Institut für Informatik
LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelorarbeit

Towards MapReduce Algorithms for the Higher Order-Singular Value Decomposition

Diego Havenstein

Aufgabensteller:	Prof. Dr. François Bry
Betreuer:	Dipl.-Inf. Christoph Wieser
Abgabetermin:	02. August 2012

Zusammenfassung

Die Singulärwertzerlegung höherer Ordnung (HO-SVD) ist die Verallgemeinerung der Singulärwertzerlegung (SVD) von Matrizen auf Tensoren. Diese Produktdarstellungen von Matrizen bzw. Tensoren besitzen ähnliche mathematische Eigenschaften und zahlreiche Anwendungen in der Wissenschaft.
Die Berechnung der SVD, und vor allem der HO-SVD, sind leider für moderne Rechner sehr aufwändige Berechnungen. Der Hauptzweck dieser Arbeit besteht darin, eine Methodik zu beschreiben mit welcher sich die HO-SVD mit einem MapReduce-Algorithmus verteilt berechnen lässt. Zusätzlich wird eine mögliche Anwendung dieser Technik im Bereich der Latenten Semantischen Analyse demonstriert, wobei die Erkennung von semantischen Strukturen in mehrdimensionalen Daten durch die erwähnten Darstellungen ermöglicht wird.

Abstract

The Higher-Order Singular Value Decomposition (HO-SVD) is the generalisation of the Singular Value Decomposition (SVD) from matrices to tensors. These decompositions have similar mathematical properties and many useful applications in science.
Unfortunately, the computation of the SVD, and especially of the HO-SVD, represent computationally very expensive tasks for modern computers. The main purpose of this thesis is to demonstrate how the HO-SVD can be computed in a partially distributed manner using a MapReduce algorithm. In addition, a possible application of this technique in the field of Latent Semantic Analysis is demonstrated, where semantic structures in multidimensional data can be detected using the mentioned decompositions.

Acknowledgements

I want to express my sincere thanks to the members of the teaching and research unit Programming and Modelling Languages of the Ludwig Maximilian University of Munich. I want to thank my supervisors, Prof. Dr. François Bry and Dipl.-Inf. Christoph Wieser, for their excellent mentoring and also for allocating a working place for me. The thesis in its actual form would not have been possible without the invaluable hints and advice Christoph Wieser gave me during the semester.

Last but not least I thank my friends and family for supporting and motivating me.

This thesis is dedicated to Rüdiger, Martina, Charlotte and Friederike von Stosch for making my studies abroad the most pleasant, unforgettable experience.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Related work . . . 2
- 1.3 Structure of this thesis . . . 3

2 Mathematical Background **5**
- 2.1 Singular Value Decomposition . . . 5
- 2.2 The QR Decomposition . . . 6
- 2.3 Higher-Order Singular Value Decomposition . . . 7
  - 2.3.1 Tensors . . . 7
  - 2.3.2 Higher-Order Singular Value Decomposition . . . 9
- 2.4 Householder reflections and Givens rotations . . . 10
  - 2.4.1 Householder reflections . . . 10
  - 2.4.2 Givens rotations . . . 10

3 Parallel algorithms for the HO-SVD **13**
- 3.1 Computing the SVD in parallel . . . 13
  - 3.1.1 A distributed parallel version of the R-SVD Algorithm . . . 14
    - 3.1.1.1 General idea of the R-SVD Algorithm . . . 14
    - 3.1.1.2 Computing the QR factorisation . . . 15
    - 3.1.1.3 Computing a bidiagonal matrix . . . 17
    - 3.1.1.4 The Golub-Kahan SVD Step . . . 17
    - 3.1.1.5 The R-SVD algorithm . . . 21
  - 3.1.2 The Hestenes' method . . . 21
    - 3.1.2.1 The general idea . . . 21
    - 3.1.2.2 Computing the rotations . . . 22
    - 3.1.2.3 Orthogonalising all column pairs . . . 23
    - 3.1.2.4 ORTH . . . 24
    - 3.1.2.5 Normalising W . . . 25
- 3.2 Perfomance issues . . . 25
  - 3.2.1 Computing the Householder vector . . . 25
  - 3.2.2 Applying Householder matrices . . . 25
  - 3.2.3 Storage of reflections . . . 26
  - 3.2.4 Computing the entries c and s of $G(i, k, \theta)$ . . . 27
  - 3.2.5 Applying a rotation . . . 28
  - 3.2.6 Optimal block sizes in the ROC algorithm . . . 28
- 3.3 Parallel computing of the HO-SVD . . . 28
  - 3.3.1 Parallel HO-SVD with R-SVD . . . 29
  - 3.3.2 Combining CubeSVD and Hestenes' method . . . 30

4 MapReduce Algorithm **31**
4.1 MapReduce basics 31
4.2 Ideas for parallel HO-SVD with MapReduce 32
4.2.1 The Golub-Reinsch Algorithm 32
4.2.1.1 `MatrixSplitter` 33
4.2.1.2 `HouseholderMR` 33
4.2.1.3 `GivensForRows` 35
4.2.1.4 Testing the `UJMP`-class `SVDMatrix` 37
4.2.1.5 Algorithm summary 38
4.2.2 Using the Hestenes' method 39

5 Latent Semantic Indexing **43**
5.1 Basic concepts 43
5.1.1 Information Retrieval 43
5.1.2 Evaluating the quality of a result 44
5.1.3 Problems of traditional Information Retrieval systems 44
5.2 The vector space model 44
5.2.1 The Singular Value Decomposition 45
5.2.1.1 The dot product as similarity measure 45
5.2.1.2 The cosine similarity 45
5.2.2 The Higher-Order Singular Value Decomposition 46

6 Conclusion and future work **49**

List of Figures **51**

List of Tables **53**

Bibliography **56**

Chapter 1

Introduction

1.1 Motivation

Nowadays, where vast amounts of information are being generated by applications in a wide range of fields in science, it has become a very complex task to actually be able to extract useful information from the data. The generated data can have the most variate origins, from computational simulations or physical experiments to information extracted from the world wide web.
When the generated data can be stored using a two-dimensional data structure, a useful technique that allows us to extract relevant information, which is not obvious at first sight, is the *Singular Value Decomposition* (SVD). This advanced procedure from the field of linear algebra reveals much of the structure of the matrix storing the data. Because this decomposition is defined for the two-dimensional case, a generalisation of this concept is necessary as soon as our data is higher-dimensional.
The *Higher-Order Singular Value Decomposition* (HO-SVD) is the generalisation of the Singular Value Decomposition. This decomposition allows us to study the structure of higher-dimensional data structures, called *tensors* in mathematics, which can have the most variate origins. For a more extensive introduction to the subject of tensor decompositions and applications please also refer to [KB09].

One of the conceivable applications of such a higher-order decomposition would be given in the web search market. If a company operating a search engine wishes to provide personalised search possibilities to their users, it is certanly insufficient to store a (sparse) two-dimensional structure specifying occurrences of terms in documents (in the field of Information Retrieval this is commonly known as a term-document matrix), since no personalisation of the results is achieved through this technique. A possible solution to this dilemma could be to maintain a term-document matrix for every registered user. This, of course, would be unwise from a purely technical point of view because of the dramatic increase of memory space requirement that would be needed for this representation. A more promising approach would be to store the users together with the available documents and corresponding terms in a *single* data structure. If we do this, a three-dimensional structure results. Since the HO-SVD is defined for tensors of *any* dimension larger than three, it is possible to examine such a structure using this technique. The only issue concerning this approach is the computation of the decomposition for a given tensor, which is a very expensive task with respect to the complexity of this operation.

With the steady increase of computing power which has been observed over the last decades, it seemed like computers would always be be able to execute highly complex computations in a reasonable amount of time. Moore's law, a rule of thumb formulated by the co-founder of the Intel Corporation, says that the number of transistors which can be placed on an integrated circuit doubles every two years approximately. Although this rule has remained valid for decades, the physical of this limits will be reached sooner or later. When these limits are reached, the only feasible way to increase computational power will presumably be the use of techniques for distributed parallel algorithms. Todays IT industry relies heavily on distributed computations, as can be observed in the cases of Google, Yahoo, Facebook and many others.
As a natural consequence of the increasing popularity and advances in distributed computing, companies relying on this kind of infrastructure began developing methods for reducing the complexity involved in the programming of distributed algorithms.
The most widely used method for distributed computations nowadays is possibly the Apache Hadoop Framework, which is based on the idea of MapReduce algorithms (first introduced in 2004 by two Google engineers, see [DG08]) and additionally includes a distributed file-system called HDFS (Hadoop Distributed File-System). There are a number of projects which aim to extend the capabilities of the core framework, such as Apache HBase (`www.hbase.apache.org`), a distributed database system, and Apache Mahout (`www.mahout.apache.org`), a project which aims to design scalable machine learning and data mining techniques.
The only approach for computing the SVD with a MapReduce algorithm included at the projects extending Apache Hadoop Core at the moment of writing this thesis relies on a stochastic SVD algorithm (*SSVD*), which is a part of the methods provided by the Apache Mahout project. This method has benefits over other known methods, such as the reduced flops required for the computation. Unfortunately, SSVD has also drawbacks when compared to classical methods, since the result will potentially be much less precise.

The focus of this thesis lies on parallel algorithms for computing the Higher-Order Singular Value Decomposition of multidimensional data. In order to process the data generated by the online game *ARTigo* (see `www.ARTigo.org`), which was inspired by the so-called *games with a purpose* [vA06], a corresponding implementation of a distributed HO-SVD algorithm will be developed.

1.2 Related work

Since the SVD and the HO-SVD are useful in a wide range of fields, the improvement of suitable methods for obtaining this decompositions has been a source of great interest in the last years. Particularly in the field of Information Retrieval, due to the polysemy and synonymy present in natural problems, it has become a very important task to be able to analyze term-document matrices with an alternative approach.

The idea of computing the SVD in a distributed manner is the topic of a paper by Liu, Li, Hammoud, Alham and Ponraj ([LLH$^+$10]). The approach relies on a clustering of the available documents, which are represented with a term-document matrix. This is achieved using a K-means algorithm, and then executing an LSI analysis on each cluster. The authors present two approaches for computing the SVD in parallel, one of them involving a MapReduce algorithm. The idea, although interesting, clearly does not represent a distributed computation of the SVD of the original term-document matrix.

Many interesting approaches concerning the parallel computation of the SVD are given in the paper [BMPS03]. This survey gives a view of known algorithms for computing the SVD of dense and sparse matrices, with emphasis on methods that are suitable for parallel computing. The `ROC` algorithm, which is one of the central topics of this thesis, is also described in this paper. The `ROC` algorithm was originally formulated in a paper by Sidje and Philippe, [SP94].

The algorithm described in section 3.1.2 of this thesis, which presents the Hestenes' approach for computing the SVD of a matrix in parallel, is described with much detail in [Sha10]. The algorithm, as formulated in [Sha10], is designed to work on a one-dimensional systolic multiprocessor array, which means that the working nodes can communicate more or less directly during the SVD computation. The method is also described in [BL85].

Finally, a very relevant work in the field of numerical analysis is the reference book by Golub and Van Loan ([GL96]), called *Matrix Computations.* The third edition provides essential information about the mathematical and algorithmical background required for the production of numerical software. The fifth chapter of this book is concerned with orthogonalisation procedures, which are of central interest for this thesis and for any method relying on matrix (or more generally, tensor) decompositions.

1.3 Structure of this thesis

- **Chapter 1** gives an introduction to the topics which will be discussed in this thesis.
- **Chapter 2** presents the mathematical background necessary for understanding the methods in later chapters.
- **Chapter 3** describes methods for computing the SVD in parallel, particularly the R-SVD algorithm and the Hestenes' method. In the second section, some important issues related to the computation, application and storage of orthogonal transformations to matrices are discussed. In the third and last section of this chapter, two possible approaches for computing the HO-SVD in parallel with MapReduce are presented.
- **Chapter 4** presents a concrete implementation of one of the parallel algorithms for computing the HO-SVD using the Apache Hadoop Framework.
- **Chapter 5** discusses a possible application in the field of information retrieval.
- **Chapter 6** concludes this thesis.

Chapter 2

Mathematical Background

In this chapter, the mathematical background necessary for understanding the algorithms in later chapters is discussed. Two matrix decompositions will be discussed, the Singular Value Decomposition (SVD) and the QR Decomposition. Additionally, the Higher-Order Singular Value Decomposition (HO-SVD) on tensors is discussed. Finally, two classes of orthogonal transformations that can more generally be used to compute canonical forms of matrices are presented, Householder reflections and Givens rotations. The content is mainly based on [Sha10], [GL96] and [KB09] and the focus lies on properties that are relevant for this thesis. All definitions and corollars are provided without corresponding proofs, since complete presentations are given in many textbooks on numerical analysis ([GL96] for instance) or related topics.

2.1 Singular Value Decomposition

In this section, the Singular Value Decomposition (SVD) of a real matrix and its properties are presented.

Singular Value Decomposition. Every matrix $A \in \mathbb{R}^{m\times n}$ can be written as the factorisation of three matrices U, Σ and V, where

$$A = U\Sigma V^T, \Sigma = diag(\sigma_1, \dots, \sigma_p) \in \mathbb{R}^{m\times n}, p = min\{m, n\} \tag{2.1}$$

with $U \in \mathbb{R}^{m\times m}$ and $V \in \mathbb{R}^{n\times n}$ being orthogonal matrices and $\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_p \geq 0$.

- $\sigma_1, \dots, \sigma_p$ are the *singular values* of A.
- The columns $u_1, \dots, u_m$ are called *left singular vectors* of A.
- The columns $v_1, \dots, v_n$ are called *right singular vectors* of A.

Notice that Σ is a *pseudo diagonal matrix.*
When the SVD of a matrix $A \in \mathbb{R}^{m\times n}$ is computed, then we can keep only the first n rows of Σ so that we have a diagonal matrix. Of course, U has to be modified accordingly. In this case we set

$$A_{red} = U_{red}\Sigma_{red}V_{red}^T$$

where

$$U_{red} = U[:, 1 : n], \Sigma_{red} = \Sigma(1 : n, 1 : n)$$

This trimmed down version of the SVD is called thin SVD [GL96]. The thin SVD will be relevant for the chapter about Latent Semantic Analysis, where it is used to detect the latent semantics of the stored two-dimensional structure.
The SVD is related to the eigen-decomposition of $AA^T \in \mathbb{R}^{m\times m}$ and $A^TA \in \mathbb{R}^{n\times n}$, since $A^TAv_i = \sigma_i^2 v_i$ and $AA^Tu_i = \sigma_i^2 u_i$ [SA03].
Some important properties of the SVD are the following:

- The singular values of a matrix are *uniquely* determined.
- $A = U\Sigma V^T \Leftrightarrow A^T = V\Sigma^T U^T$
- If $rank(A) = r$, then A has r nonzero singular values $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_r > 0$

The definitions and properties discussed above extend to complex matrices.
The SVD has an important role to play in linear algebra, being used for pseudoinverse computing, least square problems, computation of the Jordan canonical form or solving integral equations, just to mention a few.

2.2 The QR Decomposition

In this section, the QR decomposition (also called QR factorisation) of a matrix is defined.

QR Decomposition. Given a matrix $A \in \mathbb{R}^{m\times n}$, its QR decomposition is given by

$$A = QR \tag{2.2}$$

where

- $Q \in \mathbb{R}^{m\times m}$ is orthogonal
- $R \in \mathbb{R}^{m\times n}$ is upper triangular.

This factorisation can be computed in several ways, generally using a sequence of orthogonal transformations. The most intuitive algorithm consists of applying a sequence of reflections $H_1, H_2, \ldots, H_n$ to A, where H_i is a reflection that zeroes the under-diagonal elements of the ith column vector of A. Setting

$$Q = \prod_{i=1}^{n} H_i$$

and the upper triangular

$$R = \left(\prod_{i=1}^{n} H_i\right)^T A = Q^T A$$

we obtain the decomposition. Notice that in cases where Q is not explicitly needed, a factorised form representation of Q may be more economical. In order to make this clear, a small example is provided:
Consider the 5-by-4 case. After applying a sequence of 4 reflections, $A \in \mathbb{R}^{5\times 4}$ would have the following form:

$$\begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} \\ v_2^{(1)} & r_{22} & r_{23} & r_{24} \\ v_3^{(1)} & v_3^{(2)} & r_{33} & r_{34} \\ v_4^{(1)} & v_4^{(2)} & v_4^{(3)} & r_{44} \\ v_5^{(1)} & v_5^{(2)} & v_5^{(3)} & v_5^{(4)} \end{bmatrix}$$

The above matrix contains the necessary Householder vectors to zero all under-diagonal components of A, and this vectors are normalised so that the first component is 1. Clearly, this matrix explicitly stores the representation of the R factor, and the factorised form of Q is given by the following sequence of Householder vectors:

$$\begin{bmatrix} 1 \\ v_2^{(1)} \\ v_3^{(1)} \\ v_4^{(1)} \\ v_5^{(1)} \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ v_3^{(2)} \\ v_4^{(2)} \\ v_5^{(2)} \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \\ v_4^{(3)} \\ v_5^{(3)} \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ v_5^{(4)} \end{bmatrix}$$

Notice that we are storing a list of $1 \times n$ vectors instead of a list of $n \times n$ matrices (or their product), which would be obtained by computing the corresponding Householder matrices from the vectors above. Please refer to section 3.2 of this thesis for a more detailed discussion concerning computation, storage and application of orthogonal transformations.
Since not every computation requires the explicit formation of the Q factor, it is of course not always necessary to store the reflections in the under-diagonal portion of the resulting R factor.

2.3 Higher-Order Singular Value Decomposition

The following subsections will treat the formal definition of *tensors* and related operations (section 2.3.1) as well as the *Higher-Order Singular Value Decomposition* (section 2.3.2).

2.3.1 Tensors

Tensor. A tensor (also called multidimensional array or n-way array) is the generalisation of a matrix. A matrix can be seen as a two-dimensional tensor, a vector as a one-dimensional tensor. We refer to the different dimensions of a given tensor as its *modes.*
Example: the third mode of a tensor $\mathcal{A} \in \mathbb{R}^{I \times J \times K}$ is of size K.

Tensor order. The order of a tensor $\mathcal{A} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ is N. Its elements are denoted as $a_{i_1 \ldots i_n \ldots i_N}$, where $1 \leq i_n \leq I_n$ for $1 \leq n \leq N$.

Tensor unfoldings. Let $\mathcal{A} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ be a tensor. $\mathcal{A}$ has N matrix unfoldings $A_{(n)} \in \mathbb{R}^{I_n \times \prod_{k=1, k \neq n}^{N} I_k}$ for all n such that $0 < n < N + 1$ holds. These are matrix representations of the tensor. The unfoldings contain elements $a_{i_1 \ldots i_N}$ at the position with row number i_n and column number $(i_{n+1} - 1) I_{n+2} \ldots I_N I_1 \ldots I_{n-1} + (i_{n+2} - 1) I_{n+3} \ldots I_N I_1 \ldots I_{n-1} + \cdots + (i_N - 1) I_1 \ldots I_{n-1} + (i_1 - 1) I_2 \ldots I_{n-1} + (i_2 - 1) I_3 \ldots I_{n-1} + \cdots + i_{n-1}$ [LMV00].

When constructing the nth unfolding of a tensor, the operation consists of rearranging the mode-n fibers of the tensor to be the columns of the resulting matrix. As an example, consider a tensor $\mathcal{A} \in \mathbb{R}^{3 \times 3 \times 2}$, with frontal slices

$$\mathcal{A}_1 = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}, \mathcal{A}_2 = \begin{bmatrix} 10 & 13 & 16 \\ 11 & 14 & 17 \\ 12 & 15 & 18 \end{bmatrix}$$

The unfoldings of $\mathcal{A}$ are

$$A_{(1)} = \begin{bmatrix} 1 & 4 & 7 & 10 & 13 & 16 \\ 2 & 5 & 8 & 11 & 14 & 17 \\ 3 & 6 & 9 & 12 & 15 & 18 \end{bmatrix}$$

$$A_{(2)} = \begin{bmatrix} 1 & 2 & 3 & 10 & 11 & 12 \\ 4 & 5 & 6 & 13 & 14 & 15 \\ 7 & 8 & 9 & 16 & 17 & 18 \end{bmatrix}$$

$$A_{(3)} = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 \end{bmatrix}$$

Notice that the tensor unfoldings can be "filled" column-wise, and by just rearranging the indices over which must be iterated in $\mathcal{A}$, the correct entries can be determined. A vectorised version of $\mathcal{A}$ is also possible [KB09], in the example above it would be

$$vec(\mathcal{A}) = \begin{bmatrix} 1 \\ 2 \\ \vdots \\ 18 \end{bmatrix}$$

Norm of a tensor. The norm of a tensor $\mathcal{A}$ is defined by the Frobenius-norm [1] of any of its existing unfoldings.

$$\|\mathcal{A}\| := \|A_{(1)}\|_F = \cdots = \|A_{(N)}\|_F \tag{2.3}$$

Rank of a tensor. The n-rank of $\mathcal{A}$, $rank_n(\mathcal{A})$, is the dimension of the vector space spanned by the n-mode vectors and the following equation holds:

$$rank_n(A_{(n)}) = rank(\mathcal{A}) \tag{2.4}$$

Tensor product. The tensor product (denoted by the operator $\otimes$) for tensors $\mathcal{A} \in \mathbb{R}^{d_1 \times \cdots \times d_N}$, $\mathcal{B} \in \mathbb{R}^{f_1 \times \cdots \times f_M}$ is defined as:

$$c_{i_1,\ldots,i_N,j_1,\ldots,j_M} := a_{i_1,\ldots,i_N} b_{j_1,\ldots,j_M} \tag{2.5}$$

where $\mathcal{C} = \mathcal{A} \otimes \mathcal{B} \in \mathbb{R}^{d_1 \times \cdots \times d_N \times f_1 \times \cdots \times f_M}$.

Scalar product. The scalar product $\langle \cdot, \cdot \rangle$ for tensors is given by:

$$\langle \mathcal{A}, \mathcal{B} \rangle := \sum_{i_1} \cdots \sum_{i_N} b_{i_1 \ldots i_N} a_{i_1 \ldots i_N} \tag{2.6}$$

for $\mathcal{A}$,$\mathcal{B} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$.

[1] The Frobenius-norm of a complex matrix $A \in \mathcal{C}^{m \times n}$ is given by

$$\|A\|_F := \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} \mid a_{ij} \mid^2}$$

n-mode product. Let $\mathcal{A} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ and $M \in \mathbb{R}^{J_n \times I_n}$ be given.
Then $\mathcal{A} \times_n M \in \mathbb{R}^{\prod_{i=1}^{n-1} I_i \times J_n \times \prod_{i=n+1}^{N} I_i}$, and we define the new tensor elementwise as

$$(\mathcal{A} \times_n M)_{i_1 \dots i_{n-1} j_n i_{n+1} \dots i_N} := \sum_{i_n \in I_n} a_{i_1 \dots I_n} m_{j_n i_n} \tag{2.7}$$

The operator $\times_n$ is called the n-mode product.

The idea can also be expressed in terms of tensor unfoldings [KB09]:

$$\mathcal{Y} = \mathcal{A} \times_n M \Leftrightarrow \mathcal{Y}_{(n)} = M\mathcal{A}_{(n)} \tag{2.8}$$

2.3.2 Higher-Order Singular Value Decomposition

Higher-Order Singular Value Decomposition. The Higher-Order Singular Value Decomposition (HO-SVD) of a tensor $\mathcal{A} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ is defined as

$$\mathcal{A} = \mathcal{S} \times_1 U_1 \times_2 \cdots \times_N U_N \tag{2.9}$$

when the following properties hold:

- The U_i are orthogonal for $i \in \{1, \dots, N\}$. These matrices are composed by the left singular vectors of the corresponding matrix-unfoldings of $\mathcal{A}$, as we will see later.
- The *core tensor* $\mathcal{S} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$, and all subtensors with fixed index are *all-orthogonal*, which means $\langle \mathcal{S}_{i_n=a}, \mathcal{S}_{i_n=b} \rangle = 0$ for all $a, b \in \{1, \dots, N\}$ where $a \neq b$.
- $\mathcal{S} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$, and all subtensors with fixed index are *ordered* ($\|\mathcal{S}_{i_n=1}\| \geq \cdots \geq \|\mathcal{S}_{i_n=I_n}\| \geq 0$ for all possible values of n). Notice the similar properties of $\mathcal{S}$ in the HO-SVD and Σ in the SVD.
- $\sigma_{n,i} := \|\mathcal{S}_{i_n=i}\|$ are called *n-mode singular values* and the vectors $u_{n,i} \in \mathbb{R}^n$ are called *n-mode singular vectors* of $\mathcal{A}$, analogously to the SVD definitions.

Decompositions of higher-order tensors have useful applications in many different fields. The HO-SVD is used in psychometrics, chemometrics, signal processing, compression algorithms, numerical linear algebra and analysis, statistics, computer vision, OCR, data mining, personalised web-search, neuroscience and graph analysis, just to mention a few.
There is an important link between HO-SVD and SVD [Sha10]:
"Let $\mathcal{A} = \mathcal{S} \times_1 U_1 \times_2 \cdots \times_N U_N$ a HO-SVD, then the SVD of the n-mode matrix unfolding $A_{(n)}$ is: $A_{(n)} = U_n \Sigma_n V_n^T$, with $\Sigma_n = diag(\sigma_{n,1}, \dots, \sigma_{n,I_n}) \in \mathbb{R}^{I_n \times I_n}$ and $V_n^T := (\Sigma_n^{-1} S_n) \cdot (U_{n+1} \otimes \cdots \otimes U_N \otimes U_1 \otimes \cdots \otimes U_{n-1})^T \in \mathbb{R}^{I_{n+1} \dots I_N I_1 \dots I_{n-1} \times I_n}$, where $\otimes$ denotes the Kronecker product". [2]
This means that we can compute the HO-SVD of a n-mode tensor $\mathcal{A}$ by computing the SVD of its first n tensor unfoldings [Sha10]. In fact, all known methods for computing the HO-SVD of a tensor need the SVD of its unfoldings. The complete SVD of the unfoldings is actually not needed, but only the matrices containing the left singular vectors in order to compute the approximation of the tensor.

[2] The *Kronecker product* of two matrices $B \in \mathbb{R}^{m \times n}$ and $C \in \mathbb{R}^{p \times q}$ is given by:

$$B \otimes C = \begin{pmatrix} b_{11}C & b_{12}C & \cdots & b_{1n}C \\ b_{21}C & b_{22}C & \cdots & b_{2n}C \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1}C & b_{m2}C & \cdots & b_{mn}C \end{pmatrix} \in \mathbb{R}^{mp \times nq}$$

2.4 Householder reflections and Givens rotations

In this section, two important matrix transformation techniques are presented: Householder relflections and Givens rotations. These are orthogonal transformations that can be used for obtaining various canonical forms of a given matrix, for example the QR decomposition of a real matrix $A \in \mathbb{R}^{m \times n}$. This is highly relevant for the next chapters of this thesis, since the parallel SVD algorithm described in section 3.1.1 relies on a previous QR factorisation of the tensor unfoldings. Additionally, several procedures in the final algorithm make use of orthogonal transformations.

2.4.1 Householder reflections

The most useful technique for introducing zeros at grand scale in a matrix with orthogonal transformations is presented in this section. A 2-by-2 orthogonal, symmetric matrix Q having the form

$$Q = \begin{bmatrix} cos(\theta) & sin(\theta) \\ sin(\theta) & -cos(\theta) \end{bmatrix} \tag{2.10}$$

is a reflection. The vector $y = Qx = Q^T x$ is a reflection of x across the line defined by

$$span \left\{ \begin{bmatrix} cos(\theta/2) \\ sin(\theta/2) \end{bmatrix} \right\} \tag{2.11}$$

as described in [GL96].

Householder reflection. Let $v \in \mathbb{R}^n \neq 0$ be given. We call $P \in \mathbb{R}^{n \times n}$ a Householder reflection if it has the form

$$P = I - \frac{2}{v^T v} v v^T \tag{2.12}$$

where v is called the Householder vector and P the Householder matrix.

It can be easily verified that Householder matrices are symmetric and orthogonal. Householder reflections can be used to zero the components $x(2 : n, 1)$ of a vector $x \in \mathbb{R}^n$. What makes them computationally attractive is the fact that the determination of an appropriate Householder vector is simple.
There are however many practical details and things to be considered while determining, storing and applying reflections to matrices or vectors. If these details are not recognised, the computational effort can be rapidly increased by an order of a magnitude. These important perfomance issues and practical implementation details will be discussed in section 3.2.

2.4.2 Givens rotations

While Householder reflections are mostly used for introducing zeros in a matrix at grand scale, if we have to zero elements more selectively, the orthogonal transformation type of choice are Givens rotations. A 2-by-2 orthogonal matrix Q is a rotation if it has the form

$$Q = \begin{bmatrix} cos(\theta) & sin(\theta) \\ -sin(\theta) & cos(\theta) \end{bmatrix} \tag{2.13}$$

Such a matrix can be used to rotate a vector x counterclockwise through an angle θ $(y = Q^T x)$.

A Givens rotation is a matrix of the form

$$G(i,k,\theta) = \begin{bmatrix} 1 & \dots & 0 & \dots & 0 & \dots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \dots & c & \dots & s & \dots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \dots & -s & \dots & c & \dots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \dots & 0 & \dots & 0 & \dots & 1 \end{bmatrix} \tag{2.14}$$

The intersections of the ith and the kth row/columns ($i < k$) contain factors c or s, and these are defined as $cos(\theta)$ and $sin(\theta)$ respectively for a given θ. Other components of the matrix are defined by the Kronecker delta [3]. Givens rotations are clearly orthogonal (since $G(i,k,\theta)G(i,k,\theta)^T = I_m$ for $G(i,k,\theta) \in \mathbb{R}^{m \times m}$ and any θ). The effect of premultiplying a matrix with $G(i,k,\theta)$ is a counterclockwise rotation in the (i,k) coordiante plane of θ radians [GL96].
An efficient method for determining the entries c and s of $G(i,k,\theta)$ (without explicit computation of θ), as well as facts to be considered while applying and storing rotations, are discussed later in this thesis (see section 3.2).

[3] $\delta_{p,q}$ with $p,q \in \mathbb{N}$ is defined as

$$\delta_{p,q} := \begin{cases} 1, p = q \\ 0, \text{else} \end{cases}$$

Chapter 3

Parallel algorithms for the HO-SVD

In this chapter, the most important base algorithms to be implemented in chapter 4 of this thesis are described. The main steps of the final algorithm can be summarised by the following sequence:

1. Save the unfoldings of the input tensor.
2. Compute the SVD of each unfolding.
3. Compute the core tensor of the resulting HO-SVD.
4. Compute the HO-SVD of the tensor.

Two algorithms for obtaining the SVD of each tensor unfolding are described. Both can be implemented for a distributed computation of the HO-SVD of a tensor using the MapReduce concept for essential tasks. [1]

3.1 Computing the SVD in parallel

In this section, two algorithms that can be used to compute the SVD of a matrix in parallel are presented. While many algorithms that can accomplish this task are described in the literature, these algorithms were chosen because of their parallelisation potential at important, computationally expensive operations which are needed in order to obtain the desired decomposition.
The task at hand will be to compute the SVD of each unfolding of a tensor $\mathcal{A} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$. The elements in the set containing the unfoldings,

$$unf(\mathcal{A}) := \{A_{(i)} \in \mathbb{R}^{I_i \times I_1 I_2 \ldots I_{i-1} I_{i+1} \ldots I_N} : 0 < i < N+1\} \tag{3.1}$$

, will clearly have more columns than rows in most cases. Before starting the SVD computation, a boolean-array *trans* of length N is initialised, which is defined as follows for all $i \in \{0, \ldots, N-1\}$:

$$trans[i] := \begin{cases} true, \text{if } I_i > \prod_k I_k, k \neq i, 0 < k < N+1 \\ false, \text{else} \end{cases} \tag{3.2}$$

[1] Recall that the HO-SVD of a tensor $\mathcal{A} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ is obtained with the left singular vectors of the tensor unfoldings, in this case of $A_{(i)}$ for $i \in \{1, \ldots, N\}$.

Using this array, we know where to compute the decomposition of the transposed version for any element in $unf(\mathcal{A})$. We, therefore, need an efficient algorithm to compute the SVD of matrices where the row count is generally much larger than the row count. The first subsection (3.1.1) illustrates a method that relies on a QR factorisation of each tensor unfolding (R-SVD), the second (3.1.2), a method which orthogonalises all column pairs of the input matrix in parallel (Hestenes' method).

3.1.1 A distributed parallel version of the R-SVD Algorithm

The method presented in this subsection can be used to compute the SVD of a given matrix $A \in \mathbb{R}^{m \times n}$ where $m \geq n$ and preferably $m \gg n$, as discussed above. If this is not the given case, computing the SVD of A^T instead leads to the result easily.
This algorithm initially computes the QR factorisation of A in parallel, applying the orthogonal transformations discussed in chapter 2. This step will be discussed in section 3.1.1.2, and at this stage of the computation the algorithm is executed in a completely distributed manner. When the QR factorisation is complete, a bidiagonal matrix B is obtained from R with the R-Bidiagonalisation procedure stated in [Cha82], which is discussed in section 3.1.1.3. The last step, which is explained in section 3.1.1.4, is to obtain a diagonal matrix Σ with the Golub-Kahan SVD step.
While executing the bidiagonalisation and diagonalisation procedures, the factors of the intermediate decompositions have to be accumulated in order to compute the SVD at the end. Notice that the explicit formation of rotation and reflections matrices is not always necessary, and a factorised form can be used to store the result (leading to a better runtime while applying a sequence of rotations and making an economic storage possible). See [GL96] and 3.2 for more details about the perfomance of these operations.
The idea of first computing a QR factorisation and then bidiagonalising R was first introduced in [LH74] and further analised by [Cha82]. The algorithm involving a previous QR factorisation instead of forming a bidiagonal matrix directly (which is the approach followed by the Golub-Reinsch algorithm) is called R-SVD in [GL96], and it shows a better runtime whenever the input matrix A has the property $m \geq 5n/3$ when $A \in \mathbb{R}^{m \times n}$. Table 3.1 (from [GL96]) resumes the runtime complexity of both algorithms [2] , which is dependent on how many of the factors of the SVD are needed. In the case of a HO-SVD computation, the left singular vectors of the tensor unfoldings is needed.
If the left and right factors of the bidiagonalisation and diagonalisation steps were accumulated, as well as the Q factor of the QR decomposition, the SVD can be computed with them and the matrix Σ. The details are discussed below.
This section is mainly based on [BMPS03], [Cha82], [LH74] and [GL96].

3.1.1.1 General idea of the R-SVD Algorithm

The algorithm presented in this section uses a parallel scheme for basis orthogonalisation of a matrix $A \in \mathbb{R}^{m \times n}$ where $m \geq n$ (and optimally, $m \gg n$) on a system with distributed memory. Other steps that can be computed in parallel (but not on a system with distributed memory) are formulated accordingly for making a multithreaded execution possible. This is the given case when intermediate results of a computation are needed globally in order to advance, since this would not be efficient if implemented

[2]In the table, U_1 is the matrix where $A = U_1R$ holds. R is an upper triangular matrix and U_1 is orthogonal (QR decomposition)

Required	Golub-Reinsch SVD	R-SVD
Σ	$4mn^2 - 4n^3/3$	$2mn^2 + 2n^3$
Σ, V	$4mn^2 + 8n^3$	$2mn^2 + 11n^3$
Σ, U	$4m^2n - 8mn^2$	$4m^2n + 13n^3$
Σ, U_1	$14mn^2 - 2n^3$	$6mn^2 + 11n^3$
Σ, U, V	$4m^2n + 8mn^2 + 9n^3$	$4m^2n + 22n^3$
Σ, U_1, V	$14mn^2 + 8n^3$	$6mn^2 + 20n^3$

Table 3.1: Runtime for Golub-Reinsch SVD and R-SVD algorithms

in a distributed manner.
The classical procedure when $m \gg n$, the Modified Gram Schmidt algorithm, unfortunately does not offer effective parallelism for this task [BMPS03]. [3]
The general idea of the algorithm presented in the following subsections can be summarised as follows:

1. Given an initial matrix $A \in \mathbb{R}^{m\times n}$ where $m \geq n$ and optimally $m \gg n$, compute its QR factorisation with the `ROC` algorithm, obtaining $A = QR$. A factorised form for storing the resulting Q factor in this case is not trivial, since both rotations and reflections are involved in the process of forming this matrix. This procedure is partially executed in a distributed manner.
2. Apply the R-Bidiagonalisation procedure ([Cha82]) to R, obtaining $R = U_2 B V_2^H$
3. Compute a diagonal matrix Σ (Golub-Kahan step), such that $B = U_3 \Sigma V_3^H$ holds.
4. Compute $U = U_1 U_2 U_3$ and $V^H = (V_3 V_2)^H$, which leads to $A = U \Sigma V^H$

These steps are described more precisely in sections 3.1.1.2, 3.1.1.3, 3.1.1.4 and 3.1.1.5, respectively. More complete presentations of the mathematical background of these algorithms and their runtime complexity can be found in [GL96].

3.1.1.2 Computing the QR factorisation

Let

$$A_{(i)} \in \mathbb{R}^{I_i \times \prod_{k=1,k\neq i}^{N} I_k}, 0 < i < N+1 \qquad (3.3)$$

be the ith tensor unfolding of a tensor $\mathcal{A} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ and assume w.l.o.g [4] that

$$I_i > \prod_{k=1,k\neq i}^{N} I_k$$

holds. We want to compute an orthogonal matrix $Q \in \mathbb{R}^{I_i \times I_i}$ and an upper-triangular matrix $R \in \mathbb{R}^{I_i \times \prod_{k=1,k\neq i}^{N} I_k}$ such that

$$A_{(i)} = QR \qquad (3.4)$$

[3] Recall that the dimensions of the unfoldings are $I \times JK$, $J \times IK$ and $K \times IJ$ for the first, second and third unfolding respectively if we assume a tensor $\mathcal{A} \in \mathbb{R}^{I\times J\times K}$. Also recall that knowing the SVD of a matrix A implies knowing the SVD of A^T. These facts lead to the interest for the case $m \gg n$, since it will normally be the given case while computing the SVD of tensor unfoldings (and if not, the transposed unfolding can be used as input instead).

[4] Since knowing the SVD of $A_{(i)}^T$ easily leads to the SVD of $A_{(i)}$.

holds for $i \in \{1, \ldots, N\}$.
For the input matrix described above, the algorithm described in section 4.2.1.1 would additionally take an integer value $r_i < \prod_{k=1,k\neq i}^{N} I_k$ as a parameter (r_i is smaller than the column size of $A_{(i)}$) in order to split $A_{(i)}$ into submatrices, which will be processed later in a distributed manner.
Let $g : \mathbb{N} \to \mathbb{N}$ be a function specifying the size of the set of submatrices obtained for each unfolding [5]. We can say that the ith tensor unfolding has $g(i)$ elements, which are denoted as $A_{(i,0)}, \ldots, A_{(i,g(i)-1)}$. The QR factorisation of $A_{(i)}$ can be obtained in two steps according to [BMPS03]:

- `ROCDEC` computes the upper triangular matrix R of the QR factorisation using
    1. A sequence of Householder reflections for each submatrix. When they are applied, $A_{(i,0)}$ is upper triangular and only the first row of each submatrix $A_{(i,1)}, \ldots, A_{(i,g(i)-1)}$ is non-zero.
    2. A set of sequences of Givens rotations zeroes the first row of the submatrices $A_{(i,1)}, \ldots, A_{(i,g(i)-1)}$.
- `ROCVEC` accumulates the orthogonal transformations to compute the vector Qy for any y.

Since the application of the reflections and their computation in `ROCDEC` are truly independent steps, it is possible to execute these procedures in parallel or even as a distributed procedure.
The `ROCVEC` function proceeds as follows: after the R factor of the QR factorisation of $A_{(i)}$ is available ($A_{(i)} = QR = U_1R$), the following steps are realised in order to obtain the factor Q [BMPS03]:
Given the accumulated reflections, if $A_{(i)} \in \mathbb{R}^{m\times n}$ was split into g submatrices we define for each $p \in \{0, \ldots, g-1\}$ a set of matrices

$$H_j = \begin{bmatrix} H_j^{(0)} & & & \\ & H_j^{(1)} & & \\ & & \ddots & \\ & & & H_j^{(g-1)} \end{bmatrix} \tag{3.5}$$

where $H_j^{(p)}$ is the reflection created to zero the jth column in the pth submatrix if $j \in \{1, \ldots, n\}$ and $0 \le p < g$ hold. Additionally, we set

$$G_j = \prod_{i=0}^{g-1} G_{1j}^{(i)} \tag{3.6}$$

for $j \in \{1, \ldots, n\}$, where $G_{1j}^{(p)}$ is the Givens rotation used to annihilate the element $(1, j)$ of the pth submatrix if $0 \le p < g$. The resulting Q factor of $A_{(i)}$ is

$$Q = \prod_{i=1}^{n} H_i G_i \tag{3.7}$$

Notice that in the above equations, the products must be realised in ascending order for the corresponding indices, since the matrix multiplication is *not* commutative.

[5] for instance, if the ith unfolding is split into 5 submatrices, then $g(i) = 5$

At this point, the factors Q and R are explicitly available. If you need further details about this procedure, please refer to [BMPS03]. Notice that the main idea (applying reflectors and reflections to a set of submatrices in order to become the factorisation) is the same as described in [BMPS03], but the implementation details described later in this thesis do not have much in common with it.

3.1.1.3 Computing a bidiagonal matrix

Lawson and Hanson [LH74] and later Chan [Cha82] noticed that when bidiagonalising $A \in \mathbb{R}^{m\times n}$ where $m \gg n$, it is cheaper to compute the QR factorisation of A and then bidiagonalise the factor R whenever $m \geq 5n/3$. Since this will mostly be the given case (and since it will be done after a distributed QR factorisation), an important improvement in perfomance is expected whenever $m \gg n$.
The bidiagonalisation of the upper-triangular $R \in \mathbb{R}^{m\times n}$ matrix is achieved with the Householder Bidiagonalisation [GL96] algorithm, which is illustrated in Figure 3.1. The algorithm, as stated below, overwrites the input matrix A with $B = U_B^T A V_B$ using reflector matrices. Notice that the essential part of the U_j's Householder reflectors is stored in the submatrix $A(j+1:m, j)$ and for the V_j's Householder vectors analogously in $A(j, j+2:n)$, leading to a compact storage possibility of the obtained structures. A factored form representation of the reflections used and the explicit form of the obtained bidiagonal matrix B can be accessed with adequate submatrix procedures. While this procedure does not make use of the fact that R is upper triangular, the computation is much less expensive where the case $m \gg n$ is given, since only the row vectors $A(j, j:n)$ for $j \in \{1, \dots, n\}$ are non zero.
The following matrix sequence illustrates the idea for a matrix $A \in \mathbb{R}^{4\times 4}$ that is not upper triangular (this is not a loss on generality, and the principle is illustrated more clearly for this case). The U_i are used to eliminate the under-diagonal elements of A, the V_i for the row-vectors containing elements with higher column index as the first upper-bidiagonal band (if at row k, the column index must be bigger than $k+1$):

$$\begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \end{bmatrix} \xrightarrow{U_1} \begin{bmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \end{bmatrix} \xrightarrow{V_1} \begin{bmatrix} \times & \times & 0 & 0 \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \end{bmatrix} \xrightarrow{U_2} \begin{bmatrix} \times & \times & 0 & 0 \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & \times & \times \end{bmatrix}$$

$$\xrightarrow{V_2} \begin{bmatrix} \times & \times & 0 & 0 \\ 0 & \times & \times & 0 \\ 0 & 0 & \times & \times \\ 0 & 0 & \times & \times \end{bmatrix} \xrightarrow{U_3} \begin{bmatrix} \times & \times & 0 & 0 \\ 0 & \times & \times & 0 \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \end{bmatrix} = B \in \mathbb{R}^{4\times 4}$$

In this case, the reflections $U_B = U_1 U_2 U_3$ and $V_B = V_1 V_2$ satisfy $U_B^T A V_B = B \in \mathbb{R}^{m\times n}$. This can be used to bidiagonalise arbitrary real matrices. In the case where an upper triangular matrix R must be bidiagonalised, the premultiplication with the kth reflection matrix only changes the sign of the elements in the kth row of R where $k \in \{1, \dots, n\}$, so the application of the reflections should be implemented accordingly.

3.1.1.4 The Golub-Kahan SVD Step

Once the bidiagonalisation of R is achieved (for instance, with the technique described in 3.1.1.3), the remaining problem is to compute the SVD of the resulting matrix. This

```
function HOUSEHOLDER BIDIAGONALISATION($A \in \mathbb{R}^{m\times n}$ with $m \geq n$)
    for $j = 1 : n$ do
        $[v, \beta] \leftarrow getReflector(A(j : m, j))$
        $A(j : m, j : n) \leftarrow (I_{m-j+1} - \beta vv^T)A(j : m, j : n)$
        $A(j + 1 : m, j) \leftarrow v(2 : m - j + 1)$
        if $j \leq n - 2$ then
            $[v, \beta] \leftarrow getReflector(A(j, j + 1 : n)^T)$
            $A(j : m, j + 1 : n) \leftarrow A(j : m, j + 1 : n)(I_{n-j} - \beta vv^T)$
            $A(j, j + 2 : n) \leftarrow v(2 : n - j)^T$
        end if
    end for
    return $A$
end function
```

Figure 3.1: Householder Bidiagonalisation

step is called the Golub-Kahan SVD step.
Consider orthogonal matrices U_Σ and V_Σ where

$$U_\Sigma^T B V_\Sigma = diag(\sigma_1, \ldots, \sigma_n) \in \mathbb{R}^{m\times n} \tag{3.8}$$

holds. Setting $U = U_B U_\Sigma$ and $V = V_B V_\Sigma$ (where U_B and V_B are the left and right factors used to obtain the bidiagonal matrix B, respectively), the SVD of A is $U\Sigma V^T$ [GL96].
The algorithm that can be used for this purpose is illustrated in Figure 3.2. It overwrites B with $\overline{B} = \overline{U}^T B \overline{V}$ where $\overline{U}$ and $\overline{V}$ are orthogonal and $\overline{B}$ is bidiagonal. The new bidiagonal matrix $\overline{B}$ is obtained as follows:

- Using algorithm 3.3, compute the eigenvalue λ of the matrix

$$T(m : n, m : n) = \begin{bmatrix} d_m^2 + f_{m-1}^2 & d_m f_m \\ d_m f_m & d_n^2 + f_m^2 \end{bmatrix}, m = n - 1 \tag{3.9}$$

  where λ is the closest value to $d_n^2 + f_m^2$, if we assume an input matrix of the form

$$B \in \mathbb{R}^{m\times n} = \left[\begin{array}{ccccc} d_1 & f_1 & & \cdots & 0 \\ 0 & d_2 & \ddots & & \vdots \\ & \ddots & \ddots & \ddots & \\ \vdots & & \ddots & \ddots & f_{n-1} \\ 0 & \cdots & & 0 & d_n \\ \hline \multicolumn{5}{c}{0 \in \mathbb{R}^{m-n\times n}} \end{array}\right] \tag{3.10}$$

- Let $c = cos(\theta)$ and $s = sin(\theta)$ such that

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix}^T \begin{bmatrix} d_1^2 - \lambda \\ d_1 f_1 \end{bmatrix} = \begin{bmatrix} \times \\ 0 \end{bmatrix} \tag{3.11}$$

  holds. Set $G_1 = G(1, 2, \theta)$ and compute rotations $G_2, \ldots, G_{n-1}$ so that if $Q = G_1 \cdots G_{n-1}$, then $Q^T T Q$ is tridiagonal and $Qe_1 = G_1 e_1$ [GL96].

These calculations would require the explicit formation of the tridiagonal B^TB, but the rotations can also be applied to B directly. We can apply a sequence of rotations to B, where the non-zero element, which is first introduced by the application of G_1, is eliminated down the diagonal. If B is bidiagonal, the process would generate the following matrix sequence:

$$\xrightarrow{B\leftarrow BG_1}\begin{bmatrix}\times & \times & 0 & 0 & \cdots & \cdots & 0\\ + & \times & \times & 0 & 0 & \cdots & \vdots\\ 0 & 0 & \times & \times & 0 & \cdots & \vdots\\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & 0\\ 0 & \cdots & 0 & 0 & \times & \times & 0\\ 0 & \cdots & \cdots & 0 & 0 & \times & \times\\ 0 & \cdots & \cdots & \cdots & 0 & 0 & \times\end{bmatrix}\xrightarrow{B\leftarrow U_1^TB}\begin{bmatrix}\times & \times & + & 0 & \cdots & \cdots & 0\\ 0 & \times & \times & 0 & 0 & \cdots & \vdots\\ 0 & 0 & \times & \times & 0 & \cdots & \vdots\\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & 0\\ 0 & \cdots & 0 & 0 & \times & \times & 0\\ 0 & \cdots & \cdots & 0 & 0 & \times & \times\\ 0 & \cdots & \cdots & \cdots & 0 & 0 & \times\end{bmatrix}$$

$$\xrightarrow{B\leftarrow BV_2}\begin{bmatrix}\times & \times & 0 & 0 & \cdots & \cdots & 0\\ 0 & \times & \times & 0 & 0 & \cdots & \vdots\\ 0 & + & \times & \times & 0 & \cdots & \vdots\\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & 0\\ 0 & \cdots & 0 & 0 & \times & \times & 0\\ 0 & \cdots & \cdots & 0 & 0 & \times & \times\\ 0 & \cdots & \cdots & \cdots & 0 & 0 & \times\end{bmatrix}\xrightarrow{B\leftarrow U_2^TB}\begin{bmatrix}\times & \times & 0 & 0 & \cdots & \cdots & 0\\ 0 & \times & \times & + & 0 & \cdots & \vdots\\ 0 & 0 & \times & \times & 0 & \cdots & \vdots\\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & 0\\ 0 & \cdots & 0 & 0 & \times & \times & 0\\ 0 & \cdots & \cdots & 0 & 0 & \times & \times\\ 0 & \cdots & \cdots & \cdots & 0 & 0 & \times\end{bmatrix}$$

When the form $B \leftarrow (U_{n-2}^T \dots U_3^T)B(V_3 \dots V_{n-2})$ is reached, the final rotations lead to

$$\xrightarrow{B\leftarrow BV_{n-1}}\begin{bmatrix}\times & \times & 0 & 0 & \cdots & \cdots & 0\\ 0 & \times & \times & 0 & 0 & \cdots & \vdots\\ 0 & 0 & \times & \times & 0 & \cdots & \vdots\\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & 0\\ 0 & \cdots & 0 & 0 & \times & \times & 0\\ 0 & \cdots & \cdots & 0 & 0 & \times & \times\\ 0 & \cdots & \cdots & \cdots & 0 & + & \times\end{bmatrix}\xrightarrow{B\leftarrow U_{n-1}^TB}\begin{bmatrix}\times & \times & 0 & 0 & \cdots & \cdots & 0\\ 0 & \times & \times & 0 & 0 & \cdots & \vdots\\ 0 & 0 & \times & \times & 0 & \cdots & \vdots\\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & 0\\ 0 & \cdots & 0 & 0 & \times & \times & 0\\ 0 & \cdots & \cdots & 0 & 0 & \times & \times\\ 0 & \cdots & \cdots & \cdots & 0 & 0 & \times\end{bmatrix}$$

and all in all:

$$\overline{B} = (U_{n-1}^T \cdots U_1^T)B(G_1V_2 \cdots V_{n-1}) = \overline{U}^T B\overline{V} \tag{3.12}$$

Since $V_i = G(i, i+1, \theta_i)$ for $i = 2 : n-1$, it follows that $\overline{V}e_1 = Qe_1$. $\overline{V}$ $(= Q)$ is the orthogonal matrix that would be obtained by applying the algorithm in Figure 3.3 to $T = B^TB$ [GL96], which is done implicitly by this algorithm.

It is necessary to avoid zero entries in the subdiagonal of B^TB. The subdiagonal entries of B^TB are of the form d_{i-1}, f_i, so the bidiagonal band must be zero free. If $f_k = 0$ for some k, then the SVD problem splits into two smaller problems involving block matrices B_1 and B_2, where

$$B = \begin{array}{cc} \begin{array}{cc} k & n-k \end{array} & \\ \begin{bmatrix} B_1 & 0 \\ 0 & B_2 \end{bmatrix} & \begin{array}{c} k \\ n-k \end{array} \end{array} \tag{3.13}$$

Otherwise, if $d_k = 0$ for $k < n$, premultiplication by a sequence of rotations can zero f_k. A small 6-by-6 example from [GL96] where $d_3 = 0$ illustrates the principle, where rotations in row planes $(3,4),(3,5)$ and $(3,6)$ can be used:

$$B = \begin{bmatrix} \times & \times & 0 & 0 & 0 & 0 \\ 0 & \times & \times & 0 & 0 & 0 \\ 0 & 0 & 0 & \times & 0 & 0 \\ 0 & 0 & 0 & \times & \times & 0 \\ 0 & 0 & 0 & 0 & \times & \times \\ 0 & 0 & 0 & 0 & 0 & \times \end{bmatrix} \xrightarrow{(3,4)} \begin{bmatrix} \times & \times & 0 & 0 & 0 & 0 \\ 0 & \times & \times & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & + & 0 \\ 0 & 0 & 0 & \times & \times & 0 \\ 0 & 0 & 0 & 0 & \times & \times \\ 0 & 0 & 0 & 0 & 0 & \times \end{bmatrix}$$

$$\xrightarrow{(3,5)} \begin{bmatrix} \times & \times & 0 & 0 & 0 & 0 \\ 0 & \times & \times & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & + \\ 0 & 0 & 0 & \times & \times & 0 \\ 0 & 0 & 0 & 0 & \times & \times \\ 0 & 0 & 0 & 0 & 0 & \times \end{bmatrix} \xrightarrow{(3,6)} \begin{bmatrix} \times & \times & 0 & 0 & 0 & 0 \\ 0 & \times & \times & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \times & \times & 0 \\ 0 & 0 & 0 & 0 & \times & \times \\ 0 & 0 & 0 & 0 & 0 & \times \end{bmatrix}$$

If $d_n = 0$, a sequence of column rotations in planes $(n-1,n),(n-2,n),\ldots,(1,n)$ can set the last column to zero [GL96].
The algorithm which is used to determine the eigenvalue of the trailing 2×2 submatrix of $T = B^TB$ that is closer to t_{nn} is illustrated in Figure 3.3. Both algorithms are described precisely in [GL96].
Notice that the matrix $T = B^TB$ needs to be formed explicitly in order to execute these algorithms. A sparse matrix representation should be used, since this matrix is tridiagonal.

```
function GOLUB-KAHAN SVD STEP($B \in \mathbb{R}^{m\times n}$ bidiagonal)
    $\mu \leftarrow QRShift(B^TB)$                              ▷ See Figure 3.3
    $y \leftarrow t_{11} - \mu$
    $z \leftarrow t_{12}$
    for k = 1:n-1 do
        $[c,s] \leftarrow getCandS(y,z)$   ▷ such that $\begin{bmatrix} y & z \end{bmatrix} \begin{bmatrix} c & s \\ -s & c \end{bmatrix} = \begin{bmatrix} * & 0 \end{bmatrix}$
        $B \leftarrow BG(k,k+1,\theta)$
        $y \leftarrow b_{kk}$
        $z \leftarrow b_{k+1,k}$
        $[c,s] \leftarrow getCandS'(y,z)$      ▷ such that $\begin{bmatrix} c & s \\ -s & c \end{bmatrix}^T \begin{bmatrix} y \\ z \end{bmatrix} = \begin{bmatrix} * \\ 0 \end{bmatrix}$
        $B \leftarrow G(k,k+1,\theta)^TB$
        if $k < n-1$ then
            $y \leftarrow b_{k,k+1}$
            $z \leftarrow b_{k,k+2}$
        end if
    end for
    return $B$
end function
```

Figure 3.2: Golub-Kahan SVD step

```
function QRShift($T \in \mathbb{R}^{n\times n}$ tridiagonal)
    $d \leftarrow (t_{n-1,n-1} - t_{nn})/2$
    $\mu \leftarrow t_{n,n} - t_{n,n-1}^2 / \left(d + sign(d)\sqrt{d^2 + t_{n,n-1}^2}\right)$
    $x \leftarrow t_{11} - \mu$
    $z \leftarrow t_{21}$
    for $k = 1 : n-1$ do
        $[c, s] \leftarrow Givens(x, z)$                ▷ See Figure 3.8
        $T \leftarrow G_k^T T G_k$                      ▷ setting $G_k = G(k, k+1, \theta)$
        if $k < n-1$ then
            $x \leftarrow t_{k+1,k}$
            $z \leftarrow t_{k+2,k}$
        end if
    end for
    return $T$
end function
```

Figure 3.3: Implicit Symmetric QR step with Wilkinson Shift

3.1.1.5 The R-SVD algorithm

This section illustrates the sequence of steps needed to compute the SVD with the R-SVD algorithm, which executes the steps described in sections 3.1.1.2, 3.1.1.3 and 3.1.1.4. Implementation details of this procedure with MapReduce are given in section 3.1.1, a pseudo-code version of the algorithm is given in Figure 3.4. Notice that this algorithm explicitly computes all factors of the SVD, which is not always necessary.

```
function R-SVD($A \in \mathbb{R}^{m\times n}$ with $m \geq n$)
    $A \leftarrow QR$                          ▷ ROC
    $B \leftarrow U_2^T R V_2$                 ▷ R-Bidiagonalisation
    $\Sigma \leftarrow U_3^T B V_3$            ▷ Golub-Kahan SVD step
    $U \leftarrow Q U_2 U_3$
    $V^H \leftarrow (V_3 U_2)^H$
    return $U, \Sigma, V^H$
end function
```

Figure 3.4: SVD algorithm

3.1.2 The Hestenes' method

In this section, the Hestenes' method (sometimes also called *Hestenes-Jacobi method*) for computing the SVD of a matrix is presented. We aim to compute the SVD of an $m \times n$ matrix A, where $m \geq n$. When this is not given, the SVD of the matrix A^T is computed instead.

3.1.2.1 The general idea

The idea of the Hestenes' method, as described in [Sha10, BL85], is to generate an orthogonal matrix V such that $W = AV$ has orthogonal columns. After normalising the Euclidean length of each nonzero column to unity, we obtain

$$W = U\Sigma \tag{3.14}$$

In this matrix product, U is orthogonal and Σ is a non-negative diagonal matrix. Since V is orthogonal, we can left-multiply the equation $W = AV$ by V^T, obtaining $A = WV^T$ (since $V^T = V^{-1}$). Because of $W = U\Sigma$, we can replace W in the last equation and get

$$A = U\Sigma V^T \tag{3.15}$$

The difficulty in this method is to find the matrix V. This is done by computing a sequence of plane rotations, each of them orthogonalising a pair of columns until a sweep is completed (see next subsection).

3.1.2.2 Computing the rotations

The Hestenes' method uses plane rotations to construct V. An important fact to notice is that two plane rotations affecting distinct column pairs are truly independent computational steps, thus they could be computed in parallel.
We want to generate a sequence of matrices $\{A_k\}$ where $A_{k+1} = A_k Q_k$, $A_1 = A$ and Q_k is the plane rotation at the kth step. The idea is to orthogonalise a column-pair of the original matrix at each step, making sure to do it in an order that ensures convergence towards an orthogonal matrix and without repeating any pair of columns.
There are some important things to notice about this procedure:

- The plane rotation Q_k affects only two columns with indices i and j of A_k, $a_{k,i}$ and $a_{k,j}$.

- The rotation angle (θ) must be chosen appropriately in order to make $a_{k+1,i}$ and $a_{k+1,j}$ orthogonal.

The plane rotation angle θ is chosen with the formulas of Rutishauser [Rut66], since they "diminish the accumulation of rounding errors" [SA03]. It can be verified that the resulting angle θ always satisfies $\mid \theta \mid \leq \pi/4$.
Let $\alpha = \|a_{k,i}\|_2^2$, $\beta = \|a_{k,j}\|_2^2$ and $\gamma = \langle a_{k,i}, a_{k,j}\rangle = a_{k,i}^T a_{k,j}$ be given. If $\gamma = 0$, the ith and the jth column are already orthogonal and we set $\theta = 0$, in order to skip the corresponding rotation. Otherwise we set

$$\xi = \frac{\beta - \alpha}{2\gamma} \tag{3.16}$$

$$t = \frac{sign(\xi)}{\mid \xi \mid + \sqrt{1+\xi^2}} \tag{3.17}$$

$$cos\theta = \frac{1}{\sqrt{1+t^2}} \tag{3.18}$$

$$sin\theta = t \cdot cos\theta \tag{3.19}$$

The algorithm using this idea is illustrated in Figure 3.6.

3.1.2.3 Orthogonalising all column pairs

Every column-pair $(a^{(i)}, a^{(j)})$ with $i \neq j$ must be orthogonal for the Hestenes' method to work. Unfortunately, not all sequences containing all column pairs (this is normally called a *sweep*) guarantee convergence. A simple sweep like

$$(1,2),(1,3),\ldots,(1,n),(2,3),\ldots,(2,n)(3,4),\ldots,(n-1,n)$$

for instance, ensures convergence, but it is clearly not appropriate for parallel processing [BL85].
In [BL85] a sweep is proposed that works on a one-dimensional systolic multiprocessor array. The presented method obtains all pairs $(i,j), 1 \leq i < j \leq n$ in $O(n)$ steps. A pseudo-code version of the algorithm for generating the sweep can be found in [BL85] and [Sha10]. For a graphical illustration of this procedure see Figure 3.5.
Suppose that n is even (in the case of Figure 3.5 $n = 8$) and we have $P_1,\ldots,P_{\frac{n}{2}}$ processors, where P_k and P_{k+1} can communicate. Assume that every processor has two registers, which can store a column each for the orthogonalisation step. The boxes in Figure 3.5 represent the processors, and the labeled arrows the exchange of columns between step t and $t+1$ for $t \in \{0,\ldots,7\}$. Notice that every column pair occurred exactly once, and no column has been orthogonalised with another column more than once at each time step. If we had an odd number of columns, one "virtual-column" can be added so that the column count is even and we can use the same procedure for the matrix. Every time a column $a^{(i)}$ gets paired with this column, we leave $a^{(i)}$ unchanged.
The limitation of this procedure is the necessity of having $\frac{n}{2}$ communicating processors in order to orthogonalise n columns.

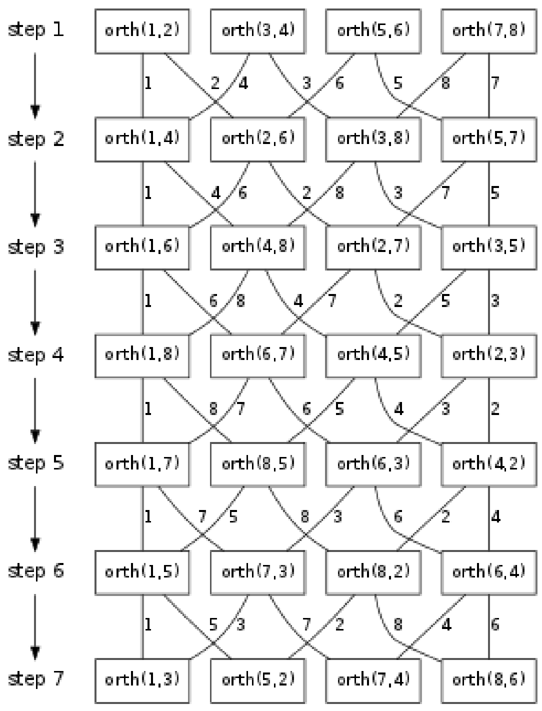

Figure 3.5: Sweep with systolic array for n=8

3.1.2.4 ORTH

The procedure which orthogonalises two columns is executed at each processor. Since it only affects two columns of the actual matrix, there is no need to send the whole matrix through the network after each time step, but only the two affected columns. When this procedure terminates, the "updated" columns must be sent back to a central instance in order to use them for the next orthogonalisation step.
The pseudo-code version of this algorithm [6] can be found in Figure 3.6, and is basically the procedure described with the formulas of Rutishauser [Rut66].
The procedure must orthogonalise both A and V. Since all column-pairs of both matrices must be rotated by the same angle θ, their processing steps are done at the same processor with the same rotation matrix.
Since we have an error tolerance while computing the SVD (HO-SVD) of a large matrix (tensor), a parameter ϵ is passed to the ORTH procedure to determine when a plane rotation should be realised. The term $\gamma = a_{k,i}^T a_{k,j} = \langle a_{k,i} a_{k,j} \rangle$ is, therefore, compared to ϵ in order to check if the columns are "orthogonal enough" for our purposes or not. The procedure returns the updated column-vectors that will be inserted in A_{k+1} and V_{k+1} for further processing. [7]

```
**function** ORTH($a_{k,i}, a_{k,j}, v_{k,i}, v_{k,j} \in \mathbb{R}^{n\times 1}, \epsilon \geq 0$)
    $\gamma \leftarrow a_{k,i}^T a_{k,j}$
    **if** $\mid \gamma \mid > \epsilon$ **then**
        $\alpha \leftarrow \|a_{k,i}\|_2^2$
        $\beta \leftarrow \|a_{k,j}\|_2^2$
        $\xi \leftarrow \frac{\beta - \alpha}{2\gamma}$
        $t \leftarrow \frac{sign(\xi)}{\mid \xi \mid + \sqrt{1+\xi^2}}$
        $cos(\theta) \leftarrow \frac{1}{\sqrt{1+\xi^2}}$
        $sin(\theta) \leftarrow t \cdot cos(\theta)$
        $[a_{k+1,i}, a_{k+1,j}] \leftarrow [a_{k,i}, a_{k,j}] \cdot \begin{pmatrix} cos(\theta) & sin(\theta) \\ -sin(\theta) & cos(\theta) \end{pmatrix}$
        $[v_{k+1,i}, v_{k+1,j}] \leftarrow [v_{k,i}, v_{k,j}] \cdot \begin{pmatrix} cos(\theta) & sin(\theta) \\ -sin(\theta) & cos(\theta) \end{pmatrix}$
    **end if**
    **return** $a_{k,i}, a_{k,j}, v_{k,i}, v_{k,j}$
**end function**
```

Figure 3.6: ORTH Procedure

[6] Notice: while this procedure only takes two pairs of columns that will be orthogonalised ($(a_{k,i}, a_{k,j})$ and $(v_{k,i}, v_{k,j})$), a procedure in a production environment should be able to handle a longer list of column-tuples in order to diminish network traffic.

[7] The ith column of A_{k+1} ($a_{k+1,i}$) stores $a_{k,i} \cdot cos(\theta) - a_{k,j} \cdot sin(\theta)$ in the pseudo-code. Analogously for $a_{k+1,j}$,$v_{k+,i}$ and $v_{k+1,j}$.

3.1.2.5 Normalising W

The last step is to normalise the Euclidean length of every column of W. We get

$$W = U\Sigma \tag{3.20}$$

where Σ is a diagonal matrix of the form

$$\Sigma = \begin{pmatrix} \|w_1\| & 0 & \dots & \dots & 0 \\ 0 & \ddots & & & \vdots \\ \vdots & & \ddots & & \vdots \\ \vdots & & & \ddots & 0 \\ 0 & \dots & \dots & 0 & \|w_n\| \end{pmatrix} \Rightarrow \Sigma^{-1} = \begin{pmatrix} 1/\|w_1\| & 0 & \dots & \dots & 0 \\ 0 & \ddots & & & \vdots \\ \vdots & & \ddots & & \vdots \\ \vdots & & & \ddots & 0 \\ 0 & \dots & \dots & 0 & 1/\|w_n\| \end{pmatrix}$$

where $U = W\Sigma^{-1}$. We are especially interested in computing U_1, U_2 and U_3 (these matrices contain the left singular vectors of $A_{(1)}$,$A_{(2)}$ and $A_{(3)}$, respectively) since we need them for computing the HO-SVD after the Euclidean length of the W_is is normalised (see 3.10).

3.2 Perfomance issues

In this section, some perfomance issues that must be recognised while implementing algorithms using orthogonal transformations are discussed.
The following subsections discuss perfomance issues necessary to effectively employ orthogonal transformations. Ideas for computing, storing and applying these transformations are discussed. Householder reflections are treated in 3.2.1 and 3.2.2, Givens rotations in 3.2.4 and 3.2.5.
The content of this section is based on [GL96].

3.2.1 Computing the Householder vector

In this subsection, an algorithm for computing the Householder vector $v \in \mathbb{R}^n$ that can be used to apply a rotation to a vector $x \in \mathbb{R}^n$ (see Figure 3.7) is presented. The same procedure also returns a scalar β such that if $P = I_n - \beta vv^T$ is the Householder matrix, the equation

$$Px = \|x\|_2 e_1 \tag{3.21}$$

holds (so $\beta = 2/v^T v$).
The first component of v is normalised so that $v(1) = 1$, and the portion $v(2:n)$ of v, which is called the essential part of the Householder vector, is stored later in the main algorithm. For a detailed discussion of this procedure please refer to [GL96].

3.2.2 Applying Householder matrices

When applying a Householder reflection to a matrix, it is critical to exploit the structure of the reflection in order to achieve maximal perfomance. If $A \in \mathbb{R}^{m\times n}$ and a Householder matrix $P = I - \beta vv^T \in \mathbb{R}^{m\times m}$ where $\beta = 2/v^T v$ are given, then we can write

$$PA = (I - \beta vv^T)A = A - vw^T \tag{3.22}$$

```
function HOUSEHOLDER VECTOR($x \in \mathbb{R}^n$)
    $\sigma \leftarrow x(2:n)^T x(2:n)$
    $v \leftarrow \begin{bmatrix} 1 \\ x(2:n) \end{bmatrix}$
    if $\sigma = 0$ then
        $\beta \leftarrow 0$
    else
        $\mu \leftarrow \sqrt{x(1)^2 + \sigma}$
        if $x(1) <= 0$ then
            $v(1) \leftarrow x(1) - \mu$
        else
            $v(1) \leftarrow -\sigma/(x(1) + \mu)$
        end if
        $\beta \leftarrow 2v(1)^2/(\sigma + v(1)^2)$
        $v \leftarrow v/v(1)$
    end if
    return $v, \beta$
end function
```

Figure 3.7: Computing the Householder vector

setting $w = \beta A^T v$. If we want to postmultiply A with a given reflection $P = I - \beta vv^T \in \mathbb{R}^{n\times n}$, we can write similarly

$$AP = A(I - \beta vv^T) = A - wv^T \tag{3.23}$$

with $w = \beta Av$ [GL96]. Having recognised this, applying the reflection involves only a matrix-vector multiplication and an outer product update. Additionally, the fact that $v(1) = 1$ (see 3.2.1) simplifies the application of the reflection.

3.2.3 Storage of reflections

Let a matrix $A \in \mathbb{R}^{m\times n}$ be given, as well as

$$Q = Q_1 Q_2 \cdots Q_r$$

for some $r \leq n$ be given, such that $Q^T A$ is upper triangular. Each reflection in this product has the form

$$Q_j = I - \beta_j v^{(j)} v^{(j)^T}$$

and the entries below the $j+1$-th row for each row j store the essential part of the corresponding Householder vector. Since only the last $n - j1$ entries of $v^{(j)}$ contain the essential part of the Householder vector, and the j-th entry is always 1 after normalising the Householder reflection, the "free room" below the main diagonal in A after applying the reflections is enough for storing the applied reflections. This is called a factorised form representation of Q.
This little trick will be of great relevance for the MapReduce implementation described later in this thesis, since it allows to return the applied reflections efficiently, using only one data structure for each submatrix.
In order to obtain the Q factor explicitly, there are two possibilities to achieve this. The first one is through forward accumulation, which means that Q is initially set to

I_n, and then the sequence of reflection matrices $Q_1, Q_2, \ldots, Q_r$ is applied to Q "from the right". The problem about this procedure is that after setting $Q = QQ_1$ in the first step of the iteration, Q is a full matrix [8] , and the further computation is not as easy as it could be if the fact that Q is initially the identity would be exploited. Thus, the technique of backward accumulation is preferred, since Q gradually becomes full as the iteration progresses. After setting $Q = I_n$, the first iteration of the backward accumulation loop sets $Q = Q_jQ$, then $Q = Q_j - 1$ and so on. The principle, as well as small pseudo-code versions of this techniques are illustrated in [GL96].

3.2.4 Computing the entries c and s of $G(i, k, \theta)$

In this section, the procedure for computing the factors c and s, which are necessary to form a plane rotation, is presented. Notice that the following procedure (illustrated in Figure 3.8) does not compute a matrix with the shape described in 2.4.2, since the application of the rotation does not involve the implicit formation of the rotation matrix (see 3.2.5).
The algorithm presented in Figure 3.8 computes the factors c and s so that the following equation holds:

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix}^T \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix} \tag{3.24}$$

```
function GIVENS($a, b \in \mathbb{R}$)
    if $b = 0$ then
        $c \leftarrow 1$
        $s \leftarrow 0$
    else
        if $|b| > |a|$ then
            $\tau \leftarrow -a/b$
            $s \leftarrow 1/\sqrt{1+\tau^2}$
            $c \leftarrow s\tau$
        else
            $\tau \leftarrow -b/a$
            $c \leftarrow 1/\sqrt{1+\tau^2}$
            $s \leftarrow c\tau$
        end if
    end if
    return $c, s$
end function
```

Figure 3.8: Computing the factors c and s

[8] Recall that the leading $(j-1)$-by-$(j-1)$ portion of Q_j is the identity.

3.2.5 Applying a rotation

While applying a rotation $G(i,k,\theta)$ to a matrix, it is essential to exploit its simple structure. Recall that a rotation $G(i,k,\theta)$ has the form

$$G(i,k,\theta) = \begin{bmatrix} 1 & \dots & 0 & \dots & 0 & \dots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \dots & c & \dots & s & \dots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \dots & -s & \dots & c & \dots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \dots & 0 & \dots & 0 & \dots & 1 \end{bmatrix}$$

Thus, premultiplying a matrix A with $G(i,k,\theta)^T$ will affect only the ith and the kth row of A (this submatrix is denoted as $A([i,k],:)$), so there is no need to initialise a matrix as presented above or to do a complete matrix multiplication, which would dramatically increase the amount of work required for this simple operation dramatically. Similarly to the case mentioned above, only the ith and the kth column of A ($A(:,[i,k])$) would change if a postmultiplication by $G(i,k,\theta)$ is applied.
More precisely, if we want to simulate the matrix product $A = G(i,k,\theta)^T A$ for a matrix $A \in \mathbb{R}^{m\times n}$, we set

$$A(i,j) = cA(i,j) - sA(k,j) \tag{3.25}$$

$$A(k,j) = sA(i,j) + cA(k,j) \tag{3.26}$$

for $j \in \{1,\dots,n\}$.
If the matrix product $A = AG(i,k,\theta)$ has to be simulated instead, set

$$A(j,i) = cA(j,i) - sA(j,k) \tag{3.27}$$

$$A(j,k) = sA(j,i) + cA(j,k) \tag{3.28}$$

for $j \in \{1,\dots,m\}$. These updates take only $6n$ and $6m$ flops, respectively.

3.2.6 Optimal block sizes in the ROC algorithm

While choosing the size of the splitting to be used (r) in the stage where the distributed QR factorisation has to be done, there are a few important things to consider. If we are splitting a matrix $A \in \mathbb{R}^{m\times n}$ with $m \geq n$ and the splitting size $r \in \mathbb{N}$ is given, there will be $\lceil \frac{m-n}{r} \rceil$ rows in the submatrix $A[n+1:m;\ :]$ that have to be set to zero by Givens rotations locally after the distributed procedure terminates. Considering this, we should choose r to be big enough so that the amount of work to do for the central program is not too extensive, but also small enough so that the worker nodes are not excessively occupied computing reflections.

3.3 Parallel computing of the HO-SVD

In this section, two methods for computing the HO-SVD of a 3-dimensional tensor are described.

3.3.1 Parallel HO-SVD with R-SVD

In this subsection, a parallel HO-SVD algorithm is presented (Figure 3.9) which uses the idea discussed in 3.1.1 in order to compute the SVD of the tensor unfoldings.
The algorithm takes as input a N-dimensional tensor $\mathcal{A} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ and a tuple $(i_1, i_2, \ldots, i_N) \in [1, I_1] \times [1, I_2] \times \cdots \times [1, I_N]$, which indicates how many of the left singular vectors composing the SVDs of the unfoldings of $\mathcal{A}$ should be dropped. After the unfoldings $A_{(1)}, \ldots, A_{(N)}$ are computed (and transposed whenever $I_i < \prod_{k=1,k\neq i}^{N} I_k$ holds), the `ROC` procedure upper-triangularises all unfoldings and returns the used orthogonal transformations and the R factors. After this, if the i-th unfolding was transposed at the beginning, the V factor contain the left singular vectors of $A_{(i)}$. In any other case, U_i can be formed by multiplying the resulting orthogonal factor for the QR factorisation, the reflections of the R-Bidiagonalisation step and the rotations of the Golub-Kahan step for the i-th unfolding.
After the U factors are trimmed accordingly to the tuple $(i_1, i_2, \ldots, i_N)$ mentioned above, the remaining computation involves a sequence of n-mode products.

```
function PARALLEL HO-SVD WITH R-SVD($\mathcal{A} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$,$(i_1, i_2, \ldots, i_N) \in [1, I_1] \times [1, I_2] \times \cdots \times [1, I_N]$)
    for $i = 1, \ldots, N$ do
        compute $A_{(i)}$
        if $I_i < \prod_{k=1,k\neq i}^{N} I_k$ then
            $A_{(i)} \leftarrow A_{(i)}^T$
        end if
    end for
    for $i = 1, \ldots, N$ do
        $[U_{i,1}, R_i] \leftarrow A_{(i)}$                                   ▷ ROC
        if $I_i < \prod_{k=1,k\neq i}^{N} I_k$ then
            $[U_{i,2}, B_i, V_{i,2}] \leftarrow R_i$            ▷ R-Bidiagonalization, V explicit
            $[U_{i,3}, \Sigma_i, V_{i,3}] \leftarrow B_i$             ▷ Golub-Kahan step, V explicit
            $U_i \leftarrow (V_{i,3} V_{i,2})^T$
        else
            $[U_{i,2}, B_i, V_{i,2}] \leftarrow R_i$            ▷ R-Bidiagonalization, U explicit
            $[U_{i,3}, \Sigma_i, V_{i,3}] \leftarrow B_i$             ▷ Golub-Kahan step, U explicit
            $U_i \leftarrow U_{i,1} U_{i,2} U_{i,3}$
        end if
    end for
    for $k = 1, \ldots, N$ do
        $U_k \leftarrow U_k[:, 1 : i_k]$
    end for
    $\mathcal{S} \leftarrow \mathcal{A} \times_1 U_1^T \times_2 U_2^T \times_3 \cdots \times_{N-1} U_{N-1}^T \times_N U_N^T$
    $\mathcal{A}_{(i_1, i_2, \ldots, i_N)} = \mathcal{S} \times_1 U_1 \times_2 U_2 \times_3 \cdots \times_{N-1} U_{N-1} \times_N U_N$
    return $\mathcal{A}_{(i_1, i_2, \ldots, i_N)} \in \mathbb{R}^{i_1 \times i_2 \times \cdots \times i_N}$
end function
```

Figure 3.9: Parallel HO-SVD with R-SVD

3.3.2 Combining CubeSVD and Hestenes' method

Shah proposed in [Sha10] a method for computing the HO-SVD of a three-dimensional tensor with a parallel algorithm. It combines the Hestenes' and the CubeSVD methods. While the Hestenes' method is used for computing the SVD (but any non-randomised parallel method could be used instead) of each unfolding of the tensor, the CubeSVD method [SZL+05] is used to obtain the HO-SVD with the left singular vectors of the SVDs. Shah simply calls this algorithm *parallel HO-SVD*.
The algorithm described here takes a N-dimensional tensor as input, as well as N integer values specifying the dimension of the decomposed, resulting tensor.
The algorithm described in [Sha10] looks similar to the pseudo-code in Figure 3.10, which is generalised to the N-dimensional case.

```
function PARALLEL HO-SVD($\mathcal{A} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$,$(i_1, i_2, \ldots, i_N) \in [1, I_1] \times [1, I_2] \times \cdots \times [1, I_N]$)
    for $i = 1, \ldots, N$ do
        compute $A_{(i)}$
    end for
    for $k = 1, \ldots, N$ do
        $[U_k, \Sigma_k, V_k^T] \leftarrow A_{(k)}$                ▷ with Hestenes' method
        $U_k \leftarrow U_k[:, 1 : i_k]$
    end for
    $\mathcal{S} \leftarrow \mathcal{A} \times_1 U_1^T \times_2 U_2^T \times_3 \cdots \times_{N-1} U_{N-1}^T \times_N U_N^T$
    $\mathcal{A}_{(i_1,i_2,\ldots,i_N)} = \mathcal{S} \times_1 U_1 \times_2 U_2 \times_3 \cdots \times_{N-1} U_{N-1} \times_N U_N$
    return $\mathcal{A}_{(i_1,i_2,\ldots,i_N)} \in \mathbb{R}^{i_1 \times i_2 \times \cdots \times i_N}$
end function
```

Figure 3.10: Parallel HO-SVD with Hestenes' method

Chapter 4

MapReduce Algorithm

The present chapter describes an algorithm which can be used to compute the HO-SVD of a tensor in parallel using the MapReduce Framework ([DG08]). For further information about the concepts behind MapReduce (and Apache Hadoop) you can refer to the book [Ven09]. Here I limit myself to explain the very basics of a MapReduce process. The Open-Source MapReduce-Framework Apache Hadoop (`www.hadoop.apache.org`) will be used for this purpose. The main programming language used in the implementation will be Java, and basic matrix operations and classes provided by the Universal Java Matrix Package (`www.ujmp.org`) will be in use. This library will be extended in order to provide advanced operations like the n-mode product, which are needed for the algorithms presented in the last chapter.

4.1 MapReduce basics

The main idea behind MapReduce is to simplify the process of development of distributed parallel applications working on big data sets. The MapReduce concept can be used for writing applications which process vast amonunts of data in parallel on large clusters or commodity hardware, doing this in a reliable, fault-tolerant and efficient manner. In order to achieve this, a corresponding Framework takes care of distributing the tasks to be realised, splitting the input, collecting the results, etc. The Framework uses the Master-Worker Model (or "Master-Worker Architecture"). In order to reach the desired parallelism, a MapReduce job splits the input data set into independent chunks, which are processed by a specified map function in a completely parallel manner. A similar principle applies to the step where the results are collected. The workflow of a MapReduce process is illustrated in Figure 4.1.
The process looks basically like this:

1. The master partitions the problem into sub-problems and assigns its portion of sub-problems to each worker.
2. Each worker receives its workload of sub-problems and works independently of other workers on its part of the solution.
3. The master receives the results obtained by the workers globally, and aggregates them to the final result that will be presented to the user (or the program which uses the actual MapReduce process as a part of its computation).

In the MapReduce process, the user is only responsible for specifying the location and format of the input, a *Map* function, a *Reduce* function, and the location and format of

the output. This way the programmer does not have to take care of the advanced issues involved with communication and monitoring present in every distributed computation and can instead work at a higher level of abstraction.
In every MapReduce program, the *Map* and *Reduce* functions have the following purposes [DG08]:
The *Map* function, written by the user, ”takes an input pair and produces a set of *intermediate* key/value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key I and passes them to the *Reduce* function”.
”The *Reduce* function, also written by the user, accepts an intermediate key I and a set of values for that key. It merges together these values to form a possibly smaller set of values. Typically just zero or one output value is produces per *Reduce* invocation. The intermediate values are supplied to the user's reduce function via an iterator. This allows us to handle lists of values that are too large to fit in memory”.

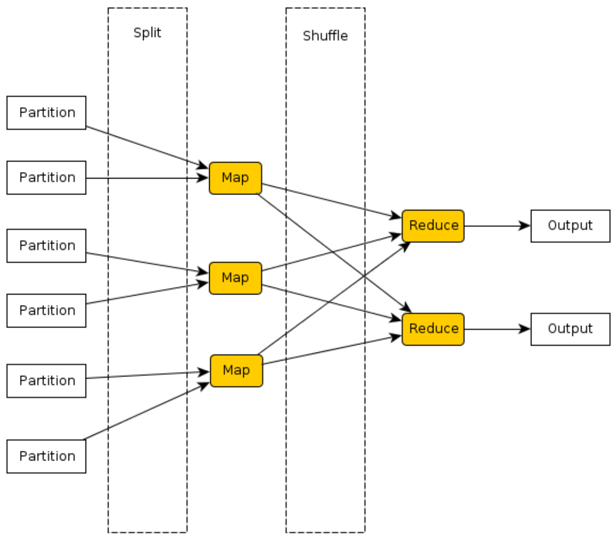

Figure 4.1: Workflow of a MapReduce process

4.2 Ideas for parallel HO-SVD with MapReduce

The next subsections discuss the implementation details of a parallel HO-SVD algorithm using the two approaches described in chapter 3.

4.2.1 The Golub-Reinsch Algorithm

In this subsection, a MapReduce implementation of the algorithm described in 3.1.1 is discussed. Since the Framework carries out parallel computations without having the possibility of sharing intermediate results between worker-nodes, not every step can be done in parallel without being inefficient (compared to the performance of a single- or multi-threaded procedure on a single machine). Only computations that are completely independent steps can be executed in a distributed manner efficiently.
Given a matrix $A_{(i)} \in \mathbb{R}^{m\times n}$, which is the ith tensor unfolding of a tensor with higher order than i, a set of block matrices is produced by splitting $A_{(i)}$ (see 4.2.1.1). The

notation used for specifying these blocks will be $A_{(i,0)}, A_{(i,1)}, \dots, A_{(i,r)}$ if we partition in r submatrices. There are a few limitations while choosing the size of the $A_{(i,k)}, k \in \{0, \dots, r\}$. While $A_{(i,0)} \in n \times n$ is the most intuitive choice, the row-size of this submatrix could also be bigger than n but not smaller (see 3.1.1). The row-size of each submatrix $A_{(i,k)}$ must be bigger than one (otherwise no reflections could be applied to the submatrices).
I describe a MapReduce process, `HouseholderMR`, which is used to zero the elements below the main diagonal for the first block, and all elements below the first one for any column and any other block. Next, the `GivensForRows` procedure is described, which is used to zero the rows that were not zeroed by `HouseholderMR`. This is an iterative process. Since the Givens rotations (applied from left) modify two rows of a matrix, it is not possible to use the kth row of the first block for zeroing out the entry at the kth column of two different rows that were not zeroed by `HouseholderMR` independently. Since the modification of the row also influences the elements with column indices j, where $k < j \leq n$, their rotations cannot be computed until the rotation for treating the element in the kth column is complete.

4.2.1.1 `MatrixSplitter`

The `MatrixSplitter` algorithm is used to obtain a suitable work distribution among available machines. There are a few things to consider while selecting the splitting size to be used. These are discussed in section 3.2.6.
When the size of the splits to be obtained is determined, this simple algorithm proceeds as specified in Figure 4.2. The list returned by `MatrixSplitter` specifies the splitting positions to be used.

```
function MatrixSplitter($A \in \mathbb{R}^{m \times n}$ with $m \geq n, r \in \mathbb{N}$)
    $splittingPositions \leftarrow []$                    ▷ List of Integers
    $splittingPositions.append(n)$                ▷ First block is in $n \times n$
    $i \leftarrow n + r$
    while $i < m$ do
        $splittingPositions.append(i)$
        $i \leftarrow i + r$
    end while
    return $splittingPositions$
end function
```

Figure 4.2: `MatrixSplitter`

4.2.1.2 `HouseholderMR`

Given a set of submatrices obtained by applying the `MatrixSplitter` procedure to a given tensor unfolding, `HouseholderMR` computes a sequence of rotations which leads to an almost upper triangular shape of the original matrices in a distributed manner.
The output of `MatrixSplitter` for all unfoldings of a given tensor is stored in a single text file, where each line represents a key-value pair (the class `KeyValueTextInputFormat` of the Hadoop Framework will be the input of the Map function). The key is a single `LongWritable` which identifies the unfolding-ID and the block-ID (if we have r blocks, we have block-IDs $0, \dots, r-1$) of the submatrix that will be the value (of type `Text`) of the corresponding pair.

In order to encode a pair of natural numbers, a simple pairing function [Ros03] will be used. The bijective Cauchy/Cantor pairing function $\pi : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$ [1] is given by

$$\pi(x,y) := y + \sum_{i=0}^{x+y} i = y + \frac{(x+y)(x+y+1)}{2} \tag{4.1}$$

The remaining question is how to get the encoded pair of natural numbers back. Since π is a bijection, we need $\pi^{-1} : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$. Let the functions

$$g : \mathbb{N}_0 \to \mathbb{N}_0, (n) \to \sum_{i=0}^{n} i \tag{4.2}$$

and

$$f : \mathbb{N}_0 \to \mathbb{N}_0, (n) \to max\{v | g(v) \leq n\} \tag{4.3}$$

be given. It can be proven that the original pair of any encoded natural number pair z is

$$(z_x = \pi_x(z), z_y = \pi_y(z)) \tag{4.4}$$

, where

$$\pi_x(z) = f(z) - \pi_y(z), \pi_y(z) = z - g(f(z)) \tag{4.5}$$

Table 4.1 illustrates how the pairs of natural numbers are encoded. Notice how the diagonals of the table are filled in ascending order with natural numbers, which are unique for a given pair of natural numbers determining the coordinates.

x/y	**0**	**1**	**2**	**3**	**4**	...
0	0	2	5	9	14	...
1	1	4	8	13	...	...
2	3	7	12	...	...	...
3	6	11	...	...	...	...
4	10	...	...	...	...	...
⋮	...	...	...	...	...	...

Table 4.1: Cauchy/Cantor pairing function

Assuming the Map of `HouseholderMR` becomes a key-value pair (k, v), it will do the following:

- If $\pi_y(k)$ equals 0, compute an upper triangular matrix, starting with the matrix which is encoded with v, using Householder reflectors. The resulting matrix and the Householder vectors that were used are emmited as a key-value pair (k, v'), where the matrix represented by v' is upper triangular and the used Householder vectors are stored in the under-diagonal part of v'.

- If $\pi_y(k) > 0$, proceed as above, but apply and store reflections that eliminate the entries with row index bigger than one.

[1] $\mathbb{N}_0 := \mathbb{N} \cup \{0\}$

After the intermediate results of the Maps are accumulated (using the values of $\pi_x(k)$ and $\pi_y(k)$ for any key k in order to restore the initial order), the process is essentially complete. For instance, if $A_{(i)}$ has 4 columns and the submatrices created by `MatrixSplitter` are square matrices, it would have the shape

$$A_{(i)} = \left[\begin{array}{cccc} \cdot & \cdot & \cdot & \cdot \\ 0 & \cdot & \cdot & \cdot \\ 0 & 0 & \cdot & \cdot \\ 0 & 0 & 0 & \cdot \\ \hline \cdot & \cdot & \cdot & \cdot \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \hline \cdots & \cdots & \cdots & \cdots \\ \hline \cdot & \cdot & \cdot & \cdot \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array}\right]$$

after the reflections are extracted.

The Householder vectors that were used are accumulated in order to compute the Q factor of the decomposition later. We use an `IdentityReducer`, since no reducing step is necessary. In fact, the output files produced by `HouseholderMR` will be used directly by the `GivensForRows` procedure, which executes the next step in the computation of the QR decomposition (4.2.1.3).

4.2.1.3 `GivensForRows`

In this section, the functionality of the `GivensForRows` procedure is described. Let

$$A_{(i)} \in \mathbb{R}^{m\times n} = \left[\begin{array}{cccc} \cdot & \cdot & \cdot & \cdot \\ 0 & \cdot & \cdot & \cdot \\ 0 & 0 & \cdot & \cdot \\ 0 & 0 & 0 & \cdot \\ \hline \cdot & \cdot & \cdot & \cdot \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \hline \cdots & \cdots & \cdots & \cdots \\ \hline \cdot & \cdot & \cdot & \cdot \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array}\right] = \left[\begin{array}{c} A_{(i,0)} \\ \hline A_{(i,1)} \\ \hline \cdots \\ \hline A_{(i,r)} \end{array}\right]$$

be the matrix that resulted after executing the `HouseholderMR` on the ith tensor unfolding of a tensor. For each non-zero row $A(i,:)$ where $i > n$, the procedure described in this section will do the following: Use the first row of the first block for computing the Givens rotation that will annihilate the first element of the first row of $A(i,:)$, then do the same (but with the updated first and ith rows, since the rotation affects these portions of the matrix) for the second element, and so on.

More precisely, let $P \subset \mathbb{N}$ be the set of row-indices of $A_{(i)}$ where $A_{(i)}[k,:] \neq 0 \forall k \in P$, and assume no index in P is lower than n. The sequence of rotations realised by `GivensForRows` is the following: For $l = 1 \dots n$

- Rotate rows $A_{(i,0)}[l,:]$ and $A_{(i,z)}[k,:]$ for $k \in P$, zeroing the entry $A_{(i,z)}[0,l]$ if $A_{(i,z)}$ contains the kth row of $A_{(i)}$.

The general idea of `ROCDEC` is illustrated below for the 12-by-4 case:

$$\left[\begin{array}{cccc} \cdot & \cdot & \cdot & \cdot \\ 0 & \cdot & \cdot & \cdot \\ 0 & 0 & \cdot & \cdot \\ 0 & 0 & 0 & \cdot \\ \hline \cdot & \cdot & \cdot & \cdot \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \hline \cdot & \cdot & \cdot & \cdot \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array}\right] \xrightarrow[l=1]{G(1,5)} \left[\begin{array}{cccc} * & * & * & * \\ 0 & \cdot & \cdot & \cdot \\ 0 & 0 & \cdot & \cdot \\ 0 & 0 & 0 & \cdot \\ \hline 0 & * & * & * \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \hline \cdot & \cdot & \cdot & \cdot \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array}\right] \xrightarrow[l=1]{G(1,9)} \left[\begin{array}{cccc} * & * & * & * \\ 0 & \cdot & \cdot & \cdot \\ 0 & 0 & \cdot & \cdot \\ 0 & 0 & 0 & \cdot \\ \hline 0 & \cdot & \cdot & \cdot \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \hline 0 & * & * & * \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array}\right] \xrightarrow[l=2]{G(2,5)} \left[\begin{array}{cccc} \cdot & \cdot & \cdot & \cdot \\ 0 & * & * & * \\ 0 & 0 & \cdot & \cdot \\ 0 & 0 & 0 & \cdot \\ \hline 0 & 0 & * & * \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \hline 0 & \cdot & \cdot & \cdot \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array}\right] \cdots \xrightarrow[l=4]{G(4,9)} R$$

After this procedure terminates and the rotations that were used are accumulated, an upper-triangular matrix R has been obtained. The files containing $A_{(i,0)}, \dots, A_{(i,r)}$ are rewritten, and the essential part of the used rotations are stored below the main diagonal for the first split and below the first row for all other blocks. The following sequence illustrates the principle for a 9×3 matrix A, where the splitting size $g = 2$ is used:

$$\left[\begin{array}{ccc} * & * & * \\ * & * & * \\ * & * & * \\ * & * & * \\ * & * & * \\ * & * & * \\ * & * & * \\ * & * & * \\ * & * & * \end{array}\right] \xrightarrow{\textit{split}} \left[\begin{array}{ccc} * & * & * \\ * & * & * \\ * & * & * \\ \hline * & * & * \\ * & * & * \\ \hline * & * & * \\ * & * & * \\ \hline * & * & * \\ * & * & * \end{array}\right] \xrightarrow{\text{ROCDEC}} \left[\begin{array}{ccc} * & * & * \\ 0 & * & * \\ 0 & 0 & * \\ \hline * & * & * \\ 0 & 0 & 0 \\ \hline * & * & * \\ 0 & 0 & 0 \\ \hline * & * & * \\ 0 & 0 & 0 \end{array}\right] \xrightarrow{\text{ROCVEC}} \left[\begin{array}{ccc} * & * & * \\ 0 & * & * \\ 0 & 0 & * \\ \hline 0 & 0 & 0 \\ 0 & 0 & 0 \\ \hline 0 & 0 & 0 \\ 0 & 0 & 0 \\ \hline 0 & 0 & 0 \\ 0 & 0 & 0 \end{array}\right]$$

The last step is to compute Q [2]. This is done as described in 3.1.1.1.
Figure 4.3 describes the process graphically. Here, I assume that

- The `MatrixSplitter` procedure splits the ith unfolding to g_i submatrices.
- The QR algorithm is executed locally for the first submatrix of each unfolding in order to reach the upper-triangular shape at the end.
- It holds for any unfolding $A_{(i)} \in \mathbb{R}^{m_i \times n_i}$ that $m_i \geq n_i$, so after `ROCDEC` only the V_i factors have to be computed explicitly.

[2] Notice that the explicit computation of Q can be *very* expensive if m is large. In order to remain efficient, this factor is only computed explicitly if the left singular vectors of A are needed in the overall computation. In the case of a HO-SVD computation, this will not be the most common case (consider how the dimensions of the tensor unfoldings are determined).

4.2.1.4 Testing the `UJMP`-class `SVDMatrix`

In this section, a comparison of the runtime necessary for computing the SVD of large sparse matrices with more rows than columns using the `SVDMatrix` class of the `UJMP` package is presented. This following analysis is interesing for this thesis since

- Sets of very large and sparse matrices will be the input of the MapReduce procedure previously described.
- The output of the previously described MapReduce procedure is also a set of very large, upper-triangular, sparse matrices.

It is interesting to observe that the computation of the SVD, when only the left singular vectors are needed, is considerably faster when the matrix has been upper-triangularised before, as can be seen in table 4.2. This kind of computation is necessary to form the HO-SVD of a tensor, as discussed in earlier chapters.

Matrix	SVDM	SVDM-R
$\mathbb{R}^{2000\times1000}$	01:17	00:42
$\mathbb{R}^{3000\times1000}$	02:09	00:43
$\mathbb{R}^{4000\times1000}$	02:32	00:42
$\mathbb{R}^{5000\times1000}$	02:59	00:44
$\mathbb{R}^{6000\times1000}$	03:30	00:41
$\mathbb{R}^{7000\times1000}$	04:12	00:42
$\mathbb{R}^{8000\times1000}$	04:50	00:42
$\mathbb{R}^{9000\times1000}$	05:32	00:42
$\mathbb{R}^{10000\times1000}$	06:47	00:43
$\mathbb{R}^{11000\times1000}$	07:19	00:43

Table 4.2: Runtime comparison of SVD computations, with and without previous QR decomposition. The right singular vectors were computed explicitly.

The test which returned the results presented in table 4.2 generated an array containing 10 matrices in the specified dimensions, where approximately 1% of the entries were non-zero. The second column of the table shows the time needed to compute the SVD, obtaining the factors Σ and V explicitly, if the corresponding matrix in the matrix array is not upper-triangularised first. The third column shows the results if the `SVDMatrix` class is used to compute the SVD after the `ROC` procedure upper-triangularised the corresponding input matrix.
Notice that *only* the time used to compute the SVD is considered for the third column in the table. Since the QR decomposition can be computed in a distributed manner, this method will be faster anyway if the input matrices have the property $m \gg n$.
While the time necessary for computing the SVD steadily increases if the input matrices aren't upper-triangular, we can see that the time is more or less constant if the input matrices are already upper triangular. This fact is critical when the input matrices are tensor unfoldings of large tensors!
As an example consider a tensor $\mathcal{A} \in \mathbb{R}^{10000\times20000\times100000}$. If we aim to compute the HO-SVD of $\mathcal{A}$, we need the U factors of the SVDs of $A_{(1)}, A_{(2)}$ and $A_{(3)}$. Since all unfoldings will have more columns than rows [3], the SVD of $A^T_{(1)}, A^T_{(2)}$ and $A^T_{(3)}$ must

[3]Recall that the dimensions of the unfoldings are $I \times JK$, $J \times IK$ and $K \times IJ$ for the first, second and third unfolding respectively if we assume a tensor $\mathcal{A} \in \mathbb{R}^{I\times J\times K}$.

be computed [4], and the left singular vectors of all computations are needed explicitly. Clearly, we will save much time and computational resources if the decompositions $A_{(i)} = Q_i R_i$ can be computed distributedly (which is done by the `ROC` algorihm), and then use the R_i as input to the SVD procedure, which will only compute the V_i factors explicitly.

4.2.1.5 Algorithm summary

This is the summary of the MapReduce algorithm for computing the HO-SVD of a 3-dimensional [5] tensor involving the use of `HouseholderMR` and `GivensForRows`, which has some steps that are the same as described in 4.2.2, but also some important differences:

1. An initial 3-dimensional tensor $\mathcal{A}$ is taken as input.

2. The matrix-unfoldings of $\mathcal{A}$ (which are denoted as $A_{(1)}$, $A_{(2)}$ and $A_{(3)}$) are determined.

3. If the column-size of a given tensor unfolding is smaller than its row-size, transpose it before computing its SVD [6].

4. For each tensor unfolding $A_{(i)}$, compute its (full) QR decomposition

$$A_{(i)} = Q_i R_i$$

   using the previously described `HouseholderMR` and `GivensForRows` procedures.

5. Compute bidiagonal matrices B_i starting from R_i, using Householder reflections (R-Bidiagonalisation) for this. We get

$$R_i = U_{2,i} B_i V_{2,i}^H$$

6. Starting with the B_i, compute diagonal matrices Σ_i using Givens rotations (Golub-Kahan SVD step). We get a factorisation

$$B_i = U_{3,i} \Sigma_i V_{3,i}^H$$

7. At this point, we compute the SVD of $A_{(i)}$, setting

$$A_{(i)} = Q_i U_{2,i} U_{3,i} \Sigma_i (V_{3,i} V_{2,i})^H = U_i \Sigma_i V_i^T$$

8. For each tensor unfolding $A_{(i)}$ where we computed the SVD of its transpose, swap the content of U_i and V_i. Alternatively, compute only V_i for unfoldings if the original row count was smaller than the column count, and only U_i if this was not the given case.

9. Compute the core tensor of $\mathcal{A}$, which is defined as

$$\mathcal{S} = \mathcal{A} \times_1 U_1^T \times_2 U_2^T \times_3 U_3^T$$

[4] Also recall that knowing the SVD of a matrix A implies knowing the SVD of A^T.

[5] For higher-dimensional tensors, the procedure can of course be generalised

[6] $A^T = U\Sigma V^T \Leftrightarrow A = V\Sigma U^T$

10. Obtain the HO-SVD of $\mathcal{A}$, obtained by setting

$$\mathcal{A}' = \mathcal{S} \times_1 U_1 \times_2 U_2 \times_3 U_3$$

For a more compact representation of this algorithm, see Figure 3.9 (here, the algorithm is generalised to the N-dimensional case).

4.2.2 Using the Hestenes' method

The idea behind the HO-SVD-algorithm using the Hestenes' method for SVD computations can be described by the following sequence:

1. An initial 3-dimensional tensor $\mathcal{A}$ is taken as input. We want to compute its approximation in form of a HO-SVD.
2. The matrix-unfoldings of $\mathcal{A}$ (we refer to them as $A_{(1)}$, $A_{(2)}$ and $A_{(3)}$) are determined.
3. For each unfolding $A_{(i)} \in \mathbb{R}^{m \times n}$, a corresponding exact copy $A_{i,1}$ ($i \in \{1, 2, 3\}$) is created as well as a unitary matrix $V_{i,1} \in \mathbb{R}^{n \times n}$. These matrices will later converge to W and to V, respectively.
   Both matrices will be updated while the orthogonalisation of its columns is processed. This will happen in a centralised manner (single point of failure), since every column-pair must be processed exactly once by the `ORTH` procedure and we need to accumulate the results of intermediate steps.
4. The $A_{i,1}$ and the $V_{i,1}$ matrices are partitioned into $r \in \mathbb{N}$ column-groups. Since the column-size of $A_{i,1}$ and $V_{i,1}$ is equal (n), the same amount of groups is generated for each matrix.
5. Distribute column-group pairs for parallel orthogonalisation (using the previously described `ORTH` procedure). The plane rotations are computed and the columns of the groups are updated. After $c - 1$ steps (if c is the column-group size of the groups), the groups consist of column-sets that are (internally) orthogonal. I call this the *group-internal orthogonalisation step*, and the sequence in which the column-pairs are orthogonalised is determined locally at each worker machine (Sweep class).
6. Every group-pair is processed in order to make their columns orthogonal. This is possible after the results of the last step have been accumulated.
   The group pairs to be processed at each time step are determined centrally, for instance, with the previously discussed Sweep procedure. After processing two groups, say G_i and G_j, any column in G_i is orthogonal to all columns in G_j and vice versa. I call this *group-pair orthogonalisation step.*
   The order in which the columns of a group-pair is orthogonalised is described in Figure 4.4. Notice that in this procedure every column-pair is processed exactly once. In the mentioned figure, the left component comes from the blue group, and the right component comes from the red group. This procedure also works for groups of different sizes.
   The group-pair orthogonalisation function (as illustrated in Figure 4.4) takes at least two pairs of columns as arguments, where the first pair comes from the

matrix A_k and the second pair from the matrix V_k. In this case, only one plane rotation takes place.

7. At this point, all column-pairs are accumulated to their corresponding matrix. Since $A_{(i,k)} \to W_i$ and $V_{(i,k)} \to V_i$, we obtain

$$A_i = W_i V_i^T$$

for $1 \geq i \geq 3$.

8. After normalising the Euclidean length of every nonzero column in $W_{1 \geq i \geq 3}$, we get the corresponding U_i and the Σ_i as previously described. The left singular vectors of A_i are stored at U_i.

9. Compute the core tensor of $\mathcal{A}$, which is given by the product

$$\mathcal{S} = \mathcal{A} \times_1 U_1^T \times_2 U_2^T \times_3 U_3^T$$

10. Obtain the HO-SVD of $\mathcal{A}$, setting

$$\mathcal{A}' = \mathcal{S} \times_1 U_1 \times_2 U_2 \times_3 U_3$$

For a more compact representation of this algorithm, see Figure 3.10 (here, the algorithm is generalised to the N-dimensional case). There are important perfomance-related issues concerning this approach however. Consider for instance the partitioning properties of this algorithm. If we aim to achieve high parallelism (that is, to generate many submatrices after the initial split), the amount of intermediate results increments notably. As a consequence of this, there is a tradeoff between parallelism and amount of intermediate results, which is not handled efficiently with the MapReduce approach. This fact leads to the preference for the method involving the `ROC` as distributed-SVD algorithm, since no intermediate results are necessary. Additionally, the only factors which are needed for a HO-SVD computation can be computed with ease after `ROC` has been applied (since the computation of the SVD, where only the V factor is computed explicitly, is easier when the input matrix is upper-triangular-shaped).

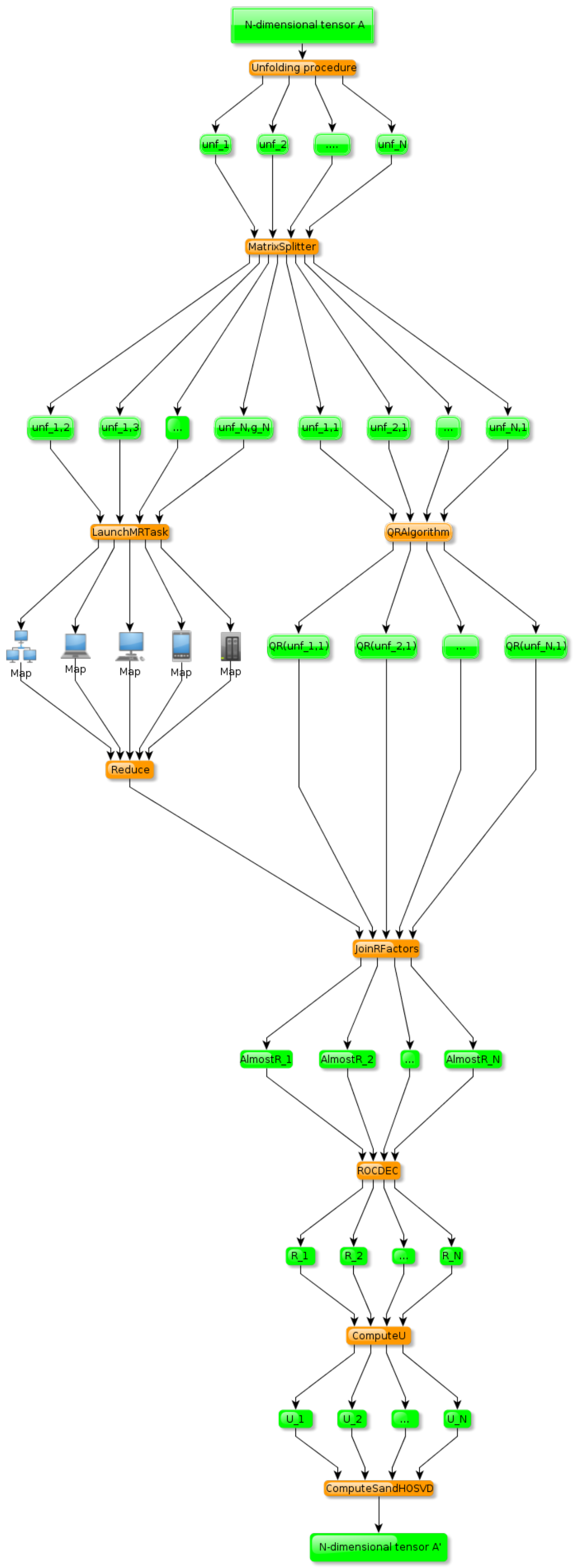

Figure 4.3: HO-SVD computation with R-SVD and `ROC` steps.

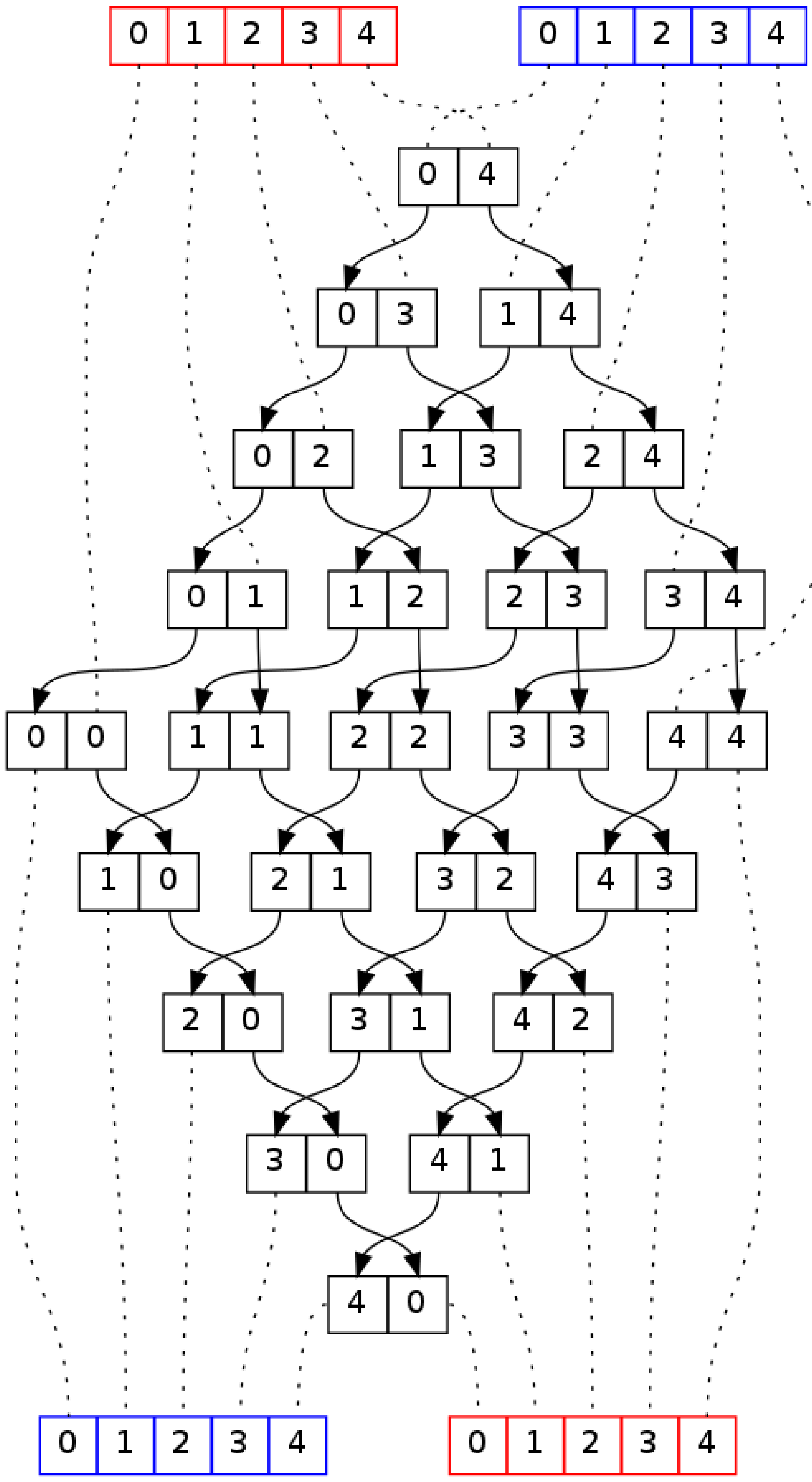

Figure 4.4: Two-Group Orthogonalisation.

Chapter 5

Latent Semantic Indexing

This chapter gives a short introduction to the topic of Latent Semantic Analysis. The content is mainly based on [DDF$^+$90].
The ideas expressed below, although formulated for the two-dimensional case (where a set of documents and a vocabulary of terms is considered), are also applicable to multidimensional data in analogous fashion after the Higher-Order Singular Value of the corresponding data structure is available.

5.1 Basic concepts

In this section some basic concepts from the information retrieval field are explained.

5.1.1 Information Retrieval

In the field of Information Retrieval, the ideal result for a given query would be the following: for the given set of documents contained in our database, find the subset containing all and only the relevant documents. The documents contain, in the sense of classical information retrieval, text in some natural language.
The most common representation of the set of documents and the terms contained in them is achieved with a term-document matrix. If our collection of documents, call it D, has n documents $(d_1 \dots d_n)$ containing m terms $(t_1 \dots t_m)$ to be indexed (which means, we consider these terms to be relevant, semantically speaking), the representation of choice is illustrated in Table 5.1, where $f(t_i, d_j)$ for $i \in \{1, \dots, m\}, j \in \{1, \dots, n\}$

	d_1	d_2	$\dots$	d_n
t_1	$f(t_1, d_1)$	$f(t_1, d_2)$	$\dots$	$f(t_1, d_n)$
t_2	$f(t_2, d_1)$	$f(t_2, d_2)$	$\dots$	$f(t_2, d_n)$
$\vdots$	$\vdots$	$\vdots$	$\dots$	$\vdots$
t_m	$f(t_n, d_1)$	$f(t_n, d_2)$	$\dots$	$f(t_n, d_n)$

Table 5.1: Term-Document matrix

determines how often the term t_i occurs in the document d_j. Issues related to the storing of this structure will not be discussed here, but it is clear that this matrix will in most cases be extremely sparse since a single document will contain only a very small fraction of the available term vocabulary of all documents in the collection.

5.1.2 Evaluating the quality of a result

There are mainly two quality measuring principles when analysing the perfomance of an information retrieval system. We can say that an algorithm has high *precision* if most of the retrieved documents are relevant. We say that an algorithm has high *recall* if most of the relevant documents existing in the database are retrieved. If you are interested in a comparative analysis of approaches for defining precision and recall, you can refer to [RJB89].

5.1.3 Problems of traditional Information Retrieval systems

One of the deficiencies of traditional information retrieval systems has its origin in the fact that users of the system very often use words to formulate their queries that are not the same as those by which the information retrieval system has indexed the underlying document database.
For example, consider a user that typed in a query containing the term "cat". It could refer to

- The mammal.
- The Caterpillar Inc., also known as "CAT".
- The Unix-Command `cat`.
- ...

We can call this the synonymity problem. After the normalisation step, many terms indexed by the information retrieval system could be classified as equivalent, so the terms $C.A.T$, CAT, cat would be in the same equivalence class. This would lead to decreasing recall perfomance according to [DDF$^+$90].
A further problem arises when identical terms are used by users in different contexts. This means that if a term is contained in a given query, it does not necessarily follow that a document in our database where this term occurs will be relevant for the user. This fact can be referred to as the polysemic problem, and it is very likely to lead to a perfomance reduction if we observe the obtained precision level.

5.2 The vector space model

The issues discussed in the section above motivate the use of a different approach for filtering relevant documents out of a collection. The vector space model is an algebraic model where documents are represented as vectors, where the vector components contain indexed terms. It is even possible to represent space the relationships of any kind of objects in a vector.
But where are such structures necessary? Consider two users A and B searching for the same term, but having different semantics in mind for the query (the polysemic problem discussed in the last section). In a traditional information retrieval system, the terms would be treated as identical (no consideration of the semantics of the term would take place), and A and B would get the same results. Since not every document in the retrieved set will be relevant for A, a manual post-filtering of the results will be necessary. If this is not possible because the resulting set is too large, the query could be reformulated along with some additional contextual information.
The most intuitive solution to this problem would be to personalise the search. Indeed,

in the field of personalised web search, the information retrieval system takes into consideration earlier queries of a user. In this scenario, we have a third dimension that has to be considered: the *users*. The relations are represented by triples, so it is intuitive to use a three-dimensional array (or equivalently, a third-order tensor).
The next subsections will discuss the techniques used in this field to "understand" the underlying latent semantics of two-dimensional structures (like a term-document matrix) and of higher-dimensional structures (like a third-order tensor with the dimensions *user*, *document* and *term*).

5.2.1 The Singular Value Decomposition

The Singular Value Decomposition (SVD) can be used to reveal the semantics underlying two-dimensional structures. Consider a term-document matrix T having m rows and n columns, which represents the indexing of a set of m documents including n different terms. Each document in our collection is represented by a row in T, each term is represented by a column in T.
While computing the SVD of T, we aim to approximate T in such a way that we have to consider fewer dimensions while measuring term-term, document-document and/or term-document similarity. The obtained reduced form of T, T', is obtained as described in section 2.1 of this thesis (see reduced SVD).
We aim to escape the "*unreliability, ambiguity and redundancy of individual terms as descriptors*" [DDF$^+$90], which is the main problem of traditional information retrieval systems. Possible ways to measure similarity in this new representation will be discussed below.

5.2.1.1 The dot product as similarity measure

One possibility to measure similarity between objects represented by our SVD is given by the *dot product.* [1] More precisely, the similarity of two terms t_i and t_j for $i, j \in \{1, n\}$ is given by the dot product of the corresponding columns of T', which is the approximation of T. The similarity of two documents d_k and d_l for $k, l \in \{1, m\}$ is analogously defined. In order to find out the relevance of a term t_j in a document d_i, the element $T[i, j]$ has to be considered.

5.2.1.2 The cosine similarity

It is also possible to define the similarity of two objects of the same type (two documents, terms, items, users, etc) in our SVD reprentation with the *cosine similarity.* Consider an $m \times n$ matrix $A = U\Sigma V^T$ and the corresponding thin SVD $A' = U_k \Sigma_k V_k^T$ (where we kept the k largest singular values $\sigma_1, \ldots, \sigma_k$ of the original Σ, for a given $k \in \{1, \ldots, rank(A)\}$). The computation of the SVD has the following effect [Sha10]: the axes of the m' dimensional column space of A are rotated in such a way that the first axis (the first column of U) represents the direction of the largest variation among the documents, the second axis the direction of the second largest variation, and so on. When considering the thin SVD, where only the k largest singular values

[1] Recall that for $a \in \mathbb{R}^n, b \in \mathbb{R}^n$, the dot product of a and b is given by

$$a \cdot b = \sum_{i=1}^{n} a_i b_i$$

were preserved, we obtain a new space which is called k-concept space, containing less "noise" than the previous exact representation.
As a direct consequence of truncating the dimensions in our matrix representation, it is necessary to project queries represented by m-dimensional vectors into the k-concept space, so that we can compare the query to existing documents (we can call the query a pseudo-document, as stated in [Sha10]). We can derive a formula for the transformed query q_k as follows [Sha10]:

$$A_k = U_k \Sigma_k V_k \Rightarrow q = U_k \Sigma_k q_k^T \Leftrightarrow U_k^T q = \Sigma_k q_k^T \Leftrightarrow \Sigma_k^{-1} U_k^T q = q_k^T \Leftrightarrow q_k = q^T U_k \Sigma_k^{-1}$$

The obtained query for the k-concept space must then be compared to every column in V_k^T. Using the cosine similarity as a measure, we compare the cosine of the angle θ between q_k and a given vector of V_k^T (columns) or V_k (rows). The cosine similarity is defined as

$$cos(\theta_i) = \frac{< q_k, v_i >}{\|q_k\|_2 \cdot \|v_i\|_2}$$

where for any space $\mathbb{R}^n$ the inner product $< a, b >$ is the dot product of a and b, and for $x \in \mathbb{R}^n$ the Euclidean norm is given by

$$\|x\|_2 = \sqrt{< x, x >} = \sqrt{\sum_{i=1}^{n} x_i^2}$$

as usual.

5.2.2 The Higher-Order Singular Value Decomposition

This section is mainly based on chapter 5 of [Sha10].
Consider a set of users using a web search engine where it is possible to create user accounts to personalise the search and get better results. The design of the underlying information retrieval system of the search engine can be such that for each user a separate term-document matrix is managed in order to get personalised results (each term-document matrix would have a unique latent semantics structure). A more interesting approach would be to use a single data structure to manage users, documents and terms, since queries of different users may be related in some form and they could benefit from a joint analysis [Sha10].
Since the SVD is only defined for two-dimensional tensors (matrices), the generalisation of the SVD would be a suitable technique for analysing higher-dimensional structures. The main issues concerning this approach are

1. The runtime complexity of an HO-SVD algorihm (about $\sim O(n^4)$).
2. How can the similarity or correlation between users, documents and/or terms be measured?

Since the tensor could be very large if we have an extensive database and a vast amount of users, it is clear that the HO-SVD algorithm is critical for overall system perfomance. The procedures presented in chapter 3 may be used to obtain the HO-SVD within a reasonable time. Issues related to updating the HO-SVD (and the SVD) are discussed in [Sha10].
Measuring similarity or correlation between users, documents and/or terms is also not trivial. Consider the game mentioned in the introductory chapter, ARTigo. The game description on the projects website is the following:

> *ARTigo is an online game with the aim to supply artworks with tags. The same artwork is simultaneously shown to you and a co-player. The game now consists in describing the artwork accurately with tags. A tag can describe what you see on the artwork or remarks on style, quality or emotions. For each artwork, you have 60 seconds for your input. You get points for the tags identical to the ones entered by your co-player or another player in a previous round. The more tags per artwork are matched, the more points you get during the five rounds of the game. Every entered tag is saved and improves the search for artworks. As for the search however, only those tags entered at least twice for an artwork and thus marked in blue are relevant.*

This game generates data that can be stored in a three-dimensional structure with the variables *user*, *item* and *tag*.
It is necessary to formally define similarity in a way that reflects our own view of similarity in an intuitive way. We would like to be able to evaulate the similarity of users, taking into consideration their tagging behaviour, or to predict the behaviour of a given user u, considering the registered tags assigned by u in the past.
As a small example, let $\mathcal{A}$ be the tensor storing the tagging behavior of a set of users in our item set. If we perform a HO-SVD on $\mathcal{A}$, obtaining an approximation $\mathcal{A}'$, we could find out the "likeliness that user u_i will tag item i_j with tag t_k" [Sha10] by analysing the value a'_{ijk} if we stored the users in the first mode, the items in the second mode and the tags in the third mode. A satisfactory similarity measuring technique for comparing two objects of the same type has not been formulated for higher-order tensors yet. The way this is accomplished for matrices is by using the cosine similarity, as discussed in the previous subsection.

Chapter 6

Conclusion and future work

This thesis presented two approaches to be considered when computing the Higher-Order Singular Value Decomposition, especially when processing ”big-sized” tensors. Since both algorithms can be implemented as partially distributed procedures using the MapReduce paradigm, an important improvement in the execution time can be expected.
In order to make clear how the presented algorithms work, the mathematical background necessary for understanding the underlying operations and structures was presented as well as practical details associated with an efficient implementation of the described techniques. Finally, one possible field of application of the described algorithms was briefly discussed in the chapter about Latent Semantic Indexing.
Further work could include the design and implementation of an efficient and reliable set of procedures which allow the updating of an already obtained decomposition, as well as the integration of my implementation in a production environment.

List of Figures

3.1 Householder Bidiagonalisation 18
3.2 Golub-Kahan SVD step 20
3.3 Implicit Symmetric QR step with Wilkinson Shift 21
3.4 SVD algorithm 21
3.5 Sweep with systolic array for n=8 23
3.6 ORTH Procedure 24
3.7 Computing the Householder vector 26
3.8 Computing the factors c and s 27
3.9 Parallel HO-SVD with R-SVD 29
3.10 Parallel HO-SVD with Hestenes' method 30

4.1 Workflow of a MapReduce process 32
4.2 `MatrixSplitter` 33
4.3 HO-SVD computation with R-SVD and `ROC` steps. 41
4.4 Two-Group Orthogonalisation. 42

List of Tables

3.1 Runtime for Golub-Reinsch SVD and R-SVD algorithms 15

4.1 Cauchy/Cantor pairing function 34
4.2 Runtime comparison of SVD computations, with and without previous QR decomposition. The right singular vectors were computed explicitly. 37

5.1 Term-Document matrix 43

Bibliography

[BL85] R. Brent and F. Luk. The solution of singular-value and symmetric eigenvalue problems on multiprocessors. *SIAM Journal on Scientific and Statistical Computing*, 6:69–84, 1985.

[BMPS03] Michael Berry, W., Dany Mezher, Bernard Philippe, and Ahmed Sameh. Parallel computation of the singular value decomposition. Rapport de recherche RR-4694, INRIA, 2003.

[Cha82] T. F. Chan. Algorithm 581: An improved algorithm for computing the singular value decomposition. *ACM Trans. Math. Software*, 8:84–88, 1982.

[DDF$^+$90] Scott Deerwester, Susan T. Dumais, George W. Furnas, Thomas K. Landauer, and Richard Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41, June 1990.

[DG08] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *CACM*, 51(1):107–113, 2008.

[GL96] G. Golub and C. Van Loan. *Matrix Computations.* John Hopkins Press, 3rd edition, 1996.

[KB09] T. G. Kolda and B. W. Bader. Tensor decompositions and applications. *SIAM Review*, 51(3):455–500, 2009.

[LH74] C. L. Lawson and R. J. Hanson. Solving least squares problems. *Prentice-Hall*, 1974.

[LLH$^+$10] Yang Liu, Maozhen Li, Suhel Hammoud, Nasullah Khalid Alham, and Mahesh Ponraj. A mapreduce based distributed LSI. In Maozhen Li, Qilian Liang, Lipo Wang, and Yibin Song, editors, *Seventh International Conference on Fuzzy Systems and Knowledge Discovery, FSKD 2010, 10-12 August 2010, Yantai, Shandong, China*, pages 2978–2982. IEEE, 2010.

[LMV00] Lieven De Lathauwer, Bart De Moor, and Joos Vandewalle. A multilinear singular value decomposition. *SIAM Journal on Matrix Analysis and Applications*, 21(4):1253–1278, 2000.

[RJB89] Vijay V. Raghavan, Gwang S. Jung, and Peter Bollmann. A critical investigation of recall and precision as measures of retrieval system performance. *ACM Transactions on Information Systems*, 7(3):205–229, July 1989. Special Issue on Research and Development in Information Retrieval.

[Ros03] Rosenberg. Efficient pairing functions–and why you should care. *IJFCS: International Journal of Foundations of Computer Science*, 14, 2003.

[Rut66] H. Rutishauser. The Jacobi method for real symmetric matrices. *Numerische Mathematik*, 9:1–10, 1966.

[SA03] Volker Strumpen and Anant Agarwal. Technical memo, October 23 2003.

[Sha10] Philipp Shah. Towards Efficient Algorithms for Higher-Order Singular Value Decomposition . 2010.

[SP94] R.B. Sidje and B. Philippe. Parallel Krylov subspace basis computation. In *Actes du deuxième colloque africain sur la recherche en informatique = Proceedings of the second African Conference on research in computer science*, Colloques et Séminaires, pages 421–439. ORSTOM, 1994.

[SZL+05] Jian-Tao Sun, Hua-Jun Zeng, Huan Liu, Yuchang Lu, and Zheng Chen. CubeSVD: a novel approach to personalized web search. In Allan Ellis and Tatsuya Hagino, editors, *Proceedings of the 14th international conference on World Wide Web, WWW 2005, Chiba, Japan, May 10-14, 2005*, pages 382–390. ACM, 2005.

[vA06] Luis von Ahn. Games with a purpose. *IEEE Computer*, 39(6):92–94, 2006.

[Ven09] Jason Venner. *Pro Hadoop (Expert's Voice in Open Source)*. Apress, 6 2009.