Project PARTENSOR (T1EΔK-3360)
Deliverable Π1.1 Efficient Algorithms for Tensor Factorization

Authored by

Paris Karakasis (TSI), George Lourakis (Neurocom), George Lykoudis (Neurocom),
Ioanna Siaminou (TSI), Christos Tsalidis (Neurocom), Athanasios Liavas (TSI),

with contributions by

Christos Kolomvakis (TUC), Ioannis Papagiannakos (TUC)

June 2019

Contents

Abstract

In this report, we summarize the work performed under the framework of Work Package 1, "Efficient Parallel Algorithms for Tensor Factorization" of project PARTENSOR during the first twelve months. More specifically,

1. we develop efficient algorithms for *dense* tensor factorization, under various constraints on the tensor factors, for arbitrary tensor order;

2. we develop parallel implementations of the above mentioned algorithms (up to order eight, with easy extension to higher orders) and test their efficiency on large multi-core computers.

The results are very encouraging, thus, our algorithms and implementations constitute the starting point for the parallel toolbox "PARTENSOR".

Chapter 1

Introduction

1.1 Tensor factorization basics

Tensors are mathematical objects that have recently gained great popularity due to their ability to model multiway data dependencies [1], [2], [3], [4]. Tensor factorization (or decomposition) into latent factors is very important for numerous tasks, such as feature selection, dimensionality reduction, compression, data visualization and interpretation. Tensor factorizations are usually computed as solutions of optimization problems [1], [2]. The Canonical Decomposition or Canonical Polyadic Decomposition (CANDECOMP or CPD), also known as Parallel Factor Analysis (PARAFAC), and the Tucker Decomposition are the two most widely used tensor factorization models. In this report, we focus on algorithms, and their distributed implementations, for the solution of the PARAFAC model, under various factor constraints, Specifically, we consider the unconstrained case as well as the cases of nonnegativity, orthogonality, and sparsity constraints.

Alternating Optimization (AO), All-at-Once Optimization (AOO), and Multiplicative Updates (MUs) are among the most commonly used techniques for tensor factorizations [2], [5]. Recent work for constrained tensor factorization/completion includes, among others, [6], [7], [8], and [9].

In [8], an Alternating Direction Method of Multipliers (ADMM) algorithm for NTF has been derived, and an architecture for its parallel implementation has been outlined. However, the convergence properties of the algorithm in ill-conditioned cases are not favorable, necessitating additional research towards their improvement. In [9], the authors consider constrained matrix/tensor factorization/completion problems. They adopt the AO framework as outer loop and use the ADMM for the solution of the inner constrained optimization problems for one matrix factor conditioned on the rest. The ADMM offers significant flexibility, due to its ability to efficiently handle a wide range of constraints.

In [10], two parallel algorithms for unconstrained tensor factorization/completion have been developed and results concerning the speedup attained by their Message Passing Interface (MPI) implementations on a multi-core system have been reported. Related work on parallel algorithms for sparse tensor decomposition includes [11] and [12].

We adopt the AO framework and solve each subproblem using optimal first-order (i.e., gradient-based) algorithms. These algorithms are very promising since they combine low computational complexity per iteration and fact convergence. In many cases, they are the only hope for the solution of large-scale optimization problems. Then, we develop parallel implementations for distributed computational environments, using Message Passing Interface (MPI).

1.1.1 Notation

$\mathbb{N}_N$ denotes the set $\{1, \ldots, N\}$. Vectors, matrices, and tensors are denoted by small, capital, and calligraphic capital bold letters, respectively; for example, $\mathbf{x}$, $\mathbf{X}$, and $\boldsymbol{\mathcal{X}}$. $\mathbb{R}_+^{I \times J \times K}$ denotes the set of $(I \times J \times K)$ real nonnegative tensors, while $\mathbb{R}_+^{I \times J}$ denotes the set of $(I \times J)$ real nonnegative matrices. For a matrix $\mathbf{A} \in \mathbb{R}^{I \times J}$, we use the elementwise references $a_{i,j}$ or $[\mathbf{A}]_{i,j}$, interchangeably, while we refer to the j-th column of the matrix as $\mathbf{a}_j$. $\mathbb{S}_{(I,J)} = \{\mathbf{X} \in \mathbb{R}^{I \times J} : \mathbf{X}^T\mathbf{X} = \mathbf{I}\}$ denotes the Stiefel manifold formed by all orthonormal J-frames in $\mathbb{R}^I$. Also, $\mathbb{SP}_s := \left\{\mathbf{A} \in \mathbb{R}^{I \times R} \ : \ \|\mathbf{A}\|_1 \leq s\right\}$ denotes the set of matrices with sparsity constraints, regulated by parameter s. $\|\cdot\|_F$ denotes the Frobenius norm of the tensor or matrix argument, $\mathbf{I}$ denotes the identity matrix of appropriate dimensions, and $(\mathbf{A})_+$ denotes the projection of matrix $\mathbf{A}$ onto the set of elementwise nonnegative matrices. Inequality $\mathbf{A} \succeq \mathbf{B}$ means that matrix $\mathbf{A} - \mathbf{B}$ is positive semidefinite, while $\mathbf{A} > \mathbf{0}$ denotes a matrix $\mathbf{A}$ that has positive elements.

1.1.2 Structure

In Chapter 2, we consider the matrix least squares problem with constraints. In Chapter 3, we consider the problem of constrained tensor factorization under the PARAFAC model. In Chapter 4, we present and describe in detail an MPI implementation of the algorithms for constrained tensor factorization. In Chapter 5, we briefly present some elements of the CUDA programming language. In Chapter 6, we test the efficiency of the proposed algorithm with numerical experiments in both serial and parallel computing environments.

Chapter 2

Matrix Least Squares Problems

In this chapter, we describe the matrix least squares problem, which will be our workhorse towards the development of efficient algorithms for tensor factorization.

2.1 Matrix Least Squares

Let $\mathbf{X} \in \mathbb{R}^{M \times N}$ and $\mathbf{B} \in \mathbb{R}^{N \times R}$, and consider the problem

$$\min_{\mathbf{A}} f(\mathbf{A}) = \frac{1}{2} \|\mathbf{X} - \mathbf{A}\mathbf{B}^T\|_F^2. \tag{2.1}$$

This problem is known as the Matrix Least Squares problem. It is easy to see that it is a convex optimization problem, hence, the existence of a global minimizer of f, $\mathbf{A}^*$, is guaranteed. The solution $\mathbf{A}^*$ must satisfy the following linear system of equations

$$\mathbf{X}\mathbf{B} = \mathbf{A}^*\mathbf{B}^T\mathbf{B}, \tag{2.2}$$

which are known are *normal equations*. Thus, $\mathbf{A}^*$ is given by

$$\mathbf{A}^* = \mathbf{X}\mathbf{B}^\dagger = \mathbf{X}\mathbf{B}\left(\mathbf{B}^T\mathbf{B}\right)^{-1}. \tag{2.3}$$

For an extensive discussion on the computational aspects of the solution of the normal equations, the reader is referred to [13].

2.2 Matrix Nonnegative Least Squares

Let $\mathbf{X} \in \mathbb{R}^{M \times N}$, $\mathbf{A} \in \mathbb{R}_+^{M \times R}$, $\mathbf{B} \in \mathbb{R}^{N \times R}$, and consider the Matrix Nonnegative Least Squares (MNLS) problem

$$\min_{\mathbf{A} \geq \mathbf{0}} f(\mathbf{A}) = \frac{1}{2} \|\mathbf{X} - \mathbf{A}\mathbf{B}^T\|_F^2. \tag{2.4}$$

The problem in (2.4) is a convex optimization problem with element-wise inequality constraints. Due to the lack of existence of a closed form solution, we use iterative numerical methods for its solution.

In the next section, we initially consider general L-smooth μ-strongly convex optimization problems and present optimal first-order algorithms for their solution. Then, we derive the corresponding optimal algorithm for the MNLS problem.

2.2.1 Optimal first-order algorithms for L-smooth μ-strongly convex optimization problems

We consider optimization problems of smooth and strongly convex functions and briefly present results concerning their information complexity and the associated first-order optimal algorithms (for a detailed exposition see [14, Chapter 2]).

We assume that $f : \mathbb{R}^n \to \mathbb{R}$ is a smooth (that is, differentiable up to a sufficiently high order) convex function, with gradient $\nabla f(\mathbf{x})$ and Hessian $\nabla^2 f(\mathbf{x})$. Our aim is to solve the problem

$$\min_{\mathbf{x}} f(\mathbf{x}), \tag{2.5}$$

within accuracy $\epsilon > 0$. The solution accuracy is defined as follows. If $f^* := \min_{\mathbf{x}} f(\mathbf{x})$, then point $\bar{\mathbf{x}} \in \mathbb{R}^n$ solves problem (2.5) within accuracy ϵ when $f(\bar{\mathbf{x}}) - f^* \leq \epsilon$.

Let $0 < \mu \leq L < \infty$. A smooth convex function f is called L-smooth or, using the notation of [14, p. 66], $f \in \mathcal{S}_{0,L}^{\infty,1}$, if

$$\mathbf{0} \preceq \nabla^2 f(\mathbf{x}) \preceq L\mathbf{I}, \quad \forall \mathbf{x} \in \mathbb{R}^n, \tag{2.6}$$

and L-smooth μ-strongly convex, or $f \in \mathcal{S}_{\mu,L}^{\infty,1}$, if

$$\mu\mathbf{I} \preceq \nabla^2 f(\mathbf{x}) \preceq L\mathbf{I}, \quad \forall\, \mathbf{x} \in \mathbb{R}^n. \tag{2.7}$$

The number of iterations that first-order methods need for the solution of problem (2.5), within accuracy ϵ, is $O\left(\frac{1}{\sqrt{\epsilon}}\right)$ if $f \in \mathcal{S}_{0,L}^{\infty,1}$, and $O\left(\sqrt{\frac{L}{\mu}} \log \frac{1}{\epsilon}\right)$ if $f \in \mathcal{S}_{\mu,L}^{\infty,1}$ [14, Theorem 2.2.2]. The convergence rate in the first case is *sublinear*, while, in the second case, it is *linear* and determined by the condition number of the problem, $\mathcal{K} := \frac{L}{\mu}$. Thus, strong convexity is a very important property that should be exploited whenever possible.

An algorithm that achieves this complexity, and, thus, is first-order optimal, appears in Algorithm 1 (see, also [14, p. 80]). This algorithm can handle both the L-smooth case, by setting $q = 0$, and the L-smooth μ-strongly convex case, by setting $q = \frac{\mu}{L} > 0$.

If the problem of interest is the constrained problem

$$\min_{\mathbf{x} \in \mathbb{X}} f(\mathbf{x}), \tag{2.8}$$

Algorithm 1: Nesterov-type algorithm for L-smooth μ-strongly convex problems

Input: $\mathbf{x}_0 \in \mathbb{R}^N$, μ, L. Set $\mathbf{y}_0 = \mathbf{x}_0$, $\alpha_0 \in (0,1)$, $q = \frac{\mu}{L}$.

1 k-th iteration

2 $\quad \mathbf{x}_{k+1} = \mathbf{y}_k - \frac{1}{L}\nabla f(\mathbf{y}_k)$

3 $\quad \alpha_{k+1} \in (0,1)$ from $\alpha_{k+1}^2 = (1-\alpha_{k+1})\alpha_k^2 + q\alpha_{k+1}$

4 $\quad \beta_{k+1} = \frac{\alpha_k(1-\alpha_k)}{\alpha_k^2+\alpha_{k+1}}$

5 $\quad \mathbf{y}_{k+1} = \mathbf{x}_{k+1} + \beta_{k+1}(\mathbf{x}_{k+1} - \mathbf{x}_k)$

where $\mathbb{X}$ is a closed convex set, then the corresponding optimal algorithm is very much alike Algorithm 1, with the only difference being in the computation of $\mathbf{x}_{k+1}$. We now have that [14, p. 90]

$$\mathbf{x}_{k+1} = \Pi_{\mathbb{X}}\left(\mathbf{y}_k - \frac{1}{L}\nabla f(\mathbf{y}_k)\right), \tag{2.9}$$

where $\Pi_{\mathbb{X}}(\cdot)$ denotes the Euclidean projection onto set $\mathbb{X}$. The convergence properties of this algorithm are the same as those of Algorithm 1. If the projection onto set $\mathbb{X}$ is easy to compute, then the algorithm is both theoretically optimal and very efficient in practice.

2.2.2 Optimal first-order methods for L-smooth μ-strongly convex MNLS problems

In this section, we present an optimal first-order algorithm for the solution of L-smooth μ-strongly convex MNLS problems. Optimal first-order methods have recently attracted great research interest because they are strong candidates and, in many cases, the only viable way for the solution of very large optimization problems.

Nesterov-type algorithm for MNLS with proximal term

Let $\mathbf{X} \in \mathbb{R}^{M\times N}$, $\mathbf{A} \in \mathbb{R}^{M\times R}$, $\mathbf{B} \in \mathbb{R}^{N\times R}$, and consider the problem

$$\min_{\mathbf{A}\geq\mathbf{0}} f(\mathbf{A}) := \frac{1}{2}\left\|\mathbf{X} - \mathbf{A}\mathbf{B}^T\right\|_F^2. \tag{2.10}$$

The gradient and Hessian of f, at point $\mathbf{A}$, are, respectively,

$$\nabla f(\mathbf{A}) = -\left(\mathbf{X} - \mathbf{A}\mathbf{B}^T\right)\mathbf{B} \tag{2.11}$$

and

$$\nabla^2 f(\mathbf{A}) := \frac{\partial^2 f(\mathbf{A})}{\partial \mathrm{vec}(\mathbf{A})\partial \mathrm{vec}(\mathbf{A})^T} = \mathbf{B}^T\mathbf{B} \otimes \mathbf{I} \succeq \mathbf{0}. \tag{2.12}$$

Let $L := \max(\mathrm{eig}(\mathbf{B}^T\mathbf{B}))$ and $\mu := \min(\mathrm{eig}(\mathbf{B}^T\mathbf{B}))$. If $\mu = 0$ (for example, if $R > N$), then problem (2.10) is L-smooth. If $\mu > 0$, then problem (2.10) is L-smooth μ-strongly convex. A

Algorithm 2: Nesterov-type algorithm for MNLS problems with proximal term

Input: $\mathbf{X} \in \mathbb{R}^{M\times N}$, $\mathbf{B} \in \mathbb{R}^{N\times R}$, $\mathbf{A}_* \in \mathbb{R}^{M\times R}$

$L = \max(\text{eig}(\mathbf{B}^T\mathbf{B}))$, $\mu = \min(\text{eig}(\mathbf{B}^T\mathbf{B}))$

$\lambda = g(L, \mu)$

$\mathbf{W} = -\mathbf{X}\mathbf{B} - \lambda\mathbf{A}_*$, $\mathbf{Z} = \mathbf{B}^T\mathbf{B} + \lambda\mathbf{I}$

$q = \frac{\mu+\lambda}{L+\lambda}$

$\mathbf{A}_0 = \mathbf{Y}_0 = \mathbf{A}_*$

$\alpha_0 = 1$, $k = 0$

while (1) **do**

  $\nabla f_{\mathrm{P}}(\mathbf{Y}_k) = \mathbf{W} + \mathbf{Y}_k\mathbf{Z}$

  if (terminating_condition is TRUE) **then**

    break

  else

    $\mathbf{A}_{k+1} = \left(\mathbf{Y}_k - \frac{1}{L+\lambda}\nabla f_{\mathrm{P}}(\mathbf{Y}_k)\right)_+$

    $\alpha_{k+1}^2 = (1-\alpha_{k+1})\alpha_k^2 + q\alpha_{k+1}$

    $\beta_{k+1} = \frac{\alpha_k(1-\alpha_k)}{\alpha_k^2+\alpha_{k+1}}$

    $\mathbf{Y}_{k+1} = \mathbf{A}_{k+1} + \beta_{k+1}(\mathbf{A}_{k+1} - \mathbf{A}_k)$

    $k = k + 1$

return $\mathbf{A}_k$.

first-order optimal algorithm for the solution of (2.10) can be derived using the approach of Section 2.2.1. We note that [15] and [16] solved problem (2.10) using a variation of Algorithm 1, which is equivalent to Algorithm 1 with $\mu = 0$. However, if $\mu > 0$, then this algorithm is *not* first-order optimal and, as we shall see later, it performs much worse than the optimal.

We note that the values of L and μ are necessary for the development of the Nesterov-type algorithm, thus, their computation is imperative.[1]

Under the AO framework, in order to avoid very ill-conditioned problems (and guarantee strong convexity), we introduce a proximal term and solve problem

$$\min_{\mathbf{A}\geq\mathbf{0}} f_{\mathrm{P}}(\mathbf{A}) := \frac{1}{2}\|\mathbf{X} - \mathbf{A}\mathbf{B}^T\|_F^2 + \frac{\lambda}{2}\|\mathbf{A} - \mathbf{A}_*\|_F^2, \tag{2.13}$$

for given $\mathbf{A}_*$ and appropriately chosen λ. We choose λ based on L and μ, and denote this functional dependence as $\lambda = g(L, \mu)$. If $\frac{\mu}{L} \ll 1$, then we may set $\lambda \approx 10\mu$, significantly

[1] An alternative to their direct computation is to estimate L using line-search techniques and overcome the computation of μ using heuristic adaptive restart techniques [17]. However, in our case, this alternative is computationally demanding, especially for large-scale problems, and shall not be considered.

improving the conditioning of the problem by putting large weight on the proximal term; however, in this case, we expect that the optimal point will be biased towards $\mathbf{A}_*$. Otherwise, we may set $\lambda \lessapprox \mu$, putting small weight on the proximal term and permitting significant progress towards the computation of $\mathbf{A}$ that satisfies approximate equality $\mathbf{X} \approx \mathbf{A}\mathbf{B}^T$ as accurately as possible.

The gradient of f_{P}, at point $\mathbf{A}$, is

$$\nabla f_{\mathrm{P}}(\mathbf{A}) = -\left(\mathbf{X} - \mathbf{A}\mathbf{B}^T\right)\mathbf{B} + \lambda(\mathbf{A} - \mathbf{A}_*). \tag{2.14}$$

The Karush-Kuhn-Tucker (KKT) conditions for problem (2.13) are [15]

$$\nabla f_{\mathrm{P}}(\mathbf{A}) \geq \mathbf{0},\ \mathbf{A} \geq \mathbf{0},\ \nabla f_{\mathrm{P}}(\mathbf{A}) \circledast \mathbf{A} = \mathbf{0}. \tag{2.15}$$

These expressions can be used in a terminating condition. For example, we may terminate the algorithm if

$$\min_{i,j}\left([\nabla f_{\mathrm{P}}(\mathbf{A})]_{i,j}\right) > -\delta_1, \max_{i,j}\left(\left|[\nabla f_{\mathrm{P}}(\mathbf{A}) \circledast \mathbf{A}]_{i,j}\right|\right) < \delta_2, \tag{2.16}$$

for small positive real numbers δ_1 and δ_2, while $\mathbf{A} \circledast \mathbf{B}$ denotes the elementwise product of matrices $\mathbf{A}$ and $\mathbf{B}$. Of course, other criteria, based, for example, on the (relative) change of the cost function can be used in terminating conditions.

A Nesterov-type algorithm for the solution of the MNLS problem with proximal term (2.13) is given in Algorithm 2. For notational convenience, we denote Algorithm 2 as

$$\mathbf{A}_{\mathrm{opt}} = \text{Nesterov\_MNLS}(\mathbf{X}, \mathbf{B}, \mathbf{A}_*).$$

Computational complexity of Algorithm 2

Quantities $\mathbf{W}$ and $\mathbf{Z}$ are computed once per algorithm call and cost, respectively, $O(MNR)$ and $O(RN^2)$ arithmetic operations. Quantities L and μ are also computed once and cost at most $O(R^3)$ operations. $\nabla f_{\mathrm{P}}(\mathbf{Y}_k)$, $\mathbf{A}_k$, and $\mathbf{Y}_k$ are updated in every iteration with cost $O(MR^2)$, $O(MR)$, and $O(MR)$ arithmetic operations, respectively.

2.3 Matrix Least Squares with Orthogonality Constraints (Orthogonal Procrustes)

Given two matrices $\mathbf{X} \in \mathbb{R}^{N\times M}$ and $\mathbf{B} \in \mathbb{R}^{M\times R}$, the optimization problem

$$\min_{\mathbf{A}\in\mathbb{S}_{(N,D)}} \quad f(\mathbf{A}) = \left\|\mathbf{X} - \mathbf{A}\mathbf{B}^T\right\|_F^2, \tag{2.17}$$

is known as Orthogonal Procrustes (OP) and has a closed form solution given by [18, 19]

$$\mathbf{A}_{\text{opt}} = \mathbf{U}\mathbf{V}^T = \mathbf{M}\left(\mathbf{M}^T\mathbf{M}\right)^{-\frac{1}{2}}, \tag{2.18}$$

where matrices $\mathbf{U} \in \mathbb{R}^{N\times R}$ and $\mathbf{V} \in \mathbb{R}^{R\times R}$ are given by the singular value decomposition of matrix $\mathbf{M} = \mathbf{X}\mathbf{B} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$.

2.3.1 Computational complexity of the OP problem

For later use, we notice that an efficient way of solving the OP problem, after calculating matrix $\mathbf{M}$ with computational complexity $\mathcal{O}(NMR)$ arithmetic operations and when $\min(N, M) > R$, is the following algorithm:

1. Calculate $\mathbf{M}^T\mathbf{M}$, with complexity $O\left(NR^2\right)$;
2. Calculate the eigen-decomposition of $\mathbf{M}^T\mathbf{M} = \mathbf{V}\mathbf{\Sigma}\mathbf{V}^T$ with complexity $O\left(R^3\right)$;
3. Set $\mathbf{G} = \mathbf{M}\mathbf{V}\mathbf{\Sigma}^{-\frac{1}{2}}\mathbf{V}^T$ with complexity $O\left(NR^2\right)$.

Thus, the overall complexity is $\mathcal{O}\left(NR^2\right)$ in contrast to computing the singular value decomposition of matrix $\mathbf{M}$ in $O\left(N^2R\right)$. The most demanding computation of this approach is the computation of matrix $\mathbf{M}$.

2.4 Sparse Matrix Least Squares

Let $\mathbf{X} \in \mathbb{R}^{M\times N}$, $\mathbf{A} \in \mathbb{R}^{M\times R}$, $\mathbf{B} \in \mathbb{R}^{N\times R}$, and consider problem

$$\min_{\mathbf{A}} f(\mathbf{A}) = \frac{1}{2}\|\mathbf{X} - \mathbf{A}\mathbf{B}^T\|_F^2 + \lambda\|\mathbf{A}\|_1. \tag{2.19}$$

The problem (2.19) is the matrix least squares with sparsity constraints. The ℓ_1-norm is used as a sparsity inducing term. This class of problems do not have a closed form solution, thus, for their solution, we rely on efficient iterative algorithms.

2.4.1 Fast Iterative Shrinkage Thresholding Algorithm for LASSO Problems

In this section, we present the Fast Iterative Shrinkage Thresholding Algorithm (FISTA) [20], an optimal first order method for the solution of optimization problems of the form

$$\min_{\mathbf{x}} f(\mathbf{x}) := f_0(\mathbf{x}) + h(\mathbf{x}), \tag{2.20}$$

where $h(\mathbf{x})$ is not differentiable. Given a point $\mathbf{y}_k$, a gradient step is performed using only $\nabla f_0(\mathbf{y}_k)$, and the new point is computed via

$$\mathbf{x}_{k+1} = \text{prox}_h(\mathbf{y}_k - \eta \nabla f_0(\mathbf{y}_k)), \tag{2.21}$$

where $\text{prox}_h(\cdot)$ is the proximal map of h.

Consider the problem of the form

$$\min_{\mathbf{x}} f(\mathbf{x}) = \frac{1}{2} \|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2^2 + \lambda \|\mathbf{x}\|_1, \tag{2.22}$$

where $\mathbf{x} \in \mathbb{R}^N$, $\mathbf{b} \in \mathbb{R}^M$, and $\mathbf{A} \in \mathbb{R}^{M \times N}$. This optimization problem is known as Least Absolute Shrinkage and Selection Operator (LASSO) [21]. By letting $f_0(\mathbf{x}) = \frac{1}{2} \|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2^2$ and $h(\mathbf{x}) = \lambda \|\mathbf{x}\|_1$, the gradient ∇f_0 at point $\mathbf{x}$ is given by

$$\nabla f_0(\mathbf{x}) = \mathbf{A}^T(\mathbf{A}\mathbf{x} - \mathbf{b}). \tag{2.23}$$

When applied to problem (2.22), the proximal map reduces to soft-thresholding. Namely,

$$\text{prox}_h(\mathbf{y}_k - (\eta \cdot \lambda)\nabla f_0(\mathbf{y}_k)) = \text{sthr}_{\eta \cdot \lambda}(\mathbf{y}_k - (\eta \cdot \lambda)\nabla f_0(\mathbf{y}_k)). \tag{2.24}$$

For the i-th element of vector $\mathbf{x}$, the soft-thresholding operator is given by

$$\text{sthr}_{\eta \cdot \lambda}(x_i) = \text{sgn}(x_i)[|x_i| - (\eta \cdot \lambda)]_+, \tag{2.25}$$

where the operator $[\cdot]_+$ is the projection onto the positive orthant. The algorithm FISTA for the optimization problem (2.22) is given in Algorithm 3.

Algorithm 3: FISTA algorithm for (2.22) optimization problems

Input: $\mathbf{x}_0 \in \mathbb{R}^N$, smooth constant of f_0 L, λ. Set $\mathbf{y}_1 = \mathbf{x}_0$, $t_1 = 1$, $\eta = 1/L$.

1 k-th iteration

2 $\mathbf{x}_k = \text{sthr}_{\eta \cdot \lambda}(\mathbf{y}_k - (\eta \cdot \lambda)\nabla f_0(\mathbf{y}_k))$

3 $t_{k+1} = \dfrac{1 + \sqrt{1 + 4t_k^2}}{2}$

4 $\mathbf{y}_{k+1} = \mathbf{x}_k + \dfrac{t_k - 1}{t_{k+1}}(\mathbf{x}_k - \mathbf{x}_{k-1})$

2.4.2 FISTA for Sparse Matrix Least Squares Problems

In the sequel, we present the extension of FISTA for the matrix least squares problems with sparsity constraints. Consider the optimization problem

$$\min_{\mathbf{A}} f(\mathbf{A}) := \frac{1}{2} \|\mathbf{X} - \mathbf{A}\mathbf{B}^T\|_F^2 + \lambda \|\mathbf{A}\|_1, \tag{2.26}$$

where $\mathbf{X} \in \mathbb{R}^{M\times N}$, $\mathbf{A} \in \mathbb{R}^{M\times R}$, $\mathbf{B} \in \mathbb{R}^{N\times R}$ and $\|\mathbf{A}\|_1$ is the sum of the absolute values of the entries of $\mathbf{A}$. The gradient of f_0 at point $\mathbf{A}$ is

$$\nabla f_0(\mathbf{A}) = -\left(\mathbf{X} - \mathbf{A}\mathbf{B}^T\right)\mathbf{B}. \tag{2.27}$$

Also, let $L := \max(\text{eig}(\mathbf{B}^T\mathbf{B}))$. Similarly to the discussion in the previous section, a standard gradient step is performed using ∇f_0, and the new estimate is computed via soft-thresholding. The FISTA-type algorithm for the LS with sparsity constraints optimization problem is given in Algorithm 4.

Algorithm 4: FISTA-type algorithm for the solution of (2.19)

Input: $\mathbf{X} \in \mathbb{R}^{M\times N}$,$\mathbf{A}_0 \in \mathbb{R}^{M\times R}$, $\mathbf{B} \in \mathbb{R}^{N\times R}$, L, λ. Set $\mathbf{Y}_1 = \mathbf{A}_0$, $t_1 = 1$, $\eta = 1/L$.

1 k-th iteration

2 $\quad \mathbf{A}_k = \text{sthr}_{\eta\cdot\lambda}(\mathbf{Y}_k - (\eta\cdot\lambda)\nabla f_0(\mathbf{Y}_k))$

3 $\quad t_{k+1} = \dfrac{1+\sqrt{1+4t_k^2}}{2}$

4 $\quad \mathbf{Y}_{k+1} = \mathbf{A}_k + \dfrac{t_k - 1}{t_{k+1}}(\mathbf{A}_k - \mathbf{A}_{k-1})$

A tolerance parameter $\text{tol}_{\text{FISTA}}$ is used for the termination of the algorithm. Specifically, a relative factor change criterion is used for the last two iterations of the algorithm. Namely

$$\frac{\|\mathbf{A}_{k+1} - \mathbf{A}_k\|_F}{\|\mathbf{A}_k\|_F} < \text{tol}_{\text{FISTA}}. \tag{2.28}$$

For convenience, we denote Algorithm 4 as

$$\mathbf{A}_{\text{opt}} = \text{Fista\_l}_1(\mathbf{X}, \mathbf{B}, \mathbf{A}_0). \tag{2.29}$$

An equivalent problem to (2.26) is

$$\begin{aligned} &\min_{\mathbf{A}} \ \frac{1}{2}\|\mathbf{X} - \mathbf{A}\mathbf{B}^T\|_F^2 \\ &\text{s.t.} \ \ \mathbf{A} \in \mathbb{SP}_{s_{\mathbf{A}}}, \end{aligned} \tag{2.30}$$

where $s_{\mathbf{A}}$ is a parameter that regulates the sparsity of matrix $\mathbf{A}$. The parameters λ and $s_{\mathbf{A}}$ are closely related with each other. The acquisition of one parameter from the other is possible through a tuning process, which is beyond the scope of this work.

Computational Complexity of Algorithm 4

Matrices $\mathbf{XB}$ and $\mathbf{B}^T\mathbf{B}$ are computed once per algorithm call, and demand $\mathcal{O}(MNR)$ and $\mathcal{O}(NR^2)$ arithmetic operations. Quantity L is also computed once and cost $\mathcal{O}(R^3)$ operations. The gradient $\nabla f_0(\mathbf{Y}_k)$, $\mathbf{A}_k$ and $\mathbf{Y}_k$ are updated in every iteration with computational cost $\mathcal{O}(MR^2)$, $\mathcal{O}(MR)$ and $\mathcal{O}(MR)$, respectively.

Chapter 3

Tensor Factorizations

3.1 Mathematical Background

3.1.1 Definitions

Definition 3.1.1 *Let* $\mathbf{a} \in \mathbb{R}^N$, $\mathbf{b} \in \mathbb{R}^P$, *and* $\mathbf{c} \in \mathbb{R}^J$. *The* ***outer product*** *of* $\mathbf{a}$ *and* $\mathbf{b}$ *is defined as the rank-one matrix with elements*

$$[\mathbf{a} \circ \mathbf{b}]_{n,p} = a_n b_p, \tag{3.1}$$

for all $n \in \mathbb{N}_N$, $p \in \mathbb{N}_P$, *and the outer product of* $\mathbf{a}$, $\mathbf{b}$, *and* $\mathbf{c}$ *is defined as the rank-one tensor with elements*

$$[\mathbf{a} \circ \mathbf{b} \circ \mathbf{c}]_{n,p,j} = a_n b_p c_j, \tag{3.2}$$

for all $n \in \mathbb{N}_N$, $p \in \mathbb{N}_P$, *and* $j \in \mathbb{N}_J$.

Definition 3.1.2 *Let* $\mathbf{A} \in \mathbb{R}^{N \times M}$ *and* $\mathbf{B} \in \mathbb{R}^{P \times K}$. *The* ***Kronecker product*** *(or tensor product) of* $\mathbf{A}$ *and* $\mathbf{B}$ *is defined as the matrix*

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{1,1}\mathbf{B} & \cdots & a_{1,M}\mathbf{B} \\ \vdots & \ddots & \vdots \\ a_{N,1}\mathbf{B} & \cdots & a_{N,M}\mathbf{B} \end{bmatrix} \in \mathbb{R}^{NP \times MK}. \tag{3.3}$$

Definition 3.1.3 *Let* $\mathbf{A} \in \mathbb{R}^{N \times M}$ *and* $\mathbf{B} \in \mathbb{R}^{P \times M}$. *The* ***Khatri-Rao product*** *of* $\mathbf{A}$ *and* $\mathbf{B}$ *is defined as the matrix*

$$\mathbf{A} \odot \mathbf{B} = \begin{bmatrix} \mathbf{a}_1 \otimes \mathbf{b}_1 & \cdots & \mathbf{a}_M \otimes \mathbf{b}_M \end{bmatrix} \in \mathbb{R}^{NP \times M}. \tag{3.4}$$

Definition 3.1.4 *Let* $\mathbf{A} \in \mathbb{R}^{N \times M}$ *and* $\mathbf{B} \in \mathbb{R}^{N \times M}$. *The* ***Hadamard Product*** *or elementwise matrix product of* $\mathbf{A}$ *and* $\mathbf{B}$, *is a matrix of size* $N \times M$, *and is defined as*

$$[\mathbf{A} \circledast \mathbf{B}]_{n,m} = a_{n,m} b_{n,m}, \tag{3.5}$$

for all $n \in \mathbb{N}_N$, $m \in \mathbb{N}_M$.

Regarding the Kronecker and Khatri-Rao products, we notice that they are both associative [4]. For example, a Khatri-Rao product of three matrices can be equivalently calculated as

$$\mathbf{A} \odot \mathbf{B} \odot \mathbf{C} = (\mathbf{A} \odot \mathbf{B}) \odot \mathbf{C} = \mathbf{A} \odot (\mathbf{B} \odot \mathbf{C}) \tag{3.6}$$

For the latter three products, the following properties hold [3]:

$$(\mathbf{A} \otimes \mathbf{B})(\mathbf{C} \otimes \mathbf{D}) = \mathbf{AC} \otimes \mathbf{BD}, \tag{3.7}$$

$$(\mathbf{A} \otimes \mathbf{B})^\dagger = \mathbf{A}^\dagger \otimes \mathbf{B}^\dagger, \tag{3.8}$$

$$(\mathbf{A} \odot \mathbf{B})^T (\mathbf{A} \odot \mathbf{B}) = \mathbf{A}^T\mathbf{A} \circledast \mathbf{B}^T\mathbf{B}, \tag{3.9}$$

$$(\mathbf{A} \odot \mathbf{B})^\dagger = ((\mathbf{A}^T\mathbf{A}) \circledast (\mathbf{B}^T\mathbf{B}))^{-1}(\mathbf{A} \odot \mathbf{B})^T, \tag{3.10}$$

where $\mathbf{A}^\dagger$ denotes the Moore-Penrose pseudoinverse of $\mathbf{A}$. Based on the associativity of these products, we can derive generalized expressions for properties (3.7), (3.9), and (3.10) for three or more operands.

Definition 3.1.5 *Let* $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ *and* $\mathbf{U}^{(i)} \in \mathbb{R}^{I_i \times R}$, *for* $i \in \mathbb{N}_N$. *We define the* ***rank*** *of* $\boldsymbol{\mathcal{X}}$ *as the minimum positive integer* R *such that*

$$\boldsymbol{\mathcal{X}} = \sum_{r=1}^{R} \mathbf{u}_r^{(1)} \circ \cdots \circ \mathbf{u}_r^{(N)}. \tag{3.11}$$

Definition 3.1.6 *Let* $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$. *The* ***matricization of the tensor with respect to the j-th mode*** *(mode-j matricization) is defined as the matrix* $\mathbf{X}_{(j)} \in \mathbb{R}^{I_j \times \prod_{k=1,k\neq j}^{N} I_k}$, *where the element* $x_{i_1,\ldots,i_j,\ldots,i_N}$ *is mapped to* $\left[\mathbf{X}_{(j)}\right]_{i_j,k}$ *according to* [3]

$$k = 1 + \sum_{\substack{p=1 \\ p\neq j}}^{N} (i_p - 1)J_p, \text{ where } J_p = \prod_{\substack{m=1 \\ m\neq j}}^{p-1} I_m.$$

This can be better understood through an example. Let a tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I \times J \times K}$. We are interested in examining the mode-1 matricization of $\boldsymbol{\mathcal{X}}$, $\mathbf{X}_{(1)} \in \mathbb{R}^{I \times JK}$. Then, element x_{i_1,i_2,i_3}, will be mapped to element $\left[\mathbf{X}_{(1)}\right]_{i_1,k}$, where

$$k = 1 + (i_2 - 1) + (i_3 - 1)I_2, \tag{3.12}$$

since $\prod_{m=2}^{1} I_m = \prod_{m\in\emptyset} I_m = 1$ (by convention) and $J_3 = \prod_{m=2}^{2} I_m = I_2$. For the mode-2 matricization, $\mathbf{X}_{(2)} \in \mathbb{R}^{J \times IK}$, element x_{i_1,i_2,i_3} will be mapped to element $\left[\mathbf{X}_{(2)}\right]_{i_2,k'}$, where

$$k' = 1 + (i_1 - 1) + (i_3 - 1)I_1,$$

since $J_1 = \prod_{m=1}^{0} I_m = 1$, $J_3 = \prod_{m=2}^{2} I_m = I_1$,, and so on.

Definition 3.1.7 *Let $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times \cdots \times I_n \cdots \times I_N}$ and $\mathbf{U} \in \mathbb{R}^{I_n \times J}$. The **n-mode product** of $\boldsymbol{\mathcal{X}}$ and $\mathbf{U}$, denoted as $\boldsymbol{\mathcal{X}} \times_n \mathbf{U}$, is a tensor of size $I_1 \times \cdots \times I_{n-1} \times J \times I_{n+1} \times \cdots \times I_N$ and it is defined, elementwise, as*

$$(\boldsymbol{\mathcal{X}} \times_n \mathbf{U})_{i_1,\ldots,i_{n-1},j,i_{n+1},\ldots,I_N} = \sum_{i_n=1}^{I_N} x_{i_1,i_2,\ldots,i_n,\ldots,i_N} u_{i_n,j}.$$

The idea can also be expressed in terms of the mode-n matricization of $\boldsymbol{\mathcal{X}}$ since

$$\boldsymbol{\mathcal{Y}} = \boldsymbol{\mathcal{X}} \times_n \mathbf{U} \Longleftrightarrow \mathbf{Y}_{(n)} = \mathbf{U}^T \mathbf{X}_{(n)}.$$

Definition 3.1.8 *Let $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times \cdots \times I_n \cdots \times I_N}$ and $\mathbf{v} \in \mathbb{R}^{I_n}$. The n**-mode vector product** of $\boldsymbol{\mathcal{X}}$ with $\mathbf{v}$ produces a new tensor $\boldsymbol{\mathcal{Y}} \in \mathbb{R}^{I_1 \times \cdots \times I_{n-1} \times I_{n+1} \times \cdots \times I_N}$ and is defined as*

$$y_{i_1,i_2,\ldots,i_{n-1},i_{n+1},\ldots,i_N} = \sum_{j=1}^{I_n} x_{i_1,i_2,\ldots,i_{n-1},j,i_{n+1},\ldots,i_N} v_j.$$

The new tensor $\boldsymbol{\mathcal{Y}}$ has order $N-1$. This operation is also called tensor-times-vector (TTV) multiplication. In addition, ordering of different TTVs does not matter; the following holds [22]

$$\boldsymbol{\mathcal{X}} \times_i \mathbf{u}_1 \times_j \mathbf{u}_2 = \boldsymbol{\mathcal{X}} \times_j \mathbf{u}_2 \times_i \mathbf{u}_1, \tag{3.13}$$

for $\boldsymbol{\mathcal{X}}^{I_1 \times \cdots \times I_i \times \cdots \times I_j \ldots I_N}$, $\mathbf{u}_1 \in \mathbb{R}^{I_i}$ and $\mathbf{u}_2 \in \mathbb{R}^{I_j}$.

Let us assume vectors $\mathbf{v}_i \in \mathbb{R}^{I_i}$, with $i \in \mathbb{N}_N$, and that we want to perform the following operation

$$\boldsymbol{\mathcal{X}} \times_1 \mathbf{v}_1 \times_2 \mathbf{v}_2 \times_3 \cdots \times_N \mathbf{v}_N. \tag{3.14}$$

A more concise way to write it, which we adopt for the rest of this report, is the following

$$\boldsymbol{\mathcal{X}} \times_{i \in \mathbb{N}_N} \mathbf{v}_i. \tag{3.15}$$

3.2 PARAFAC model

Let tensor $\boldsymbol{\mathcal{X}}^o \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ admit a factorization of the form

$$\boldsymbol{\mathcal{X}}^o = [\![\mathbf{U}^{o(1)}, \ldots, \mathbf{U}^{o(N)}]\!] = \sum_{r=1}^{R} \mathbf{u}_r^{o(1)} \circ \cdots \circ \mathbf{u}_r^{o(N)}, \tag{3.16}$$

where $\mathbf{U}^{o(i)} = [\mathbf{u}_1^{o(i)} \ \cdots \ \mathbf{u}_R^{o(i)}] \in \mathbb{R}^{I_i \times R}$, with $i \in \mathbb{N}_N$. We observe the noisy tensor $\boldsymbol{\mathcal{X}} = \boldsymbol{\mathcal{X}}^o + \boldsymbol{\mathcal{E}}$, where $\boldsymbol{\mathcal{E}}$ is the additive noise. Then, estimates of $\mathbf{U}^{o(i)}$ can be obtained by computing matrices

$\mathbf{U}^{(i)} \in \mathbb{R}^{I_i \times R}$, for $i \in \mathbb{N}_N$, that solve the optimization problem

$$\min_{\mathbf{U}^{(1)},\ldots,\mathbf{U}^{(N)}} f_{\boldsymbol{\mathcal{X}}}\left(\mathbf{U}^{(1)},\ldots,\mathbf{U}^{(N)}\right), \tag{3.17}$$

where $f_{\boldsymbol{\mathcal{X}}}$ is a function measuring the quality of the factorization. A common choice for $f_{\boldsymbol{\mathcal{X}}}$ is

$$f_{\boldsymbol{\mathcal{X}}}\left(\mathbf{U}^{(1)},\ldots,\mathbf{U}^{(N)}\right) = \frac{1}{2}\left\|\boldsymbol{\mathcal{X}} - [\![\mathbf{U}^{(1)},\ldots,\mathbf{U}^{(N)}]\!]\right\|_F^2. \tag{3.18}$$

If $\boldsymbol{\mathcal{Y}} = [\![\mathbf{U}^{(1)},\ldots,\mathbf{U}^{(N)}]\!]$, then, for an arbitrary mode i, the corresponding matrix unfolding is given by [3]

$$\mathbf{Y}_{(i)} = \mathbf{U}^{(i)}\mathbf{K}^{(i)^T}, \tag{3.19}$$

where

$$\mathbf{K}^{(i)} = \mathbf{U}^{(N)} \odot \cdots \odot \mathbf{U}^{(i+1)} \odot \mathbf{U}^{(i-1)} \odot \cdots \odot \mathbf{U}^{(1)}. \tag{3.20}$$

Thus, $f_{\boldsymbol{\mathcal{X}}}$ can be expressed as

$$f_{\boldsymbol{\mathcal{X}}}(\mathbf{U}^{(1)},\ldots,\mathbf{U}^{(N)}) = \frac{1}{2}\left\|\mathbf{X}_{(i)} - \mathbf{U}^{(i)}\mathbf{K}^{(i)^T}\right\|_F^2, \quad i \in \mathbb{N}_N. \tag{3.21}$$

These expressions form the basis of the alternating least squares algorithm (ALS) for tensor factorization, in the sense that, for fixed matrix factors $\mathbf{U}^{(j)}$, with $j \neq i$, we can update $\mathbf{U}^{(i)}$ by solving a matrix least squares problem.

3.3 Dimension Trees

As we mentioned above, for fixed matrix factors $\mathbf{U}^{(j)}$, with $j \neq i$, updating $\mathbf{U}^{(i)}$ can be reduced to the solution of a matrix least squares problem. This reduces to the solution of the normal equations, which, for the PARAFAC model, are given by

$$\mathbf{Y}_{(i)}\mathbf{K}^{(i)} = \mathbf{U}^{(i)}\mathbf{H}^{(i)}, \tag{3.22}$$

where $\mathbf{H}^{(i)} = \mathbf{K}^{(i)^T}\mathbf{K}^{(i)} = \overset{N}{\underset{j=1,j\neq i}{\circledast}} \mathbf{U}^{(j)^T}\mathbf{U}^{(j)}$. The left side term of the above equation is referred to as matricized tensor times Khatri-Rao product (MTTKRP). For the PARAFAC decomposition, under the ALS framework, the calculation of MTTKRP constitutes the main computational bottleneck and finding ways to reduce its computational complexity has attracted much interest recently.

Dimension trees are data structures that aim to avoid recalculating expressions that are common among computations of different MTTKRPs during a full cycle of factor updates (one ALS outer iteration). We introduce them by initially examining how they are used in the case of a third-order tensor, and then of a fourth-order tensor. Based on the analysis for the fourth-order tensor, the approach can be easily generalized for the case of Nth-order tensors, with $N > 4$.

Third-Order Tensors

Let us consider a tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I \times J \times K}$ and let $\mathbf{U}^{(1)} \in \mathbb{R}^{I \times R}$, $\mathbf{U}^{(2)} \in \mathbb{R}^{J \times R}$, and $\mathbf{U}^{(3)} \in \mathbb{R}^{K \times R}$ be the approximating factors. Then, the MTTKRPs that correspond to factors $\mathbf{U}^{(1)}$ and $\mathbf{U}^{(2)}$, respectively, are given by

$$\begin{aligned}\mathbf{M}^{(1)} &= \underline{\mathbf{X}_{(1)}(\mathbf{U}^{(3)}} \odot \mathbf{U}^{(2)}),\\ \mathbf{M}^{(2)} &= \underline{\mathbf{X}_{(2)}(\mathbf{U}^{(3)}} \odot \mathbf{U}^{(1)}).\end{aligned}$$

In the sequel, we show that the underlined (temporary) quantities correspond to a common term which is actually a tensor. We will refer to these common tensors as partial MTTKRPs. Each element of matrix $\mathbf{M}^{(1)}$ can be computed as

$$m^{(1)}_{i,r} = \sum_{j,k} x_{i,j,k} u^{(2)}_{j,r} u^{(3)}_{k,r} = \sum_{j} u^{(2)}_{j,r} \sum_{k} x_{i,j,k} u^{(3)}_{k,r} = \sum_{j} u^{(2)}_{j,r} t_{i,j,r},$$

where $\boldsymbol{\mathcal{T}} \in \mathbb{R}^{I \times J \times R}$ is a tensor that can be reused for the computation of $\mathbf{M}^{(2)}$, since

$$m^{(2)}_{j,r} = \sum_{i,k} x_{i,j,k} u^{(1)}_{i,r} u^{(3)}_{k,r} = \sum_{i} u^{(1)}_{i,r} \sum_{k} x_{i,j,k} u^{(3)}_{k,r} = \sum_{i} u^{(1)}_{i,r} t_{i,j,r}.$$

However, the MTTKRP that corresponds to the factor $\mathbf{U}^{(3)}$ has to be computed from ground.

Fourth-Order Tensors

Let $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I \times J \times K \times L}$ and let $\mathbf{U}^{(1)} \in \mathbb{R}^{I \times R}$, $\mathbf{U}^{(2)} \in \mathbb{R}^{J \times R}$, $\mathbf{U}^{(3)} \in \mathbb{R}^{K \times R}$, and $\mathbf{U}^{(4)} \in \mathbb{R}^{L \times R}$ be the approximating factors. Then, the corresponding MTTKRPs are given by

$$\begin{aligned}\mathbf{M}^{(1)} &= \underline{\mathbf{X}^{(1)}(\mathbf{U}^{(4)} \odot \mathbf{U}^{(3)}} \odot \mathbf{U}^{(2)}),\\ \mathbf{M}^{(2)} &= \underline{\mathbf{X}^{(2)}(\mathbf{U}^{(4)} \odot \mathbf{U}^{(3)}} \odot \mathbf{U}^{(1)}),\\ \mathbf{M}^{(3)} &= \underline{\mathbf{X}^{(3)}}(\mathbf{U}^{(4)} \odot \underline{\mathbf{U}^{(2)} \odot \mathbf{U}^{(1)}}),\\ \mathbf{M}^{(4)} &= \underline{\mathbf{X}^{(4)}}(\mathbf{U}^{(3)} \odot \underline{\mathbf{U}^{(2)} \odot \mathbf{U}^{(1)}}).\end{aligned}$$

Working in the same manner as above, we obtain

$$\begin{aligned}m^{(1)}_{i,r} &= \sum_{j,k,l} x_{i,j,k,l} u^{(2)}_{j,r} u^{(3)}_{k,r} u^{(4)}_{l,r} = \sum_{j} u^{(2)}_{j,r} r^{1}_{i,j,r},\\ m^{(2)}_{j,r} &= \sum_{i,k,l} x_{i,j,k,l} u^{(1)}_{i,r} u^{(3)}_{k,r} u^{(4)}_{l,r} = \sum_{i} u^{(1)}_{i,r} r^{1}_{i,j,r},\\ m^{(3)}_{k,r} &= \sum_{i,j,l} x_{i,j,k,l} u^{(2)}_{j,r} u^{(1)}_{i,r} u^{(4)}_{l,r} = \sum_{l} u^{(4)}_{l,r} r^{2}_{k,l,r},\\ m^{(4)}_{l,r} &= \sum_{i,j,k} x_{i,j,k,l} u^{(2)}_{j,r} u^{(1)}_{i,r} u^{(3)}_{k,r} = \sum_{k} u^{(3)}_{k,r} r^{2}_{k,l,r},\end{aligned} \tag{3.23}$$

where $\boldsymbol{\mathcal{R}}^{(1)} \in \mathbb{R}^{I \times J \times R}$ is the result of R TTV products between $\boldsymbol{\mathcal{T}}^{(1)}_r$ and $\mathbf{u}^{(3)}_r$, for $r \in \mathbb{N}_R$, while $\boldsymbol{\mathcal{R}}^{(2)} \in \mathbb{R}^{K \times L \times R}$ is defined in a similar way.

Computing MTTKRP through tensor operations

An alternate way of computing the MTTKRP, through tensor algebra operations, is by using TTVs. The i-th factor's MTTKRP, $\mathbf{M}^{(i)}$, can be computed columnwise. The r-th column, for $r \in \mathbb{N}_R$, is the result of the $(N-1)$ TTVs

$$\mathbf{m}_r^{(i)} = \boldsymbol{\mathcal{X}} \times_{j \in \mathbb{N}_N \setminus i} \mathbf{u}_r^{(j)}. \tag{3.24}$$

3.3.1 Dimension Tree Based ALS

As presented in the previous cases, in each ALS outer iteration different MTTKRPs share common terms, the partial MTTKRPs, that can be calculated once and saved to convenient data structures. One popular choice for the needs of the ALS-based PARAFAC decomposition is the deployment of a specific kind of binary trees, known as *dimension trees*. As we show in Chapter 6, this strategy can offer significant speedup in computing tensor decompositions.

Constructing a Dimension Tree

Given a tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$, a dimension tree follows the structure of a binary tree, where each node of the tree is associated with a tensor and is labeled by a set of indices. The cardinality of this set determines the order of the associated tensor, while the i-th index defines the i-th dimension of the associated tensor, in correspondence to the dimensions of the initial tensor $\boldsymbol{\mathcal{X}}$. Specifically, regarding the root node, the associated tensor will be $\boldsymbol{\mathcal{X}}$ and will be labeled by the set $\mathbb{N}_N$, while the non-root nodes will be associated with tensors of order $\leq N$, with the last dimension being equal to the decomposition rank R, and being labeled with a subset of $\mathbb{N}_N$. This specification of the non-root nodes emerges from the fact that the associated tensors, in the case of non-root nodes, are produced by partial MTTKRP operations. The reader should notice at this point that, for the case of the leaf nodes, the associated tensor will have a maximum order of 2, and by extension to an associated matrix that will coincide with one of the MTTKRP terms.

Next, we demonstrate an example of a dimension tree for the case of a fourth-order tensor. Let us assume that we are interested in factorizing a tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I \times J \times K \times L}$. Then, a dimension tree that could be used along with the ALS algorithm is presented in Figure 3.1. In accordance to the relations in (3.23), nodes $\{1,2\}$ and $\{3,4\}$ will be associated to tensors $\boldsymbol{\mathcal{R}}^{(1)}$ and $\boldsymbol{\mathcal{R}}^{(2)}$, respectively, while nodes $\{1\}$, $\{2\}$, $\{3\}$, and $\{4\}$ will be associated to matrices $\mathbf{M}^{(1)}$, $\mathbf{M}^{(2)}$, $\mathbf{M}^{(3)}$, and $\mathbf{M}^{(4)}$, respectively.

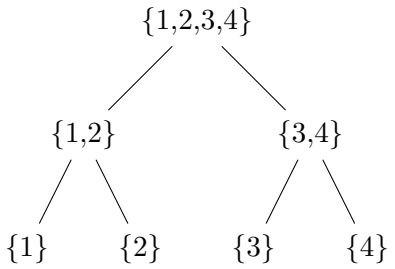

Figure 3.1: Dimension tree for a Tensor of order 4. In general, each node is associated with a tensor, and a set of indices $\mathcal{S} \subseteq \mathbb{N}_{order(\boldsymbol{\mathcal{X}})}$, which give us information regarding the dimensions of the node's tensor.

Updating a Dimension Tree

In this section, we discuss the algorithms that are presented in [22]. Assume that factor $\mathbf{U}^{(i)}$ has been successfully updated after taking into consideration the associated matrix of leaf $\{i\}$. It should be clear that several contents of the dimension tree are no more up to date and proceeding with the course of the ALS algorithm would lead to inaccurate results. Hence, after updating a matrix factor, the need of efficiently updating the dimension tree arises. In order to describe the actions one should perform to achieve this, we have to introduce some notation. A dimension tree generated by a tensor $\boldsymbol{\mathcal{X}}$ is denoted as $\mathcal{T}(\boldsymbol{\mathcal{X}})$. The leaf node labeled by $\{i\}$ is denoted as l_i. For a node t, we denote the set of indices node t has as $\mu(t)$, while we define $\mu'(t)$ to be the complement of $\mu(t)$ with respect to $\mu(\text{root})$, namely $\mu'(t) = \mathbb{N}_N \setminus \mu(t)$. The parent of node t is denoted as $\mathcal{P}(t)$, while the left and right children nodes are denoted as $\mathcal{L}(t)$ and $\mathcal{R}(t)$, respectively. The associated tensor to node t is denoted by $\boldsymbol{\mathcal{X}}^{(t)}$. At last, we let $\delta(t)$ to be the set of indices that the sibling node of t has, namely, $\delta(t) = \mu'(t) \setminus \mu'(\mathcal{P}(t)) = \mu(\mathcal{P}(t)) \setminus \mu(t)$.

Updating a node t requires the parent node $\mathcal{P}(t)$ to be updated. Hence, in order to update node t, all the nodes that belong to the path connecting t and the root node have to be traversed, beginning from node t, until an updated node be met. Note that the existence of an updated node in every ending to the root path is guaranteed, since the content of the root node does not change during the execution of the ALS algorithm and, therefore, can always be considered as updated. As a result, updating node t can be achieved in a recursive fashion. Algorithms 5 and 7, presented in [22], are devised within this framework for the problem of efficient updating a dimension tree for the needs of PARAFAC decomposition via ALS optimization.

Dimension Tree based ALS

Before presenting the algorithms, we introduce some additional notation. By $\boldsymbol{\mathcal{X}}_r^{(t)}$ we refer to a subtensor of $\boldsymbol{\mathcal{X}}^{(t)}$, where the last index is fixed at value r, with $r \in \mathbb{N}_R$. By exception, for the root node we have that $\boldsymbol{\mathcal{X}}_r^{(root)} = \boldsymbol{\mathcal{X}}$, for all values of r.

Algorithm 5 is a recursive algorithm that takes as input argument a node $t \in \mathcal{T}(\boldsymbol{\mathcal{X}})$, deals with the update of all nodes on the path connecting node t and the root node, and returns an updated version of $\boldsymbol{\mathcal{X}}^{(t)}$. From the perspective of [22], a node t' is assumed to be updated when tensor $\boldsymbol{\mathcal{X}}^{(t')}$ exists, since when node t' becomes outdated Algorithm 7 proceeds into the destruction of $\boldsymbol{\mathcal{X}}^{(t')}$. Algorithm 7 can be characterized as a customized version of vanilla ALS PARAFAC, where the interaction of the dimension tree is supported by functions *Construct_Dimension_Tree*, *Destroy*, and *Dtree_TTV*.

Algorithm 5: DTree_TTV: Dimension tree-based TTV with R vectors in each mode

Input: Node $t \in \mathcal{T}(\boldsymbol{\mathcal{X}})$.

Output: Tensor $\boldsymbol{\mathcal{X}}^{(t)}$.

if *exists*$(\boldsymbol{\mathcal{X}}^{(t)})$ **then**
  return
end
$\boldsymbol{\mathcal{X}}^{\mathcal{P}(t)} = \text{DTree\_TTV}(\mathcal{P}(t))$
for $r = 1, \ldots, R$ **do**
  $\boldsymbol{\mathcal{X}}_r^{(t)} = \boldsymbol{\mathcal{X}}_r^{(\mathcal{P}(t))} \times_{d \in \delta(t)} \mathbf{u}_r^{(d)}$
end
return $\boldsymbol{\mathcal{X}}^{(t)}$

Alongside the standard ALS algorithm, we use two functions which have been proven very useful in our experiments, in the sense that they significantly reduce the number of outer iterations necessary to reach convergence.

Function "Normalize" receives as arguments a matrix $\mathbf{U} \in \mathbb{R}^{I \times R}$ and a vector $\boldsymbol{\lambda} \in \mathbb{R}^R$. The Euclidean norm of each column $\mathbf{u}_r$ is calculated and is used to update the element λ_r, multiplicatively. Then, $\mathbf{u}_r$ is set to have unit Euclidean norm. Notice that the j-th element of vector $\boldsymbol{\lambda}$, with $j \in \mathbb{N}_R$, contains the Frobenius norm of the rank-1 tensor $\mathbf{u}_j^{(1)} \circ \cdots \circ \mathbf{u}_j^{(N)}$.

Function "Accelerate" implements an acceleration mechanism. The development of efficient acceleration mechanisms is a very important research topic, see, for example, [23], [24], but is beyond the scope of this report. In our experiments, we adopted the simple acceleration technique used in the function `parafac` of the n-way toolbox [25], which is briefly described as follows.

At iteration $k+1 > k_0$, after the computation and normalization of $\{\mathbf{U}_k^{(i)}\}_{i=1}^N$, we compute

$$\mathbf{U}_{\text{new}}^{(i)} = \mathbf{U}_k^{(i)} + s_{k+1}\left(\mathbf{U}_{k+1}^{(i)} - \mathbf{U}_k^{(i)}\right), \tag{3.25}$$

where s_{k+1} is a small positive number; a simple choice for s_{k+1} is $s_{k+1} = (k+1)^{\frac{1}{n}}$, where n is initialized as $n = 3$ and its value may change as the algorithm progresses. In an analogous manner, we compute $\mathbf{U}_{\text{new}}^{(j)}$, with $j \neq i$. If $f_{\boldsymbol{\mathcal{X}}}(\mathbf{U}_{\text{new}}^{(1)}, \ldots, \mathbf{U}_{\text{new}}^{(N)}) \leq f_{\boldsymbol{\mathcal{X}}}(\mathbf{U}_{k+1}^{(1)}, \ldots, \mathbf{U}_{k+1}^{(N)})$, then the acceleration step is successful, and we set $\{\mathbf{U}_{k+1}^{(j)}\}_{j=1}^N = \{\mathbf{U}_{\text{new}}^{(j)}\}_{j=1}^N$. If the acceleration step fails, then the factor matrix estimates $\{\mathbf{U}_{k+1}^{(j)}\}_{j=1}^N$ remain unchanged, and are given as input to the next AO iteration. If the acceleration step fails for n_0 iterations, then we set $n = n + 1$, thus, decreasing the exponent of the acceleration step. Typical values of k_0 and n_0 are $k_0 = 5$ and $n_0 = 5$.

Algorithm 6: DTree-PARAFAC-ALS: A dimension tree-based PARAFAC-ALS algorithm.

Input: $\boldsymbol{\mathcal{X}}$: An N-mode tensor,
R: The rank of PARAFAC decomposition.

Output: $[\boldsymbol{\lambda}, \mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)}]$: Rank-$R$ PARAFAC decomposition of $\boldsymbol{\mathcal{X}}$.

$k = 0$
$\boldsymbol{\lambda} = \mathbf{1}$
$\mathcal{T}(\boldsymbol{\mathcal{X}}) = \text{Construct\_Dimension\_Tree}(\boldsymbol{\mathcal{X}})$
for $i = 1, \ldots, N$ **do**
  $\mathbf{W}^{(i)} = \mathbf{U}_k^{(i)T}\mathbf{U}_k^{(i)}$
end
repeat
  for $i = 1, \ldots, N$ **do**
    for $t \in \mathcal{T}(\boldsymbol{\mathcal{X}})$ **do**
      if $i \in \mu'(t)$ **then**
        Destroy($\boldsymbol{\mathcal{X}}^{(t)}$)
      end
    end
    $\mathbf{M}^{(i)} = \text{DTree\_TTV}(l_i)$
    $\mathbf{H}^{(i)} = \circledast_{j=1, j\neq i}^{N} \mathbf{W}^{(i)}$
    $\mathbf{U}_{k+1}^{(i)} = \mathbf{M}^{(i)}\mathbf{H}^{(i)\dagger}$
    Normalize($\mathbf{U}_{k+1}^{(i)}, \boldsymbol{\lambda}$)
    $\mathbf{W}^{(i)} = \left(\mathbf{U}_{k+1}^{(i)}\right)^T \mathbf{U}_{k+1}^{(i)}$
  end
  Accelerate($\mathbf{U}_k^{(1)}, \mathbf{U}_{k+1}^{(1)}, \ldots, \mathbf{U}_k^{(N)}, \mathbf{U}_{k+1}^{(N)}$)
  $k = k + 1$
until *convergence is achieved or maximum number of iterations has been reached*;
return $[\boldsymbol{\lambda}, \mathbf{U}_k^{(1)}, \ldots, \mathbf{U}_k^{(N)}]$

3.4 Constrained Tensor Decomposition

In many applications, we are interested in tensor decompositions where the requested factors are desired to comply with constraints emerging from underlying models or for interpretability

reasons. Specifically, let tensor $\boldsymbol{\mathcal{X}}^o \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ admit a factorization of the form

$$\boldsymbol{\mathcal{X}}^o = [\![\mathbf{U}^{(1)o}, \ldots, \mathbf{U}^{(N)o}]\!] = \sum_{r=1}^{R} \mathbf{u}_r^{(1)o} \circ \cdots \circ \mathbf{u}_r^{(N)o}, \tag{3.26}$$

where $\mathbf{U}^{(i)o} = \begin{bmatrix} \mathbf{u}_1^{(i)o} & \cdots & \mathbf{u}_R^{(i)o} \end{bmatrix} \in \mathbb{B}^{(i)} \subseteq \mathbb{R}^{I_i \times R}$, with $i \in \mathbb{N}_N$. We observe the noisy tensor $\boldsymbol{\mathcal{X}} = \boldsymbol{\mathcal{X}}^o + \boldsymbol{\mathcal{E}}$, where $\boldsymbol{\mathcal{E}} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ is the additive noise. Then, the problem of finding estimates of the factors $\mathbf{U}^{(i)o}$ can be formulated as

$$\begin{aligned} \min_{\mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)}} & f_{\boldsymbol{\mathcal{X}}} \left(\mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)} \right) \\ \text{s.t.} \quad & \mathbf{U}^{(i)} \in \mathbb{B}^{(i)}, \quad i \in \mathbb{N}_N, \end{aligned} \tag{3.27}$$

where $f_{\boldsymbol{\mathcal{X}}}$ is a function measuring the quality of the factorization. As in the unconstrained case, we focus on the sum of squared errors cost function

$$f_{\boldsymbol{\mathcal{X}}} \left(\mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)} \right) = \frac{1}{2} \left\| \boldsymbol{\mathcal{X}} - [\![\mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)}]\!] \right\|_F^2. \tag{3.28}$$

Under the ALS framework, each factor can be updated via solving an unconstrained/constrained matrix least squares problem. It can be formulated as

$$\begin{aligned} \min_{\mathbf{U}^{(i)}} \quad & \frac{1}{2} \|\mathbf{X}_{(i)} - \mathbf{U}^{(i)} \mathbf{K}^{(i)}\|_F^2 \\ s.t. \ & \mathbf{U}^{(i)} \in \mathbb{B}^{(i)}, \quad i \in \mathbb{N}_N, \end{aligned} \tag{3.29}$$

where $\mathbf{K}^{(i)}$ is defined in (3.20). In case $\mathbb{B}^{(i)}$ is the set $\mathbb{R}^{I_i \times R}$, factor $\mathbf{U}^{(i)}$ is updated via solving an unconstrained LS problem (see Section 2.1). In case $\mathbf{U}^{(i)} \in \mathbb{R}_+^{I_i \times R}$, in order to update factor $\mathbf{U}^{(i)}$, it suffices to solve a MNLS problem, as introduced in Section 2.2. The special case, where $\mathbf{U}^{(i)} \in \mathbb{R}_+^{I_i \times R}$, for all $i \in \mathbb{N}_N$, is known as Nonnegative Tensor Factorization (NTF). In this formulation, one can adopt the ALS method, where a sequence of MNLS problems is solved at every AO iteration [26].

For the case where $\mathbb{B}^{(i)}$ is the set $\mathbb{S}_{(I_i,R)}$, we refer to Orthogonal Procrustes problem (OP) in Section 2.3. Specifically, where $\mathbb{B}^{(i)}$ is the set $\mathbb{S}_{(I_i,R)}$, while all the remaining factors are unconstrained, we refer to the emerging tensor factorization problem as Unimodal Orthogonality Constrained Tensor Factorization (UOTF). A FISTA-type algorithm (Algorithm 4) is used to update factor $\mathbf{U}^{(i)}$, when $\mathbb{B}^{(i)}$ is the set of sparsity constraints $\mathbb{SP}_{s_i}$ (Section 2.4). As we explain in Section 2.4, for this case, the problem (3.26) can be rewritten as

$$\min_{\mathbf{U}^{(i)}} \frac{1}{2} \|\mathbf{X}_{(i)} - \mathbf{U}^{(i)} \mathbf{K}^{(i)}\|_F^2 + \lambda_i \|\mathbf{U}^{(i)}\|_1. \tag{3.30}$$

The parameters λ_i and s_i are closely related and regulate the sparsity level of the factor $\mathbf{U}^{(i)}$.

Algorithm 7: DTree-PARAFAC-ALS: A dimension tree-based PARAFAC-ALS algorithm, for GTF problems.

Input: $\boldsymbol{\mathcal{X}}$: An N-mode tensor,
R: The rank of PARAFAC decomposition,
constraints: Vector of factor constraints.

Output: $[\boldsymbol{\lambda}, \mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)}]$: Rank-$R$ PARAFAC decomposition of $\boldsymbol{\mathcal{X}}$.

$k = 0$
$\boldsymbol{\lambda} = \mathbf{1}$
$\mathcal{T}(\boldsymbol{\mathcal{X}}) = \text{Construct\_Dimension\_Tree}(\boldsymbol{\mathcal{X}})$
for $i = 1, \ldots, N$ **do**
  $\mathbf{W}^{(i)} = \mathbf{U}_k^{(i)T}\mathbf{U}_k^{(i)}$
end
repeat
  for $i = 1, \ldots, N$ **do**
    for $t \in \mathcal{T}(\boldsymbol{\mathcal{X}})$ **do**
      if $i \in \mu'(t)$ **then**
        $\text{Destroy}(\boldsymbol{\mathcal{X}}^{(t)})$
      end
    end
    $\mathbf{M}^{(i)} = \text{DTree\_TTV}(l_i)$
    $\mathbf{H}^{(i)} = \circledast_{j=1, j\neq i}^{N} \mathbf{W}^{(i)}$
    $\mathbf{U}_{k+1}^{(i)} = \text{Update\_Factor}(\mathbf{M}^{(i)}, \mathbf{H}^{(i)}, \mathbf{U}_k^{(i)}, constraints_i)$
    $\text{Normalize}(\mathbf{U}_{k+1}^{(i)}, \boldsymbol{\lambda})$
    $\mathbf{W}^{(i)} = \left(\mathbf{U}_{k+1}^{(i)}\right)^T \mathbf{U}_{k+1}^{(i)}$
  end
  $\text{Accelerate}(\mathbf{U}_k^{(1)}, \mathbf{U}_{k+1}^{(1)}, \ldots, \mathbf{U}_k^{(N)}, \mathbf{U}_{k+1}^{(N)})$
  $k = k + 1$
until *convergence is achieved or maximum number of iterations has been reached*;
return $[\boldsymbol{\lambda}, \mathbf{U}_k^{(1)}, \ldots, \mathbf{U}_k^{(N)}]$

Algorithm 8: The Update_Factor function

Input: $\mathbf{M}^{(i)}$: MTTKRP for factor i.

$\mathbf{H}^{(i)}$:

$\mathbf{U}_k^{(i)}$: Factor to be updated.

constraint: Constraint indicator.

Result: $\mathbf{U}_{k+1}^{(i)}$: The updated factor.

if *constraint = 'unconstrained'* **then**
  $\mathbf{U}_{k+1}^{(i)} = \mathbf{M}^{(i)}\mathbf{H}^{(i)^\dagger}$
else if *constraint = 'nonnegative'* **then**
  $\mathbf{U}_{k+1}^{(i)} = \text{Nesterov\_MNLS}(\mathbf{M}^{(i)}, \mathbf{H}^{(i)}, \mathbf{U}_k^{(i)})$
else if *constraint = 'orthogonal'* **then**
  $\mathbf{U}_{k+1}^{(i)} = \text{OP\_Update}(\mathbf{M}^{(i)}, \mathbf{H}^{(i)})$
else if *constraint = 'sparsity'* **then**
  $\mathbf{U}_{k+1}^{(i)} = \text{Fista\_l1}(\mathbf{M}^{(i)}, \mathbf{H}^{(i)}, \mathbf{U}_k^{(i)})$
return $\mathbf{U}_{k+1}^{(i)}$

Chapter 4

Parallel Implementations

In this chapter, we assume that we have at our disposal $p = \prod_{i=1}^{N} p_i$ processing elements and describe a parallel implementation of the AO GTF algorithm, which has been motivated by the medium-grained approach of [11].[1] The p processors form an n-dimensional Cartesian grid and are denoted as $p_{i_1,\ldots,i_N}$, with $i_j \in \mathbb{N}_{p_j}$ and $j \in \mathbb{N}_N$.

4.1 Message Passing Implementations

4.1.1 Variable partitioning and data allocation

In order to describe the parallel implementation, given a decomposition problem of a tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ into a set of factors $\mathbf{U}^{(i)} \in \mathbb{R}^{I_i \times R}$, for all $i \in \mathbb{N}_N$, we introduce certain partitionings of the factor matrices and tensor $\boldsymbol{\mathcal{X}}$. Specifically, we partition each factor matrix $\mathbf{U}^{(i)}$ into p_i block rows as

$$\mathbf{U}^{(i)} = \left[\begin{array}{ccc} \left(\mathbf{U}^{(i)^1}\right)^T & \cdots & \left(\mathbf{U}^{(i)^{p_i}}\right)^T \end{array} \right]^T, \tag{4.1}$$

with $\mathbf{U}^{(i)^j} \in \mathbb{R}^{\frac{I_i}{p_i} \times R}$, for all $j \in \mathbb{N}_{p_i}$. Additionally, we partition tensor $\boldsymbol{\mathcal{X}}$ into p subtensors as

$$\boldsymbol{\mathcal{X}}^{i_1,\ldots,i_N} = \boldsymbol{\mathcal{X}} \left((i_1 - 1)\frac{I_1}{p_1} + 1 : i_1 \frac{I_1}{p_1}, \ldots, (i_N - 1)\frac{I_N}{p_N} + 1 : i_N \frac{I_N}{p_N} \right) \in \mathbb{R}^{\frac{I_1}{p_1} \times \cdots \times \frac{I_N}{p_N}}, \tag{4.2}$$

where $i_j \in \mathbb{N}_{p_j}$ and $j \in \mathbb{N}_N$.

From the perspective of the processors, processor $p_{i_1,\ldots,i_N}$ receives subtensor $\boldsymbol{\mathcal{X}}^{i_1,\ldots,i_N}$ and contributes into updating the i_j-th part of factor $\mathbf{U}^{(j)}$, $\mathbf{U}^{(j)^{i_j}}$, for all $j \in \mathbb{N}_N$. At last, we assume that, at the end of the k-th outer AO iteration,

[1] We note that both the single-core and the multi-core implementations solve the same problem, thus problems that are identifiable in single-core environments remain identifiable in multi-core environments and the solutions, in both cases, are practically the same.

(a) processor $p_{i_1,\ldots,i_N}$ knows $\mathbf{U}_k^{(1)^{i_1}}$, $\mathbf{U}_k^{(2)^{i_2}}$, $\ldots$, $\mathbf{U}_k^{(N)^{i_N}}$

(b) *all* processors know $\left(\mathbf{U}_k^{(i)}\right)^T \mathbf{U}_k^{(i)}$, for all $i \in \mathbb{N}_N$.

4.1.2 Communication groups

We define certain communication groups, also known as communicators [28], over subsets of the p processors, which are used for the efficient collaborative implementation of specific computational tasks, as explained in detail below.

First, we define as $\mathcal{C}_{i,j}$, for $i \in \mathbb{N}_N$ and $j \in \mathbb{N}_{p_i}$, to be the $(N-1)$–dimensional group of processors, involving the $\prod_{k=1,k\neq i}^{N} p_k$ processors having the i-th index equal to j, which are used for the collaborative update of $\mathbf{U}_k^{(i)^j}$. Additionally, we define as $\mathcal{D}_{i,\mathcal{J}^i}$, for $i \in \mathbb{N}_N$ and $\mathcal{J}^i \in \underset{n=1,n\neq i}{\times} \mathcal{S}^n$, with $\mathcal{S}^n = \mathbb{N}_{p_n}$, to be the one-dimensional processor groups, each involving the p_i processors that differ only at the i-th index. Each of these groups is used for the collaborative computation of $\left(\mathbf{U}_{k+1}^{(j)}\right)^T \mathbf{U}_{k+1}^{(j)}$, for $j \in \mathbb{N}_N$.

4.2 Parallel implementation of AO GTF

4.2.1 Factor update implementation

In this section, we consider the case of updating the factor matrix $\mathbf{U}_k^{(1)}$. In order to facilitate our analysis, we introduce the following partitioning of a mode-1 matricization of the tensor $\boldsymbol{\mathcal{X}}$ as

$$\mathbf{X}_{(1)} = \left[\begin{array}{ccc} \left(\mathbf{X}_{(1)}^1\right)^T & \cdots & \left(\mathbf{X}_{(n)}^{p_1}\right)^T \end{array} \right]^T, \tag{4.3}$$

with $\mathbf{X}_{(1)}^j \in \mathbb{R}^{\frac{I_1}{p_1} \times \prod_{n=2}^{N} I_n}$, for all $j \in \mathbb{N}_{p_1}$. We describe in detail the update of $\mathbf{U}_k^{(1)}$, which is achieved via the parallel updates of $\mathbf{U}_k^{(1)^{i_1}}$, for all $i_1 \in \mathbb{N}_{p_1}$, and consists of the following stages:

1. Processors in $\mathcal{C}_{1,i_1}$, for all $i_1 \in \mathbb{N}_{p_1}$, collaboratively compute the $\frac{I_1}{p_1} \times R$ matrix

$$\mathbf{M}^{(1)^{i_1}} = \mathbf{X}_{(1)}^{i_1} \mathbf{K}^{(1)}, \tag{4.4}$$

and the result is scattered among the processors in the group; thus, each processor in the group receives $\frac{I}{\prod_{i=1}^{N} p_i}$ successive rows of $\mathbf{M}^{(1)^{i_1}}$. Term $\mathbf{M}^{(1)^{i_1}}$ can be computed collaboratively because

$$\mathbf{X}_{(1)}^{i_1} \mathbf{K}^{(1)} = \sum_{i_2=1}^{p_2} \cdots \sum_{i_N=1}^{p_N} \mathbf{X}_{(1)}^{i_1,\ldots,i_N} (\mathbf{U}_k^{(N)^{i_N}} \odot \cdots \odot \mathbf{U}_k^{(2)^{i_2}}), \tag{4.5}$$

where $\mathbf{X}_{(1)}^{i_1,\ldots,i_N}$ is the matricization of $\boldsymbol{\mathcal{X}}^{i_1,\ldots,i_N}$ with respect to the first mode. Each processor $p_{i_1,\ldots,i_n}$ in $\mathcal{C}_{1,i_1}$ knows $\mathbf{X}_{(1)}^{i_1,\ldots,i_N}$, $\mathbf{U}_k^{(2)^{i_2}}$, $\ldots$, $\mathbf{U}_k^{(N)^{i_N}}$ and computes the corresponding term of (4.5). The sum is computed and scattered among all processors in $\mathcal{C}_{1,i_1}$ via a `reduce-scatter` operation.

2. Each processor in the group $\mathcal{C}_{1,i_1}$ uses the scattered part of $\mathbf{M}^{(1)^{i_1}}$, matrix $\mathbf{K}^{(1)}$ and partial factor $\mathbf{U}_k^{(1)^{i_1}}$, in order to compute the updated part of $\mathbf{U}_{k+1}^{(1)^{i_1}}$, via the `update factor` algorithm.

3. The updated parts of $\mathbf{U}_{k+1}^{(1)^{i_1}}$ are all-gathered at the processors of the group $\mathcal{C}_{1,i_1}$, so that *all* processors in the group learn the updated factor $\mathbf{U}_{k+1}^{(1)^{i_1}}$.

4. By applying an `all-reduce` operation to $\left(\mathbf{U}_{k+1}^{(1)^{i_1}}\right)^T \mathbf{U}_{k+1}^{(1)^{i_1}}$, for all $i_1 \in \mathbb{N}_{p_1}$, on each of the single-dimensional processor groups $\mathcal{D}_{1,\mathcal{J}^1}$, *all* p processors learn $\mathbf{U}_{k+1}^{(1)^T}\mathbf{U}_{k+1}^{(1)}$.[2]

As for the rest of the factors, the updates can be implemented by following analogous steps.

The Euclidean norms of the columns of each factor $\mathbf{U}_k^{(i)}$ appear on the diagonals of $\mathbf{U}_k^{(i)^T}\mathbf{U}_k^{(i)}$, which is known to all processors. Thus, no communication is necessary for the normalization of the updated matrix factors.

After the completion of the $(k+1)$-*st* AO iteration, processor $p_{i_1,\ldots,i_N}$ knows the parts of the updated and normalized factors, that is, $\left\{\mathbf{U}_{k+1}^{(j)^{i_j}}\right\}_{j=1}^{N}$, as well as $\left\{\mathbf{U}_k^{(j)^{i_j}}\right\}_{j=1}^{N}$. It is now able to compute $\left\{\mathbf{U}_{new}^{(j)^{i_j}}\right\}_{j=1}^{N}$, (see (3.25)). The computation of the cost function $f_{\boldsymbol{\mathcal{X}}}$ at points $\left\{\mathbf{U}_{k+1}^{(j)}\right\}_{j=1}^{N}$ and $\left\{\mathbf{U}_{new}^{(j)}\right\}_{j=1}^{N}$ is implemented collaboratively. Each processing element computes its local contribution and, via an all-reduce operation over the whole processor grid, the values of the cost function are computed and become known to all processors, thus, all processors make the same decision regarding the success or failure of the acceleration step.

4.2.2 Communication cost

We focus on the parallel updates of $\mathbf{U}_k^{(1)^{i_1}}$, for all $i_1 \in \mathbb{N}_{p_1}$, and present results concerning the associated communication cost. Analogous results hold for the updates of the remaining factor matrices $\mathbf{U}_k^{(j)^{i_j}}$, for all $j \in \{2,\ldots,N\}$ and $i_j \in \mathbb{N}_{p_j}$.

[2]In the cases where $R \gtrapprox \frac{I_1}{p_1}$ it seems preferable to compute $\left(\mathbf{U}_{k+1}^{(1)}\right)^T \left(\mathbf{U}_{k+1}^{(1)}\right)$ via an all-gather operation on terms $\mathbf{U}_{k+1}^{(1)^{i_1}}$, for all $i_1 \in \mathbb{N}_{p_1}$, on each of the single-dimensional processor groups $\mathcal{D}_{1,\mathcal{J}^1}$. However, in this work, we mainly focus on small-rank factorizations, thus, in our communication cost analysis and experiments we do not present results for this alternative.

We assume that an m-word message is transferred from one process to another with communication cost $t_s + t_w m$, where t_s is the latency, or startup time for the data transfer, and t_w is the word transfer time [28].

Communication occurs at three algorithm execution points.

1. The $\frac{I_1}{p_1} \times R$ matrix $\mathbf{M}^{1^{i_1}}$ is computed and scattered among the $\prod_{j=2}^{N} p_j$ processors of group $\mathcal{C}_{1,i_1}$, using a reduce-scatter operation, with communication cost [28, §4.2]

$$\mathcal{C}ost_1^{\mathbf{1}} = t_s \left(\sum_{j=2}^{N} p_j - (N-1) \right) + t_w \frac{I_1 R}{p} \left(\prod_{j=2}^{N} p_j - 1 \right).$$

2. Processors in $\mathcal{C}_{1,i_1}$ learn the updated $\mathbf{U}_{k+1}^{(1)^{i_1}}$ through an all-gather operation on its updated parts, each of dimension $\frac{I}{p} \times R$, with communication cost [28, §4.2]

$$\mathcal{C}ost_2^{\mathbf{1}} = t_s \left(\sum_{j=2}^{N} p_j - (N-1) \right) + t_w \frac{I_1 R}{p} \left(\prod_{j=2}^{N} p_j - 1 \right).$$

3. Finally, $\left(\mathbf{U}_{k+1}^{(1)}\right)^T \mathbf{U}_{k+1}^{(1)}$ is computed by using an all-reduce operation on quantities $\left(\mathbf{U}_{k+1}^{(1)^{i_1}}\right)^T \mathbf{U}_{k+1}^{(1)^{i_1}}$, for all $i_1 \in \mathbb{N}_{p_1}$, on each single-dimensional processor group $\mathcal{D}_{1,\mathcal{J}^1}$, with communication cost [28, §4.3]

$$\mathcal{C}ost_3^{\mathbf{1}} = \left(t_s + t_w R^2\right) \log_2 p_1. \tag{4.6}$$

The communication that takes place during the acceleration step involves scalar quantities and, thus, is ignored.

When we are dealing with large messages, the t_w terms dominate the communication cost. Thus, if we ignore the startup time, the total communication time is

$$\begin{aligned} \mathcal{C}^{\mathbf{1}} &= t_w \left(\frac{2 I_1 R}{p} \left(\prod_{j=2}^{N} p_j - 1 \right) + R^2 \log_2 p_1 \right) \\ &\approx t_w \left(\frac{2 I_1 R}{p_1} + R^2 \log_2 p_1 \right) \\ &\approx \frac{2 I_1 R\, t_w}{p_1}, \end{aligned} \tag{4.7}$$

with the second approximation being accurate for $R \ll \frac{I_1}{p_1}$. The presence of p_1 in the denominator of the last expression of (4.7) implies that our implementation is scalable in the sense that, if we double I_1, then we can have (approximately) the same communication cost per processor by doubling p_1. Again, analogous results hold for the updates of the remaining factor matrices $\mathbf{U}_k^{(j)^{i_j}}$, for all $j = \{2, \ldots, N\}$ and $i_j \in \mathbb{N}_{p_j}$.

4.2.3 The case of Orthogonality Constraints

The update of a matrix factor $\mathbf{U}_k^{(j)}$ under orthogonality constraints deviates from the procedure presented in the previous section. However, it can be achieved via the parallel updates of $\mathbf{U}_k^{(j)^{i_j}}$, for all $i_j \in \mathbb{N}_{p_j}$, and consists of the following stages:

1. Processors in $\mathcal{C}_{j,\mathcal{J}^j}$, for all $i_j \in \mathbb{N}_{p_j}$, collaboratively compute the $\frac{I_j}{p_j} \times R$ matrix

$$\mathbf{M}^{(j)^{i_j}} = \mathbf{X}_{(j)}^{i_j} \mathbf{K}^{(j)}, \tag{4.8}$$

   by applying an all-reduce operation, since

$$\mathbf{X}_{(j)}^{i_j} \mathbf{K}^{(j)} = \sum_{\mathcal{C}_{j,i_j}} \mathbf{X}_{(j)}^{i_1,\ldots,i_N} \left(\mathbf{U}_k^{(N)^{i_N}} \odot \cdots \odot \mathbf{U}_k^{(j+1)^{i_{j+1}}} \odot \mathbf{U}_{k+1}^{(j-1)^{i_{j-1}}} \odot \cdots \odot \mathbf{U}_{k+1}^{(1)^{i_1}} \right),$$

   where $\mathbf{X}_{(j)}^{i_1,\ldots,i_N}$ is the matricization of $\boldsymbol{\mathcal{X}}^{i_1,\ldots,i_N}$, with respect to the j-th mode.

2. Processors in $\mathcal{D}_{j,\mathcal{J}^j}$ collaboratively compute the $R \times R$ matrix

$$\left(\mathbf{M}^{(j)}\right)^T \mathbf{M}^{(j)} = \sum_{i_j=1}^{p_j} \left(\mathbf{M}^{(j)^{i_j}}\right)^T \mathbf{M}^{(j)^{i_j}}, \tag{4.9}$$

   by applying an all-reduce operation. We notice that at the end of this step, *all* p processors know matrix $\left(\mathbf{M}^{(j)}\right)^T \mathbf{M}^{(j)}$.

3. Each processor $p_{i_1,\ldots,i_N}$, for all $i_n \in \mathbb{N}_{p_n}$ and $n \in \mathbb{N}_N$, computes the updated partial factor $\mathbf{U}_{k+1}^{(j)^{i_j}}$ as

$$\mathbf{U}_{k+1}^{(j)^{i_j}} = \mathbf{M}^{(j)^{i_j}} \left(\left(\mathbf{M}^{(j)}\right)^T \mathbf{M}^{(j)} \right)^{-\frac{1}{2}}. \tag{4.10}$$

4.2.4 Communication Cost

As for the communication costs of the procedure needed for updating a factor with orthogonality constraints in parallel, since communication occurs at two algorithm execution points, we have

1. The $\frac{I_j}{p_j} \times R$ matrix $\mathbf{M}^{(j)^{i_j}}$ is computed among the $\prod_{k=1,k\neq j}^{N} p_k$ processors of group $\mathcal{C}_{j,i_j}$, using an all-reduce operation, with communication cost [28, §4.2]

$$\mathcal{C}ost_1^{\mathbf{j}} = \left(t_s + t_w \frac{I_j}{p_j} R \right) \log_2 \left(\prod_{k=1,k\neq j}^{N} p_k \right).$$

2. Matrix $\left(\mathbf{M}^{(j)}\right)^T \mathbf{M}^{(j)}$ is computed by using an all-reduce operation on $\left(\mathbf{M}^{(j)^{i_j}}\right)^T \mathbf{M}^{(j)^{i_j}}$ within each single-dimensional processor group $\mathcal{D}_{j,\mathcal{J}^j}$, with communication cost [28, §4.3]

$$\mathcal{C}_2^{\mathbf{j}} = \left(t_s + t_w R^2\right) \log_2 p_j. \tag{4.11}$$

When we are dealing with large messages, the t_w terms dominate the communication cost. Thus, if we ignore the startup time, the total communication time, for updating factor $\mathbf{U}_k^{(j)}$, is

$$\begin{aligned} \mathcal{C}^{\mathbf{j}} &= t_w \left(\frac{I_j R}{p_j} \log_2 \left(\prod_{k=1, k\neq j}^{N} p_k \right) + R^2 \log_2 p_j \right) \\ &\approx t_w \left(\frac{I_j R}{p_j} \log_2 \left(\prod_{k=1, k\neq j}^{N} p_k \right) \right) \end{aligned} \tag{4.12}$$

with the approximation being accurate for $R \ll \frac{I_j}{p_j}$. We again observe that our implementation is scalable in the above mentioned sense.

Chapter 5

Cuda Implementations

5.1 Implementations on Heterogeneous Platforms using CUDA

5.1.1 A short introduction to Heterogeneous Platforms

A modern heterogeneous computational node consists of multicore CPU sockets and hardware accelerators such as GPUs or even FPGAs. The most common hardware accelerator are GPUs, used to accelerate the execution of a program section. Before analyzing the architecture of a Heterogeneous Platform, we have to focus on each device's architecture and characteristics. CPUs and GPUs do not share the same architecture. In general, CPUs are classified as *Single Instruction Single Data* (SISD) when a single core executes a single instruction stream on single data and *Single Instruction Multiple Data* (SIMD) if a single core executes one instruction stream which operates on multiple data. Modern CPUs with multiple cores are classified as *Multiple Instructions Multiple Data* (MIMD), where each core executes an independent instruction which performs on single or on multiple data.

In contrast, GPUs are classified as *Single Instruction Multiple Threads* (SIMT), a combination of SIMD and multithreading execution model. Furthermore, CPUs and GPUs have different type of cores. Modern CPUs have multiple large cores, where each core is designed with a complicated control and is optimized to execute sequential tasks. On the contrary, GPUs have many cores (often hundreds or even thousands) smaller in size, with a simpler control and are ideal for data-parallel tasks. Also, GPU cores can handle many more threads than the CPU ones due to the fact that they are extremely light-weight and are organized in groups making them more easy to schedule.

A GPU, often called device, operates in conjunction with a CPU, also referred as host. In a program implemented on Heterogeneous Platforms, the host is responsible for data initialization, synchronization across all devices, small data size computations etc. On the

other hand, devices are responsible for large data size computations, where parallelism can be highly exploited.

5.1.2 Available Frameworks for Heterogeneous Platforms

There are many frameworks for writing programs that execute across heterogeneous platforms, but the most common are OpenCL and CUDA (Compute Unified Device Architecture). In both platforms, a program consists of two parts, the host code, which runs on CPU and device code, also referred to as "kernel," which runs on GPU.

OpenCL is an open standard that runs on any hardware type and by multiple vendors, which is a huge advantage compared to CUDA, which is developed and compatible only to NVIDIA GPUs. On the other hand, CUDA promises speedup on runtime performance, due to the fact that kernels are compiled once and not at runtime execution. So the generated code will be optimized to the target GPU, exploiting its unique features.

CUDA can be described shortly as a general-purpose parallel computing platform and application programming interface (API). Using CUDA, one can access the GPU for computation, likewise it has been done on the CPU. This platform is designed to work with plenty of programming languages, such as C, C++, Python, Fortran etc. It comes also with plenty of math libraries like CUBLAS (Basic Linear Algebra Subroutines), CUFFT (Fast Fourier Transform) and many more, increasing performance especially when operating on large data.

The aforementioned advantages of CUDA over OpenCL, led us to choose CUDA for our implementation.

Chapter 6

Numerical experiments

6.1 NTF Problem with third-order tensors

6.1.1 Matlab environment

In this subsection, we test the effectiveness of the Nesterov-based AO NTF algorithm with numerical experiments performed in Matlab.

At first, we compare the performance of the algorithm we propose in Algorithm 2 for the solution of the MNLS problem (2.10) with that of the algorithm proposed in [15] and [16] (for the moment, we ignore the proximal term, thus, we put $\lambda = 0$ in Algorithm 2). As we mentioned in subsection 2.2.2, if matrix $\mathbf{B}^T\mathbf{B}$ is rank deficient, that is, if $\mu = 0$, then both algorithms have practically the same behavior. However, if $\mathbf{B}^T\mathbf{B}$ is full-rank, then the two algorithms exhibit different behavior. In order to illustrate their difference, we perform the following experiment. We generate random matrices $\mathbf{X} \in \mathbb{R}^{m\times n}$ and $\mathbf{B} \in \mathbb{R}^{n\times r}$ with $m = 300$, $n = 200$, and $r = 100$, with independent and identically distributed (i.i.d) elements, taking values uniformly at random in the interval $[0, 1]$. Then, we solve problem (2.10) with the two algorithms, starting from the same random point. The terminating conditions are determined by parameters $\delta_1 = \delta_2 = 10^{-3}$. In Figure 6.1, we plot the number of iterations needed by the two algorithms to converge over 100 independent realizations. We observe that the Nesterov-type algorithm which exploits strong convexity is much more efficient than the algorithm which does not. Thus, in the sequel, we shall not present performance results involving the algorithm of [15].

Next, we compare the performance of a Matlab implementation of the proposed non-negative tensor factorization algorithm with routines `parafac` of the n-way toolbox [25] and `sdf_nls` of tensorlab [29]. Our aim is to provide some general observations about the difficulty of the problems and the behavior of the algorithms and not a strict ranking of the

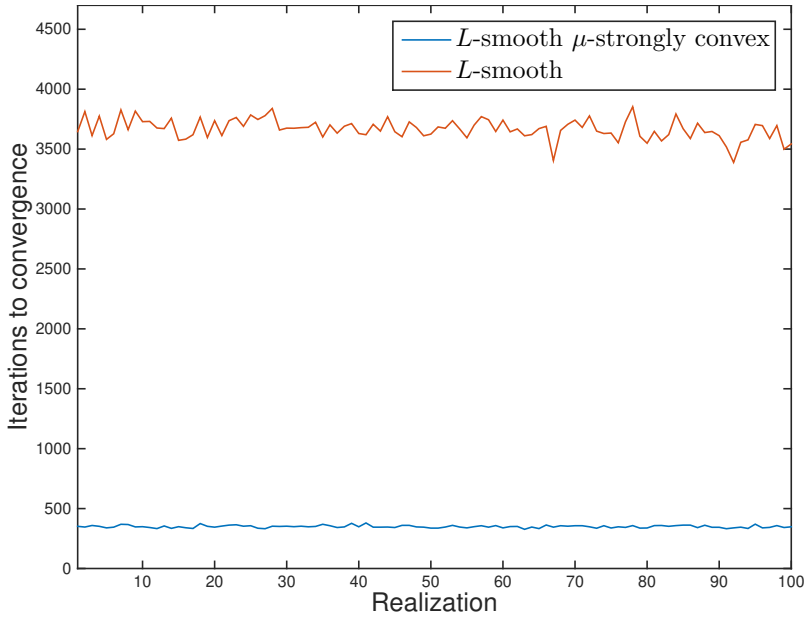

Figure 6.1: Number of iterations to convergence for (blue line) Algorithm 2, with $\lambda = 0$, and (red line) algorithm of [15].

algorithms.[1]

The `parafac` routine essentially implements an AO NTF algorithm, where each MNLS problem is solved via the function `fastnnls`, which is based on [30, §23.3]. It also incorporates the normalization and acceleration schemes briefly described in Section **??**. The `sdf_nls` routine for NTF first applies a "squaring" transformation to the problem variables [31] and then solves an unconstrained problem via an AOO-based Gauss-Newton method.

In our experiments with synthetic data, we focus on the `cputime` and the Maximum, over the three latent factors, Relative Factor Error (MRFE), which is computed via function `cpd_err` of tensorlab.

In the numerical experiments we present in this subsection, we choose the parameter values that determine the terminating conditions so that all algorithms achieve (approximately) the same average MRFEs (of course, this is not always possible with one set of parameter values). Thus, we set Tol $= 10^{-5}$ for `parafac`, TolFun $= 10^{-9}$ for `sdf_nls`, and δ_1 and δ_2, which determine the terminating conditions for the Nesterov-based MNLS, are set to $\delta_1 = \delta_2 = 10^{-2}$. The outer iterations of the Nesterov-based AO NTF terminate if the relative

[1]For our experiments, we run Matlab 2014a on a MacBook Pro with a 2.5 GHz Intel Core i7 Intel processor and 16 GB RAM.

changes of the normalized latent factors become sufficiently small, that is,

$$\frac{\|\mathbf{M}_{k+1}^{\mathcal{N}} - \mathbf{M}_{k}^{\mathcal{N}}\|_F}{\|\mathbf{M}_{k}^{\mathcal{N}}\|_F} < \text{tol}_{\text{AO}}, \text{ for } \mathbf{M} = \mathbf{A}, \mathbf{B}, \mathbf{C}, \tag{6.1}$$

where $\text{tol}_{\text{AO}} = 10^{-4}$.

The proximal parameter λ is computed as

$$\lambda := g(L, \mu) = \begin{cases} 10\mu, & \text{if } \frac{L}{\mu} > 10^6, \\ \mu, & \text{if } 10^6 > \frac{L}{\mu} > 10^4, \\ \frac{\mu}{10}, & \text{if } 10^4 > \frac{L}{\mu}. \end{cases} \tag{6.2}$$

All algorithms start from the same triple of random matrices, $(\mathbf{A}_0, \mathbf{B}_0, \mathbf{C}_0)$, which have i.i.d. elements, uniformly distributed in $[0, 1]$.

True latent factors with i.i.d. elements

We start with synthetic data by assuming that the true latent factors consist of i.i.d. elements, uniformly distributed in $[0, 1]$. The additive noise is zero-mean white Gaussian with variance σ_N^2.

In Table 6.3, we present the average, over 10 realizations, `cputime` and MRFE for various tensor "shapes," ranks $R = 15, 50$, and noise variances $\sigma_N^2 = 10^{-2}, 10^{-4}$. We observe that the Nesterov-based AO NTF is very competitive in all cases, in the sense that it converges fast, achieving very good accuracy in most of the cases.

True latent factors with correlated elements

It is well-known that, if some columns of (at least) one latent factor are almost collinear, convergence of the AO algorithm tends to be slow (these cases are known as "bottlenecks") [23]. In the sequel, we test the behavior of the three algorithms in cases with one, two, and three bottlenecks. More specifically, we generate the true latent factors with i.i.d. elements as before and we create a single "bottleneck" by modifying the last two columns of one latent factor so that each becomes highly correlated with another column of the same latent factor (the correlation coefficient is larger that 0.98). In an analogous way, we generate double and triple "bottlenecks."

In Table 6.2, we focus on the case $I = J = K = 300$, $R = 50$, $\sigma_N^2 = 10^{-4}$, and present the average, over 10 realizations, `cputime` and MRFE. We observe that the problems become more difficult as the number of bottlenecks increases, in the sense that both the `cputime` and the MRFE increase as the number of bottlenecks increases. Again, the Nesterov-based AO NTF algorithm is very efficient in all cases. Analogous observations have been made in extensive numerical experiments with other tensor shapes and noise levels.

Table 6.1: Average, over 10 realizations, `cputime` and maximum relative factor error for Nesterov-based AO NTF, `sdf_nls`, and `parafac`, for true latent factors with i.i.d. entries, uniform in $[0, 1]$

Size	R	σ_N^2	AO-Nesterov		`sdf_nls`		`parafac`	
			`cputime`	`MRFE` $\times 10^4$	`cputime`	`MRFE` $\times 10^4$	`cputime`	`MRFE` $\times 10^4$
$1000 \times 100 \times 100$	15	10^{-2}	29	80	56	79	44	85
		10^{-4}	27	10	52	13	53	8
	50	10^{-2}	77	89	217	91	191	91
		10^{-4}	76	13	221	24	251	9
$500 \times 500 \times 100$	15	10^{-2}	63	35	126	37	72	42
		10^{-4}	64	5	132	10	105	4
	50	10^{-2}	119	39	347	43	250	42
		10^{-4}	124	8	331	20	327	5
$300 \times 300 \times 300$	15	10^{-2}	72	27	84	27	70	38
		10^{-4}	71	5	87	7	106	3
	50	10^{-2}	114	31	171	32	230	34
		10^{-4}	119	8	174	13	279	4

Table 6.2: Average, over 10 realizations, `cputime` and maximum relative factor error for Nesterov-based AO NTF, `sdf_nls`, and `parafac`, for true latent factors with correlated entries

Size	R	σ_N^2	Bottleneck	AO-Nesterov		`sdf_nls`		`parafac`	
				`cputime`	`MRFE`	`cputime`	`MRFE`	`cputime`	`MRFE`
$300 \times 300 \times 300$	50	10^{-4}	$\mathbf{A}$	132	0.0074	194	0.0075	356	0.0073
			$\mathbf{A}, \mathbf{B}$	204	0.0116	254	0.0195	412	0.0122
			$\mathbf{A}, \mathbf{B}, \mathbf{C}$	271	0.0206	370	0.1007	779	0.0168

6.1.2 Parallel environment - MPI

We now present results obtained from the MPI implementation described in detail in Section 4.2, for nonnegative tensor factorization of third-order tensors. The program is executed on a DELL PowerEdge R820 system with SandyBridge - Intel(R) Xeon(R) CPU E5 − 4650v2 (in total, 16 nodes with 40 cores each at 2.4 Gz) and 512 GB RAM per node. The matrix operations are implemented using routines of the C++ library Eigen (matrix module) [32]. We assume a noiseless tensor $\boldsymbol{\mathcal{X}}$, whose true latent factors have i.i.d elements, uniformly distributed in $[0, 1]$. The terminating conditions for MNLS are determined by values $\delta_1 = \delta_2 = 10^{-2}$.

The AO terminates at iteration k if (recall that tensor $\boldsymbol{\mathcal{X}}$ is noiseless)

$$\texttt{RFE}(\mathbf{A}_k, \mathbf{B}_k, \mathbf{C}_k) < 10^{-3}.$$

We test the behavior of our implementation for various tensor sizes and rank $R = 15, 50, 100$. The performance metric we compute is the speedup attained using $p = p_{\mathbf{A}} \times p_{\mathbf{B}} \times p_{\mathbf{C}}$ processors.

In Figures 6.2-6.4, we plot the speedup for the following cases (in all cases with synthetic data, the tensor $\boldsymbol{\mathcal{X}}$ has eight billion entries):

1. Cubic tensor: we set $I = J = K = 2000$ and implement the algorithm on a grid with $p_{\mathbf{A}} = p_{\mathbf{B}} = p_{\mathbf{C}} = \sqrt[3]{p}$, for $p = 1, 8, 27, 64, 125, 216, 343, 512$.

2. One large dimension: we set $I = 400$, $J = 400$, $K = 50000$ and implement the algorithm on a grid with $p_{\mathbf{A}} = p_{\mathbf{B}} = 1$, $p_{\mathbf{C}} = p$, for $p = 1, 8, 27, 64, 125, 216, 343, 512$.

3. Two large dimensions: we set $I = 5000$, $J = 320$, $K = 5000$ and implement the algorithm on a grid with $p_{\mathbf{A}} = p_{\mathbf{C}} = \sqrt{p}$, $p_{\mathbf{B}} = 1$, for $p = 1, 9, 36, 64, 121, 225, 361, 529$.

We observe that, in all cases, we attain significant speedup, which is rather insensitive to the tensor shape and rank.

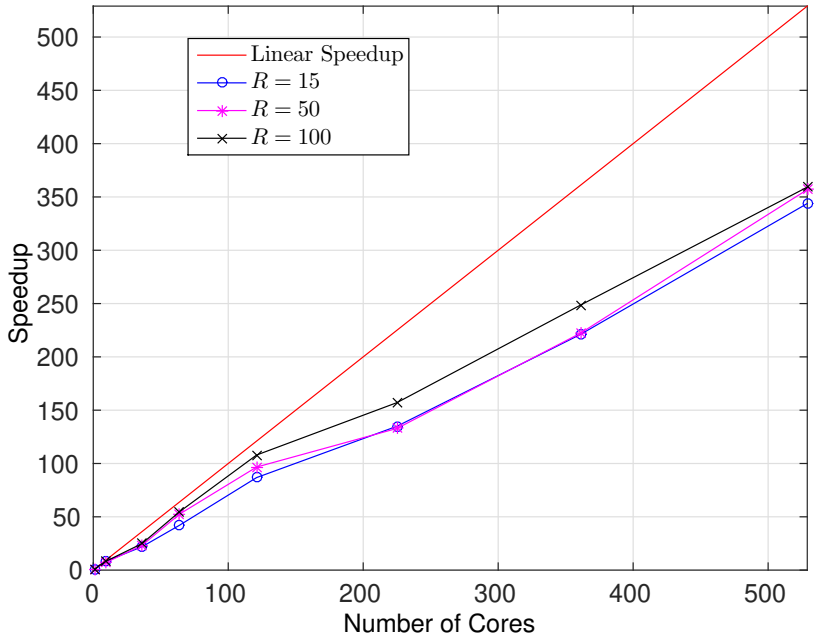

Figure 6.2: Speedup achieved for a $2000 \times 2000 \times 2000$ tensor with p cores, for $p = 1, 8, 27, 64, 125, 216, 343, 512$.

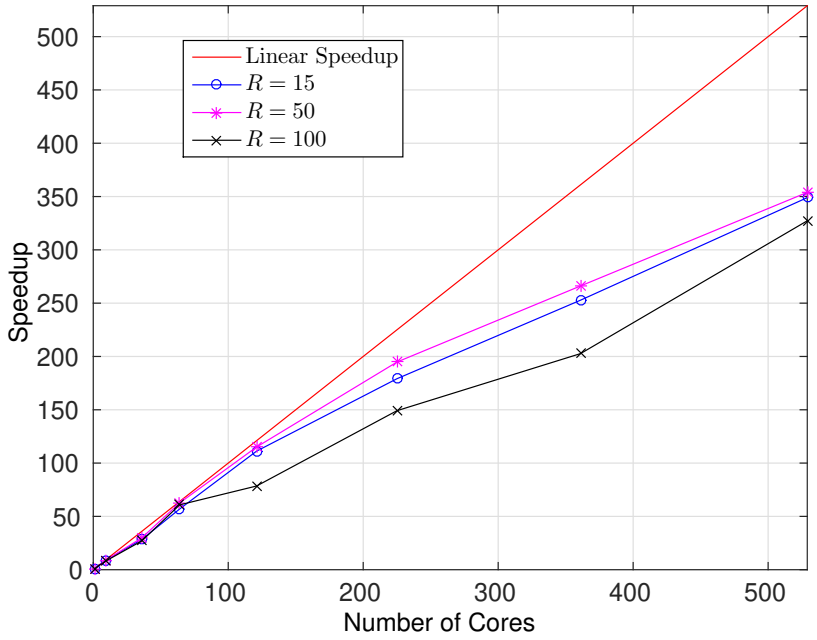

Figure 6.3: Speedup achieved for a $400 \times 400 \times 50000$ tensor with p cores, for $p = 1, 8, 27, 64, 125, 216, 343, 512$.

6.2 Numerical UOTF

6.2.1 Numerical experiments

In this section, we present results obtained from the MPI implementation described in detail in Section 4.2 and subsection 4.2.3. The program is executed on a DELL PowerEdge R820 system with SandyBridge - Intel(R) Xeon(R) CPU E5 − 4650v2 (in total, 16 nodes with 40 cores each at 2.4 Gz) and 512 GB RAM per node. The matrix operations are implemented using routines of the C++ library Eigen (matrix module) [32]. We assume a noiseless tensor $\boldsymbol{\mathcal{X}}$, whose true latent factors $\mathbf{A}^o$ and $\mathbf{C}^o$ have i.i.d elements, uniformly distributed in $[0, 1]$, while true latent factor $\mathbf{B}^o$ was produced from the left singular vectors of a matrix with i.i.d elements, uniformly distributed in $[0, 1]$.

The AO terminates at iteration k if

$$\texttt{RFE}(\mathbf{A}_k, \mathbf{B}_k, \mathbf{C}_k) < 10^{-3},$$

where

$$\text{RFE}\left(\mathbf{A}, \mathbf{B}, \mathbf{C}\right) := \frac{\|\boldsymbol{\mathcal{X}} - [\![\mathbf{A}, \mathbf{B}, \mathbf{C}]\!]\|_F}{\|\boldsymbol{\mathcal{X}}\|_F}. \tag{6.3}$$

We test the behavior of our implementation for various tensor sizes and rank $R = 15, 50, 100$. The performance metric we compute is the speedup attained using $p = p_{\mathbf{A}} \times p_{\mathbf{B}} \times p_{\mathbf{C}}$ processors.

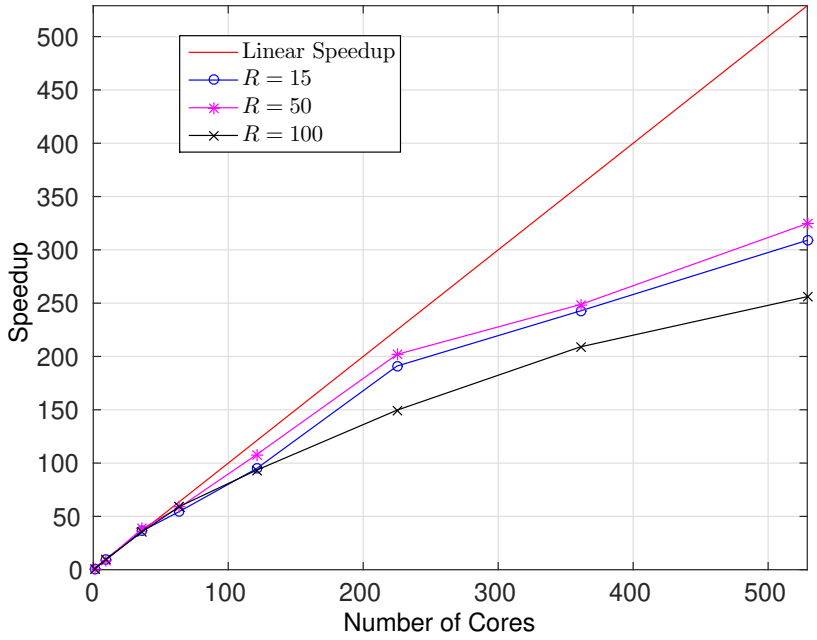

Figure 6.4: Speedup achieved for a $5000 \times 320 \times 5000$ tensor with p cores, for $p = 1, 9, 36, 64, 121, 225, 361, 529$.

In Figures 6.5–6.7, we plot the speedup for the following cases[2]:

1. Cubic tensor: we set $I = J = K = 2000$ and implement the algorithm on a grid with $p_{\mathbf{A}} = p_{\mathbf{B}} = p_{\mathbf{C}} = \sqrt[3]{p}$, for $p = 1, 8, 27, 64, 125, 216, 343$.

2. Two large dimensions: we set $I = 5000$, $J = 320$, $K = 5000$ and implement the algorithm on a grid with $p_{\mathbf{A}} = p_{\mathbf{C}} = \sqrt{p}$, $p_{\mathbf{B}} = 1$, for $p = 1, 4, 9, 36, 64, 121, 225, 361$.

3. One large dimension: we set $I = 400$, $J = 50000$, $K = 400$ and implement the algorithm on a grid with $p_{\mathbf{A}} = p_{\mathbf{C}} = 1$, $p_{\mathbf{B}} = p$, for $p = 1, 8, 27, 64, 125, 216, 343$.

In order to highlight the need of parallel processing for the decomposition of very large tensors, we quote the serial execution times ($p = 1$) in Table 6.3. We observe that, in all cases, we attain significant speedup, which is rather insensitive to the tensor shape and rank.

[2]To the best of our knowledge, there is no other parallel algorithm solving the UOTF problem, thus, we cannot compare with any competing state-of-the art algorithm.

Table 6.3: Execution times for $p = 1$ over different tensor sizes and ranks

Size	R	Time of Execution (sec)
$2000 \times 2000 \times 2000$	15	$6,272.35$
	50	14744.09
	100	34381.45
$5000 \times 320 \times 5000$	15	$7,545.13$
	50	17908.56
	100	42628.99
$400 \times 5000 \times 400$	15	$6,434.69$
	50	$17,790.75$
	100	$36,599.60$

6.3 Solving the NTF problem with Eigen's Tensor module

6.3.1 Parallel environment - MPI

We now present results obtained from the MPI implementation described in detail in Section 4.2. The program is executed on a DELL PowerEdge R820 system with SandyBridge - Intel(R) Xeon(R) CPU E5 − 4650v2 (in total, 16 nodes with 40 cores each at 2.4 Gz) and 490 GB RAM per node. The tensor and matrix operations are implemented using routines of the C++ library Eigen's *unsupported tensor module* [32]. We assume a noiseless tensor $\boldsymbol{\mathcal{X}}$, whose true latent factors have i.i.d elements, uniformly distributed in $[0, 1]$. The terminating conditions for MNLS are determined by values $\delta_1 = \delta_2 = 10^{-2}$. In addition, we set the limit for the number of AO iterations to 10, and for the Nesterov algorithm iterations to 50.

The AO terminates at iteration k if (recall that tensor $\boldsymbol{\mathcal{X}}$ is noiseless)

$$\texttt{RFE}(\mathbf{U}_k^{(1)}, \mathbf{U}_k^{(2)}, \ldots, \mathbf{U}_k^{(N)}) < 10^{-3}, \tag{6.4}$$

where

$$\texttt{RFE}(\mathbf{U}^{(1)}, \mathbf{U}^{(2)}, \ldots, \mathbf{U}^{(N)}) := \frac{\left\|\boldsymbol{\mathcal{X}} - [\![\mathbf{U}^{(1)}, \mathbf{U}^{(2)}, \ldots, \mathbf{U}^{(N)}]\!]\right\|_F}{\|\boldsymbol{\mathcal{X}}\|_F} \tag{6.5}$$

We test the behavior of our implementation for various tensor orders and rank $R = 10, 50$. The performance metric we compute is the speedup attained using $p = \prod_{i=1}^{N} p_i$ processors.

In the sequel, we plot the speedup for the following cases (in all cases with synthetic data, the tensor $\boldsymbol{\mathcal{X}}$ has one billion entries):

1. Third order tensor: we set $I = J = K = 1000$.

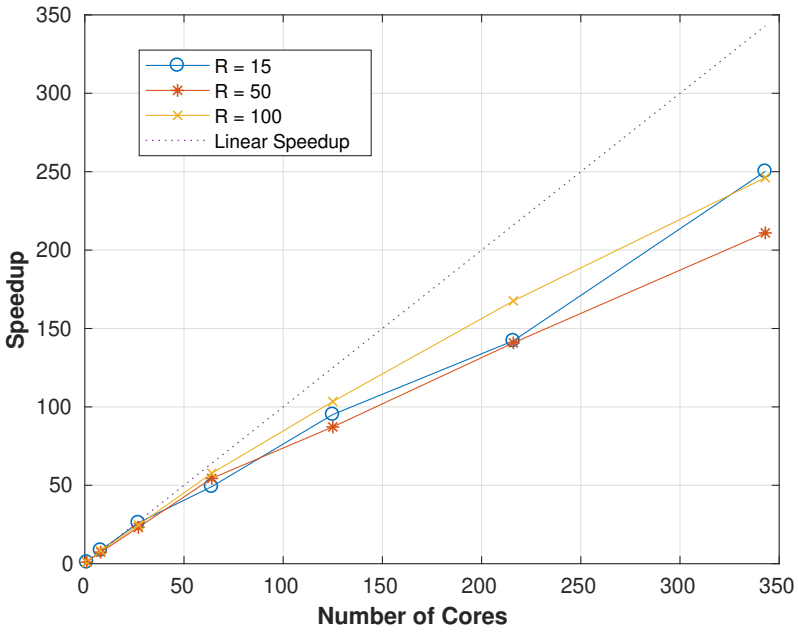

Figure 6.5: UOTF: speedup achieved for a $2000 \times 2000 \times 2000$ tensor with p cores, for $p = 1, 8, 27, 64, 125, 216, 343$.

2. Fourth order tensor: we set $I = J = K = L = 178$.

3. Fifth order tensor: we set $I = J = K = L = M = 64$.

The number of processors used in the trials were $p = 1, 2, 4, 8, 16, 32, 64, 128, 256$. We first present the formation of the grid for the various experiments carried out:

Plots of the speedup for the cases described above are presented next:

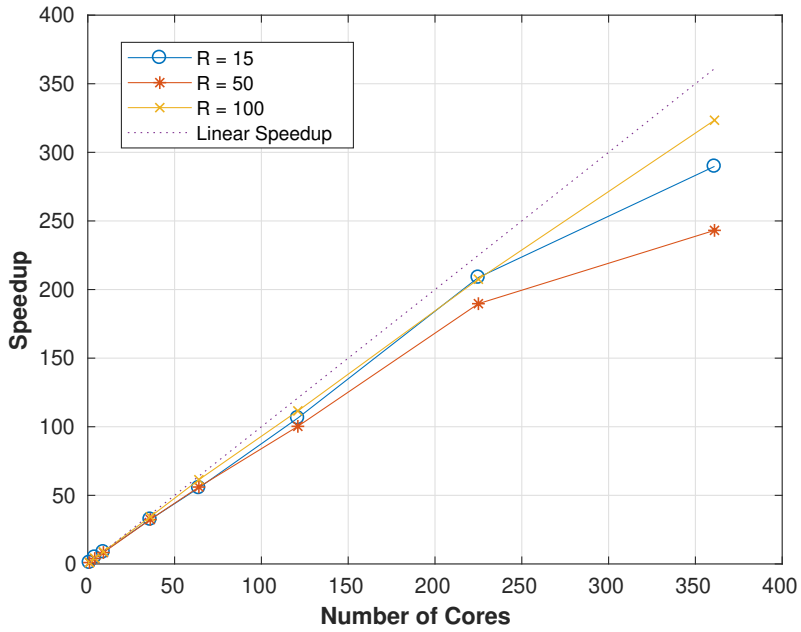

Figure 6.6: UOTF: speedup achieved for a $5000 \times 320 \times 5000$ tensor with p cores, for $p = 1, 4, 9, 36, 64, 121, 225, 361$.

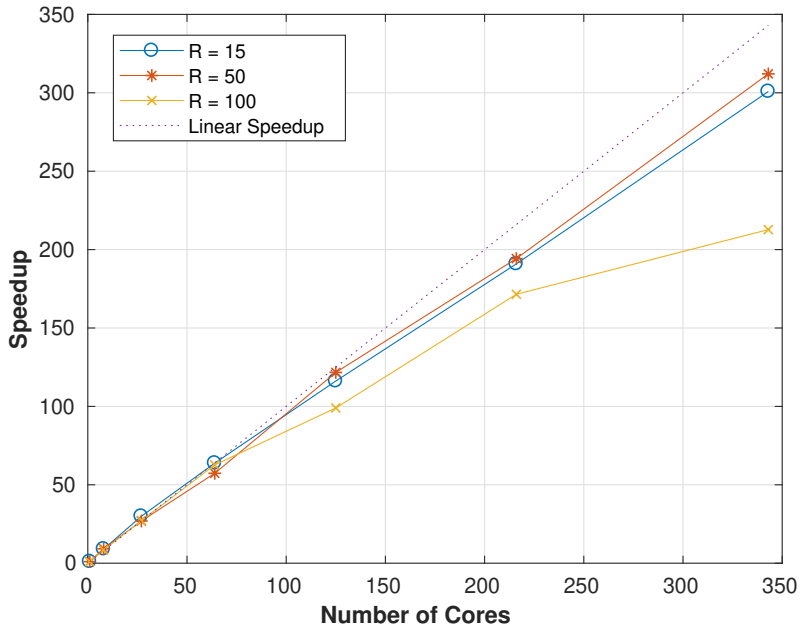

Figure 6.7: UOTF: speedup achieved for a $400 \times 50000 \times 400$ tensor with p cores, for $p = 1, 8, 27, 64, 125, 216, 343$.

Table 6.4: Grid formations used.

Cores	Third order tensor	Fourth order tensor	Fifth order tensor
1	$1 \times 1 \times 1$	$1 \times 1 \times 1 \times 1$	$1 \times 1 \times 1 \times 1 \times 1$
2	$2 \times 1 \times 1$	$2 \times 1 \times 1 \times 1$	$2 \times 1 \times 1 \times 1 \times 1$
4	$2 \times 2 \times 1$	$2 \times 2 \times 1 \times 1$	$2 \times 2 \times 1 \times 1 \times 1$
8	$2 \times 2 \times 2$	$2 \times 2 \times 2 \times 1$	$2 \times 2 \times 2 \times 1 \times 1$
16	$4 \times 2 \times 2$	$2 \times 2 \times 2 \times 2$	$2 \times 2 \times 2 \times 2 \times 1$
32	$4 \times 4 \times 2$	$4 \times 2 \times 2 \times 2$	$2 \times 2 \times 2 \times 2 \times 2$
64	$4 \times 4 \times 4$	$4 \times 4 \times 2 \times 2$	$4 \times 2 \times 2 \times 2 \times 2$
128	$8 \times 4 \times 4$	$4 \times 4 \times 4 \times 2$	$4 \times 4 \times 2 \times 2 \times 2$
256	$8 \times 8 \times 4$	$4 \times 4 \times 4 \times 4$	$4 \times 4 \times 4 \times 2 \times 2$

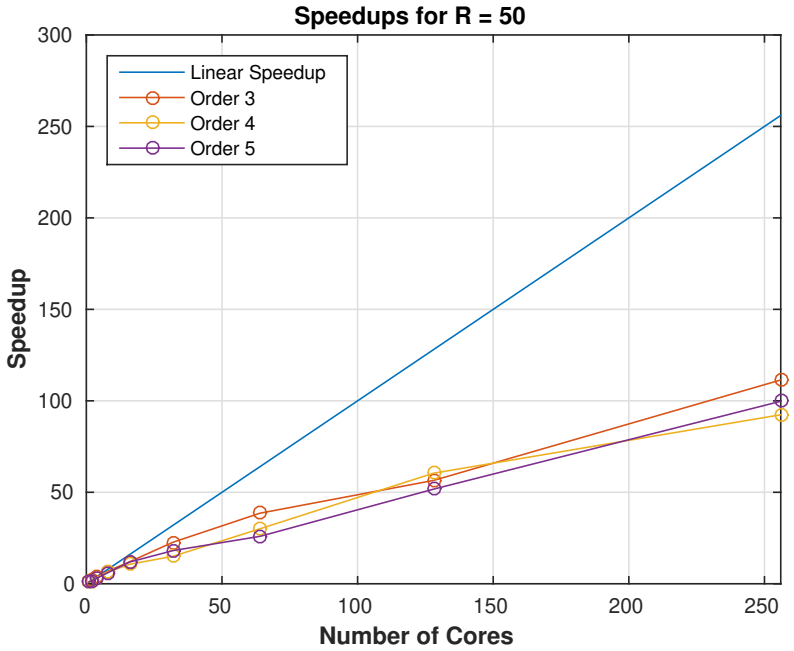

Figure 6.8: Speedups for $R = 50$. The number of cores p takes the values $p = 1, 2, 4, 8, 16, 32, 64, 128, 256$, while the third-order tensor has dimensions $1000 \times 1000 \times 1000$, the fourth-order has dimensions $178 \times 178 \times 178 \times 178$ and the fifth-order has dimensions $64 \times 64 \times 64 \times 64 \times 64$.

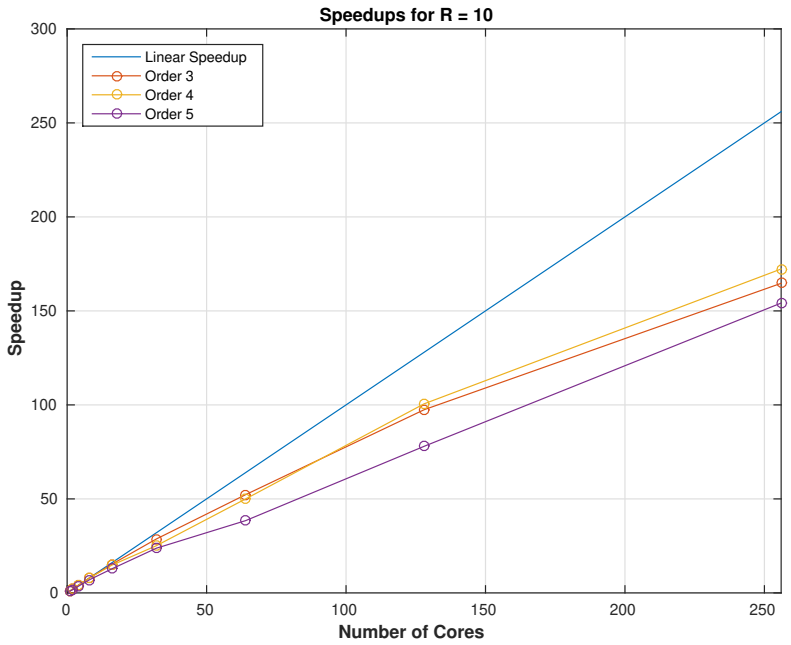

Figure 6.9: Speedups for $R = 10$. The number of cores p takes the values $p = 1, 2, 4, 8, 16, 32, 64, 128, 256$, while the third - order tensor has dimensions $1000 \times 1000 \times 1000$, the fourth-order has dimensions $178 \times 178 \times 178 \times 178$ and the fifth-order has dimensions $64 \times 64 \times 64 \times 64 \times 64$.

Chapter 7

Conclusion

In this report, we considered the problem of *dense* tensor factorization assuming that the factors are unconstrained, nonnegative, orthogonal, or sparse. We adopted the optimal first-order framework and developed efficient algorithms. We implemented the algorithms in distributed environments, using MPI, and assessed the achieved speedup. We observed that our implementations offer significant speedup, making our approach very effective, and rendering our algorithms very strong candidates for the solution of very large dense tensor factorization problems.

Our plan is to proceed to the sparse case, and develop and implement efficient algorithms for this very important case.

Our results will be submitted for publication to top international journals and conferences.

Bibliography

[1] P. M. Kroonenberg, *Applied Multiway Data Analysis*. Wiley-Interscience, 2008.

[2] A. Cichocki, R. Zdunek, A. H. Phan, and S. Amari, *Nonnegative Matrix and Tensor Factorizations*. Wiley, 2009.

[3] T. G. Kolda and B. W. Bader, "Tensor decompositions and applications," *SIAM Review*, vol. 51, no. 3, pp. 455–500, September 2009.

[4] N. D. Sidiropoulos, L. De Lathauwer, X. Fu, K. Huang, E. E. Papalexakis, and C. Faloutsos, "Tensor decomposition for signal processing and machine learning," *IEEE Transactions on Signal Processing*, vol. 65, no. 13, pp. 3551–3582, 2017.

[5] A. Cichocki, D. Mandic, L. De Lathauwer, G. Zhou, Q. Zhao, C. Caiafa, and H. A. Phan, "Tensor decompositions for signal processing applications: From two-way to multiway component analysis," *Signal Processing Magazine, IEEE*, vol. 32, no. 2, pp. 145–163, 2015.

[6] Y. Xu and W. Yin, "A block coordinate descent method for regularized multiconvex optimization with applications to nonnegative tensor factorization and completion," *SIAM Journal on imaging sciences*, vol. 6, no. 3, pp. 1758–1789, 2013.

[7] L. Sorber, M. Van Barel, and L. De Lathauwer, "Structured data fusion." *IEEE Journal on Selected Topics in Signal Processing*, vol. 9, no. 4, pp. 586–600, 2015.

[8] A. P. Liavas and N. D. Sidiropoulos, "Parallel algorithms for constrained tensor factorization via alternating direction method of multipliers," *IEEE Transactions on Signal Processing*, vol. 63, no. 20, pp. 5450–5463, 2015.

[9] K. Huang, N. D. Sidiropoulos, and A. P. Liavas, "A flexible and efficient framework for constrained matrix and tensor factorization," *IEEE Transactions on Signal Processing*, accepted for publication, May 2016.

[10] L. Karlsson, D. Kressner, and A. Uschmajew, "Parallel algorithms for tensor completion in the CP format," *Parallel Computing*, 2015.

[11] S. Smith and G. Karypis, "A medium-grained algorithm for distributed sparse tensor factorization," *30th IEEE International Parallel & Distributed Processing Symposium*, 2016.

[12] O. Kaya and B. Uçar, "Scalable sparse tensor decompositions in distributed memory systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2015, p. 77.

[13] J. Nocedal and S. Wright, *Numerical optimization*. Springer Science & Business Media, 2006.

[14] Y. Nesterov, *Introductory lectures on convex optimization*. Kluwer Academic Publishers, 2004.

[15] N. Guan, D. Tao, Z. Luo, and B. Yuan, "Nenmf: An optimal gradient method for nonnegative matrix factorization," *IEEE Transactions on Signal Processing*, vol. 60, no. 6, pp. 2882–2898, 2012.

[16] Y. Zhang, G. Zhou, Q. Zhao, A. Cichocki, and X. Wang, "Fast nonnegative tensor factorization based on accelerated proximal gradient and low-rank approximation," *Neurocomputing*, vol. 198, no. Supplement C, pp. 148 – 154, 2016.

[17] B. O' Donoghue and E. Candes, "Adaptive restart for accelerated gradient schemes," *Foundations of computational mathematics*, vol. 15, no. 3, pp. 715–732, 2015.

[18] L. Eldén and H. Park, "A procrustes problem on the stiefel manifold," *Numerische Mathematik*, vol. 82, no. 4, pp. 599–619, 1999.

[19] R. A. Harshman and M. E. Lundy, "Parafac: Parallel factor analysis," *Computational Statistics & Data Analysis*, vol. 18, no. 1, pp. 39–72, 1994.

[20] A. Beck and M. Teboulle, "A fast iterative shrinkage-thresholding algorithm for linear inverse problems," *SIAM journal on imaging sciences*, vol. 2, no. 1, pp. 183–202, 2009.

[21] R. Tibshirani, "Regression shrinkage and selection via the lasso," *Journal of the Royal Statistical Society: Series B (Methodological)*, vol. 58, no. 1, pp. 267–288, 1996.

[22] O. Kaya and B. Uçar, "Parallel Candecomp/Parafac Decomposition of Sparse Tensors Using Dimension Trees," *SIAM Journal on Scientific Computing*, vol. 40, no. 1, pp. C99 – C130, 2018. [Online]. Available: https://hal.inria.fr/hal-01397464

[23] M. Rajih, P. Comon, and R. A. Harshman, "Enhanced line search: A novel method to accelerate parafac," *SIAM Journal on Matrix Analysis and Applications*, vol. 30, no. 3, pp. 1128–1147, 2008.

[24] L. Sorber, I. Domanov, M. Van Barel, and L. De Lathauwer, "Exact line and plane search for tensor optimization," *Computational Optimization and Applications*, vol. 63, no. 1, pp. 121–142, 2016.

[25] C. A. Andersson and R. Bro, "The n-way toolbox for matlab." [Online]. Available: http://www.models.life.ku.dk/source/nwaytoolbox

[26] A. P. Liavas, G. Kostoulas, G. Lourakis, K. Huang, and N. D. Sidiropoulos, "Nesterov-based parallel algorithm for large-scale nonnegative tensor factorization," in *2017 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2017, New Orleans, USA, March 5-9, 2017*. IEEE, 2017.

[27] M. Razaviyayn, M. Hong, and Z.-Q. Luo, "A unified convergence analysis of block successive minimization methods for nonsmooth optimization," *SIAM Journal on Optimization*, vol. 23, no. 2, pp. 1126–1153, 2013.

[28] A. Grama, G. Karypis, V. Kumar, and A. Gupta, *Introduction to Parallel Computing (2nd Edition)*. Pearson, 2003.

[29] N. Vervliet, O. Debals, L. Sorber, M. Van Barel, and L. De Lathauwer, "Tensorlab 3.0," Mar. 2016. [Online]. Available: http://www.tensorlab.net/

[30] C. L. Lawson and R. J. Hanson, *Solving Lease Squares Problems*. SIAM, 1995.

[31] J.-P. Royer, N. Thirion-Moreau, and P. Comon, "Computing the polyadic decomposition of nonnegative third order tensors," *Signal Processing*, vol. 91, no. 9, pp. 2159–2171, 2011.

[32] G. Guennebaud, B. Jacob *et al.*, "Eigen v3," http://eigen.tuxfamily.org, 2010.