf{x}_\ell^{(N)}.$$

This is done approximately, by performing a few iterations of rank-one ALS.

The \hat{n} th step of rank-one ALS consists of minimizing (2) with respect to the single vector $\mathbf{x}_{\hat{\ell}}^{(\hat{n})}$. According to (5), we replace the \hat{i} th entry of $\mathbf{x}_{\hat{\ell}}^{(\hat{n})}$ by

$$z^* = \frac{\sum_{\substack{\mathbf{i} \in \Omega \\ i_{\hat{n}} = \hat{i}}} \left(a_{\mathbf{i}} - \sum_{\substack{\ell=1 \\ \ell \neq \hat{\ell}}}^L \prod_{n=1}^N [\mathbf{x}_\ell^{(n)}]_{i_n} \right) \prod_{\substack{n=1 \\ n \neq \hat{n}}}^N [\mathbf{x}_{\hat{\ell}}^{(n)}]_{i_n}}{\lambda + \sum_{\substack{\mathbf{i} \in \Omega \\ i_{\hat{n}} = \hat{i}}} \left(\prod_{\substack{n=1 \\ n \neq \hat{n}}}^N [\mathbf{x}_{\hat{\ell}}^{(n)}]_{i_n} \right)^2}. \quad (6)$$

This can be done for all $\hat{i} = 1, 2, \dots, I_{\hat{n}}$ in parallel.

In principle, the block coordinate update rule (6) for $\mathbf{x}_{\hat{\ell}}^{(\hat{n})}$ can be applied in any order, but the choice of this order will affect convergence. Following the matrix case [9, 10], CCD++ optimizes all entries for a given rank-one term ($\hat{\ell}$), by sweeping through $\hat{n} = 1, 2, \dots, N$, possibly multiple times, before moving on to the next rank-one term. This leads to the nested loop structure shown in Algorithm 2.

Algorithm 2: CCD++

```

1 Initialize all vectors  $\mathbf{x}_\ell^{(n)}$  and  $\mathcal{R}$  for  $\hat{\ell} = 1$  using (7);
2 for each outer iteration do
3   for  $\hat{\ell} = 1, 2, \dots, L$  do
4     for each inner iteration do
5       for  $\hat{n} = 1, 2, \dots, N$  do
6         for  $\hat{i} = 1, 2, \dots, I_{\hat{n}}$  do in parallel
7           Update entry  $[\mathbf{x}_{\hat{\ell}}^{(\hat{n})}]_{\hat{i}}$  using (6);
8         Update  $\mathcal{R}$  in parallel using (8) with  $\hat{\ell}_{\text{old}} = \hat{\ell}$  and  $\hat{\ell}_{\text{new}} = 1 + (\hat{\ell} \bmod L)$ ;

```

Efficient evaluation of (6). The quantities

$$r_{\mathbf{i}} := a_{\mathbf{i}} - \sum_{\substack{\ell=1 \\ \ell \neq \hat{\ell}}}^L \prod_{n=1}^N [\mathbf{x}_\ell^{(n)}]_{i_n}, \quad \mathbf{i} \in \Omega, \quad (7)$$

appearing in the numerator of (6) do not depend on \hat{n} . Therefore, they need to be evaluated only once when sweeping through $\hat{n} = 1, 2, \dots, N$. This trick effectively reduces the number of evaluations by a factor of $1/N$, at the expense of storing all these quantities in a sparse tensor \mathcal{R} with the same sparsity pattern as \mathcal{A} .

When moving from one rank-one term to the next, the value of $\hat{\ell}$ changes from $\hat{\ell}_{\text{old}}$ to $\hat{\ell}_{\text{new}}$. Consequently, the sum that appears in the definition (7) of $r_{\mathbf{i}}$ loses one term (corresponding to $\hat{\ell}_{\text{new}}$) and gains one term (corresponding to $\hat{\ell}_{\text{old}}$). To compensate for these changes, we perform the update

$$r_{\mathbf{i}} \leftarrow r_{\mathbf{i}} - \prod_{n=1}^N [\mathbf{x}_{\hat{\ell}_{\text{old}}}^{(n)}]_{i_n} + \prod_{n=1}^N [\mathbf{x}_{\hat{\ell}_{\text{new}}}^{(n)}]_{i_n}. \quad (8)$$

This natural rule was also used in [9, 10] for matrices ($N = 2$). Especially for large ranks L , it is much more efficient than recomputing \mathcal{R} from scratch whenever $\hat{\ell}$ changes. Note that the update rule may be prone to cancellation, potentially limiting the attainable accuracy. We have not observed such an effect for the examples described in Section 4, but it may become a concern in applications with very high accuracy requirements.

Inner iterations. Using more than one inner iteration in Algorithm 2 can sometimes be beneficial; it potentially allows to make progress without having to perform the update (8) of \mathcal{R} . For the matrix case, Yu et al. [9, 10] proposed an adaptive scheme that selects the number of inner iterations based on the observed convergence behavior, leading to a considerable improvement of the performance. However, to simplify the interpretation of the results, we use only one inner iteration in our numerical experiments. As demonstrated in Section 4.4, choosing two inner iterations in CCD++ does not necessarily lead to improved performance in the tensor case.

2.3. Convergence

In the following, we provide a convergence analysis of Algorithm 2 for the case of one inner iteration. Our analysis basically consists of validating the assumptions of a recent convergence theorem [15] for cyclic BCD methods applied to multi-convex functions.

We treat the parameters $\mathbf{x}_\ell^{(n)}$ of our model (1) as disjoint blocks of a single array \mathbf{X} arranged in CCD++ order, that is,

$$\mathbf{X} = [\mathbf{x}_1^{(1)}, \dots, \mathbf{x}_1^{(N)}, \mathbf{x}_2^{(1)}, \dots, \mathbf{x}_2^{(N)}, \dots, \mathbf{x}_L^{(1)}, \dots, \mathbf{x}_L^{(N)}].$$

Moreover, we split the objective function (2) into the model error

$$E(\mathbf{X}) = \left\| P_\Omega \left(\mathcal{A} - \sum_{\ell=1}^L \mathbf{x}_\ell^{(1)} \circ \mathbf{x}_\ell^{(2)} \circ \dots \circ \mathbf{x}_\ell^{(N)} \right) \right\|_2^2 \quad (9)$$

and the squared Frobenius norm $\|\mathbf{X}\|_F^2 = \sum_{\ell=1}^L \sum_{n=1}^N \|\mathbf{x}_\ell^{(n)}\|_2^2$ of the matrix containing the representation parameters. The CCD++ algorithm is simply the cyclic BCD method with exact line search applied to the objective function

$$f_\lambda(\mathbf{X}) = E(\mathbf{X}) + \lambda \|\mathbf{X}\|_F^2. \quad (10)$$

The case $\lambda = 0$ (no regularization) leads to a number of difficulties in the analysis, due to the lack of closedness [17] and the scaling indeterminacy [18] of the CP format. To avoid these difficulties, we assume that regularization is present ($\lambda > 0$). Then the sublevel sets of f_λ are bounded so that (2) has at least one globally optimal solution. Although we cannot ensure that the algorithm will find such a global solution, the multi-convexity of the objective function allows us to prove convergence to a critical point.

Proposition 1. *Assume $\lambda > 0$ and let \mathbf{X}_k denote the parameter array produced by k outer iterations of Algorithm 2 (using one inner iteration). Then the sequence (\mathbf{X}_k) converges to a critical point \mathbf{X}_* of f_λ and $\|\mathbf{X}_k - \mathbf{X}_*\|_F \leq Ck^{-\alpha}$ for constants $C > 0$ and $\alpha \in (1, \infty)$. If, additionally, the Hessian $\nabla^2 f_\lambda(\mathbf{X}_*)$ is positive definite, then \mathbf{X}_* is a local minimum, and the convergence is asymptotically Q -linear: $\|\mathbf{X}_k - \mathbf{X}_*\|_F \leq q\|\mathbf{X}_{k-1} - \mathbf{X}_*\|_F$ for $q \in (0, 1)$ and sufficiently large k . The same statements hold for Algorithm 1.*

Proof. As noted above, Algorithm 2 is cyclic BCD applied to the real-analytic function f_λ . Consequently, the first part follows from Theorem 2.8 in [15], provided that the optimization problem with respect to any of the block variables $\mathbf{x}_\ell^{(n)}$ is strongly convex with a lower bound on the Hessian independent of the current iterate. To see that this is indeed the case, let $\varphi(\mathbf{x}_\ell^{(n)})$ denote the restriction of f_λ to $\mathbf{x}_\ell^{(n)}$ with all other blocks fixed. As the restriction of E in (9) to $\mathbf{x}_\ell^{(n)}$ is a non-negative quadratic function, it follows directly from (10) that the smallest eigenvalue of $\nabla^2 \varphi$ is bounded from below by λ at any point. The convergence rate is as stated in Theorem 2.9 in [15]. Although this theorem also covers the second part (that is, the case of a positive definite Hessian), it is worth noting that this part follows from textbook results [19, Section 10.3]. The arguments for Algorithm 1 are the same, as ALS is cyclic BCD with respect to the block variables (3). \square

The positive definiteness of $\nabla^2 f_\lambda(\mathbf{X}_*)$ can be enforced by a sufficiently strong regularization.

Proposition 2. *Fix $R > 0$. There exists $\lambda_0 > 0$ such that for all $\lambda \geq \lambda_0$ and all starting points \mathbf{X}_0 with $\|\mathbf{X}_0\|_F \leq R$ the following holds: The Hessian at all points in the sublevel set $\mathcal{L}_\lambda(\mathbf{X}_0) = \{\mathbf{X} \mid f_\lambda(\mathbf{X}) \leq f_\lambda(\mathbf{X}_0)\}$ has eigenvalues bounded from below by λ , which ensures linear convergence according to Theorem 1.*

Proof. Let $\mathbf{X} \in \mathcal{L}_\lambda(\mathbf{X}_0)$. By (10), the fact that E is never negative implies $\|\mathbf{X}\|_F^2 \leq E(\mathbf{X}_0)/\lambda + R \leq E^*/\lambda + R$, where E^* is the maximum of E on the zero-centered ball of radius R . Let σ_λ denote the minimum of the smallest eigenvalues of ∇E^2 on a zero-centered ball of radius $(E^*/\lambda + R)^{1/2}$. Then the smallest eigenvalue of $\nabla^2 f_\lambda(\mathbf{X})$ is bounded from below by $\sigma_\lambda + 2\lambda$. This implies the assertion, considering that σ_λ does not decrease when λ increases. \square

The convexification of the problem on a sublevel set suggested by Proposition 2 is likely to change its qualitative properties. For completeness, we note that the global convergence of cyclic BCD for strongly convex problems at an asymptotically linear rate has been obtained in earlier works like [20].

The *qualitative* convergence result of Proposition 1 is identical for CCD++ (with one inner iteration) and ALS. In fact, it is true for any cyclic BCD method applied to (10). The attained convergence rate q may of course differ, but it remains unspecified as a consequence of the non-constructive technique from [15], upon which our proof is based. Intuitively, one expects that q becomes smaller when larger blocks are used. Therefore, we expect ALS to make larger progress than CCD++ per outer iteration, as can be verified in many of the experiments in Section 4. However, updating larger blocks comes at a higher price and therefore CCD++ still can be equally fast or even faster to reach a certain accuracy.

3. Parallel algorithms

In this section, we present parallel formulations of ALS (Algorithm 1) and CCD++ (Algorithm 2) in the message-passing paradigm.

The parallel tensor completion algorithms proposed below fundamentally differ from previously published algorithms for the matrix case. In particular, the parallel ALS algorithms presented in [6, 7] and the parallel CCD++ algorithm presented in [9, 10] have in common that they distribute the *variables* and let each processor independently update its own subset of the variables. However, this comes at the cost of duplicating the storage for the incomplete matrix and, in the case of CCD++, also duplicating a significant portion of the work. Generalizing the approach to order N tensors would require N times the storage (and N times the work in the case of CCD++). In contrast, our algorithms are based on the idea of distributing the *incomplete tensor* instead of the variables and let the processors *collaboratively* update the variables. This approach does not suffer from the described problem and requires only a single copy of the incomplete tensor. Another benefit is an increase in the *degree of concurrency* (and hence improved scalability) from $\max_n \{I_n\}$ to the typically much larger $|\Omega|$.

The primary objects are the data tensor \mathcal{A} , the CP tensor \mathcal{X} , and in the case of CCD++ also the intermediate tensor \mathcal{R} . In our parallel algorithms, the data tensor \mathcal{A} can have any arbitrary distribution. Let $\Omega = \Omega^{(1)} \cup \dots \cup \Omega^{(p)}$ denote a partition of the index set Ω over p processors. Specifically, processor q owns $a_{\mathbf{i}}$ (and also $r_{\mathbf{i}}$ in the case of CCD++) for all $\mathbf{i} \in \Omega^{(q)}$. The factor matrices $\mathbf{X}^{(n)}$ for $n = 1, \dots, N$ of the CP tensor \mathcal{X} are in both cases replicated on all p processors.

3.1. Parallel formulation of ALS

Algorithm 3: Parallel formulation of ALS.

- 1 Initialize all vectors $\mathbf{x}_\ell^{(n)}$ in parallel;
 - 2 **for** each outer iteration **do**
 - 3 **for** $\hat{n} = 1, 2, \dots, N$ **do**
 - 4 Construct local contributions to each of the $I_{\hat{n}}$ normal equations (5) in the form of a local coefficient matrix $\mathbf{C}_{\hat{i}}$ of size $L \times L$ and a local right-hand side vector $\mathbf{d}_{\hat{i}}$ of length L for each $\hat{i} = 1, 2, \dots, I_{\hat{n}}$;
 - 5 Reduce the local contributions to the $I_{\hat{n}}$ normal equations defined by the pairs $(\mathbf{C}_{\hat{i}}, \mathbf{d}_{\hat{i}})$ by summation;
 - 6 Scatter the $I_{\hat{n}}$ normal equations across all processors;
 - 7 Solve the (roughly $I_{\hat{n}}/p$) local normal equations $(\mathbf{C}_{\hat{i}} + \lambda\mathbf{I})\mathbf{z}_{\hat{i}} = \mathbf{d}_{\hat{i}}$ for $\mathbf{z}_{\hat{i}}$;
 - 8 Replicate the updated factor matrix $\mathbf{X}^{(\hat{n})}$;
-

To understand the parallel formulation of ALS described in Algorithm 3, consider how to update row \hat{i} of $\mathbf{X}^{(\hat{n})}$. To apply the updating rule (5), one needs to compute the $L \times L$ matrix product $\mathbf{C}_{\hat{i}} := \mathbf{H}_{\hat{n}, \hat{i}}^\top \mathbf{H}_{\hat{n}, \hat{i}}$ and the matrix–vector product $\mathbf{d}_{\hat{i}} := \mathbf{H}_{\hat{n}, \hat{i}}^\top \mathbf{a}_{\hat{n}, \hat{i}}$. These are the most time-consuming tasks unless the rank L is very large. Then one can add the regularization term to the coefficient matrix and solve the normal equations to obtain the updated row. Each row of $\mathbf{H}_{\hat{n}, \hat{i}}$ and the corresponding element of $\mathbf{a}_{\hat{n}, \hat{i}}$ are associated with an index $\mathbf{i} \in \Omega_{\hat{n}, \hat{i}}$. The elements of a row of $\mathbf{H}_{\hat{n}, \hat{i}}$ are computed from the CP factor matrices given the row index. The matrix $\mathbf{C}_{\hat{i}}$ and the vector $\mathbf{d}_{\hat{i}}$ can be computed as a sum of vector outer products and vectors, respectively, as in

$$\mathbf{C}_{\hat{i}} = \mathbf{c}_1 \mathbf{c}_1^\top + \mathbf{c}_2 \mathbf{c}_2^\top + \dots + \mathbf{c}_m \mathbf{c}_m^\top \quad \text{and} \quad \mathbf{d}_{\hat{i}} = \mathbf{d}_1 + \mathbf{d}_2 + \dots + \mathbf{d}_m.$$

Here, \mathbf{c}_i is one row of $\mathbf{H}_{\hat{n}, \hat{i}}$ and \mathbf{d}_i is that same row scaled by the corresponding tensor element. Each pair $(\mathbf{c}_i, \mathbf{d}_i)$ is associated with a tensor index $\mathbf{i} \in \Omega_{\hat{n}, \hat{i}}$ and computed from the CP factor matrices and the tensor element $a_{\mathbf{i}}$. Since the tensor is distributed over p processors, each processor holds only a subset of the tensor indices and elements and can thus compute only the corresponding subset of the pairs $(\mathbf{c}_i, \mathbf{d}_i)$. Our parallel formulation lets each processor compute its own local contribution to the global matrix $\mathbf{C}_{\hat{i}}$ and the global vector $\mathbf{d}_{\hat{i}}$ by summing together its locally available terms. The p local contributions are then

reduced by inter-processor communication to form the global objects and ultimately solve the normal equations and update row \hat{i} of $\mathbf{X}^{(\hat{n})}$.

Knowing how to update *one* row of the current factor matrix, let us describe how to simultaneously update *all* rows. To update one row, one needs access only to a subset of the tensor indices/elements. Conversely, each index/element participates in the updating of one particular row. Our parallel formulation exploits this by constructing *all* $I_{\hat{n}}$ normal equations at the same time by scanning the tensor indices/elements once and for each one adding the resulting terms to the corresponding normal equations. This occurs on line 4 in Algorithm 3 and results in p local contributions $(\mathbf{C}_i, \mathbf{d}_i)$ for each of the $I_{\hat{n}}$ normal equations. The local contribution for each set of normal equations then needs to be summed together by an inter-process reduction operation. This communication occurs on line 5. The normal equations are then scattered across all processors on line 6 and each processor then solves its subset of the normal equations on line 7. After this step, the current factor matrix has been fully updated but each processor holds only a subset of the updated rows. Therefore, the updated rows are replicated across all processors on line 8.

3.2. Parallel formulation of CCD++

The inner loop of CCD++ (Algorithm 2) consists of applying the update rule (6) to each of the $I_{\hat{n}}$ entries of the $\hat{\ell}$ -th column $\mathbf{x}_{\hat{\ell}}^{(\hat{n})}$ of $\mathbf{X}^{(\hat{n})}$. Using the intermediate tensor \mathcal{R} defined in (7), the update rule can be expressed as

$$[\mathbf{x}_{\hat{\ell}}^{(\hat{n})}]_{\hat{i}} \leftarrow \frac{\alpha_{\hat{i}}}{\lambda + \beta_{\hat{i}}}, \quad \text{where} \quad \alpha_{\hat{i}} = \sum_{\substack{\mathbf{i} \in \Omega \\ i_{\hat{n}} = \hat{i}}} r_{\mathbf{i}} \gamma_{\mathbf{i}}, \quad \beta_{\hat{i}} = \sum_{\substack{\mathbf{i} \in \Omega \\ i_{\hat{n}} = \hat{i}}} \gamma_{\mathbf{i}}^2, \quad \gamma_{\mathbf{i}} = \prod_{\substack{n=1 \\ n \neq \hat{n}}}^N [\mathbf{x}_{\hat{\ell}}^{(n)}]_{i_n}. \quad (11)$$

Since the tensor \mathcal{R} has an arbitrary distribution but the factor matrices are replicated, the data necessary to compute the tensor contractions in (11) for a fixed \hat{i} are in general distributed over all processors and each processor can compute only a subset of the terms in the sums that define $\alpha_{\hat{i}}$ and $\beta_{\hat{i}}$.

Our parallel formulation (Algorithm 4) has many similarities with our parallel ALS formulation described in the previous section. Each processor first computes their local contributions to the tensor contractions required to update the $I_{\hat{n}}$ variables $\mathbf{x}_{\hat{\ell}}^{(\hat{n})}$ in the current step. This is accomplished while scanning the tensor indices/elements once on line 6 in Algorithm 4. The local contributions are then reduced (line 7) to obtain the global objects necessary to apply the updating rule (line 8). The updated variables are finally replicated across all processors (line 9). The main difference between ALS and CCD++ is the latter's need to initialize and update an intermediate tensor \mathcal{R} . Since the factor matrices are replicated, the initialization (line 1) can be performed in a perfectly parallel fashion using (7). The same reasoning applies to the update (line 10) using (8).

3.3. Cost analysis

Below we compare our parallel formulations of ALS and CCD++ with respect to memory usage, computation and communication costs, and the degree of concurrency.

Algorithm 4: Parallel formulation of CCD++.

```

1 Initialize in parallel all vectors  $\mathbf{x}_{\hat{\ell}}^{(n)}$  and the intermediate tensor  $\mathcal{R}$  for  $\hat{\ell} = 1$  using (7);
2 for each outer iteration do
3   for  $\hat{\ell} = 1, 2, \dots, L$  do
4     for each inner iteration do
5       for  $\hat{n} = 1, 2, \dots, N$  do
6         Construct local contributions to each of the  $2I_{\hat{n}}$  tensor contractions  $\alpha_{\hat{i}}$ 
7         and  $\beta_{\hat{i}}$  for  $\hat{i} = 1, 2, \dots, I_{\hat{n}}$ ;
8         Reduce the local contributions to the tensor contractions from the
9         previous step by summation;
10        Update the  $I_{\hat{n}}$  variables  $\mathbf{x}_{\hat{\ell}}^{(\hat{n})}$  on the master processor;
11        Replicate the updated variables  $\mathbf{x}_{\hat{\ell}}^{(\hat{n})}$ ;
12      Update  $\mathcal{R}$  in parallel using (8) with  $\hat{\ell}_{\text{old}} = \hat{\ell}$  and  $\hat{\ell}_{\text{new}} = 1 + (\hat{\ell} \bmod L)$ ;

```

Memory usage. The memory usage is measured per processor. Both algorithms replicate the factor matrices of the tensor \mathcal{X} across all p processors, which requires $\Theta(L \sum_{n=1}^N I_n)$ bytes. The incomplete tensor \mathcal{A} (and the intermediate tensor \mathcal{R} in the case of CCD++) are distributed uniformly across all processors, which requires $\Theta(|\Omega|/p)$ bytes. The only substantial difference between the two algorithms is in their use of temporary storage with ALS requiring a factor of L^2 more memory. Specifically, ALS needs $\Theta(L^2 \max_n I_n)$ bytes to hold the local contributions, whereas CCD++ needs only $\Theta(\max_n I_n)$ bytes. For large ranks, the high memory cost of ALS can become prohibitive, but such exceptional cases can be handled for example by partitioning the largest dimension(s) of the tensor to effectively reduce the $\max_n I_n$ factor. Another remedy is to not update all rows at once but instead scan the tensor several times or first sort the tensor elements.

Computation and communication costs. The computation and communication costs are measured per *outer* iteration, i.e., a complete update of all variables. To construct the local contributions, ALS has a computation cost of $\Theta(L^2 N |\Omega|/p)$ while CCD++ has a cost of $\Theta(LN |\Omega|/p)$. In other words, ALS has an L times higher computation cost in this step. To reduce the local contributions, ALS first reduces the local contributions for a communication cost of $\mathcal{O}(N(t_s + t_w L^2 \max_n I_n) \log p)$, where t_s is the latency and t_w the bandwidth cost, and then scatters the normal equations for a cost of $\mathcal{O}(N(t_s \log p + t_w L^2 \max_n I_n))$. CCD++ reduces the local contributions for a communication cost of $\mathcal{O}(LN(t_s + t_w \max_n I_n) \log p)$. This means that CCD++ sends L times more messages but ALS sends L times more data. To update the variables, ALS has a computation cost of $\Theta(L^3 \sum_{n=1}^N I_n/p)$ while CCD++ has a cost of $\Theta(\sum_{n=1}^N I_n)$. Hence, the cost of CCD++ is in this step negligible while significant for ALS if the rank is large. To replicate the updated factor matrices, ALS has a communication cost of $\mathcal{O}(N(t_s \log p + t_w L \max_n I_n))$ while CCD++ has a cost of $\mathcal{O}(LN(t_s + t_w \max_n I_n) \log p)$. Again, CCD++ sends L times more messages but ALS sends L times more data. To update

the intermediate tensor, CCD++ has a computation cost of $\Theta(LN|\Omega|/p)$.

Degree of concurrency. Both algorithms are massively parallel in the sense that the most expensive steps are also perfectly parallel with a large degree of concurrency. ALS has a potential sequential bottleneck in the updating of the variables for dimensions that are smaller than p if the rank is also large since the degree of concurrency in this step is equal to the size of the current dimension. However, the computation cost is proportional to the dimension so the negative effect is thereby inherently dampened.

Summary. Both algorithms are weakly scalable and their most computationally expensive parts are massively parallel. ALS uses more temporary storage and sends L times more data and has a potential sequential bottleneck when the rank is large. CCD++ sends L times more messages. Table 1 summarizes a few key metrics that can be used to compare the two algorithms.

	Parallel ALS	Parallel CCD++
Memory usage	$\Theta(L \sum_n I_n + \Omega /p + L^2 \max_n I_n)$	$\Theta(L \sum_n I_n + \Omega /p)$
Computation cost	$\Theta(L^2 N \Omega /p + L^3 \sum_n I_n/p)$	$\Theta(LN \Omega /p + \sum_n I_n)$
Number of messages	$\Theta(N \log p)$	$\Theta(LN \log p)$
Bandwidth cost	$\Theta(L^2 N \max_n I_n \log p)$	$\Theta(LN \max_n I_n \log p)$
Degree of concurrency	$ \Omega $	$ \Omega $

Table 1: Comparison of the parallel formulations of ALS and CCD++ per outer iteration.

4. Numerical Experiments

The parallel algorithms were tested on the Abisko distributed memory system at HPC2N in Sweden. Each node contains four sockets with AMD Opteron 6238 processors and runs the Ubuntu 12.04 operating system. Each processor contains 12 cores partitioned into two NUMA domains. Hence, each node consists of 48 cores. The nodes are interconnected with a 40 Gb/s Mellanox Infiniband.

The code is written in C++ with MPI and was compiled with the PathScale C++ compiler with `-O2` optimization and native code generation. The compiled code was then linked with the OpenMPI implementation of the MPI. The parallel code is pure MPI and is not aware of and does not take advantage of the hierarchical structure of the hardware beyond what the MPI library internally does.

In all experiments presented below, the factor matrices (3) were initialized with normally distributed random numbers and one inner iteration was used in CCD++. The tensor completion accuracy was measured in a *test phase* on a randomly sampled set of entries disjoint from the set Ω used in a *training phase*, and is measured by the root mean square error (RMSE) or the normalized root mean square error (NRMSE) (RMSE divided by the difference between maximum and minimum entry) on the test set.

4.1. Synthetic data

We consider a random tensor $\mathcal{A} \in \mathbb{R}^{100 \times 100 \times 100 \times 100}$ with prescribed tensor rank $L \in \{10, 20, 30, 40, 50\}$, using exactly this rank in the algorithms. The aim was to investigate the scalability of our parallel implementations of ALS and CCD++. The index set Ω was chosen randomly such that $|\Omega| = 10^6$ (1%) and λ was set to 10^{-3} . The number of cores were set to $p \in \{1, 6, 12, \dots, 96\}$, that is, up to two full nodes were utilized. We stopped the CCD++ algorithm after 20 and ALS after 10 iterations, which leads to approximately the same execution time on a single core. The median execution time spent on one outer iteration was measured, including the evaluation of the objective function. Figures 1a and 1b show the resulting relative speedup (i.e., the ratio of the time on one core to the time on p cores) as the number of cores increases. As expected (see Section 3.1), the scalability of ALS deteriorates with growing rank L , while CCD++ scales independently of L .

4.2. Function-related tensor

To account for the growing importance of function-related tensors in scientific computing [21], we have tested our algorithms on a tensor obtained from discretizing the function $\mathbf{x} \mapsto \exp(-\|\mathbf{x}\|_2)$, which has a cusp singularity at the origin, on a tensor product grid. Functions of this type play an important role as *Slater-type orbitals* in quantum chemistry. Tensor completion can be used, for example, to learn the entire function from a few function values.

The considered domain $[-1, 1]^N$ is discretized by a regular grid with n^N grid points (an odd value n has been chosen such that the grid contains the origin). This gives a tensor \mathcal{A} with entries

$$a_{i_1, i_2, \dots, i_N} = \exp\left(-\sqrt{\xi_{i_1}^2 + \xi_{i_2}^2 + \dots + \xi_{i_N}^2}\right)$$

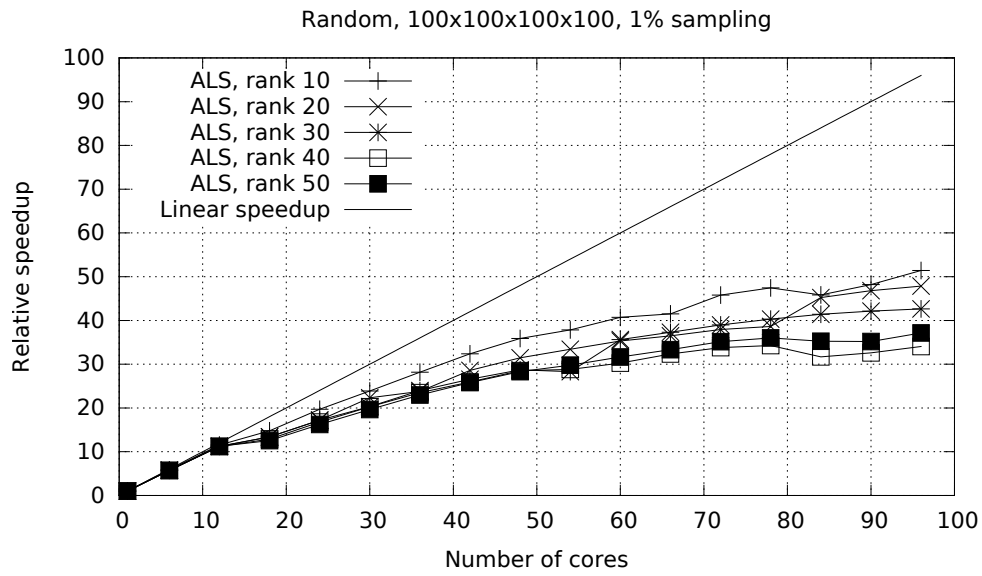
where $\xi_{i_1}, \xi_{i_2}, \dots, \xi_{i_N}$ are the grid points. We sample $10nNL$ entries of \mathcal{A} and apply our algorithms.

We used $N = 5$, $n = 51$, $L = 100$, and $10nNL = 255\,000$ samples in both methods. We have performed five trial runs with different random initial guesses. The resulting accuracies for the training phase are presented in Figures 2b and 2a. They suggest that ALS is more accurate and also less sensitive to the initial guess. For some trial runs CCD++ seems to offer a slight advantage in the initial phase.

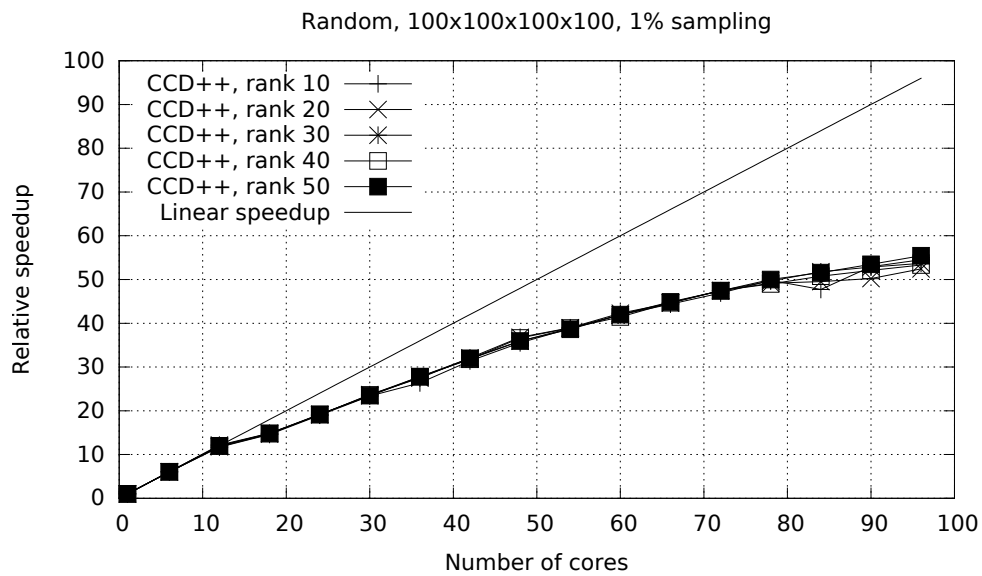
4.3. Applications

We now assess the performance of our parallel implementations of ALS and CCD++ for two data sets commonly used in the literature.

Hyperspectral image. The hyperspectral image ‘‘Ribeira’’ [22] corresponds to a tensor of size $1017 \times 1340 \times 33$. The index set Ω was chosen randomly to contain 10% of the entries (4 497 174 samples). The test set is of the same cardinality. Figures 3a and 3b report NRMSE versus execution time on one core and one full node (48 cores), respectively, with $L = 100$ and $\lambda = 10^{-3}$. In this and the next example, the timings do not include the time needed for evaluating the test scores, which were instead obtained in separate runs using the same

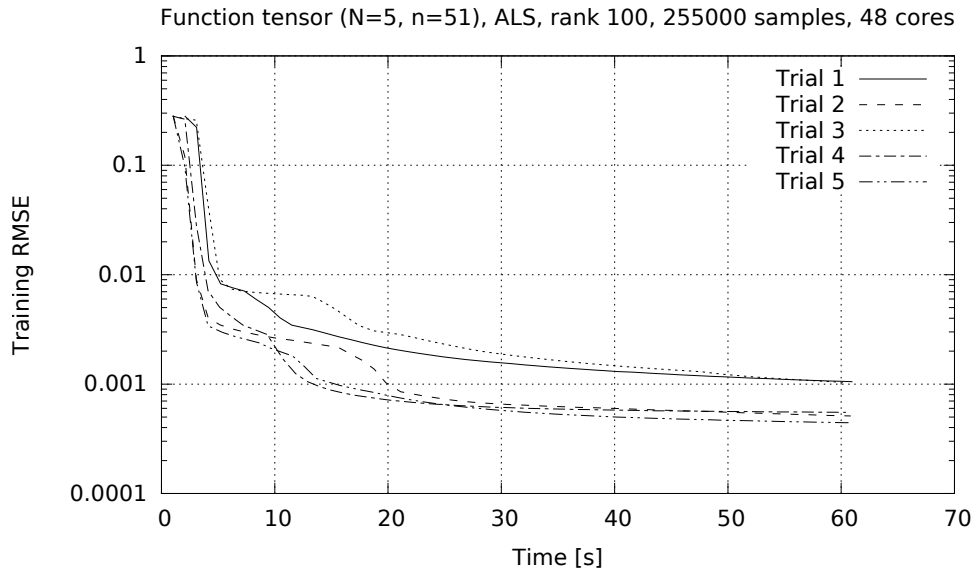


(a) Relative speed-up of ALS.

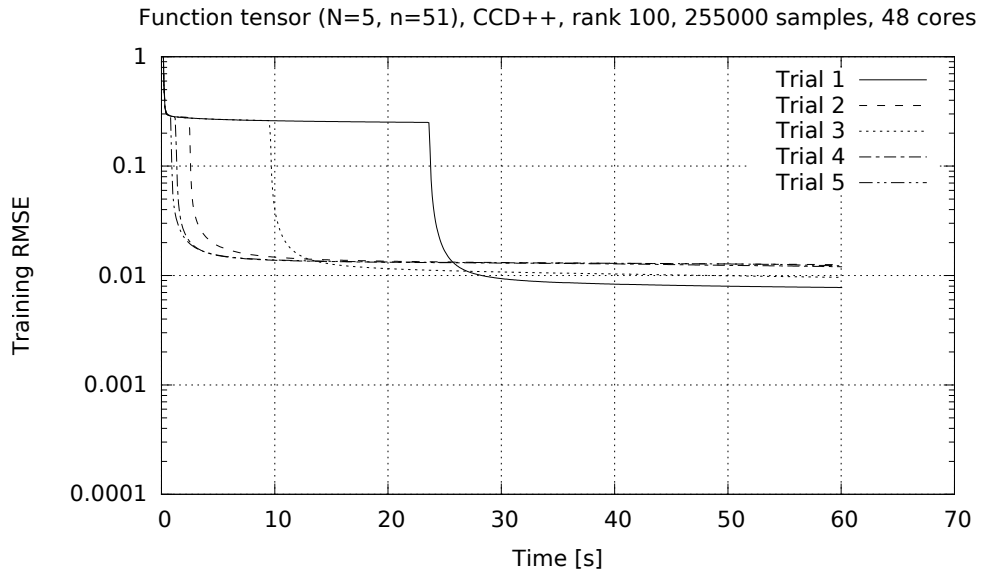


(b) Relative speed-up of CCD++.

Figure 1: Performance of parallel ALS and CCD++ for synthetic data.



(a) ALS on the function tensor.



(b) CCD++ on the function tensor.

Figure 2: Performance of parallel ALS and CCD++ for a function-related tensor.

inputs. The iterations in the figures have been non-uniformly sampled (more frequently in the beginning, and less frequently towards the end). For this data, ALS is not only faster, but it also achieves a better reconstruction on the test set.

MovieLens 10M. The MovieLens 10M dataset¹ contains timestamped ratings of movies. Binning the time into 7-day wide bins results in a tensor of size $71\,567 \times 10\,677 \times 731$. The number of samples is 10 000 054 (0.002%). For this example, the distribution of samples is highly nonuniform and it is therefore beneficial to replace $\lambda \|\mathbf{X}\|_{\mathbb{F}}^2$ by a weighted regularization term. Following [7, 9], we propose to use

$$\lambda \sum_{n=1}^N \|\Sigma^{(n)} \mathbf{X}^{(n)}\|_{\mathbb{F}}^2,$$

with the diagonal matrix $\Sigma^{(n)} = \text{diag}(\sigma_1^{(n)}, \dots, \sigma_{I_n}^{(n)})$ containing the number of known entries in the corresponding slice of the tensor: $\sigma_i^{(n)} = |\Omega_{n,i}|$ for $i = 1, \dots, I_n$. The resulting changes in ALS and CCD++ are straightforward. One should also note that corresponding convergence statements as in Section 2.3 hold for such a weighted regularization.

Figures 4a and 4b report RMSE versus execution time on one core and one full node (48 cores), respectively, with rank $L = 40$ and $\lambda = 10^{-1}$. The two methods perform similarly on one core but CCD++ clearly outperforms ALS on 48 cores due to better parallel scaling. Both methods reach a similar accuracy, but we observed that this is sensitive to the details of the regularization. While both methods reach a final test RMSE of 0.87 when using the weighted regularization suggested above, they only reach 0.99 (ALS) and 1.07 (CCD++) when using the standard regularization.

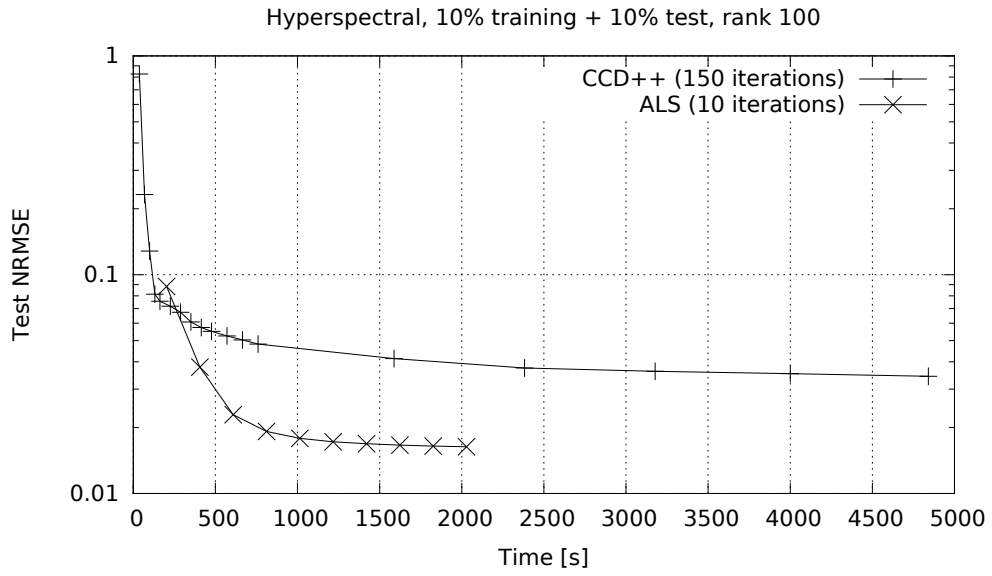
4.4. CCD++ with two inner iterations

We repeat the experiments from the previous section for the CCD++ algorithm with two instead of one inner iteration, in order to check if this results in any improvements. For both datasets, it turns out that the cost for two inner iterations is not compensated by the possibly improved rank-one approximations. The whole process slows down, as can be seen in Figures 5a and 5b.

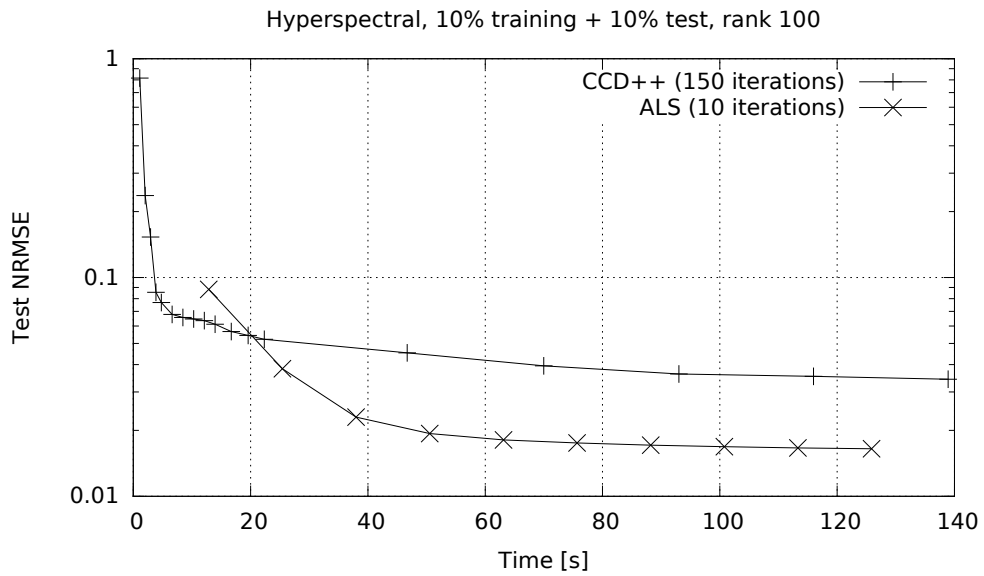
5. Summary and conclusions

We have presented and analyzed two parallel algorithms (ALS and CCD++) for tensor completion in the CP format, provided convergence results for them, as well as numerical results on synthetic and real-world data. Concerning the comparison between the two algorithms, we have to draw a mixed conclusion. In contrast to CCD++, the parallel scalability of ALS appears to be sensitive to the prescribed CP rank and, consequently, CCD++ scales better than ALS for larger ranks when applied to random data. For data

¹<http://grouplens.org/datasets/movielens/>

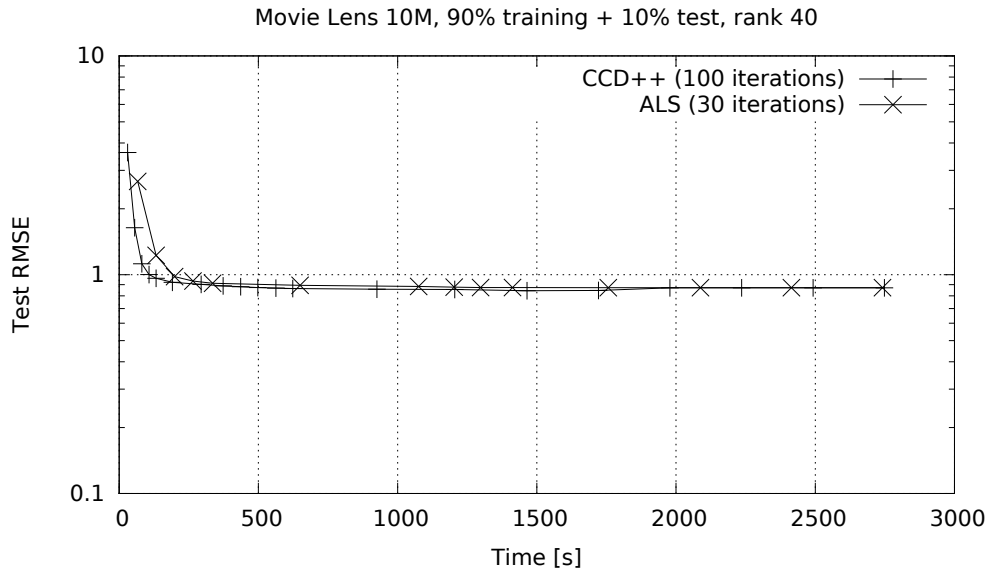


(a) Hyperspectral image on 1 core.

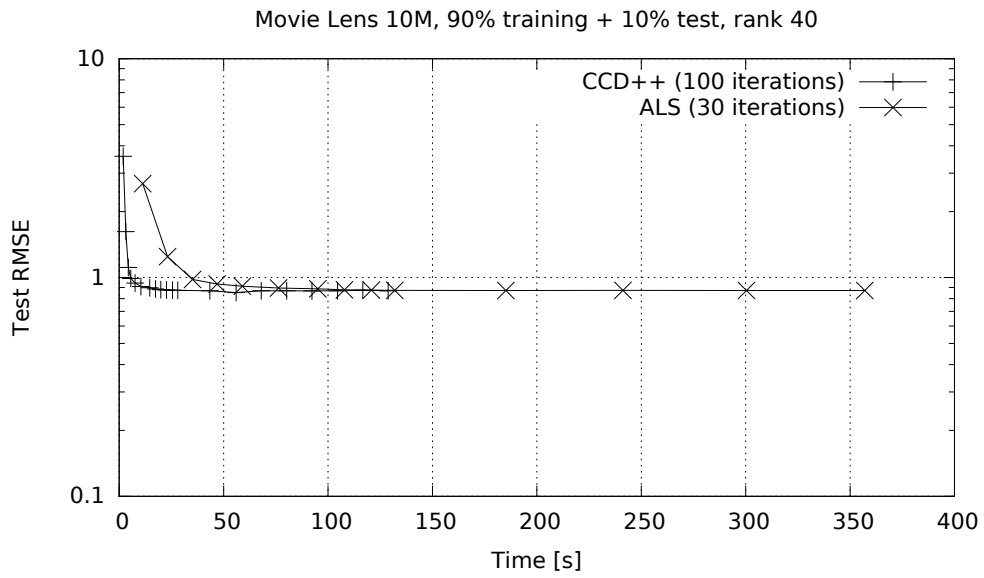


(b) Hyperspectral image on 48 cores.

Figure 3: Performance of parallel ALS and CCD++ for the hyperspectral image “Ribeira”.

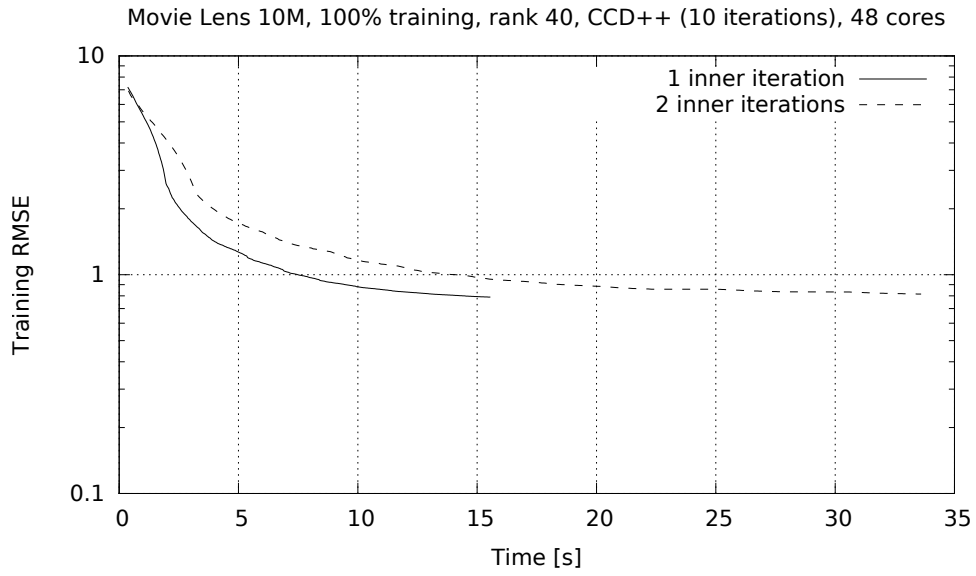


(a) Movie Lens 10M on 1 core.

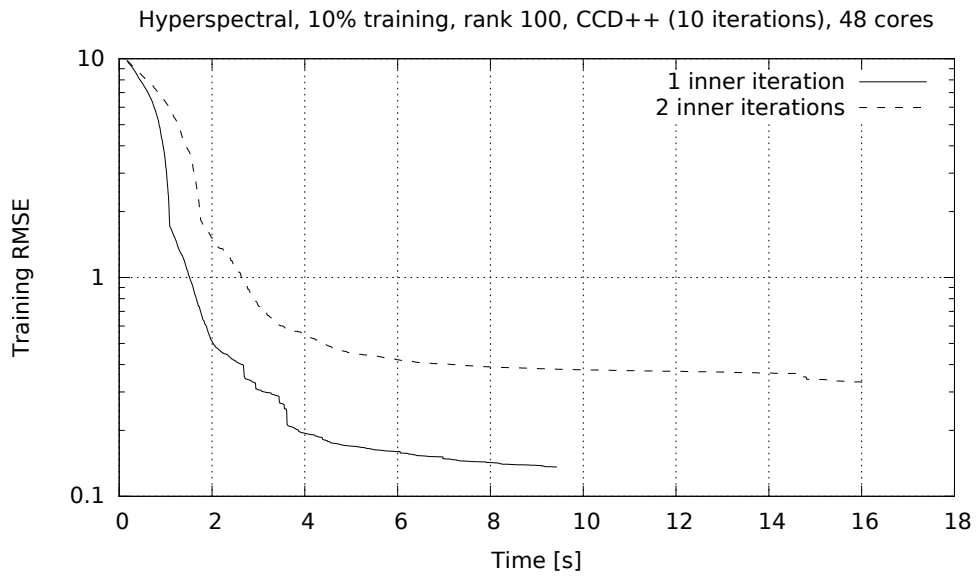


(b) Movie Lens 10M on 48 cores.

Figure 4: Performance of parallel ALS and CCD++ for the Movie Lens 10M data set.



(a) Movie Lens 10M.



(b) Hyperspectral image.

Figure 5: One vs. two inner iterations of parallel CCD++.

sets from specific applications, like hyperspectral images, we have observed on 48 cores that this disadvantage can be more than compensated by the faster convergence of ALS. For other applications, like MovieLens 10M, ALS is still slower. In terms of attained quality of approximation, ALS appears to be never worse and sometimes better than CCD++. In summary, this makes ALS one of the most powerful low-rank optimization algorithms currently at hand and CCD++ becomes a valuable alternative when scalability is critical.

Acknowledgments

We thank the anonymous reviewers for their comments which helped improve the quality of the paper.

Financial support has been provided by UMIT Research Lab via Balticgruppen and eSENCE, a strategic collaborative eScience programme funded by the Swedish Research Council. This research was conducted using the resources of High Performance Computing Center North (HPC2N).

References

- [1] T. G. Kolda, B. W. Bader, Tensor decompositions and applications, *SIAM Rev.* 51 (3) (2009) 455–500. doi:10.1137/07070111X.
- [2] B. Hidasi, D. Tikk, Fast ALS-based tensor factorization for context-aware recommendation from implicit feedback, in: P. A. Flach, T. Bie, N. Cristianini (Eds.), *Machine Learning and Knowledge Discovery in Databases*, Springer, 2012, pp. 67–82. doi:10.1007/978-3-642-33486-3_5.
- [3] G. Beylkin, J. Garcke, M. J. Mohlenkamp, Multivariate regression and machine learning with sums of separable functions, *SIAM J. Sci. Comput.* 31 (3) (2009) 1840–1857. doi:10.1137/070710524.
- [4] M. Signoretto, R. Van de Plas, B. De Moor, J. A. K. Suykens, Tensor versus matrix completion: A comparison with application to spectral data, *IEEE Signal Proc. Let.* 18 (7) (2011) 403–406. doi:10.1109/LSP.2011.2151856.
- [5] C. Da Silva, F. J. Herrmann, Hierarchical Tucker tensor optimization - applications to tensor completion, in: *SampTA 2013*, Bremen, Germany, 2013, pp. 384–387.
- [6] C. Teflioudi, F. Makari, R. Gemulla, Distributed matrix completion, in: *IEEE 12th International Conference on Data Mining*, IEEE Computer Society, Los Alamitos, CA, USA, 2012, pp. 655–664. doi:10.1109/ICDM.2012.120.
- [7] Y. Zhou, D. Wilkinson, R. Schreiber, R. Pan, Large-scale parallel collaborative filtering for the Netflix prize, in: R. Fleischer, J. Xu (Eds.), *Algorithmic Aspects in Information and Management*, Springer, 2008, pp. 337–348. doi:10.1007/978-3-540-68880-8_32.
- [8] I. Pilászy, D. Zibriczky, D. Tikk, Fast ALS-based matrix factorization for explicit and implicit feedback datasets, in: *Proceedings of the Fourth ACM Conference on Recommender Systems, RecSys '10*, ACM, New York, NY, USA, 2010, pp. 71–78. doi:10.1145/1864708.1864726.
- [9] H.-F. Yu, C.-J. Hsieh, S. Si, I. Dhillon, Scalable coordinate descent approaches to parallel matrix factorization for recommender systems, in: *IEEE 12th International Conference on Data Mining*, Los Alamitos, CA, USA, 2012, pp. 765–774. doi:10.1109/ICDM.2012.168.
- [10] H.-F. Yu, C.-J. Hsieh, S. Si, I. S. Dhillon, Parallel matrix factorization for recommender systems, *Knowl. Inf. Syst.* In press. doi:10.1007/s10115-013-0682-2.
- [11] R. Gemulla, E. Nijkamp, P. J. Haas, 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. doi:10.1145/2020408.2020426.
- [12] B. Recht, C. Ré, Parallel stochastic gradient algorithms for large-scale matrix completion, *Math. Program. Comput.* 5 (2) (2013) 201–226. doi:10.1007/s12532-013-0053-8.

- [13] E. Acar, D. M. Dunlavy, T. G. Kolda, M. Morup, Scalable tensor factorizations for incomplete data, *Chemometr. Intell. Lab.* 106 (1) (2011) 41–56. doi:10.1016/j.chemolab.2010.08.004.
- [14] G. Tomasi, R. Bro, PARAFAC and missing values, *Chemometr. Intell. Lab.* 75 (2) (2005) 163–180. doi:10.1016/j.chemolab.2004.07.003.
- [15] Y. Xu, W. Yin, A block coordinate descent method for regularized multiconvex optimization with applications to nonnegative tensor factorization and completion, *SIAM J. Imaging Sci.* 6 (3) (2013) 1758–1789. doi:10.1137/120887795.
- [16] A. H. Phan, A. Cichocki, PARAFAC algorithms for large-scale problems, *Neurocomputing* 74 (11) (2011) 1970–1984. doi:10.1016/j.neucom.2010.06.030.
- [17] V. de Silva, L.-H. Lim, Tensor rank and the ill-posedness of the best low-rank approximation problem, *SIAM J. Matrix Anal. Appl.* 30 (3) (2008) 1084–1127. doi:10.1137/06066518X.
- [18] A. Uschmajew, Local convergence of the alternating least squares algorithm for canonical tensor approximation, *SIAM J. Matrix Anal. Appl.* 33 (2) (2012) 639–652. doi:10.1137/110843587.
- [19] J. M. Ortega, W. C. Rheinboldt, *Iterative solution of nonlinear equations in several variables*, Academic Press, New York, 1970.
- [20] Z. Q. Luo, P. Tseng, On the convergence of the coordinate descent method for convex differentiable minimization, *J. Optim. Theory Appl.* 72 (1) (1992) 7–35. doi:10.1007/BF00939948.
- [21] L. Grasedyck, D. Kressner, C. Tobler, A literature survey of low-rank tensor approximation techniques, *GAMM-Mitt.* 36 (1) (2013) 53–78. doi:10.1002/gamm.201310004.
- [22] D. H. Foster, S. M. C. Nascimento, K. Amano, Information limits on neural identification of colored surfaces in natural scenes, *Vis Neurosci.* 21 (2004) 331–336, see http://personalpages.manchester.ac.uk/staff/david.foster/Hyperspectral_images_of_natural_scenes_04.html (Scene 7). doi:10.1017/S0952523804213335.