FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

LARGE SCALE DATA PROCESSING USING MAPREDUCE

A dissertation submitted in partial fulfillment of the

requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Zhengguo Sun

2011

To: Dean Amir Mirmiran

College of Engineering and Computing

This dissertation, written by Zhengguo Sun, and entitled Large Scale Data Processing Using MapReduce, having been approved in respect to style and intellectual content, is referred to you for judgment.

We have read this dissertation and recommend that it be approved.

Date of Defence: March 25, 2011

The dissertation of Zhengguo Sun is approved.

Florida International University, 2011

DEDICATION

To my beloved parents.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor, Professor Naphtali Rishe, who provides me the great opportunity to study in HPDRC and offers me support and guidance through my Ph.D. in FIU.

I would also like to thank other members of my dissertation committee, who taught me many things in and outside of dissertation writing. Special thanks to Dr. Tao Li, who gave me much help and suggestions on my research.

Next, I want to thank members of HPDRC, from whom I learned a lot during daily work and discussions.

Finally and most important, I would like to thank my wife and my parents in law, who provides me with enormous support and encouragement during my dissertation writing.

ABSTRACT OF THE DISSERTATION

LARGE SCALE DATA PROCESSING USING MAPREDUCE

by

Zhengguo Sun

Florida International University, 2011

Miami, Florida

Professor Naphtali Rishe, Major Professor

As massive data sets become increasingly available, people are facing the problem of how to effectively process and understand these data. Traditional sequential computing models are giving way to parallel and distributed computing models, such as MapReduce, both due to the large size of the data sets and their high dimensionality.

This dissertation, as in the same direction of other researches that are based on MapReduce, tries to develop effective techniques and applications using MapReduce that can help people solve large-scale problems.

Three different problems are tackled in the dissertation. The first one deals with processing terabytes of raster data in a spatial data management system. Aerial imagery files are broken into tiles to enable data parallel computation. The second and third problems deal with dimension reduction techniques that can be used to handle data sets of high dimensionality. Three variants of the nonnegative matrix factorization technique are scaled up to factorize matrices of dimensions in the order of millions in MapReduce based on different matrix multiplication implementations. Two algorithms, which compute CANDECOMP/PARAFAC and Tucker tensor decompositions respectively, are parallelized in MapReduce based on carefully partitioning the data and arranging the computation to maximize data locality and parallelism.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

1.1. Web-scale Data

Although it is a cliché, it is true that we live in an information explosion era. On the personal side, filled with applications, documents, photos, and videos, PCs with terabytes hard drives are not uncommon today. On the enterprise side, InformationWeek reported that the retail giant, Wal-Mart, had 583 terabytes sales data in 2006. The telecommunication company AT&T has 312 terabytes of data in its Daytona database [1]. And the search giant Google processes an estimated twenty peta-bytes of data per day [2]. The situation in the science community is not slightly better. Vast data stores are generated by scientific instruments and computer simulations that need to be analyzed. For example, the World Data Center for Climate has 220 terabytes of climate research data readily accessible online and six petabytes worth of additional data stored on tape in 2005 [3].

Our capability to process and analyze data has not kept pace with our ability to capture and store data [4]. There is an urgent need for tools that can process tera-, peta- or even exa-bytes scale data. Several researches have reacted to this complexity with novel ideas and tools [5] [6]. MapReduce [5] is one of them. Originally developed in Google as a tool to process large amounts of data such as crawled documents, web request logs, and etc., it has become popular both in industrial and academic to solve large-scale problems.

This research, as in the same direction of other researches that based on MapReduce, tries to solve large-scale problems we are facing today using MapReduce and gain more insights into the model as to how can this model be used under different scenarios.

1.2.My Work

There are three major piece of works presented in this dissertation. The first one is related to large scale raster data processing in spatial database. This piece of work belongs to the traditional category of data intensive computing, for which MapReduce is designed. The second and third pieces of work have a more algorithmic flavor. They explore the domain of how MapReduce could be used to express and parallelize algorithms.

As a major type of data processed by spatial database, raster data (aerial photography and satellite imagery), is being generated by satellite or aircraft-mounted cameras periodically in the scale of Tera-byte or Peta-byte. How to efficiently process and manage such huge amount of data is a challenge for modern spatial database. As a case study to show how MapReduce could be applied to spatial database to process large amount of raster data, we present two problems in the context of TerraFly [7] – an online spatial data management system. The first problem deals with loading data into the system and the second one computes some statistics over a large store of raster data. We present the design and implementation of two MapReduce applications to solve these two problems. Some techniques are demonstrated in the process. We experimentally evaluate our applications on a real-world cluster and get very good result, which in turn prove the applicability of MapReduce to data intensive problems.

A less explored application domain of MapReduce is the algorithm design field, or, contrary to data intensive problems, computation intensive problems. We present two attempts made to investigate the applicability of MapReduce in this field in the dissertation.

An implicit consequence of the increase of the size of data set is the increase of the dimensionality of the data set. For example, the most popular social network website, Facebook, has more than 600 million active users as of January 2011. Every user is a node in this giant social graph, which could be represented as a sparse matrix with millions of rows and millions of columns. Another example is the WWW, which now contains billions of web pages and is an enormous repository for text mining applications. Term-document matrix with millions or even billions of rows and columns could easily be derived from this repository. The ability to understand the structure and recover the hidden information from such data sets is of vital importance. There exist techniques and algorithms to reduce the dimensionality of data set and extract the hidden structure, but most of them do not scale well and consequently cannot cope with data sets at this scale. One of the popular algorithms is called nonnegative matrix factorization (NMF), which was originally proposed for parts-of-whole interpretation of matrix factors. NMF has attracted a lot of attentions recently in data mining and machine learning communities, and has been shown to be useful in a variety of applied settings, including environmetrics, chemometrics, pattern recognition, multimedia data analysis, text mining, web mining, and DNA gene expression analysis.

Our first attempt is to scale up the NMF algorithm to be able to factorize matrices of dimensions in the orders of millions. We propose three different matrix multiplication implementations using MapReduce, based on which we successfully parallelize three different types of NMF algorithms on the MapReduce platform. We evaluate our algorithms on a cluster provided by Google and IBM through the NSF Cluster Exploratory (CLuE) program [33] using both synthetic and real data sets and obtain linear scalability of the proposed algorithm.

Our second attempt deals with tensors, a natural extension of vectors and matrices in high-order dimensions. Tensors are multidimensional arrays. Many data sets could be naturally represented by tensors. For example, we could add a time dimension to the Facebook's social graph so that we could track the changes over time. The WWW is another example. As its size increases, it becomes more and more important to analyze link structure considering context as well. Multilinear algebra provides a novel tool for incorporating anchor text and other information into the authority computation used by link analysis methods such as Hyperlink-Induced Topic Search (HITS) [8]. T. Kolda and B.Bader [9] proposed Topical HITS (TOPHITS) method which uses a higher-order analogue of the matrix SVD called the PARAFAC model to analyze a three-way representation of web data. They compute hubs and authorities together with the terms that are used in the anchor text of the links between them. Adding a third dimension to the data greatly extends the applicability of HITS because the TOPHITS analysis can be performed in advance and offline. More applications of high-order tensor decomposition could be found in [10].

We consider two major tensor decomposition algorithms in this dissertation, namely, the CANDECOMP/PARAFAC decomposition and the Tucker decomposition. We decompose the algorithms into some basic operations such as matrix multiplication, the Hadamard product, the Khatri-Rao product, and the Moore-Penrose pseudo inverse. We implement each of them in MapReduce and chain them together to do the decomposition. Our implementation is experimentally evaluated on the Google & IBM cluster and the results show excellent scalability.

1.3.Organization of the Dissertation

The dissertation is constructed in the following way: we briefly introduce MapReduce and cloud computing in chapter two. We put emphasis on its programming model and application. Also demonstrated is the real-world usage with an open source implementation – Hadoop. As the major background in which MapReduce was born, we discuss cloud computing and its relationship to MapReduce in the end of chapter two. Chapter three presents how to solve large scale raster data processing problems in MapReduce by using two case studies from the TerraFly project. Chapter four discusses how we scale up the nonnegative matrix factorization algorithm in MapReduce. The design and implementation of two tensor decomposition algorithms are presented in chapter five. We conclude our work in chapter six and briefly discuss future work.

2. MAPREDUCE AND CLOUD COMPUTING

In this chapter, we give a brief introduction to MapReduce, which includes its restricted programming model, various applications of this model, its implementation, and how do you use such an implementation in the real world. The larger background, cloud computing, in which MapReduce origins, is also discussed in section 2.2 with its relationship to distributed and parallel computing.

2.1.Introduction to MapReduce

2.1.1. The Model and Its Application

MapReduce, originally proposed by Google in 2004 [5] for solving many problems related to large data sets in the company, is a restricted programming model that is applicable for processing and generating large data sets. It has intrigued many related researches upon its newborn, which includes but not limited to works to porting this model to different architectures [11] [12] [13] [14] [15], extensions to the model [16] [17] [18], and various applications in machine learning and data mining [19] [20] [21] [22] [23], information retrieval [24] [25], video processing [26], spatial data processing [27] [28], and so on so forth.

The popularity of MapReduce is largely related to the simplicity of its programming model. The users of a MapReduce implementation just have to specify *map* and *reduce* two functions. Everything else, including automatic parallelization of tasks, fault-tolerance handling, locality optimization, and loading balancing, is taken care of by the underlying implementation. The model takes a key/value pair perspective on the data to be processed. In other words, the input data of the computation is presented as a set of key/value pairs; the computation itself is carried out on the key/value pairs; and the output of the computation is saved as another set of key/value pairs too.

More specifically, the *map* function, specified by the user, takes an input pair and produces one or more intermediate key/value pairs, which will be consumed by the reduce function later. The underlying MapReduce implementation groups together all intermediate pairs that have the same key and passes them to the *reduce* function.

The *reduce* function, also specified by the user, consumes all the pairs that associate with the same key and produces another set of key/value pairs.

Conceptually, the process can be described by the following two formulas:

$$map(k_1, v_1) \rightarrow list(k_2, v_2)$$

$$reduce\big(k_2, list(v_2)\big) \rightarrow list(k_3, v_3).$$

Taking the classic word count problem as an example, given a large collection of documents, you are supposed to count the number of occurrences of each word in the collection. The *map* function and *reduce* function for this problem is similar to the pseudo-code in Figure 1. From a relational algebra point of view, the whole process is more like a select projection via sequential scan (*map*), followed by hash partitioning, sort and group-by (the runtime system) and aggregation (*reduce*). More formal analysis about this model can be found in [29].

```
Map(key, value)
// key: document name
// value: document contents
    For each word w in value:
        Emitintermediate(w, 1)

Reduce(key, values)
// key: a word
// values: a list of counts
    res = 0
    For each v in values:
        Result += v
    Emit(res)
```

Figure 1 MapReduce pseudo-code for word count problem

This model is very appealing to programmers; for there are only two high-level declarative primitives (*map* and *reduce*) needs to be specified to enable parallel processing. Even programmers with little experience in parallel programming can do it. All the benefits of data partitioning, task scheduling, fault-tolerance, and load balancing come with the framework. However, every coin has two sides. Those benefits don't come for free. The assumption is that the problem to be solved has a solution that can be expressed in the MapReduce model. The one-input, two-stage data flow of the model is somewhat constrained, thus, restricting the applicability of the model to certain kinds of problems. We don't want to overemphasize its limitation, for the various applications developed on this model have already proved its expressiveness. Not to mention that there exist extensions that overcomes its limitations [16] [18].

The original target applications of MapReduce are data intensive applications. As it becomes more and more popular, more applications have been considered, especially computation intensive applications. Data intensive application, such as the word count example, distributed grep, count of URL access frequency, reverse web-link graph, inverted index, distributed sort and others [5] are usually simple and straightforward to express in MapReduce. The whole process usually entails applying some small computation on each pair of input in the map function and aggregating the results in the reduce function. On the other hand, computation intensive applications, such as some machine learning algorithms [19], integer factorization [30], and the problems we are going to study in Chapter 4 and Chapter 5, are not so intuitive to be expressed in MapReduce due to the complex nature of the algorithms. Still, solution exists after careful analysis of the problems and sometimes some small adjustments of the model as demonstrated in [19] [20] [21] and Chapter 4 and 5 of this dissertation. The process is beneficial in that we gain deep insight into the capability of the MapReduce model and accumulates experience for solving new problems using the model.

2.1.2. Its Implementation and Real-World Usage

We give some introduction to the programming model of MapReduce and its applications in the previous section. In this section, we will review the implementation of the model and see how to use such framework in the real-world.

Figure 2 shows the execution overview of a MapReduce job. Google's implementation of MapReduce is proprietary and not available for public use. Fortunately, an open source project, called Hadoop [31], completely implements the idea of

MapReduce. We are going to discuss its major components. From a theoretical point of view, there is no different between Hadoop and Google's implementation of MapReduce. Our discussion is more focused from a user's point of view. Thus, some components that are not supposed to be used by a MapReduce user are skipped. Interested readers are encouraged to consult the original paper [5] and Hadoop's online documentation [31]. In the rest of this dissertation, we interchangeably use the term MapReduce and Hadoop. When we use MapReduce, we are referring to the Hadoop implementation.

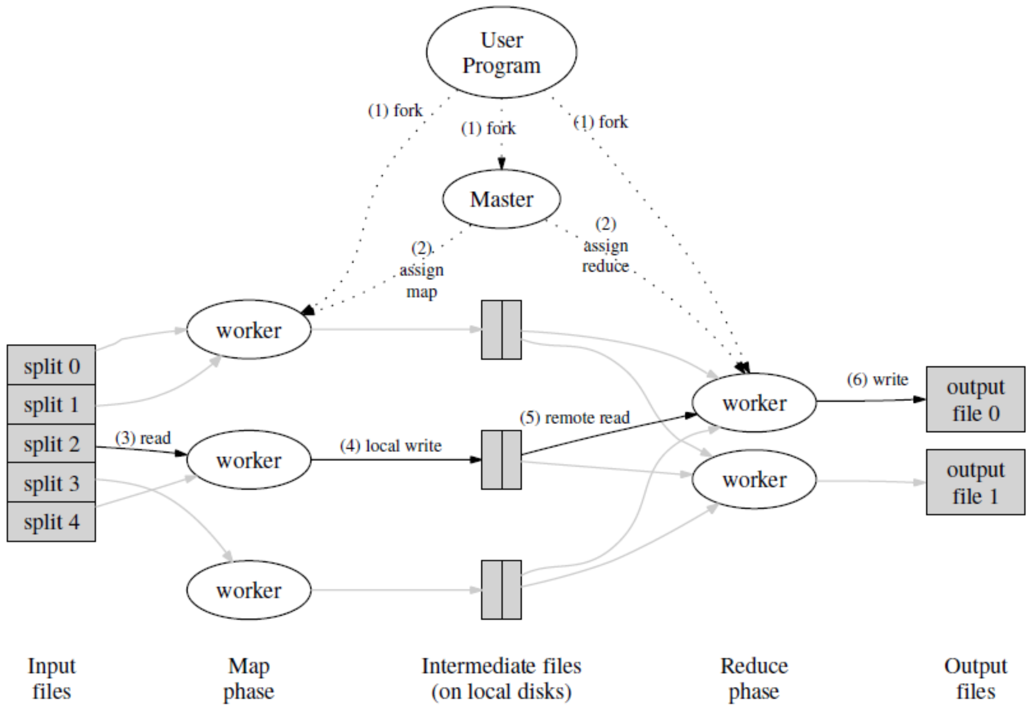


Figure 2 Execution overview of MapReduce*

*image from Google's original paper [5]

There are three major components in Hadoop, namely, the Mapper, the Reducer, and the Partitioner. The Mapper corresponds to the *map* function and the Reducer corresponds to the *reduce* function in the model. The Partitioner controls the partitioning of the keys of the intermediate outputs of map function.

There are some other components that are not always used in every Hadoop job, but are very helpful under certain conditions. For example, the InputFormat interface describes the input specification of a MapReduce job. Together with the InputSplit and the RecordReader interfaces, they make a MapReduce job capable of reading input from different sources (e.g. from files, databases e.t.c). Another example is the SortingComparator and GroupingComparator interfaces. They are responsible for sorting and grouping the intermediate outputs of Mappers. Hadoop only guarantees that the keys are sorted in the input of a Reducer, but not the values. Certain tricky use of SortingComparator and GoupingComparator can simulate secondary sort on values as well, which we will see this use in Chapter 3.

All the experiments of this dissertation are run on a large shared cluster provided by the Google and IBM Academic Cluster Computing Initiative [32] through the NSF Cluster Exploratory (CluE) program [33]. We describe its configuration and how we use this cluster as a real-world example of using MapReduce. There are about 500 machines in the cluster, each of which has two single-core 2.8 GHz Xeon processors, 8 GB memory, two 400 GB hard drives, and one gigabit Ethernet connection. Hadoop 0.20.1 runs on one 64-bit Xen virtual machine running on top of the above physical hardware with access to all the memory (minus some overhead), all the execution threads of the processors and all of the disks. The VM runs CentOS 5.3 and the host operating system

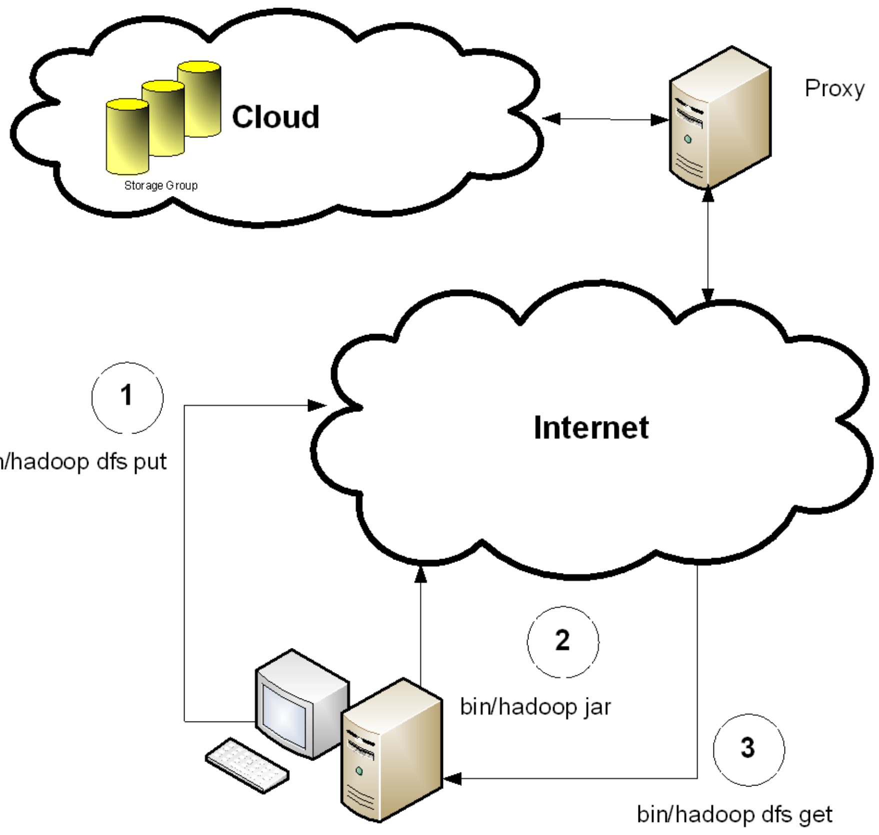


Figure 3 Google, IBM academic cluster overview

runs Fedora Core 8. Access to the cluster is provided through the Internet by a SOCKS proxy server. SOCKS is an Internet protocol that secures client-server communications over a non-secure network.

There are three main steps in interacting with the cluster, as shown in Figure 3. (1) Input data is uploaded into the cluster. The user uses file system shell scripts provided by the Hadoop Distributed File System (HDFS), which is an integral part of the Apache Hadoop project; HDFS is a clone project of Google’s files system GFS [34]. (2) A user develops a Hadoop application and submits it to the cluster via Hadoop command.

Hadoop applications are usually developed in Java, but other languages are supported, like C++ and Python. (3) After application execution is completed, the output is downloaded to the user's local site with Hadoop file system shell scripts. A data set is stored as a set of files in HDFS, which are in turn stored as a sequence of blocks (typically of 64MB in size) that are replicated on multiple nodes to provide fault-tolerance.

2.2.Cloud Computing

It is incomplete to introduce MapReduce without mentioning cloud computing, because cloud computing is the large background in which MapReduce is born. As a buzz word, the term cloud computing entails many things and means different concepts to different people. The National Institute of Standards and Technology (NIST) defines it as follow:

"Could computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. [35]"

It may not be complete, but it does describe the way we use the Google & IBM cluster. Some of its characteristics are:

- Cloud computing provides computing resources as an on-demand service.
- The user of cloud computing doesn't know the physical location of the computing resources he is using.

- Cloud computing is elastic and scalable.
- The computing resources are shared between many users.

As can be seen from these characteristics, cloud computing does bear some similarities to the traditional terms “utility computing” and “grid computing”, for example, providing computing resources as a public service is the essence of utility computing, and offering shared computation and storage over long distances is the same as grid computing.

So how does MapReduce fit into this big picture? From a user’s point of view, MapReduce is the tool to utilize the power of cloud computing, especially the infinite scalability, to solve various kinds of large scale problems. And from a cloud service providers’ point of view, MapReduce is an essential piece of software that consists of their cloud services.

3. RASTER DATA PROCESSING IN MAPREDUCE

3.1. Background

One of the major types of data that is handled by spatial databases is raster data, namely, aerial photography and satellite imagery. This type of data is being generated by satellite or aircraft-mounted cameras periodically and is in the scale of Tera-byte or Peta-byte. How to efficiently process such a huge amount of data is a challenge for modern spatial databases. Although the size of the data is huge, the operations performed on the data are relatively simple. Two case studies are given in this section to show how MapReduce could be used to solve such kind of problems.

3.1.1. TerraFly

Both of the problems described in this section have their real world applications in the TerraFly project [7]. Terrafly is a web application that enables easy access and manipulation of remotely sensed data (aerial photography and satellite imagery) and other spatial data. Using only a standard web browser, an average user can seamlessly fly over various remotely sensed data of different sources, spatial resolutions and image formats integrated with graphical maps and other spatial data sets such as US census data and demographic data. The raster data sets collected by TerraFly cover the whole United States in 1 meter spatial resolution, most of its major cities in 30 cm resolution, and some areas in 1 foot (most Florida Counties) or 3 inch (Miami) resolution. The size of this data set is about 40 TB currently and is still increasing. The first problem deals with the loading of this huge and ever increasing raster data set.

3.1.2. The image loading problem

In order to provide fast and high quality aerial photography and satellite imagery viewing experience, TerraFly converts and tiles the image files that it collects from various sources (USGS, GeoEye, or GIS department of Counties). Internally, Terrafly uses a grid structure based on the Universal Transverse Mercator geographic coordinate system (UTM) to index this raster data set. The UTM system divides the whole global into 60 longitudinal projection zones numbered 1 to 60 starting at 180 °W, each of which is 6 degrees wide except a few regions around Norway and Svalbard. Within each

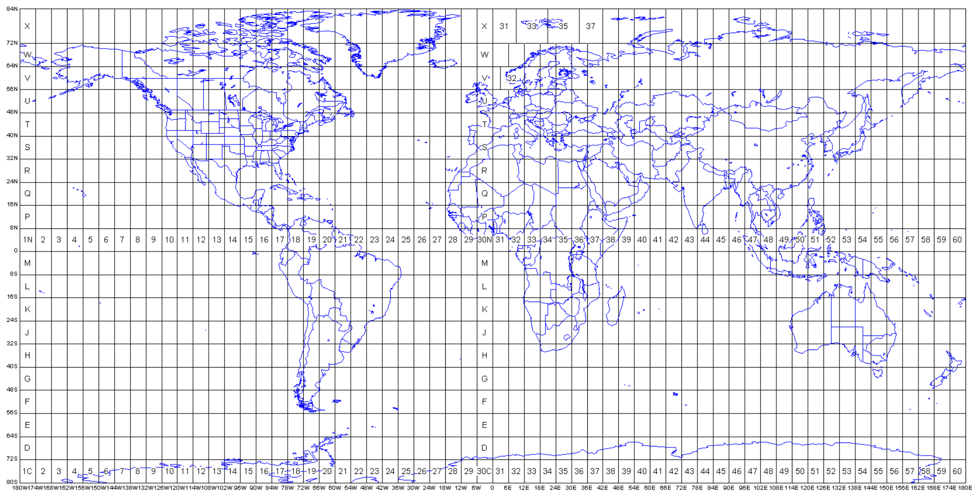


Figure 4 the Universal Transverse Mercator System

longitudinal zone the Transverse Mercator Projection is used to give co-ordinates (eastings and northings) in metres. For the easting, the origin is defined as a point 500,000 meters west of the central meridian of each longitudinal zone, giving an easting of 500,000 meters at the central meridian. For the northing in the northern hemisphere, the origin is defined as the equator. For the northing in the southern hemisphere, the origin is defined as a point 10,000,000 meters south of the equator. Figure 4 shows the whole global in the UTM system. The continental United States is covered by UTM Zone 10 to UTM Zone 19.

The index structure consists of several parts. The first part is the name of the imagery source. Each source has its own name space. The second part is the UTM Zone. We further divide each zone into grids of size 102400 by 102400 in pixels. Each of these grids has its unique id inside a particular UTM Zone. We create a file for each of these grids, each of which contains records that describe image files within or intersect with this grid. These image files are in an internal format that contains a descriptive header and a bunch of JPEG image tiles, each of which is of 400 by 400 in pixels. This is the smallest unit of this index system and we call it a tile. We also do resampling for each image file to facilitate viewing of the same area at different scales. The resampled image files are also put into the index system marked with a zoom level that indicates their resolutions after the zone number. For example, Miami, FL is in UTM Zone 17 and in grid 275. Let’s say we have a data set covers this city and its name is USGS_30CM, which indicates that it is from USGS and is of spatial resolution 30 cm. We will converts and tiles each image files in this data sets and put index records into our index system. We could find these files by following the index path usgs_30cm/17.1/275. And path usgs_30cm/17.2/275 contains the resampled image files which are of resolution 0.6 meter per pixel. And so on so forth. We usually stop resampling at 2^{10} of the original resolution, because the viewable area becomes very small at this scale.

All the raster data collected by TerraFly are converted into the internal format mentioned above, and tiled under the above index system. We call this process loading. This loading process is very slow considering the huge size of the data sets using traditional single processor architecture. But this procedure could be easily implemented in an embarrassingly parallelizable way. Each file is independent of other files and can be converted and tiled in parallel. Special care needs to be taken at the boundary of image files, for a tile could potentially span four different neighboring files. Under such conditions, the Reducers are employed to handle the merging of such tiles.

3.1.3. The image quality assessment problem

The second problem given in this section is also common in aerial photography and satellite image processing. It is related to the quality of service provided by TerraFly. Let’s say that we have loaded a data set that covers a particular geographical region. At the boundary of this data set, there could be incomplete tiles, for the coverage of the data set does not necessarily fit exactly into our index structure. In other words, the files we receive are not necessarily of sizes that are a multiple of 400 by 400 in pixels, which is the tile size of our index system. Later, we have a second data set that covers these incomplete tiles. Thus, we want to merge data from these two data sets for those incomplete tiles. This situation could become especially obvious when you zoom out into a states level or country level, at which scale every city becomes so small that a data set doesn’t even cover a tile. Figure 5 shows an example tile in Florida that is mosaiced from several data sets. Such situation not only occurs at the boundary of data sets, but may also in the middle of a data set due to faulty instrument or incorrect processing steps in the data acquisition phase.

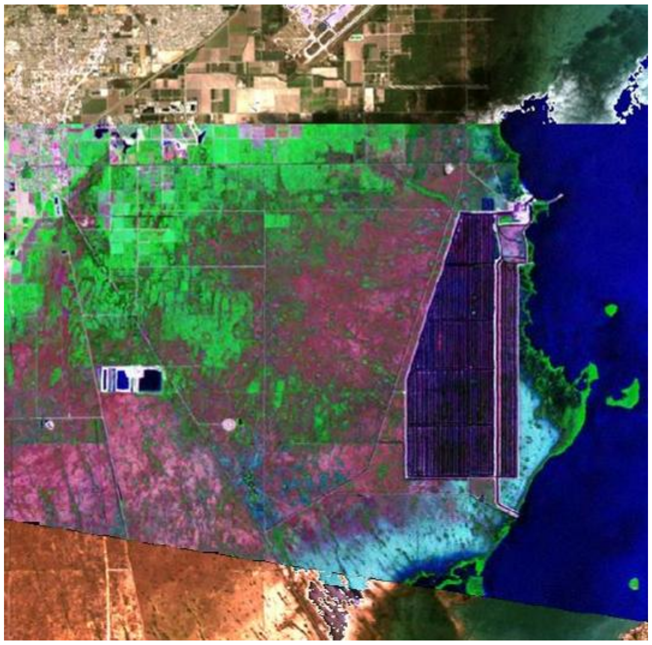


Figure 5 A tile that is mosaiced from multiple data sets

Thus, a mosaicing or patching step is needed when rendering the images. The necessity of such a step should be determined before it is carried out. Instead of dynamically analyze the image at runtime, which is computationally expensive, the result of the analysis is usually pre-calculated and stored in some format. This pre-computation could again take advantage of the parallelizing power of MapReduce, for such local analysis on each image files doesn't need information from other files and can be processed independently.

In summary, the key contributions of this work are as follow:

- We develop two real world applications in Hadoop to process spatial rater data in TerraFly.
- We demonstrate how to process spatial raster data in MapReduce by solving two real world problems.
- We experimentally evaluate our MapReduce algorithms using real data sets on a real world cluster.

The rest of this chapter is organized as follows. Section 3.2 discusses the related work to our problems. Section 3.3 presents the detailed design and implementation of the two algorithms. Experiment setup and results are shown in section 3.4. We conclude and summarize our work in section 3.5.

3.2.Related Work

3.2.1. Image Processing

The image we are referring to in this paper is digital image, which can be defined as a two-dimensional function, *f(x, y)*, where *x* and *y* are spatial coordinates, and the value of *f* at a particular coordinate *(x, y)* is called the intensity or gray level of the image at that point. The domain and range of *f* should both be finite and discrete in order for the image to be a digital image. Digital images are consists of basic elements, which are usually called *Pixel*. Each pixel occupies a specific location *(x, y)* and has some value *f(x, y)*. More formal discussions can be found in [36].

Processing digital images using a digital computer is generally called digital image processing. Closely related to but different from the concept of image processing are the concepts of image analysis and computer vision. Although there is no agreement among authors regarding the boundaries between these concepts, the concept of image processing is more towards low-level processing, such as noise reduction, contrast enhancement, and image sharpening, while the concept of computer vision is more towards high-level processing such as object recognition. Image analysis sits in between the two other concepts. A useful criterion that can be used to differentiate them is the resultant structure of the processing. The result of lower level primitive processing usually is still an image. For example, after noise reduction, the output image is still the for example, the answer to the question: whether this image contains a human face? [36] [37]. The image processing tasks considered in this paper is limited to low level processing.

Various software packages are available for carrying out general image processing tasks [38] [39]. Aerial photography and satellite imagery are usually handled by specialized software packages such as ArcGIS [40] or ERDAS [41]. In addition to some of the basic low level image processing tasks, they provides functionalities such as image reprojection, image mosaic, map composition, same as the input image except some portion of noise has been removed. This also applies to contrast enhancement and image sharpening. On the other hand, the result of high level processing is usually not an image. And it could as simply as a “YES” or “NO”, spatial modeling and other GIS related features.

3.2.2. Parallel Processing

Parallel processing is the term used in computing that is the contrary of sequential processing. In sequential processing, instructions are executed one after another. There is only one execution thread at any given point of time. While in parallel processing, there are many calculations carried out simultaneously.

Parallel processing can be classified into four categories by Flynn's taxonomy [42], namely single instruction single data (SISD), single instruction multiple data (SIMD), multiple instruction single data (MISD), and multiple instruction multiple data (MIMD). This classification is based on instruction and data two dimensions. The SISD class corresponds to sequential processing. The MISD class is rarely used and the MIMD class is the most common type of parallel processing, which requires synchronizations between different subtasks of the computation. The SIMD class, which is also called data parallel systems, is the most similar one to MapReduce and involves doing the same operations on a large data set. In such a system, all PEs (processing elements), being it a processor or a node in a cluster, execute the same instructions on its own portion of data. There is no need for communications between PEs and error-prone synchronization mechanisms such as mutex or semaphores, thus simplifying the design of the system.

Low level image processing turns out to be a very good candidate for parallel processing, for many operations require only information from a limited neighborhood of a pixel or even operate without data from other pixels. Researchers have been making efforts on many different aspects to bring the benefits of parallel computing to the image processing community. Various special architectures have been proposed in the literature for parallel image processing [43] [44]. In an earlier work, Anthony [45] surveyed the architectures of many parallel systems that have been developed for image processing and proposed a simple SIMD/MIMD computational model for comparison with such systems. Recent advancement includes using GPU (graphic processing unit) to accelerate image processing tasks [46]. Another branch of researches exploit developing high level languages and libraries for parallel image processing that can be applied to a range of different architectures. [47], which presents an application-specific high level programming language intended for implementing low level image processing applications on parallel architectures, is in this category. Similarly, [48] presents a library of data parallel low level image processing operations that is based on so-called "parallelizable pattern". Thomas et. al [49] give an extensive discussion on the topic of parallel image processing.

3.3.Design and Implementation

The detailed design and implementation of the above two problems are presented in this section. A brief description of the organization of the imagery data is given first, followed by the MapReduce algorithms to process the data. And finally, some implementation issues are discussed.

The aerial/satellite imagery data TerraFly collects is usually DOQQ files in GeoTiff format [50]. This is the input for the image loading problem. After conversion and tiling, they are transformed into an internal format. More specifically, this format contains a descriptive header and a bunch of JPEG image tiles, each of which is of size

400 by 400 pixels and covers a particular geographic area. The result of the imagery loading problem is the input for the image quality assessment problem.

3.3.1. MapReduce Algorithm for the image loading problem

The algorithm for the imagery loading problem is discussed in the following. The input for this problem is a data set that consists of GeoTiff files and covers a specific geographical region, such as a city or a county, and the output is a bunch of tiled files in the format mentioned above and their corresponding index files. All the output files will be of the same size measured in terms of width and height in pixels (for example, 25600 x 25600), which is the parameter of the actual Hadoop job. Thus, the number of output files may not be the same as input. In fact, usually it is less than the number of input files and the size is always chosen to be a multiple of 400 so that the output file doesn't contain incomplete tiles (each tile is 400 by 400 in size). All paths of the input files are compiled into one file, which contains the path of one file on each line and is the input for the Hadoop job. TextInputFormat is used to parse the input into one line per record. Each Mapper will be responsible for portion of the input files. The Mapper opens the GeoTiff file and converts it into a bitmap. After that it tiles the bitmap according to its geographic coordinates, compresses each tile, and emits a record with (*gid*, *fid*, *zoomlevel*, *tid*) as its key and the compressed tile as its value, where *gid* is the grid id in the index system, *fid* is the output file id that this tile belongs to, *zoomlevel* is the current zoom level, and *tid* is the tile id in the output file.

```
Map (offset, path)

open Geotiff and convert it into bitmap
zoomlevel = 0
while zoomlevel < targetlevel
    for each tile in current bitmap
        compress and emit intermediate result
    resample bitmap
    zoomlevel += 1
```

Figure 6 Pseudo-code for the map function of the image loading problem

The Mapper resamples the bitmap and repeats the above process until the desired zoom level is reached. Thus, the Mapper creates a hierarchical pyramid of the input image at different resolutions, which can be rendered at runtime to show the same region at different scales. Figure 6 shows the pseudo-code for the map function of this problem.

The Reducer is supposed to receive all the tiles that belongs to one output file and merges them together, but this is not the default behavior in Hadoop. Recall from section 2.1.2 that MapReduce groups all the intermediate values with the same intermediate key and sends it to a reduce function. Since the key is different for each tile (because *tid* is different for each tile), tiles belong to one output file may be shuffled to different Reducers. The reason we also put *tid* into the key is that we want to simulate a secondary sort on the tiles that belong to the same output file, which will facilitate the processing in Reducers. In other words, tiles are sorted by their positions in a file so that the Reducer can write to HDFS directly when iterating through its input. We achieve such result by overriding the Partitioner and the GroupingComparator interfaces.

```
Reduce (key, tiles)
f = open file for writing
for each tile in tiles
    f.write(tile)
```

Figure 7 pseudo-code for the reduce function of the image loading problem

As mentioned in the previous chapter, there is a shuffling and sorting phase between the map and the reduce phase. Intermediate outputs with the same keys are shuffled to one Reducer and are guaranteed to be sorted according to their keys. The Partitioner interface is responsible for the partitioning of the keys of the intermediate outputs of Mappers. It is typically implemented as a hash function. Thus, a key is hashed into one of the Reducers. We override the hash function so that it only considers the *gid* part of the key. This ensures all the tiles belong to the same grid will be shuffled to one Reducer, but it doesn't specify the order by which the groups are created. There are two Comparator interfaces in the Hadoop framework that controls the sorting and grouping of intermediate outputs, namely, the SortingComparator and the GroupingComparator. The SortingComparator is responsible for sorting all the intermediate outputs according to their keys and the GroupingComparator is responsible for grouping the intermediate outputs and sending each group as one call to the Reducers. In our case, the SortingComparator will sort all the tiles in the order according to *gid*, *fid*, *zoomlevel*, and *tid*. This gives the correct ordering of all the tiles for the Reducer to write them to HDFS, but the GroupingComparator will treat each tile as one group by default, for each of them has different keys. Again we override this GroupingComparator so that it only considers the *fid* and *zoomlevel* part of the key (the *gid* part is the same for all the values that are shuffled to one Reducer). Thus, tiles belong to the same output file are sorted and grouped in one call to the Reducer. Figure 7 shows the pseudo-code for the Reducer, which is very concise due to the work we have done in Partitioner and GroupingComparator. The input and output key/value pairs for this problem is summarized in Table 1. We use brackets to represent key value pair and square brackets to represent a list of values.

Table 1 Input/output for the converting and tiling Hadoop job

Function	Input	Output
Map	$\langle offset, GeoTiff\ file\ path\rangle$	$\langle gid + fid + level + tid, tile\ content\rangle$
Reduce	$\langle fid + level + tid, [tiles]\rangle$	$[tiled\ image\ files]$

3.3.2. Algorithm for the image quality assessment problem

We introduce some notations before describing the algorithm for the image quality assessment problem. Let *d* be an image file and *t* be a tile inside *d*. *d.name* is the file name and *t.q* is the quality information of tile *t*. Figure 8 depicts the execution overview of the MapReduce algorithm to compute the image quality. The algorithm runs on a tile by tile basis within the boundary of a given image file. It computes a bitmap for each tile where a set bit represents a good pixel in the tile and an unset bit represents a bad pixel. A pixel is defined as “bad” if all the values of its samples are below or above some predefined value. More complex operations could be carried out in the process, but

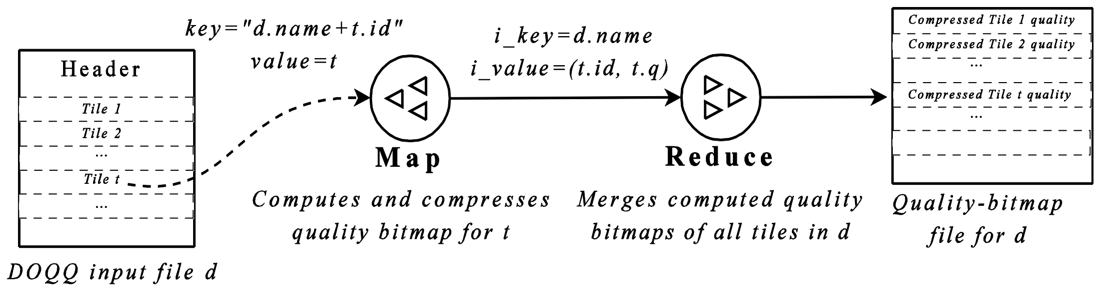

key="d.name+t.id"
value=t
Header
Tile 1
Tile 2
...
Tile t
...
DOQQ input file d
Map
Computes and compresses quality bitmap for t
i_key=d.name
i_value=(t.id, t.q)
Reduce
Merges computed quality bitmaps of all tiles in d
Compressed Tile 1 quality
Compressed Tile 2 quality
...
Compressed Tile t quality
...
Quality-bitmap file for d

Figure 8 Overview of the aerial imagery quality computation algorithm

this simple scheme is enough for the application purpose.

Each image file is first partitioned into several splits, each of which is then processed by a separate Mapper. Since the underlying distributed file system (HDFS) has no knowledge of tiles, a specialized RecordReader is implemented to respect the tile boundaries. Each tile is parsed out from a split by this RecordReader, combined with the file name and tile id as its key, and sent to Mappers as an input record.

The input and output key/value pairs for Mappers and Reducers are described in Table 2. The Mapper decompresses the JPEG tile *t*, iterates through each pixel of *t* to obtain quality information *t.q* (a bitmap, one bit per pixel) and compresses it using Run-length encoding. After that, it emits the intermediate key/value pair with *d.name* as the key and *t.q* as the value. The Reducer merges all the quality bitmaps that belong to the same file and writes them to an output file as shown in Figure 8.

Table 2 Input/output of map and reduce functions for image quality assessment

Function	Input: (Key, Value)	Output: (Key, Value)
Map	$\langle d.name + t.id, t \rangle$	$\langle d.name, t.q \rangle$
Reduce	$\langle d.name, [t_i.q] \rangle$	$\langle quality\ information\ of\ d \rangle$

3.4.Experiments

3.4.1. Cluster Setup

All the experiments in this section and in the following two sections were conducted on a large shared cluster of approximately 500 machines provided by Google and IBM [32] through the NSF Cluster Exploratory (CluE) program [33], which we described in section 2.1.2.

3.4.2. Data Sets

We used two different data sets for the two different problems. The data set used in the image loading problem is a 1-foot resolution aerial imagery of Hendry County of Florida. The data set used in the image quality assessment problem is a 3-inch resolution aerial imagery of Miami Dade County of Florida. Some statistics about these two data sets are summarized in Table 3.

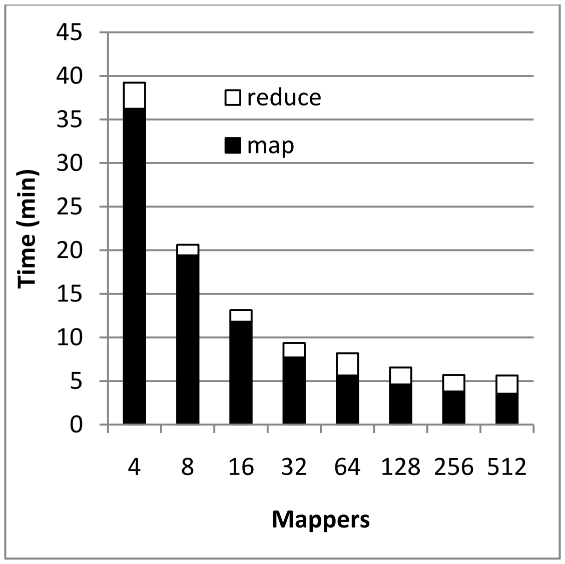
Table 3 Data sets statistics for raster data processing in MapReduce

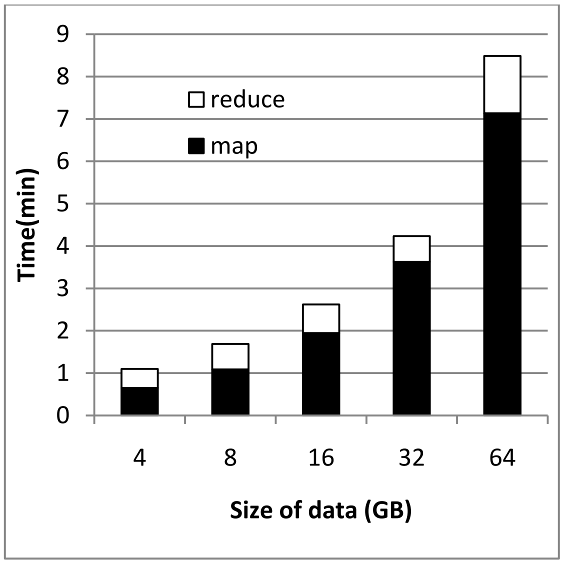
Data set	Used for	Size	Num of files
Hendry County	Image loading	100 GB	1486
Miami Dade County	Image quality assessment	520 GB	482

As can be seen in the table, the size of the data set is about 520 GB without compression. Since the final result is compressed in JPEG, its actual size is about 52 GB. This data set is stored in the format described in section 3.1.3. There are 482 files, each of which contains 4096 tiles. Each tile is 400 by 400 pixels and has 3 bytes for each pixel as the red, green, and blue channel. The size for each tile is 480,000 bytes uncompressed and compressed tile is about 50 KB each.

3.4.3. Experimental Results

Two experiments are carried out for each of the data set. The first experiment varies the number of Mappers/Reducers and the second one varies the size of the input data. Both of the experiments record the elapsed time of the MapReduce jobs.





a) Variable Mappers b) Variable input data size

Figure 9 MapReduce job completion time for the image loading problem

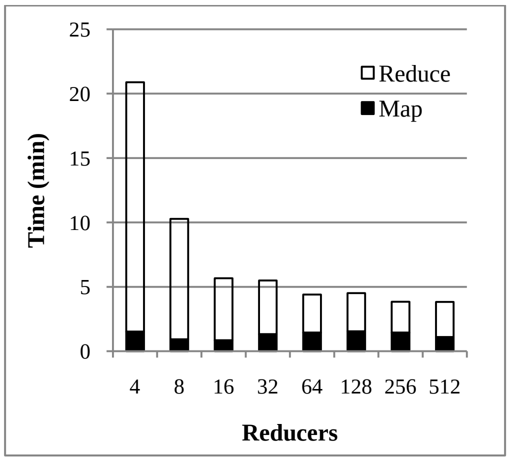
Figure 9 shows the experiment results for the image loading problem. Both of the experiments load the whole or partial of the data set for nine zoom levels. The first experiment uses the whole data set, fixed the number of Reducers to 4 and varies the number of Mappers from 4 to 512.

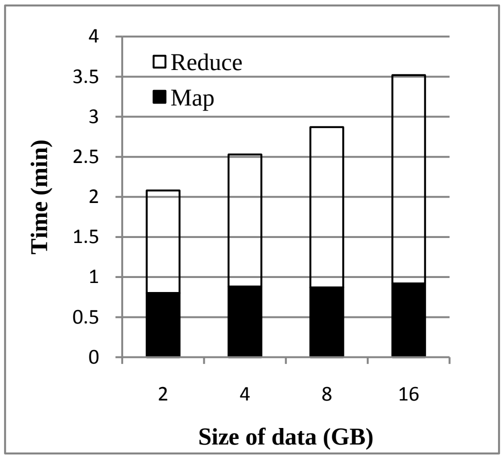
As can be seen from Figure 9 a), the time spent in the reduce phase is almost the same across all the different configurations of Mappers. This is because the work done in the reduce phase is the same for all the configurations, namely merging all the tiles to their final files. On the other hand, the time spent in the map phase drops sharply at first and stabilizes to around 4 minutes as the number of Mappers increase. It drops initially because each Mapper processes less data as the number of Mapper increase, but as the number continues to increase (especially when it’s larger than 64), the time spent in the computation of Mappers becomes relatively small compared to the time spent in launching and coordinating all the Mappers. Also, the chances that one of the Mappers is much slower than others increase as the number of Mappers increase.

The result of the second experiments is shown in Figure 9 b). We vary the input data size and fix the number of Mappers and Reducers to 16. Sub linear complexity has been observed in this figure. This means the computation power of Mapper haven’t been saturated yet. Thus, doubling the size of the data doesn’t increase the time to twice of the original.

The first experiment for the image quality assessment problem uses a subset of the whole data set that is a re-sampled version of the original one. It is about 20GB and has 482 files with 1024 tiles each. The size of the files ranges from several megabytes to around 80 megabytes, and the number of Reducers is varied from 4 to 512. The second experiment uses different sized subsets of the original data set. The size of the files ranges from 2GB to 16GB, and the number of Reducers is fixed at 256.

In the first experiment, the number of Mappers is also fixed, determined by the data set size. Thus, the execution time of the map phase is similar through different runs, as can be seen in Figure 10 (a). The execution time slightly fluctuates because there were other concurrent jobs running in the cluster at the same time. As the number of Reducers increases, the execution time of the reduce phase largely decreases for smaller number of reducers, and less improvements are obtained for larger number of reducers. This is because the same amount of work is now shared by more Reducers. When the number of Reducers is larger than 64, the execution time of the reduce phase stabilizes to around 2.5 minutes. This could be explained by the launching time of Reducers dominating the whole time at this point. With 64 Reducers, each of them will be writing around 482/64





a) Variable Reducers b) Variable input data size

Figure 10 MapReduce job completion time for image quality computation

$\approx$ 8 files. The time taken to write 8, 4 (128 Reducers) or even less files is negligible compared with the launching time of that many Reducers.

In the second experiment, Figure 10 (b), as the size of the data set increases with constant number of reducers (256), the execution time of the map phase hardly changes, which is consistent with the data parallelization provided by the MapReduce model, that is, more Mappers are engaged in processing the data. The execution time of the reduce phase increases because there are now more files to be written with the same number of Reducers.

3.5.Summary

Due to its inherent parallelizable nature of many low level image processing operations, parallel computing has been a great tool for researchers of the image processing community to dramatically reduce the processing time of large data sets. As a new member in the parallel computing domain, MapReduce is not an exception. Its data parallel model turns out to be another invaluable tool to speedup many low level but important tasks in image processing before further analysis can be carried out. In this chapter, we have studied two real world applications that utilize the computation power of MapReduce to speedup raster data processing in spatial database domain. Experiments have shown the linear scalability of our algorithms.

Another lesson worth noting is that although the two phase (map and reduce) model of MapReduce seems to be stringent at first; some components are flexible enough for the model to be adapted to different situations. We have shown two such examples in our applications. The first one is to overriding the default rule in the shuffling phase of the image loading problem, which results different sorting and grouping criteria in the reduce phase. The other one is to define customized InputFormat to handle new format of the input data in the image quality assessment problem. These two examples showcase the flexibility existing in MapReduce.

We continue our discussion with MapReduce in the following chapters, but we switch to another domain. We show how MapReduce could be utilized in the algorithmic analysis domain. More specifically, we give two case studies, namely, Nonnegative Matrix Factorization (Chapter 4) and Tensor Factorization (Chapter 5).

4. NONNEGATIVE MATRIX FACTORIZATION

4.1.Background

Nonnegative matrix factorization (NMF) factorizes an input nonnegative matrix into nonnegative matrices of lower rank. Originally proposed for parts-of-whole interpretation of matrix factors, NMF has attracted a lot of attentions recently in data mining and machine learning communities. It is recently discovered that NMF can be used to solve challenging data mining and machine learning problems. For example, NMF with the sum of squared error cost function is equivalent to a relaxed K-means clustering, the most widely unsupervised learning algorithm [51] [52]. In addition, NMF with the I-divergence cost function is equivalent to probabilistic latent semantic indexing, another unsupervised learning method popularly used in text analysis [53] [54]. Consequently, NMF has been shown to be useful in a variety of applied settings, including environmetrics, chemometrics, pattern recognition, multimedia data analysis, text mining, web mining, and DNA gene expression analysis.

Due to the increasing availability of massive data sets, researchers are facing the problem of factorizing matrices of dimensions in the orders of millions. Recent research [55] has shown that it is possible to factorize such gigantic matrices within tens of hours using the MapReduce distributed programming platform. In this section, we propose three different matrix multiplication implementations using MapReduce, based on which we successfully parallelize three different types of NMF algorithms on the MapReduce platform. We evaluate our algorithms on a cluster provided by Google and IBM through the NSF Cluster Exploratory (CLuE) program [33] using both synthetic and real data sets. Results have shown that the performance of our proposed algorithm is at least as good as previous efforts.

Different from the work by Liu et al. [55] whose implementations are directly built on the updating rules, we reduce the problem of NMF to a series of different matrix multiplication implementations on the MapReduce platform, which makes our algorithms applicable to most matrix factorization algorithms that are using the multiplicative updating scheme.

In summary, the key contributions of this work are summarized below:

- We propose three different matrix multiplication implementations on MapReduce.
- We scale up three different types of NMF algorithms on MapReduce based on our proposed matrix multiplication implementations.
- We implement and experimentally evaluate our algorithms using both synthetic and real data sets on a real-world cluster.

The rest of this chapter is organized as follows. Section 4.2 discusses related works, mainly focusing on parallel matrix multiplication and large-scale data mining two aspects. Section 4.3 defines three different types of nonnegative matrix factorization and introduces their updating algorithms respectively. Section 4.4 describes three schemes to perform matrix multiplication using MapReduce based on the properties of the matrices. These schemes are then used to implement the updating algorithms in Section 4.3. We report the experimental evaluation in Section 4.5 and conclude the work in Section 4.6.

4.2.Related Work

Two particular lines of research are related to our work. One is existing works in parallel matrix multiplication and NMF. The other is using MapReduce in large-scale data mining and machine learning.

4.2.1. Parallel Matrix Multiplication and NMF

Because of the importance of matrix multiplication as a basic operation in linear algebra, parallel matrix multiplication has been studied extensively. One of the most popular algorithms might be Cannon's algorithm [56]. Although most of the algorithms assume special data layout and are tied to a particular parallel architecture, some basic ideas still could be applied to MapReduce. Algorithms designed for single instruction multiple data (SIMD) systems are of particular interests, for the resembalance between MapReduce and SIMD. For example, the MM-2 scheme we proposed in Section 4.4.1 could also be implemented directly on an SIMD system by replacing a machine in the cluster with a processor in the system. From a relational algebra perspective, what MM-2 does is a join on the column id of the left matrix and the row id of the right matrix. More detailed discussions could be found in [57] [58].

Because of the popularity of NMF, there are many works in trying to parallelize it [59] [60]. Since data sharing and communication are no longer light-weight in distributed clusters like MapReduce, those methods cannot be ported directly to MapReduce.

4.2.2. Large-scale data mining using MapReduce

Although the initial purpose of MapReduce is to perform large-scale data processing [5], it turns out that this model is much more expressive than that [5]. Chu et al. investigated the possibility to implement machine learning algorithms using MapReduce on multicore [19]. Their conclusion is that a variety of learning algorithms that fit the statistic query model [61] could be parallelized using MapReduce. An open source project Mahout [62] has been started to port those algorithms to Hadoop. Papadimitriou and Sun have done a case study in data mining on co-clustering [20] towards petabyte scale of data using MapReduce. MapReduce has also been used in many other tasks of data mining and machine learning. Those works include but not limited to Kang et al. on graph mining [21], Panda et al. on tree ensembles [22], Liu et al. on bayesian browsing model [23], and Chen et al. on behavioral targeting [63].

In particular, Liu et al. successfully scaled up the classical NMF [64] for web-scale dyadic data analysis on MapReduce [55]. They assumed the matrix to be factorized is stored as $(i, j, A_{i,j})$ tuples that are spread across machines and proposed different partitions for the factors. Although the techniques they used are different, they could be reduced to one of the matrix multiplication schemes we proposed in Section 4.4.

4.3.Nonnegative Matrix Factorization

Let $X \in \mathbb{R}^{+^{m\times n}}$ be the input data matrix, we define three different matrix factorizations and introduce their updating rules in the following.

A. *NMF*

The classical NMF [64] could be written as

$$X \approx FG^T$$

where $F \in \mathbb{R}^{+^{m\times k}}, G \in \mathbb{R}^{+^{n\times k}}$. The updating rules for NMF are listed as follows:

$$F_{ik} \leftarrow F_{ik}\frac{(XG)_{ik}}{(FG^TG)_{ik}},$$

$$G_{jk} \leftarrow G_{jk}\frac{(X^TF)_{jk}}{(GF^TF)_{jk}}.$$

B. *Convex-NMF*

In general, the basis vectors F have the meaning of cluster centroids. To enforce this geometry meaning, F can be restricted to be a convex combination of the input data points [65]. In other words, in addition to have F to be nonnegative, we restrict each column of F to lie within the space spanned by the columns of X, i.e., $f_l = w_{1l}x_1 + \cdots + w_{nl}x_n = Xw_l$ or $F = XW$, where f_l and $x_i$is a column vector of $F$and X, $w_{il}$is an element of $W, and\ w_{il} \geq 0$. We call this restricted form of factorization as Convex-NMF [65]. The updating rules of this factorization are

$$G_{ik} \leftarrow G_{ik}\sqrt{\frac{(X^TXW)_{ik}}{(GW^TX^TXW)_{ik}}},$$

$$W_{ik} \leftarrow W_{ik}\sqrt{\frac{(X^TXG)_{ik}}{(X^TXWG^TG)_{ik}}}.$$

C. Tri-Factorization

Tri-factorization [66] is useful to simultaneously cluster the rows and the columns of the input data matrix X. We consider the following nonnegative 3-factor decomposition

$X \approx FSG^T$, where $F \in \mathbb{R}^{+^{m\times k}}, S \in \mathbb{R}^{+^{k\times k}}, G \in \mathbb{R}^{+^{n\times k}}$. Note that S provides additional degrees of freedom such that the low-rank matrix representation remains accurate while F gives row clusters and G gives column clusters. The updating rules of tri-factorization are

$$G_{jk} \leftarrow G_{jk}\sqrt{\frac{(X^TFS)_{jk}}{(GG^TX^TFS)_{jk}}},$$

$$F_{ik} \leftarrow F_{ik}\sqrt{\frac{(XGS^T)_{ik}}{(FF^TXGS^T)_{ik}}},$$

$$S_{ik} \leftarrow S_{ik}\sqrt{\frac{(F^TXG)_{ik}}{(F^TFSG^TG)_{ik}}}.$$

4.4.NMF using MapReduce

4.4.1. Matrix Multiplication in MapReduce

The major operation in the updating rules of all three types of factorizations is matrix multiplication. Thus, efficient implementation of matrix multiplication on MapReduce is the key to scale up matrix factorization.

Assume $A \in \mathbb{R}^{+^{m\times n}}$ and $B \in \mathbb{R}^{+^{n\times k}}$. Traditionally, AB is defined as the inner product of the row vectors of A and column vectors of B. Thus,

$$AB = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_m \end{pmatrix} \begin{pmatrix} b_1 & b_2 & \cdots & b_k \end{pmatrix} = \begin{pmatrix} a_1 b_1 & \cdots & a_1 b_k \\ \vdots & \ddots & \vdots \\ a_m b_1 & \cdots & a_m b_k \end{pmatrix}.$$

This kind of decomposition is useful if we could share B across all rows of A. Each row of AB could be computed in parallel. A MapReduce job MM-1 is formulized in Table 4 to carry out this computation.

Table 4 Three Matrix Multiplication Schemes

Scheme		Input/Output	
		Input	Output
MM-1	Map	$\langle i, a_{i*} \rangle$	$\langle i, a_{i*}B \rangle$
	Reduce	None	
MM-2	Map	$\langle i, a_{*i} \rangle$ or $\langle i, b_{i*} \rangle$	$\langle i, a_{*i} \rangle$ or $\langle i, b_{i*} \rangle$
	Reduce	$\langle i, [a_i, b_i] \rangle$	$[\langle j, c_{j*} \rangle]$ or $[\langle j, c_{*j} \rangle]$
	Map	$\langle j, c_j \rangle$	$\langle j, c_j \rangle$
	Reduce	$\langle j, [c_{j_1} c_{j_2} \cdots] \rangle$	$\langle j, c_j' \rangle$
MM-3	Map	$\langle i, a_{i*} \rangle$	$[\langle j, c_j \rangle]$
	Reduce	$\langle j, [c_{j_1} c_{j_2} \cdots] \rangle$	$\langle j, c_j' \rangle$

We use a_{i*} to denote a row vector with id i and a_{*i} to denote a column vector. We use a_i to denote either a row vector or column vector if it doesn't matter which one it is. a_i' is used to denote the final result aggregated from all the a_i `s. We use an angle bracket to denote a key/value pair and a square bracket to denote a list of key/value pairs.

As can be seen from Table 4, no reduce phase is needed for MM-1. The input to the map phase is the row vectors of A. When the size of B gets large enough, this scheme will fail since the cost of sharing a large matrix across many nodes in a cluster is too expensive.

Alternatively, we could decompose A into column vectors and B into row vectors. Thus,

$$AB = (a_1 \quad a_2 \quad \cdots \quad a_n)\begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} = \sum a_i \otimes b_i.$$

The implication of this decomposition is that if we could partition A and B in such a way that the corresponding columns of A and rows of B are on the same machine, we could compute the outer product in parallel and merge them together to obtain *AB*. This scheme fits the MapReduce model perfectly. No data sharing is required as in the previous scheme. We could use two MapReduce jobs to implement this scheme MM-2, which is shown in Table 4.

Now let's examine these two jobs of MM-2 in detail. The first job does multiplication and the second job does summation. The map phase of the multiplication job accepts a column vector a_{*i} or a row vector b_{i*}, and emits an intermediate pair with column id or row id as key and the column or row itself as value. Column and row vectors with the same id are shuffled to the same machine as the input for the reduce phase. In the reduce phase, we compute the outer product of a_{*i} and b_{i*} and, depending on the context in which this multiplication is used, we either output a bunch of row vectors or column vectors, which is denoted in Table 4 as $[\langle j, c_{j*}\rangle]$ or $[\langle j, c_{*j}\rangle]$. These row or column vectors are partial results of the vector. We merge them into the final vector in the summation job. The map phase of the summation job is an identity function, which just outputs the input key/value pairs for shuffling. The reduce phase of this job merges all the row vectors with the same id or column vectors with the same id and emits a final row vector or column vector.

This scheme can be used to compute the product of any matrices, no matter it is dense or sparse, as long as the left matrix is partitioned into column vectors and the right matrix is partitioned into row vectors. Under certain circumstance, the two MapReduce jobs can be merged into one. For example, consider the matrix multiplication $A^T A$. Assume A is partitioned in row vector, then A^T in column vector is the same as A in row vector. Thus, we could compute $A^T A$ using only one MapReduce job as shown in MM-3 of Table 4 where multiplication is conducted in map phase and summation is performed in reduce phase. This shortcut saves one MapReduce job and turns out to be very useful in matrix factorization, where multiplications like $A^T A$ appear frequently.

4.4.2. NMF

We partition X, F and G into row vectors, which renders the following view:

$$X = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{pmatrix} F = \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_m \end{pmatrix} \; and \; G = \begin{pmatrix} g_1 \\ g_2 \\ \vdots \\ g_n \end{pmatrix},$$

where x_i is a n-dimensional row vector and f_i and g_i are k dimensional row vectors. All of them are stored as sets of $\langle i, x_i \rangle$, $\langle i, f_i \rangle$ and $\langle i, g_i \rangle$ key/value pairs. There are three steps to update G. The first step is to compute the numerator $X^T F$ in (2). The second step is to compute the denominator $GF^T F$ in (2). And the last step is to update G.

Figure 11 shows the flow chart of updating G. We discuss them in the following one by one.

A. *Computing $A = X^T F$*

Since we partition X and F in row vectors, $X^T F$ is ready for multiplication. Using the scheme MM-2 in Table 4, we could compute $X^T F$ with two MapReduce jobs.

B. *Computing $B = GF^T F$*

There are two ways to compute $GF^T F$. The first one is to compute GF^T first and $GF^T F$ afterwards. This requires 4 MapReduce jobs, 2 for each multiplication. The second one is much faster. By using MM-3 to compute $F^T F$ first, and using MM-1 to compute $GF^T F$, only two jobs are needed. Noting that the result of $F^T F$ is a $k \times k$ matrix which is reasonably small to be fit into memory for computing $GF^T F$ with G.

C. *Updating G*

Once both $X^T F$ and $GF^T F$ have been computed; only one job is needed to update F, which is summarized as follow:

- Map: Map $\langle i, g_i \rangle$, $\langle i, a_i \rangle$ and $\langle i, b_i \rangle$ as they are
- Reduce: Take $\langle i, [g_i, a_i, b_i] \rangle$ and emit$\langle i, g_i' \rangle$, where $g_i' = g_{i,j} \frac{a_{i,j}}{b_{i,j}}$.

This finishes the update of G. F is updated in the same manner, except that when computing XG, we need X to be partitioned into column vectors, whereas it is stored as row vectors. Thus, an additional MapReduce job is needed to transform X from row vectors to column vectors. This won't affect the overall performance, since only one such job is needed for the entire updating cycle, which usually consists of a few iterations.

4.4.3. Convex-NMF and Tri-Factorization

Convex-NMF and Tri-Factorization are essentially computed in the same manner as NMF, namely, to update a matrix, first compute the numerator of the updating rule, and then compute the denominator, and finally update the original matrix. In the following, we focus on their difference with NMF on the updating rules, rather than the details.

A. Convex-NMF

We partition X, G and W into row vectors as we did in NMF. One notable pattern of the updating rules of Convex-NMF is that $X^T XW$ has appeared three times, two in (3) and one in (4). Thus, it is beneficial to first compute $X^T XW$ and use the result later.

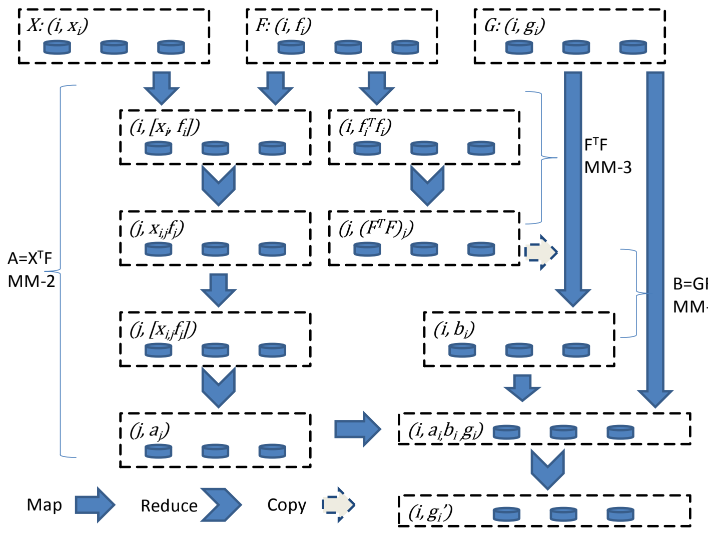


Figure 11 Computing $G_{jk} \leftarrow G_{jk} \frac{(X^T F)_{jk}}{(GF^T F)_{jk}}$.

Again, we could use the MM-3 to compute $X^T X$ first and $X^T XW$ afterwards. However, this time we decide not to do that, because the result of $X^T X$ would be a huge dense matrix that could introduce a lot of data shuffling in later computation. We choose to compute XW first and X^TXW afterwards using MM-2 in Table 4.

After computing XW, we use two additional MM-2s to compute the denominator in (3). The numerator in (4) is computed in the same manner as X^TXW. When computing the denominator in (4), we first use MM-3 to compute G^TG, and then take the result of X^TXW and do a MM-1 job to finish the computation.

B. Tri-Factorization

Again, we partition X,G, F and S into row vector. Repeated patterns like X^TFS in (5) and XGS^Tin (6) are utilized to reduce the computation. To update G, we first compute the numerator X^TFS in (5) using MM-2 ($A = X^TF$) and MM-1 ($B = AS$). The result is then used to compute the denominator in (5) using MM-2 ($C = G^TB$) and MM-1 (GC). F is updated similarly.

The updating rule of S has its own structure and S is updated slightly differently. First of all, the numerator of S is computed using two MM-2s ($A = F^TX\ and\ AG$). We use MM-3 to compute both $B = F^TF$ and $C = G^TG$ in (7). And then two MM-1 operations are used to compute $D = BS$ and DC. Finally, an update job is used to update the new value of S.

4.5.Experiments

We use the open source implementation of MapReduce – Hadoop [31] in our experiments. All three factorization algorithms have been implemented in Java for Hadoop.

4.5.1. Experiments Setup

All the experiments were conducted in the environment described in section 3.4.1. Both synthetic and real data sets were used in the experiments. Synthetic data sets were generated by a random matrix generator, which generates a matrix $X \in {\mathbb{R}^+}^{m \times n}$ with sparsity δ on given parameters m, n and δ. We varied m, n and δ to see how the performance varies. While the number of participating machines in the map phase cannot be set directly, we varied the number of machines R in the reduce phase to see how more participating machines could improve the performance. Unless explicitly pointed out, all the reported time is for one iteration of the algorithms.

The blog data was collected by an NEC in-house blog crawler. Given seeds of manually picked highly ranked blogs, the crawler discovered blogs that are densely connected with the seeds, resulting in an expanded set of blogs that communicate with each other [67][68]. The data set is represented as a sparse matrix that is of dimension 274,679 by 5,304. Each row is a blog entry and the columns contain word frequencies in the blogs. Table 5 summarizes the characteristics of the data sets used in the experiments.

Table 5 Data sets description

Data Sets	M	N	δ	Size(MB)
Synthetic data sets	$2^{12} \sim 2^{24}$	$2^{11} \sim 2^{12}$	$2^{-10} \sim 2^{-5}$	1~6605
Real data set	274,649	5,304	0.008	47

In the following sections, we first examine the computation cost of each component in (2). Then, we examine the scalability of the algorithms with the size of the input matrices. We also present how the performance varies w.r.t δ, k and R in section 4.5.3. Finally, we evaluate the algorithms on the real data set.

4.5.2. Computation Cost

The computation cost of each MapReduce job in updating G is shown in Table 6. We recorded the data shuffled between Map phase and Reduce phase in MB and the total elapsed time in seconds for each job. We choose $m = 2^{22}, n = 2^{12}, k = 8, \delta = 2^{-7}$ for the matrix being factorized in this experiment, which is typical for a large data set.

TABLE 6 COMPUTATION COST OF UPDATING G IN NMF

Component		$\boldsymbol{m = 2^{22}, n = 2^{12}, k = 8, \delta = 2^{-7} and\ R = 8}$	
		Shuffle(MB)	Time(sec)
$X^T F$	Multiplication	2099	242
	Summation	62	96
$F^T F$		0.007	141
$GF^T F$		0	26
Update G		1.4	45

As can be seen from Table 6, the major cost in updating $G$is the computation of $X^T F$and$F^T F$, which accounts for 87 percent of the total elapsed time. This is understandable because the inner dimension of both of the two multiplications is in the order of million. Another notable pattern is the imbalance in the size of data shuffling. The multiplication job of $X^T F$ shuffled more than 2 GB of data, while $F^T F$ only shuffled 7 KB. This is because we used MM-2 to compute $X^T F$, whose Map phase only read the input for grouping. Thus, both $X^T$and F were shuffled. On the other hand, we used MM-3 to compute $F^T F$, where the multiplication was performed in Map phase and only the small resulting $k \times k$ matrix was shuffled.

4.5.3. Scalability and Performance *w.r.t* δ, k and R

We also report the elapsed time of one iteration for all three algorithms in Figure 14. In these experiments, we fixed $k = 8, \delta = 2^{-7}, R = 8 \; and \; n = 2^{11}$and varied m from 2^{12} to 2^{24}.

Figure 15 shows how the performance varies w.r.t the sparsity δ. As δ goes from 2^{-10} to 2^{-5}, the number of nonzero elements in the matrix increases from 16 million to 600 million, and the elapsed time also increases in proportion to that.

Figure 12 reveals the linearity between elapsed time and the dimensionality of k. As k doubles from 8 to 128, the elapsed time gradually increases from 10 minutes to 40 minutes. The slope is smaller than 1, which is good for large data sets.

Finally, we plot the speedup achieved by doubling the number of machines in the reduce phases of MM-2 and MM-3 in Figure 13. Two series are plotted. One shows the speedup for one single iteration of the NMF algorithm; the other shows the speedup achieved by the affected MapReduce jobs only, that is, MM-2 and MM-3. There are jobs that don’t benefit from adding more machines, such as MM-1 and the updating of the original matrix. Thus, the speedup for one single iteration is smaller than that of MM-2 and MM-3.

4.5.4. Experiments with Real Data Set

Finally, we ran our algorithm on the real world blog data set. The result is reported in Figure 16. We divided this data set into five partitions and ran our algorithms on 20%, 40%, 60%, 80%, and the whole data set. Again, we observed linear scalability with regarding to this data set.

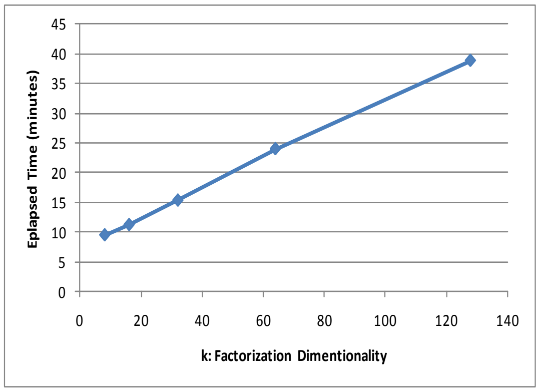


Figure 12 Elapsed time w.r.t k

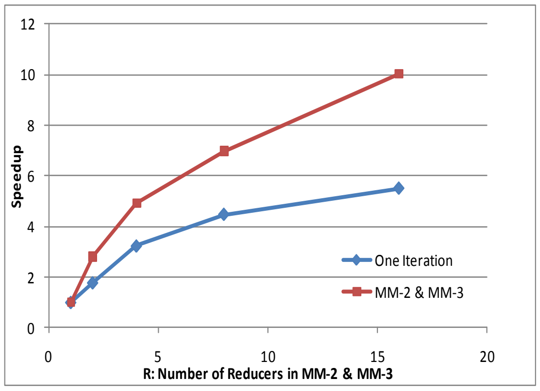
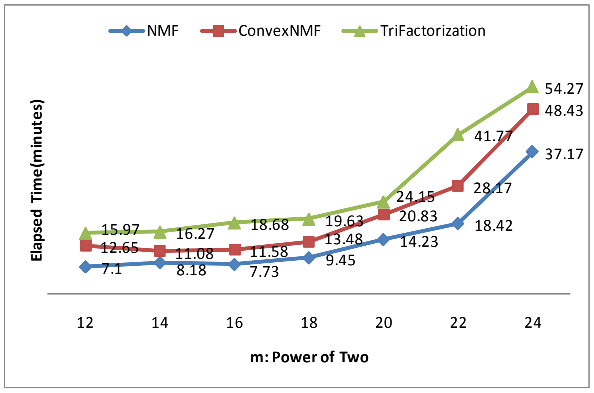


Figure 13 Speedup w.r.t R



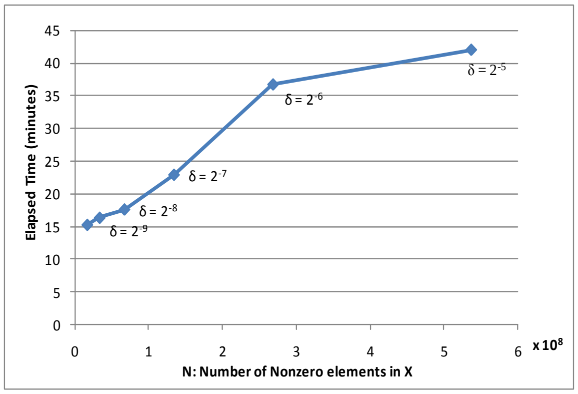


Figure 14 Elapsed time w.r.t m

Figure 15 Elapsed time w.r.t δ

Due to the relatively small size of this data set compared to the synthetic data sets, most of the elapsed time is spent in starting and cleaning up MapReduce jobs, which results a near-flat line in Figure 16.

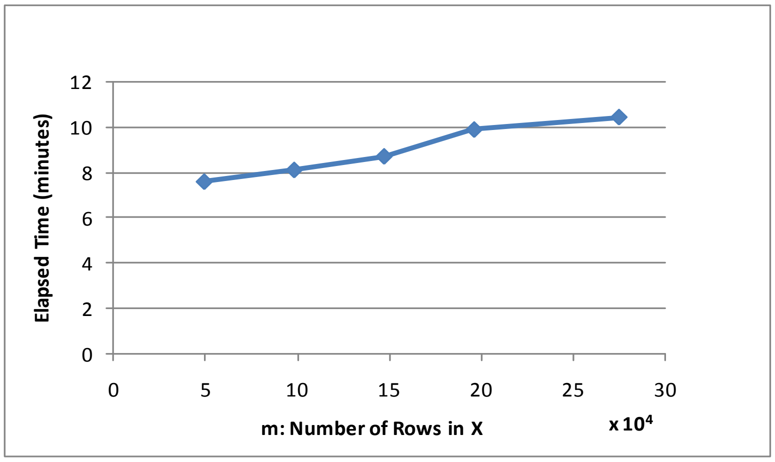


Figure 16 Elapsed time w.r.t m for real data set

4.6.Summary

In this chapter, we presented three different implementations of matrix multiplication on MapReduce depending on the properties of the matrices. Based on that, we successfully scaled up three different types of nonnegative matrix factorization algorithms. Matrices of dimension million-by-thousand with millions of nonzero elements can be factorized within several hours on a MapReduce cluster.

Ad demonstrated by the design of these algorithms, MapReduce can also be used to do computation intensive tasks such as matrix multiplication. The key is how to formulate the algorithm in a functional way that can be expressed in MapReduce.

There are many avenues for future work on large-scale matrix factorization. First, our current work is focused on the multiplicative updating rules used in NMF which could be reduced to applications of different matrix multiplications. One interesting direction is to investigate schemes for scaling up other NMF algorithm such as alternating non-negative least squares [69] and projective gradient descent [70]. Second, recently tensor factorization, as a generalization of matrix factorization, has attracted a lot of research attention [71]. It is thus interesting to study schemes for scaling up large-scale tensor factorization, which is the subject of next chapter. Last but not least, we would also like to explore various applications of NMF on large-scale data sets.

5. TENSOR DECOMPOSITION

5.1. Background

Tensors are multidimensional arrays. The earliest work dates back to 1927 from Hitchcock [72] [73]. There has been active research on tensor decompositions since ‘60s. The work of Tucker [74] [75] [76] and Carroll and Chang [77] and Harshman [78] in psychometrics have brought great attention to this topic. Appellof and Davidson [79] introduced tensor decomposition into the field of chemometrics. And a lot of works [80] [81] [82] [83] have appeared in that field afterwards. In the field of algebraic complexity, there are also many works. The most famous example might be Strassen matrix multiplication, which is an application of a decomposition of a $4 \times 4 \times 4$ tensor to describe 2×2 matrix multiplication [84] [85].

The popularity of tensor decomposition doesn’t stop in the field of psychometrics and chemometrics. In the recent years, interest has expanded to many other fields, such as signal processing [86], computer vision [87], data mining [88], machine learning [89], graph analysis [90], and so on so forth.

In this chapter, we discuss how to scale up two most famous tensor decompositions in MapReduce, namely, the CANDECOMP/PARAFAC (CP) [77] [78] and Tucker [76] tensor decomposition.

The contributions of this work are as follow:

- We propose MapReduce implementations for the Hadamard product, the Khatri-Rao product, and the Moore-Penrose pseudo inverse.
- We scale up the CP and Tucker tensor decomposition in MapReduce based on our implementation of the Hadamard product, the Khatri-Rao product, and the Moore-Penrose pseudo inverse.
- We implement and experimentally evaluate our algorithms using synthetic data sets on a real-world cluster.

The remainder of this chapter is organized as follows. Section 5.1 briefly reviews the history of tensor decomposition and introduces some notations and concepts on tensor. Rather than a complete review, the emphasis of this introduction is on the concepts that are necessary for the understanding of the two decompositions. More detailed discussions could be found in other surveys [10]. Efforts are made to keep the discussion as consistent as possible with the terminology of previous publications in the field. Section 5.3 discusses the related works. The CP decomposition and Tucker decomposition are introduced in section 5.2 with the most popular algorithms to solve them. Followed are our design and implementation of the two decompositions in MapReduce. We evaluated our algorithms in section 5.5 and conclude our work in section 5.6.

5.1.1. Tensor and its Notation

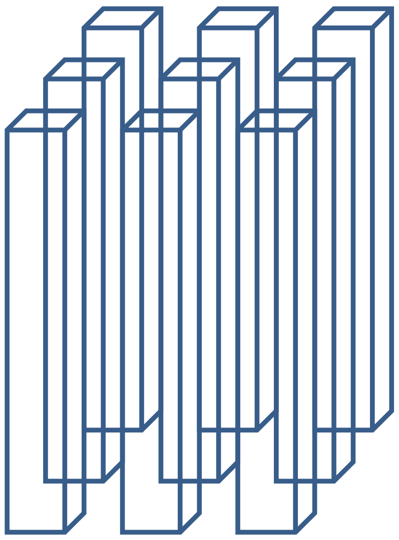
Put it in the simplest way, a tensor is a multidimensional or N-way array. More formally, an N-way or Nth-order tensor is an element of the tensor product of N vector spaces, each of which has its own coordinate system. The notations used in our discussion are very similar to [10], which in turn resembles to that proposed by Kiers [91].

The number of dimensions of a tensor is called its order, also known as ways or modes. A first-order tensor is a vector, a second-order tensor is a matrix and tensors of order three or higher are called higher-order tensors. In our discussion, vectors are denoted by boldface lowercase letters, e.g., **a**. Matrices are denoted by boldface capital letters, e.g., **A**. Higher-order tensors are denoted by boldface italic capital letters, e.g., ***A***. Scalars are denoted by lowercase letters, e.g., a.

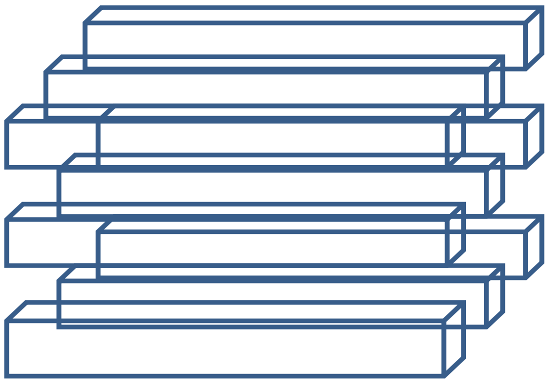
The *i*th element of a vector **a** is denoted by a_i, element (i, j) of a matrix **A** is denoted by a_{ij}, and element (i, j, k) of a third-order tensor ***A*** is denoted by a_{ijk}. Indices usually range from 1 to their capital version, e.g., $i = 1, \cdots,$ I. The *n*th element in a sequence is denoted by a superscript in parentheses, e.g., $\mathbf{A}^{(n)}$ denotes the *n*th matrix in a sequence.

A colon is used to indicate all elements of a mode. Thus, for matrices, the *j*th column of **A** is denoted by $\boldsymbol{a}_{:j}$, and the *i*th row of a matrix **A** is denoted by $\boldsymbol{a}_{i:}$. Sometimes we omit the colon for compactness if it is unambiguous in the context.

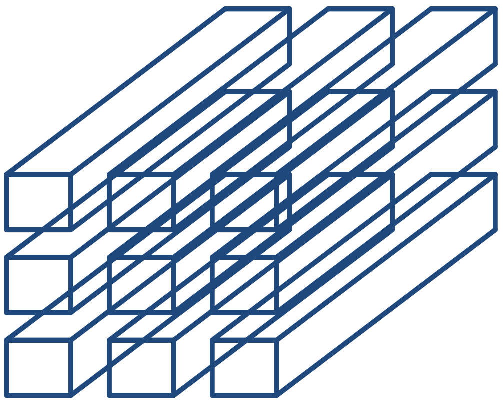
Higher-order analogue of matrix rows and columns are called fibers. A *fiber* is defined by fixing every index but one. Mode-1 fiber is a matrix column and mode-2 fiber is a matrix row. Thus, third-order tensors have three different kinds of fibers, namely, column, row, and tube fibers, denoted by $\boldsymbol{a}_{:jk}$, $\boldsymbol{a}_{i:k}$, and $\boldsymbol{a}_{ij:}$. Figure 17 shows the fibers of an example of third-order tensor.



a) Mode-1(column) fiber

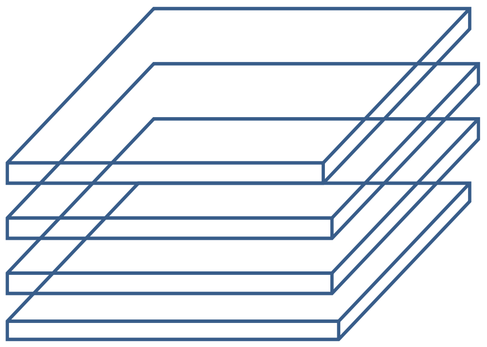


b) Mode-2(row) fiber

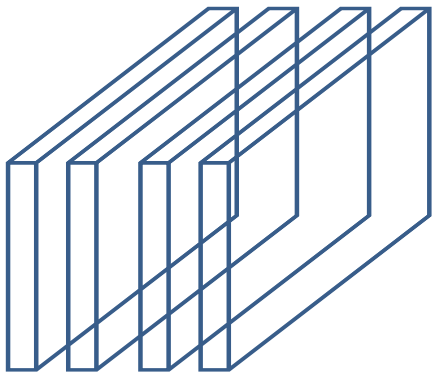


c) Mode-3(tube) fiber

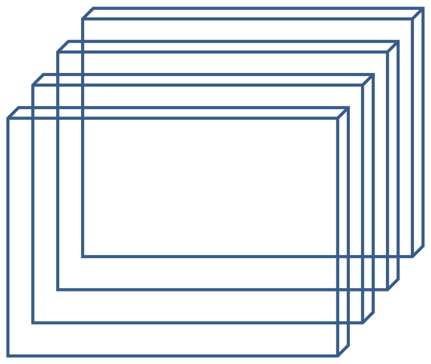
Figure 17 Fibers of a third-order 3×3×3 tensor



a) Horizontal slices



b) Lateral slices



c) Frontal slices

Figure 18 Slices of a 3rd-order tensor

When all indices but two are fixed, a *slice* is defined. It can be viewed as two-dimensional sections of a tensor. Figure 18 shows the horizontal, lateral, and frontal slides of a third-order tensor $\boldsymbol{A}$, denoted by $\boldsymbol{A}_{i::}$, $\boldsymbol{A}_{:j:}$, and $\boldsymbol{A}_{::k}$. For the sake of compactness, we sometimes omit the colons in the notation of slice. Thus, the kth frontal slice of a third-order tensor, $\boldsymbol{A}_{::k}$, would be denoted as $\boldsymbol{A}_k$.

An N-way tensor $\boldsymbol{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ is rank one if it can be written as the outer product of N vectors, i.e.,

$$\boldsymbol{X} = \boldsymbol{a}^{(1)} \circ \boldsymbol{a}^{(2)} \circ \cdots \circ \boldsymbol{a}^{(N)}$$

The symbol "○" represents the vector outer product. Thus, each element of the tensor is the product of the corresponding vector elements:

$$x_{i_1 i_2 \cdots i_N} = \boldsymbol{a}_{i_1}^{(1)} \boldsymbol{a}_{i_2}^{(2)} \cdots \boldsymbol{a}_{i_N}^{(N)} \text{ for all } 1 \le i_n \le I_n.$$

Figure 19 shows a third-order rank-one tensor.

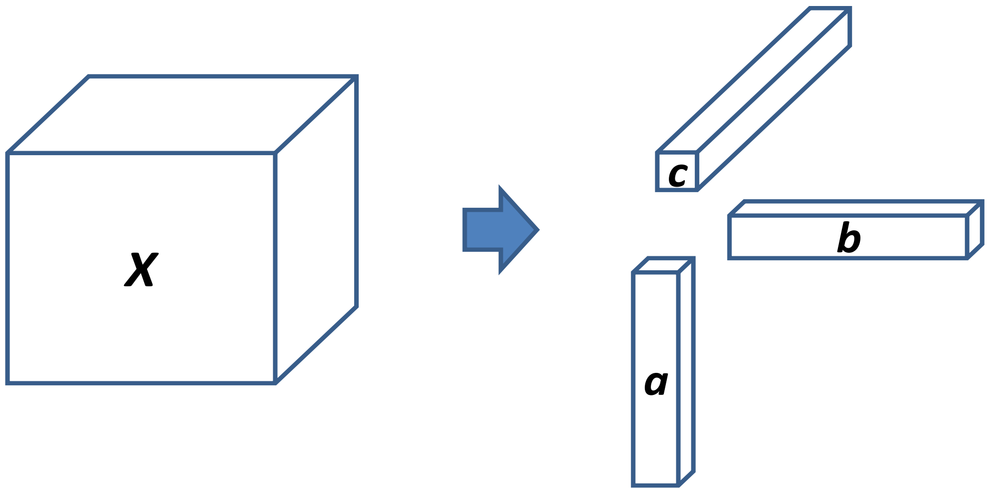


Figure 19 Rank-one third-order tensor

5.1.2. Matricization

Matricization is the process of reordering the elements of a tensor into a matrix, which is also known as *unfolding* or *flattening*. For example, a $2\times 3\times 4$ tensor can be arranged as a 6×4 matrix, a 3×8 matrix or a 2×12 matrix depending on which mode of the fibers you take. The mode-n matricization of a tensor $\boldsymbol{X} \in \mathbb{R}^{I_1\times I_2\times\cdots\times I_N}$ is denoted by $\boldsymbol{X}_{(n)}$ and arranges the mode-n fibers to be the columns of the resulting matrix. An example is given below to illustrate this concept. Let the frontal slices of $\boldsymbol{X} \in \mathbb{R}^{2\times 3\times 2}$ be $\boldsymbol{X}_{(1)} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$, $\boldsymbol{X}_{(2)} = \begin{bmatrix} 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix}$. The three mode-n matricizations are

$$\boldsymbol{X}_{(1)} = \begin{bmatrix} 1 & 2 & 3 & 7 & 8 & 9 \\ 4 & 5 & 6 & 10 & 11 & 12 \end{bmatrix},$$

$$\boldsymbol{X}_{(2)} = \begin{bmatrix} 1 & 4 & 7 & 10 \\ 2 & 5 & 8 & 11 \\ 3 & 6 & 9 & 12 \end{bmatrix},$$

$$\boldsymbol{X}_{(3)} = \begin{bmatrix} 1 & 4 & 2 & 5 & 3 & 6 \\ 7 & 10 & 8 & 11 & 9 & 12 \end{bmatrix}.$$

5.1.3. Tensor Multiplication

Tensor multiplication is much more complex than matrices. We only consider the tensor *n*-mode product here, i.e., multiplying a tensor by a matrix (or a vector) in mode *n*. More detailed discussions could be found in Bader and Kolda [92].

The *n-mode product* of a tensor $\boldsymbol{X} \in \mathbb{R}^{I_1\times I_2\times\cdots\times I_N}$ with a matrix $\mathbf{U} \in \mathbb{R}^{J\times I_n}$ is denoted by $\boldsymbol{X} \times_{\boldsymbol{n}} \mathbf{U}$ and is of size $I_1 \times \cdots \times I_{n-1} \times J \times I_{n+1} \times \cdots \times I_N$. Each element of the product is defined as:

$$(\boldsymbol{X} \times_{\boldsymbol{n}} \mathbf{U})_{i_1 \cdots i_{n-1} j i_{n+1} \cdots i_N} = \sum_{i_n=1}^{I_n} x_{i_1 i_2 \cdots i_N} u_{j i_n}.$$

Another interpretation of this product is that each mode-*n* fiber is multiplied by the matrix **U**. In the terminology of matricization, this could be expressed as:

$$\boldsymbol{Y} = \boldsymbol{X} \times_n \mathbf{U} \Leftrightarrow \mathbf{Y}_{(n)} = \mathbf{U}\mathbf{X}_{(n)}$$

5.1.4. Matrix Kronecker, Khatri-Rao, and Hadamard Products

Some matrix products are important in tensor decomposition, so we briefly discuss them in this section.

The *Kronecker product* of matrices $\mathbf{A} \in \mathbb{R}^{\mathrm{I} \times \mathrm{J}}$ and $\mathbf{B} \in \mathbb{R}^{\mathrm{K} \times \mathrm{L}}$ is denoted by $\mathbf{A} \otimes \mathbf{B}$. The result is a matrix of size $(IK) \times (JL)$ and defined by

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{11}\mathbf{B} & a_{12}\mathbf{B} & \cdots & a_{1J}\mathbf{B} \\ a_{21}\mathbf{B} & a_{22}\mathbf{B} & \cdots & a_{2J}\mathbf{B} \\ \vdots & \vdots & \ddots & \vdots \\ a_{I1}\mathbf{B} & a_{I2}\mathbf{B} & \cdots & a_{IJ}\mathbf{B} \end{bmatrix}$$

$$= [\boldsymbol{a}_1 \otimes \boldsymbol{b}_1 \quad \boldsymbol{a}_1 \otimes \boldsymbol{b}_2 \quad \boldsymbol{a}_1 \otimes \boldsymbol{b}_3 \quad \cdots \quad \boldsymbol{a}_J \otimes \boldsymbol{b}_{L-1} \quad a_J \otimes \boldsymbol{b}_L].$$

The *Khatri-Rao product* is the "matching columnwise" Kronecker product [93]. It is denoted by $\mathbf{A} \odot \mathbf{B}$, where $\mathbf{A} \in \mathbb{R}^{\mathrm{I} \times \mathrm{K}}$ and $\mathbf{B} \in \mathbb{R}^{\mathrm{J} \times \mathrm{K}}$. The result is a matrix of size $(IJ) \times K$ defined by

$$A \odot B = [\boldsymbol{a}_1 \otimes \boldsymbol{b}_1 \quad \boldsymbol{a}_2 \otimes \boldsymbol{b}_2 \quad \cdots \quad \boldsymbol{a}_K \otimes \boldsymbol{b}_K].$$

The *Hadamard product* is the element-wise matrix product. Given two matrices **A** and **B** of the same size $I \times J$, their Hadamard product is denoted by $\mathbf{A} * \mathbf{B}$. The result is also of size $I \times J$ and defined by

$$\mathbf{A} * \mathbf{B} = \begin{bmatrix} a_{11}b_{11} & a_{12}b_{12} & \cdots & a_{1J}b_{1J} \\ a_{21}b_{21} & a_{22}b_{22} & \cdots & a_{2J}b_{2J} \\ \vdots & \vdots & \ddots & \vdots \\ a_{I1}b_{I1} & a_{I2}b_{I2} & \cdots & a_{IJ}b_{IJ} \end{bmatrix}.$$

5.2. Tensor Decomposition

5.2.1. The CANDECOMP/PARAFAC Decomposition

The CANDECOMP/PARAFAC decomposition was introduced to the psychometrics community by Carroll and Chang (canonical decomposition) [77] and

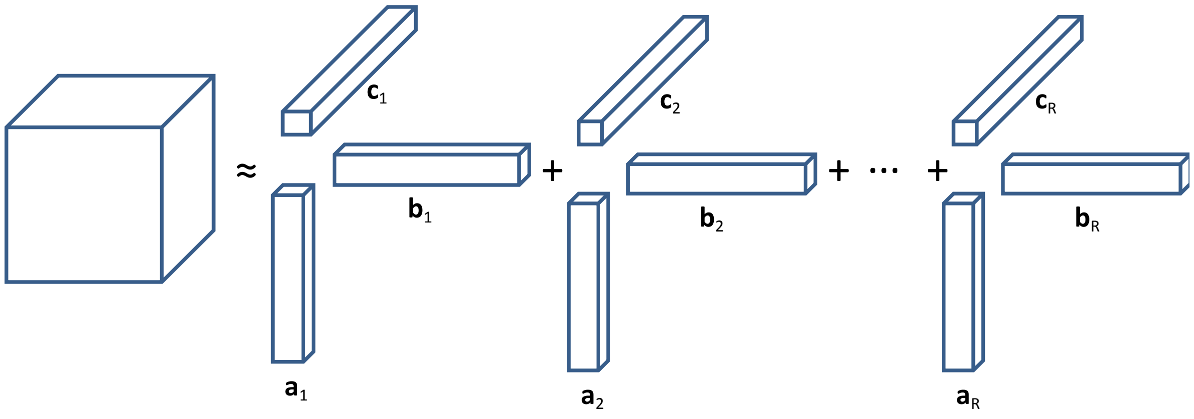


Figure 20 CP decomposition of a 3rd-order tensor.

Harshman (parallel factors) [78]. Although it was proposed by Hitchcock in 1927 [72] [73], it didn't become popular until the 70s. We refer to the CANDECOMP/PARAFAC decomposition as CP, per Kiers [91].

Figure 20 shows the CP decomposition of a 3rd-order tensor. The CP decomposition factorizes a tensor into a sum of component rank-one tensors. Recall that a rank-one tensor can be written as the outer product of *N* vectors, where *N* is the mode of the tensor. Take a third-order tensor $\boldsymbol{X} \in \mathbb{R}^{\boldsymbol{I\times J\times K}}$ as an example. We wish to write it as

$$\boldsymbol{X} \approx \sum_{r=1}^{R} \boldsymbol{a}_r \circ \boldsymbol{b}_r \circ \boldsymbol{c}_r,$$

where R is a positive integer and $\boldsymbol{a}_r \in \mathbb{R}^I$, $\boldsymbol{b}_r \in \mathbb{R}^J$, and $\boldsymbol{c}_r \in \mathbb{R}^K$ for $r = 1, \cdots, R$. Each element of $\boldsymbol{X}$ can be written as

$$x_{ijk} \approx \sum_{r=1}^{R} a_{ir}\, b_{jr} c_{kr}\ for\ i = 1, \cdots, I, j = 1, \cdots, J, k = 1, \cdots, K.$$

By combining the vectors from the rank-one components, we obtain the *factor matrices*, i.e., $\mathbf{A} = [\mathbf{a}_1 \quad \mathbf{a}_2 \quad \cdots \quad \mathbf{a}_\mathrm{R}]$ and likewise for **B** and **C**. Using this definition, we can rewrite the CP decomposition in matricized form:

$$\mathbf{X}_{(1)} \approx \mathbf{A}(\mathbf{C} \odot \mathbf{B})^\mathrm{T},$$

$$\mathbf{X}_{(2)} \approx \mathbf{B}(\mathbf{C} \odot \mathbf{A})^\mathrm{T},$$

$$\mathbf{X}_{(3)} \approx \mathbf{C}(\mathbf{B} \odot \mathbf{A})^\mathrm{T}.$$

Recall that $\odot$ is the Khatri-Rao product we introduced in section 5.1.4. We usually normalize the columns of A, B, and C to length one. Thus, there is a vector $\lambda \in \mathbb{R}^R$ which absorbs the weights. More concisely, the CP model could be expressed as

$$\boldsymbol{X} \approx [\![\mathbf{A}, \mathbf{B}, \mathbf{C}]\!] \equiv \sum_{r=1}^{R} \boldsymbol{a}_r \circ \boldsymbol{b}_r \circ \boldsymbol{c}_r.$$

We only focus on the three-way case in our discussion because it is widely applicable and sufficient for many needs.

Due to its simplicity, the alternating least squares (ALS) method, which was proposed by Carroll and Chang [77] and Harshman [78] in their original papers, has

```
1  Procedure CP-ALS($\boldsymbol{X}$, R)
2      Initialize $\boldsymbol{A}^{(n)} \in \mathbb{R}^{I_n \times R}$ for $n = 1, \ldots, N$
3      Repeat
4          For  $n = 1, \ldots, N$ do
5              $\boldsymbol{V} \leftarrow \boldsymbol{A}^{(1)T}\boldsymbol{A}^{(1)} * \cdots * \boldsymbol{A}^{(n-1)T}\boldsymbol{A}^{(n-1)} * \boldsymbol{A}^{(n+1)T}\boldsymbol{A}^{(n+1)} * \cdots * \boldsymbol{A}^{(N)T}\boldsymbol{A}^{(N)}$
6              $\boldsymbol{A}^{(n)} \leftarrow \boldsymbol{X}^{(n)}\left(\boldsymbol{A}^{(N)} \odot \cdots \odot \boldsymbol{A}^{(n+1)} \odot \boldsymbol{A}^{(n-1)} \odot \cdots \odot \boldsymbol{A}^{(1)}\right)\boldsymbol{V}^{\dagger}$
7              Normalize columns of $\boldsymbol{A}^{(n)}$ (storing norms as $\lambda$)
8          End for
9      Until fit ceases to improve or maximum iterations
   exhausted
10 Return $\lambda, \boldsymbol{A}^{(1)}, \boldsymbol{A}^{(2)}, \ldots, \boldsymbol{A}^{(N)}$
11 End procedure
```

Figure 21 ALS algorithm to compute a CP decomposition with R components for an Nth-order tensor X of size $I_1 \times I_2 \times \cdots \times I_N$.

*$\mathbf{V}^{\dagger}$ denotes the Moore-Penrose pseudoinverse of $\mathbf{V}$

become the major algorithm today to compute the CP decomposition. This algorithm assumes the number of components, which is R in our notation, is specified. We briefly describe the algorithm for the three-way case in the following. The full algorithm for an N-way tensor is presented in Figure 21.

Let $\boldsymbol{X} \in \mathbb{R}^{I \times J \times K}$be a third-order tensor. We are going to decompose $\boldsymbol{X}$ into R rank-one components that best approximate $\boldsymbol{X}$, i.e., to find

$$\min_{\widehat{\boldsymbol{X}}} \left\| \boldsymbol{X} - \widehat{\boldsymbol{X}} \right\| \; with \; \widehat{\boldsymbol{X}} = \sum_{r=1}^{R} \lambda_r \boldsymbol{a}_r \circ \boldsymbol{b}_r \circ \boldsymbol{c}_r = [\![\lambda; \mathbf{A}, \mathbf{B}, \mathbf{C}]\!].$$

The ALS approach fixes two of the matrices to solve for the third one during each iteration. Namely, it fixed **B** and **C** to solve for **A**, then fixes **A** and **C** to solve for **B**, then fixes **A** and **B** to solve for **C**. It continues to repeat the entire procedure until some convergence criterion is satisfied.

5.2.2. The Tucker Decomposition

The Tucker decomposition is another important tensor decomposition model that was proposed by Tucker in 1963 [74] and improved by Levin [94] and Tucker [75] [76] in their subsequent papers.

The Tucker decomposition is well known as a form of higher-order principal component analysis (PCA). It decomposes a tensor into a core tensor multiplied by a matrix along each mode. Take the three-way case as an example, for $\boldsymbol{X} \in \mathbb{R}^{I \times J \times K}$, we have

$$\boldsymbol{X} \approx \boldsymbol{G} \times_1 \mathbf{A} \times_2 \mathbf{B} \times_3 \mathbf{C} = \sum_{\mathbf{p}=1}^{\mathbf{P}} \sum_{\mathbf{q}=1}^{\mathbf{Q}} \sum_{\mathbf{r}=1}^{\mathbf{R}} \mathbf{g}_{\mathbf{pqr}} \mathbf{a}_{\mathbf{p}} \circ \mathbf{b}_{\mathbf{q}} \circ \mathbf{c}_{\mathbf{r}} = [\![\boldsymbol{G}; \mathbf{A}, \mathbf{B}, \mathbf{C}]\!].$$

The tensor $\boldsymbol{G} \in \mathbb{R}^{\boldsymbol{P}\times\boldsymbol{Q}\times\boldsymbol{R}}$ is called the core tensor and its entries show the level of interaction between the different components. The matrices $\mathbf{A} \in \mathbb{R}^{I\times P}, \mathbf{B} \in \mathbb{R}^{J\times Q}, and\ \mathbf{C} \in \mathbb{R}^{K\times R}$ are the factor matrices and can be thought of as the principal components in each mode. Alternatively, we can represent the Tucker decomposition element-wise as following:

$$x_{ijk} \approx \sum_{p=1}^{P}\sum_{q=1}^{Q}\sum_{r=1}^{R} \boldsymbol{g}_{pqr}\boldsymbol{a}_{ip}\boldsymbol{b}_{jq}\boldsymbol{c}_{kr}\ for\ i = 1, \dots, I, j = 1, \dots, J, k = 1, \dots, K.$$

P, *Q*, and *R* are the number of components in the factor matrices **A**, **B**, and **C**, respectively. If *P*, *Q*, *R* are smaller than *I*, *J*, *K*, the core tensor ***G*** can be viewed as a compressed version of ***X***. This feature makes the Tucker decomposition appeal for data compression applications. Figure 22 illustrates the Tucker decomposition of a three-way tensor.

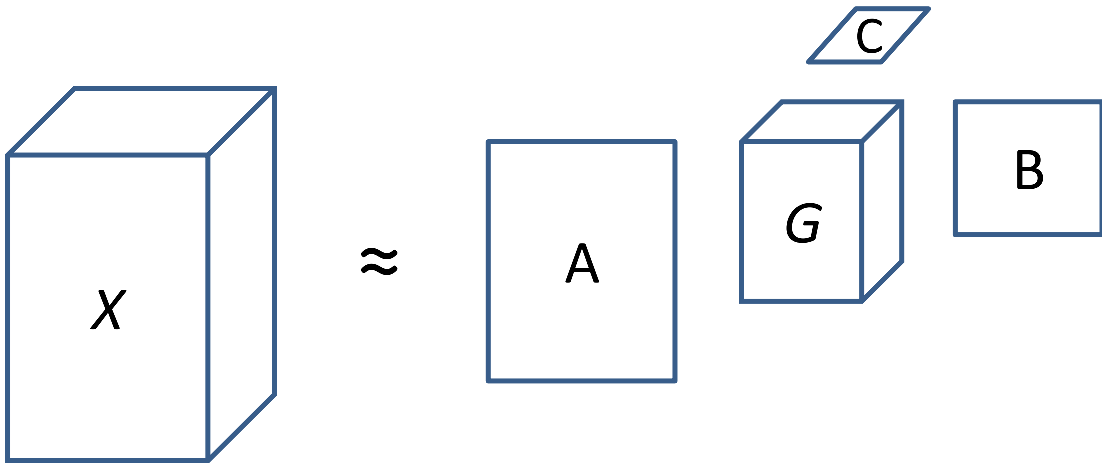


Figure 22 Tucker decomposition of a three-way tensor

Before we continue with the discussion of the algorithm to compute a Tucker decomposition, we introduce the concept of *n-Rank*. Let $\boldsymbol{X}$ be an Nth-order tensor of size $I_1 \times I_2 \times \cdots \times I_N$. The n-rank of $\boldsymbol{X}$, denoted $rank_n(\boldsymbol{X})$, is the column rank of $\mathbf{X}_{(n)}$. We say $\boldsymbol{X}$ is a rank-$(R_1, R_2, \cdots, R_N)$ tensor if $R_n = rank_n(\boldsymbol{X})$. For a given tensor $\boldsymbol{X}$, it is easy to compute an exact Tucker decomposition of rank $(R_1, R_2, \cdots, R_N)$, where $R_n = rank_n(\boldsymbol{X})$. If for one or more n, we restrict$R_n < rank_n(\boldsymbol{X})$, then the result will be inexact and more difficult to compute.

In his original paper [76], Tucker described three methods to compute a Tucker decomposition. We only discuss the first one here, which is better known as the *higher-order SVD* (HOSVD) from the work of De Lathauwer, De Moor, and Vandewalle [95]. They have shown that the HOSVD is a convincing generalization of the matrix SVD. Figure 23 shows the pseudo code of HOSVD for a rank-$(R_1, R_2, \cdots, R_N)$ Tucker decomposition.

```
Procedure HOSVD ($\boldsymbol{X}, R_1, R_2, \dots, R_N$)
    For $n = 1, \dots, N$ do
        $\boldsymbol{A}^{(n)} \leftarrow R_n$ leading left singular vectors of $\boldsymbol{X}_{(n)}$
    End for
    $\boldsymbol{G} \leftarrow \boldsymbol{X} \times_1 \boldsymbol{A}^{(1)T} \times_2 \boldsymbol{A}^{(2)T} \cdots \times_N \boldsymbol{A}^{(N)T}$
    Return $\boldsymbol{G}, \boldsymbol{A}^{(1)}, \boldsymbol{A}^{(2)}, \cdots, \boldsymbol{A}^{(N)}$
End procedure
```

Figure 23 the HOSVD algorithm to compute a rank-$(R_1, R_2, \cdots, R_N)$ Tucker decomposition

The basic idea of this algorithm is to find those components that best capture the variation in mode n, independent of the other modes.

5.3.Related Work

Most of the works on tensor decomposition still focus on the quality of solutions, for the simple ALS method doesn't guarantee to converge to a global minimum. The final solution can be heavily dependent on the initial guess. In one of the works, Faber et al. [96] compare ALS with six different methods, none of which is better than ALS in terms of quality of solutions.

There is increasing interests in using CP for large-scale, sparse tensors recently. Kolda et al. [90] developed a "greedy" CP for sparse tensors that computes one triad at a time via an ALS method. Kolda and Bader [9] [97] adapted the standard ALS algorithm to sparse tensors in their subsequent work.

Parallel and distributed implementation is definitely an important way to handle large-scale tensors. Sears, Bader, and Kolda [98] implemented several algorithms which compute PARAFAC and Tucker representations of tensors using C++/MPI. They evaluated their implementation on several data sets that are of order three with dimensions in the order of thousands and have millions of nonzero entries. Speedups of 10 to 20 are achieved with regarding to serial implementation. Zhang et al. [99] parallelized the Nonnegative Tensor Factorization (NTF) method, with the purposes of distributing large datasets into each cluster node and thus reducing the demand on a single node, blocking and localizing the computation at the maximal degree, and finally minimizing the memory use for storing matrices or tensors by exploiting their structural relationships. The data set used in their experiments is relatively small (order three tensor with dimensions in the order of hundreds). A sublinear speedup was achieved for 2 to 8 processors with an approximate peak speedup of 6.8. On a similar effort, Antikainen et al. [100] tried to accelerate the NTF method using GPGPU. Third order tensors with two of the dimensions varying between 100×100 and 1000×1000 and the third dimension being either 31 or 62 were used in their experiments. Great speedups ($100 \times$ compared with the CPU implementation) were achieved due to the massive parallelism provided by GPGPU. However, limited by the memory capacity of graphic cards, their method can only process third order tensors with up to 800 elements in each dimension.

5.4.Tensor Decomposition in MapReduce

We describe in this section how we implement the CP-ALS and the HOSVD algorithms in MapReduce to compute the CP decomposition and Tucker decomposition for a given tensor $\boldsymbol{X}$. We decompose the algorithms into several basic operations and implement each of them in MapReduce.

5.4.1. Matrix Multiplication

Matrix multiplication is used in both of the two algorithms. In CP-ALS, it is used in line 5 and 6 of Figure 21 to compute the intermediate matrix **V**. While in HOSVD, it is used indirectly in the tensor n-mode matrix product in line 5 of Figure 23.

General matrix multiplication implementation in MapReduce has been discussed extensively in section 4.4.1 and [101]. In particular, we want to point out that the multiplication in line 5 of Figure 21 is a special form of multiplication, which is of a matrix multiplying with its transpose, which can be implemented in MapReduce very efficiently using only one job.

5.4.2. The Hadamard Product

Recall from section 5.1.4 that the *Hadamard product* is the elementwise matrix product. A straight forward MapReduce job can be derived from its definition. The Mappers match the positions of the matrix elements and the Reducers do the multiplication. The input/output key/value pairs for the Hadoop job is summarized in Table 7, assuming the input of this job are two matrices **A** and **B**, whose elements are stored as $\langle i, j, a_{ij} \rangle$ or $\langle i, j, b_{ij} \rangle$ on disk.

Table 7 Input/output for the Hadamard product Hadoop job

Function	Input	Output
Map	$\langle\langle i, j\rangle, a_{ij}\rangle$ or $\langle\langle i, j\rangle, b_{ij}\rangle$	$\langle\langle i, j\rangle, a_{ij}\rangle$ or $\langle\langle i, j\rangle, b_{ij}\rangle$
Reduce	$\langle\langle i, j\rangle, [a_{ij}, b_{ij}]\rangle$	$\langle\langle i, j\rangle, a_{ij} * b_{ij}\rangle$

As can be seen from this table, the map function is basically an identity function. Since there are only two matrices in the input, there could be only two values in the input of the reduce function. The reduce function just multiply the two values and emit the result.

5.4.3. The Khatri-Rao Product

As mentioned in section 5.1.4, the *Khatri-Rao product* [93] is the "matching columnwise" Kronecker product. Given matrices $\mathbf{A} \in \mathbb{R}^{\mathrm{I}\times\mathrm{K}}$ and $\mathbf{B} \in \mathbb{R}^{\mathrm{J}\times\mathrm{K}}$, the result is a matrix of size $(IJ) \times K$ defined by

$$\mathbf{A} \odot \mathbf{B} = [\boldsymbol{a}_1 \otimes \boldsymbol{b}_1 \quad \boldsymbol{a}_2 \otimes \boldsymbol{b}_2 \quad \cdots \quad \boldsymbol{a}_K \otimes \boldsymbol{b}_K].$$

It is used in line 6 of the CP-ALS algorithm.

Similar to the Hadamard product, a straightforward MapReduce job can be derived from its definition. The map function matches the corresponding columns from the two input matrices. And the reduce function does the multiplication on the two matching columns. With the same assumption in the previous section, we summarize the input/output of the MapReduce job to compute the Khatri-Rao product in Table 8.

Table 8 Input/output for the Khatri-Rao product Hadoop job

Function	Input	Output
Map	$\langle\langle i,k\rangle, a_{ik}\rangle$ or $\langle\langle j,k\rangle, b_{jk}\rangle$	$\langle k, a_{ik}\rangle$ or $\langle k, b_{jk}\rangle$
Reduce	$\langle k, [a_{1k}, a_{2k}, \cdots, a_{Ik}, b_{1k}, b_{2k}, \cdots, b_{Jk}]\rangle$	$\boldsymbol{a}_k \otimes \boldsymbol{b}_k$

As can be seen from Table 8, the map function emits the column id as key for each element it reads. Thus, all the elements that belong to the same column will be shuffled to the same Reducer. But how do we distinguish elements from matrix **A** from elements from matrix **B**? If we don't put some special information into the matrix itself, there is no way to tell where the current element comes from. Fortunately, Hadoop provides a class called MultipleInputs, which let you use different map functions for different inputs. Thus, we apply different map functions to the two input matrices. We negate the row value of the elements in the map function for one of the input matrix. Therefore, we can group the input of the reduce function into two groups by checking their row values (one is negative; the other is positive). After that, we do the Khatri-Rao product in the reduce function and emit the results.

5.4.4. The Moore-Penrose Pseudoinverse

The Moore-Penrose pseudoinverse [57], denoted by $\mathbf{A}^{\dagger}$, of a matrix $\mathbf{A}$, is a generalization of the inverse matrix. It has various applications in applied linear algebra. In the context of tensor decomposition, it is used in line 6 of the CP-ALS algorithm.

There are many ways to compute the pseudoinverse. One of them is by using the singular value decomposition (SVD) [57], which itself is an important matrix factorization method. The reason we choose SVD is that it is computationally simple and accurate and there is an "embarrassingly parallel" way to compute the SVD of a matrix $\mathbf{A}$, namely, the Lanczos algorithm [57]. Once we have the singular value decomposition of $\mathbf{A}$, $\mathbf{A} = \mathbf{U}\sum\mathbf{V}^{\mathrm{T}}$, then $\mathbf{A}^{\dagger} = \mathbf{V}\sum^{\dagger}\mathbf{U}^{\mathrm{T}}$. Since $\sum$ is a diagonal matrix, we can get its pseudoinverse by taking the reciprocal of each non-zero element on the diagonal, leaving the zeros as they are, and transposing the resulting matrix.

With all the above, the problem is reduced to how to implement the Lanczos algorithm in MapReduce. As an adaption of the power method to compute the eigenvalues and eigenvectors of a square matrix, the Lanczos algorithm is an iterative algorithm and can be implemented in an embarrassingly parallel way. The basic idea is that starting from a random vector **v** with norm one, we repeatedly multiply the matrix **A** to **v** and computes the diagonal and off-diagonal elements of a matrix **T**. After *m* iterations (*m* is usually much smaller than the size of **A**), we obtain a tridigonal and symmetric matrix **T**, from which we can get the eigenvalues and eigenvectors of **A**. The major computation in these iterations is the matrix vector multiplication, which could be easily parallelized using the techniques presented in the previous chapter. Thus, the number of MapReduce jobs is the same as the number of iterations.

5.4.5. The CP decomposition

Once all the sub operations are ready in MapReduce, implementing the CP decomposition is just a task of assembly. All of the four operations mentioned in the previous sections are utilized in the CP decomposition.

As depicted in Figure 21, the CP-ALS algorithm starts with initializing N factor matrices $\mathbf{A}^{(n)} \in \mathbb{R}^{I_n \times R}$. This can be done using one MapReduce job by asking the map function to generate a random matrix for each input record. The input is a text file that contains one number each line as the id of the matrix to be generated. We write to HDFS directly in the map function, for there is no need to shuffling the generated elements and have a reduce phase. The input/output of this job is summarized in Table 9.

Table 9 Input/output for initializing factor matrices in the CP-ALS algorithm.

Function	Input	Output
Map	$\langle\ , n\rangle$	$\mathbf{A}^{(n)}$
Reduce	None	

The major computation happens in the inner for loop of the algorithm. For each of the *N* factor matrices, we update it to the next iteration and normalize it. The updating happens in two steps. We first compute an intermediate matrix **V** by multiplying $\mathbf{A}^{(n)\mathrm{T}}$with $\mathbf{A}^{(n)}$ for every factor matrix except the one we are computing and doing Hadamard product on all the results of the multiplication. Each of these matrix multiplication and Hadamard product takes one MapReduce job respectively. Thus, there are $2n - 3$ jobs in this step. Next, we take the pseudoinverse of **V**, do the Khatri-Rao product of *N-1* factor matrices, and multiply mode-*n* matricization of the input tensor with the results of the former two computations. Each of the Khatri-Rao product takes one MapReduce job. Adding the 4 jobs needed for the two matrix multiplications, there are $3n - 1$ plus the number of jobs needed to compute the pseudo inverse in total in this step. Finally, we can use one job to normalize the columns of the current factor matrix. Adding all the numbers together, we spend linear number of MapReduce jobs as the order of the tensor in total in the inner for loop of the algorithm.

The outer loop stops when fit ceases to improve or maximum iterations are exhausted. For simplicity, we implement a fixed number of iterations that can be tuned as a parameter of the job. Checking whether the fit has stopped to improve only requires one job which compares the newly computed factor matrices with the old ones. The map function will be responsible to match the corresponding factor matrices and the reduce function does the comparison.

5.4.6. The Tucker decomposition

A very important operation in the HOSVD algorithm is the singular value decomposition we have used in the CP-ALS algorithm. Instead of using it to compute the pseudo inverse of a matrix in the CP-ALS algorithm, it is used to initialize *N* factor matrices in the HOSVD algorithm.

The HOSVD algorithm starts with the initialization of *N* factor matrices with the R_n leading left singular vectors of $\mathbf{X}_{(n)}$. Each of the factor matrices requires computing the SVD of the corresponding mode-*n* matricization of the input tensor. Thus, $N *$ R_nMapReduce jobs are needed to finish the initialization part.

The second step is to compute the core tensor by doing mode-*n* tensor matrix product on the *N* factor matrices we get in the first step with the input tensor. Each of such products could be computed using the general matrix multiplication scheme we proposed in chapter four, which requires two MapReduce jobs. However, since the factor matrices are usually dense and small, we could put it into memory and distribute the much larger matricization of tensor to different machines to do the computation. Thus, the product could be computed in one MapReduce job, and only $(N - 1)$ jobs are needed for this step.

Table 10 shows the input/output for a tensor matrix product job. What is special about this job is that we not only merge elements of tensor into fibers in the reduce phase, but also perform the matrix vector multiplication after we form the fibers.

After we compute the core tensor ***G***, the algorithm returns it with all the factor matrices.

Table 10 MapReduce Input/Output for tensor matrix product.

Function	Input	Output
Map	$\langle [i,j,k], v \rangle$	$\langle [i,j], \left[[i,j,k], v\right] \rangle$ or $\langle [j,k], \left[[i,j,k], v\right] \rangle$ or $\langle [i,k], \left[[i,j,k], v\right] \rangle$
Reduce	$\langle [p,q], \left[\left[[i,j,k], v\right]\right] \rangle$	$\langle [i,j,k], v' \rangle$

5.5.Experiments

5.5.1. Experiments Setup

We evaluate our implementations on random generated third-order tensors, which is widely applicable and sufficient for many needs. Experiments were conducted on the HOSVD algorithm to see how the performance changes w.r.t the size of the data sets. The first and second dimension of the data set are fixed at 2^{15} and 2^{12}. The third dimension varies from 2^5 to 2^9. The sparsity of the data sets is fixed at 0.002. The size of the data sets thus varies from 200 MB to 3.7 GB.

The data sets are generated by our random tensor generation program in our experiments. Similar to our random matrix generator, the program generate random sparse third-order tensors $\boldsymbol{X} \in \mathbb{R}^{I \times J \times K}$ with δ as its sparsity, *I*, *J*, and *K* as the size of its three dimensions. All the experiments were conducted in the environment we described in section 2.1.2. We report the total time for the HOSVD algorithm.

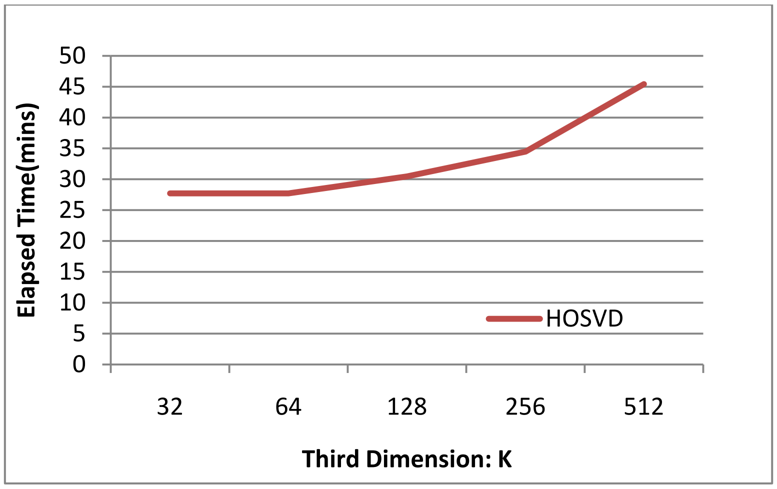


Figure 24 Elapsed time w.r.t K

5.5.2. Scalability and Performance

Figure 24 shows the performance of the HOSVD algorithm. As can be seen from the figure, the HOSVD implementation can easily handle tensors with third dimension up to 512, which has hundreds of millions nonzero entries. Figure 25 shows the computation time breakdown for the experiments. First observation is that the initialization part accounts for a large portion of the total computation time for all the data sets. However, this portion of time doesn't vary much despite the change of size of the data sets. Even for the largest data set, it takes more or less the same amount of time. The second observation is that the time takes to compute the core tensor of the Tucker decomposition grows linearly with the size of the data sets. For the largest data set, it has already become commeasurable to the time of initialization. We would expect it account for the major computation time for even larger data sets.

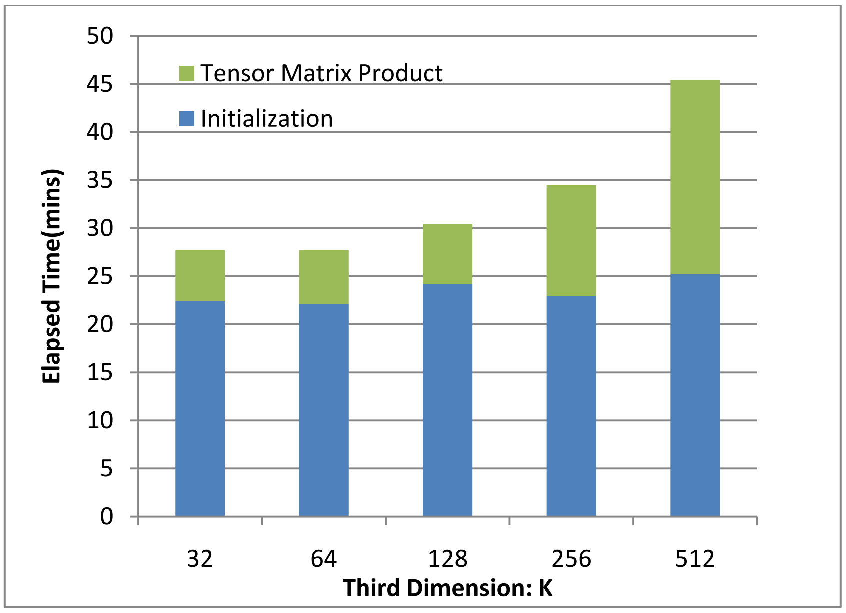


Figure 25 Computation time breakdown for the HOSVD algorithm.

5.6.Summary

Tensors are great abstractions to represent data sets of high dimensionality. While being useful for reducing the noise and recovering the hidden information in data sets, tensor decomposition techniques are notoriously unscalable due to its high computation demand. In this chapter, we presented several techniques to implement the Hadamard product, the KhatriRao product, and the Moore-Penrose Pseudoinverse in MapReduce. From these implementations, we assembled our implementation of the CP-ALS algorithm that computes the CP decomposition and the HOSVD algorithm that computes the

Tucker decomposition. Great scalability has been shown in the experiments we conducted on the HOSVD algorithm.

6. CONCLUSIONS

6.1.Conclusions

As an effort towards solving the general problem of how to efficiently process and understand large-scale data sets, this dissertation develops techniques and algorithms in large-scale raster data processing and algorithm parallelizing using MapReduce.

Two meanings for the term “large-scale” are considered, namely the size of data sets and the dimensionality of data sets. In terms of the size of data sets, this dissertation addresses problems in loading and computing statistics over terabytes of raster data in an online spatial data management system - TerraFly. Although the problems themselves are specific to TerraFly, the developed techniques are generally applicable to the area of large-scale raster data processing. Solutions of the problems are formed, implemented and experimentally evaluated in a real-world cluster.

In terms of the dimensionality of data sets, this dissertation addresses algorithms in two general categories of the domain of dimensionality reduction, namely matrix and tensor factorization. Three forms of nonnegative matrix factorization and two major tensor factorization algorithms were successfully scaled up in MapReduce. Extensive experiments were conducted to verify the scalability of the Hadoop implementation of these algorithms.

In summary, this dissertation demonstrates and advances the capability of MapReduce in processing large-scale raster data and scaling up certain kinds of algorithms. The developed techniques are not only effective in solving these specific problems, but also generally applicable to the field of data intensive and computation intensive computing using MapReduce.

6.2.Future Work

There are many directions for future work. As another major type of data in spatial database, vector data, is completely different from raster data. They are relatively small compared with raster data, but needs more complex operations over each byte of the data. It looks like MapReduce is not a good option for direct application, but may be best suited to generate the index structures that algorithms operates on [27]. More investigations need to be done on how MapReduce could be applied to vector data.

The nonnegative matrix factorization algorithms we considered in chapter four are all based on multiplicative updating rules, upon which we reduce the algorithm to different implementations of matrix multiplication and successfully scaled them up in MapReduce. We would like to investigate other NMF algorithms such as alternating non-negative least squares [69] and projective gradient descent [70] to see if there is an efficient MapReduce implementation.

Similar to matrix factorization, there are many other tensor decompositions, including INDSCAL, PARAFAC2, CANDELINC, DEDICOM, and PARATUCK2 as well as nonnegative variants of all the above. One interesting direction is to investigate schemes for scaling up these algorithms.

Lastly, we would also like to explore various applications of NMF and tensor decomposition to large-scale data sets.

LIST OF REFERENCES

[1] AT&T Daytona: http://www2.research.att.com/~daytona/

[2] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. Commun. ACM 51, 1 (January 2008), 107-113.

[3] World Data Center for Climate: http://www.mad.zmaw.de/fileadmin/extern/PI_Linux_DB_final.pdf

[4] Jim Gray, David T. Liu, Maria Nieto-Santisteban, Alex Szalay, David J. DeWitt, and Gerd Heber. 2005. Scientific data management in the coming decade. SIGMOD Rec. 34, 4 (December 2005), 34-41.

[5] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6 (San Francisco, CA, December 06 - 08, 2004). Operating Systems Design and Implementation. USENIX Association, Berkeley, CA, 10-10.

[6] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: distributed data-parallel programs from sequential building blocks. SIGOPS Oper. Syst. Rev. 41, 3 (March 2007), 59-72.

[7] TerraFly Project: http://terrafly.fiu.edu/tf-whitepaper.pdf

[8] Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. JOURNAL OF THE ACM, 46(5):604–632, 1999.

[9] Tamara Kolda and Brett Bader. The TOPHITS model for higher-order web link analysis. In Proceedings of the SIAM Data Mining Conference Workshop on Link Analysis, Counterterrorism and Security, 2006.

[10] Tamara G. Kolda and Brett W. Bader. 2009. Tensor Decompositions and Applications. SIAM Rev. 51, 3 (August 2009), 455-500.

[11] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. 2007. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture (HPCA '07). IEEE Computer Society, Washington, DC, USA, 13-24.

[12] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. 2008. Mars: a MapReduce framework on graphics processors. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques* (PACT '08). ACM, New York, NY, USA, 260-269.

[13] M. Mustafa Rafique, Benjamin Rose, Ali R. Butt, and Dimitrios S. Nikolopoulos. 2009. CellMR: A framework for supporting mapreduce on asymmetric cell-based clusters. In Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing (IPDPS '09). IEEE Computer Society, Washington, DC, USA, 1-12.

[14] Yi Shan, Bo Wang, Jing Yan, Yu Wang, Ningyi Xu, and Huazhong Yang. 2010. FPMR: MapReduce framework on FPGA. In Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays (FPGA '10). ACM, New York, NY, USA, 93-102.

[15] M. Mustafa Rafique, Benjamin Rose, Ali R. Butt, and Dimitrios S. Nikolopoulos. 2009. Supporting MapReduce on large-scale asymmetric multi-core clusters. SIGOPS Oper. Syst. Rev. 43, 2 (April 2009), 25-34.

[16] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. 2008. Pig latin: a not-so-foreign language for data processing. In Proceedings of the 2008 ACM SIGMOD international conference on Management of data (SIGMOD '08). ACM, New York, NY, USA, 1099-1110.

[17] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel Abadi, Avi Silberschatz, and Alexander Rasin. 2009. HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads. Proc. VLDB Endow. 2, 1 (August 2009), 922-933.

[18] Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and D. Stott Parker. 2007. Map-reduce-merge: simplified relational data processing on large clusters. In Proceedings of the 2007 ACM SIGMOD international conference on Management of data (SIGMOD '07). ACM, New York, NY, USA, 1029-1040.

[19] C. Chu et al, “MapReduce for Machine Learning on Multicore,” the Twentieth Annual Conference on Neural Information Processing Systems, 2006.

[20] S. Papadimitriou and J. Sun, “DisCo: Distributed Co-clustering with Map-Reduce: A Case Study towards Petabyte-Scale End-to-End Mining,” Proceedings of the 2008 Eighth IEEE international Conference on Data Mining (December 15 - 19, 2008), ICDM, IEEE Computer Society, Washington, DC, 2008, 512-521, doi:10.1109/ICDM.2008.142.

[21] U. Kang, C. Tsourakakis, and C. Faloutsos, “PEGASUS: A Peta-Scale Graph Mining System - Implementation and Observations,” the IEEE International Conference on Data Mining 2009.

[22] B. Panda, J. Herbach, S. Basu, and R. Baryado, “PLANET: Massively Parallel Learning of Tree Ensembles with MapReduce,” the 35th International Conference on Very Large Data Bases, 2009.

[23] C. Liu, F. Guo, and C. Faloutsos, “BBM: bayesian browsing model from petabyte-scale data,” Proceedings of the 15th ACM SIGKDD international Conference on Knowledge Discovery and Data Mining (Paris, France, June 28 - July 01, 2009), KDD '09, ACM, New York, NY, 2009, 537-546, doi:10.1145/1557019.1557081.

[24] Richard M. C. McCreadie, Craig Macdonald, and Iadh Ounis. 2009. On single-pass indexing with MapReduce. In Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval (SIGIR '09). ACM, New York, NY, USA, 742-743.

[25] Richard Cyganiak, Holger Stenzhorn, Renaud Delbru, Stefan Decker, and Giovanni Tummarello. 2008. Semantic sitemaps: efficient and flexible access to datasets on the semantic web. In Proceedings of the 5th European semantic web conference on The semantic web: research and applications (ESWC'08), Sean Bechhofer, Manfred Hauswirth, Hoffmann, and Manolis Koubarakis (Eds.). Springer-Verlag, Berlin, Heidelberg, 690-704.

[26] Thomas Sandholm and Kevin Lai. 2009. MapReduce optimization using regulated dynamic prioritization. In Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems (SIGMETRICS '09). ACM, New York, NY, USA, 299-310.

[27] Ariel Cary, Zhengguo Sun, Vagelis Hristidis, and Naphtali Rishe. 2009. Experiences on Processing Spatial Data with MapReduce. In Proceedings of the 21st International Conference on Scientific and Statistical Database Management (SSDBM 2009), Marianne Winslett (Ed.). Springer-Verlag, Berlin, Heidelberg, 302-319.

[28] Shubin Zhang, Jizhong Han, Zhiyong Liu, Kai Wang, and Shengzhong Feng. 2009. Spatial Queries Evaluation with MapReduce. In Proceedings of the 2009 Eighth International Conference on Grid and Cooperative Computing (GCC '09). IEEE Computer Society, Washington, DC, USA, 287-292.

[29] Ralf Lämmel. 2007. Google's MapReduce programming model -- Revisited. Sci. Comput. Program. 68, 3 (October 2007), 208-237.

[30] J Tordable. MapReduce for Integer Factorization. Arxiv preprint arXiv:1001.0421, 2010.

[31] Apache Hadoop Project: http://hadoop.apache.org/mapreduce/

[32] Google & IBM Academic Cluster Computing Initiative, http://www.google.com/intl/en/press/pressrel/20071008_ibm_univ.html

[33] NSF Cluster Exploratory Program: http://www.nsf.gov/pubs/2008/nsf08560/nsf08560.htm

[34] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. SIGOPS Oper. Syst. Rev. 37, 5 (October 2003), 29-43.

[35] http://csrc.nist.gov/groups/SNS/cloud-computing/

[36] Gonzalez, R., Woods, R., Digital Image Processing, Addison-Wesley, Reading MA, 1992.

[37] Br äunl, T., with Feyrer, S., Rapf, W., Reinhardt, M.,Parallel Image Processing, Springer Verlag, Heidelberg, 2000.

[38] http://www.adobe.com/products/photoshop/compare/

[39] http://www.gimp.org/

[40] http://www.esri.com/software/arcgis/

[41] http://www.erdas.com/Homepage.aspx

[42] Flynn, M. (1972). "Some Computer Organizations and Their Effectiveness". IEEE Trans. Comput. C-21: 948.

[43] Rosenfeld, A, Parallel image processing using cellular arrays. Computer. Vol. 16, no. 1, pp. 14-20. 1983

[44] DYER, C R, A VLSI pyramid machine for hierarchical parallel image processing , Conference on Pattern Recognition and Image Processing, Dallas, TX; United States; 3-5 Aug. 1981. pp. 381-386. 1981

[45] Anthony P. Reeves, "Parallel computer architectures for image processing," Computer Vision, Graphics, and Image Processing, Volume 25, Issue 1, January 1984, Pages 68-88.

[46] Farrugia, J.-P.; Horain, P.; Guehenneux, E.; Alusse, Y.; GPUCV: A Framework for Image Processing Acceleration with Graphics Processors,

[47] John Browna and Danny Crookes, A high level language for parallel image processing, Image and Vision Computing, Volume 12, Issue 2, March 1994, Pages 67-79

[48] F.J. Seinstra, D. Koelma, "The Lazy Programmers Approach to Building a Parallel Image Processing Library," IPDPS, vol. 3, pp.30115b, 15th International Parallel and Distributed Processing Symposium (IPDPS'01) Workshops, 2001.

[49] Thomas Br äunl, Stefan Feyrer, Wolfgang Rapf, Michael Reinhardt, Parallel image processing, Springer, 2001.

[50] http://geospatial.osu.edu/resources/DOQ.pdf

[51] C. Ding, X. He, and H.D. Simon, “On the equivalence of nonnegative matrix factorization and spectral clustering,” Proc. SIAM Data Mining Conf, 2005.

[52] T. Li and C. Ding, “The Relationships among Various Nonnegative Matrix Factorization Methods for Clustering,” Proceedings of the 2006 IEEE International Conference on Data Mining (ICDM 2006), Pages 362-371, 2006.

[53] E. Gaussier and C. Goutte, “Relation between PLSA and NMF and implications,” Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval, August 15-19, 2005.

[54] C. Ding, T. Li, and W. Peng, “On the Equivalence Between Nonnegative Matrix Factorization and Probabilistic Latent Semantic Indexing,” Computational Statistics and Data Analysis, 52(8): 3913-3927, 2008.

[55] C. Liu, H. Yang, J. Fan, L. He, and Y. Wang, “Distributed nonnegative matrix factorization for web-scale dyadic data analysis on mapreduce,” Proceedings of the 19th international Conference on World Wide Web (Raleigh, North Carolina, USA, April 26 - 30, 2010), WWW '10. ACM, New York, NY, 2010, 681-690.

[56] Lynn Elliot Cannon, “A cellular computer to implement the kalman filter algorithm,” Technical report, Ph.D. Thesis, Montana State University, 14 July 1969.

[57] Gene H. Golub and Charles F. Van Loan, Matrix Computations. 3rd ed, The Johns Hopkins University Press, 1996.

[58] J. J. MODI, Parallel Algorithms and Matrix Computation. Oxford: Clarendon Press, 1988.

[59] S. A. Robila and L. G. Maciak, “A parallel unmixing algorithm for hyperspectral images,” Intelligent Robots and Computer Vision XXIV, 2006.

[60] E. Batternberg and D. Wessel, “Accelarating Non-Negative Matrix Factorization for Audio Source Separation on Multi-core and Many-core Architectures,” 10th International Society for Music Information Retrieval Conference (ISMIR 2009).

[61] M. Kearns, “Efficient noise-tolerant learning from statistical queries,” J. ACM 45, 6, 983-1006, 1998.

[62] The Apache Mahout Project: http://mahout.apache.org/

[63] Y. Chen, D. Pavlov, and J. F. Canny, “Large-scale behavioral targeting,” Proceedings of the 15th ACM SIGKDD international Conference on Knowledge

Discovery and Data Mining (Paris, France, June 28 - July 01, 2009), KDD '09, ACM, New York, NY, 2009, 209-218.

[64] D. D. Lee and H. S. Seung, “Algorithms for Non-Negative Matrix Factorization,” NIPS, 2000.

[65] C. Ding, T. Li, and M. I. Jordan, “Convex and Semi-Nonnegative Matrix Factorizations,” IEEE Transactions on Pattern Analysis and Machine Intelligence, 32(1): 45-55, 2010.

[66] C. Ding, T. Li, W. Peng, and H. Park, “Orthogonal nonnegative matrix tri-factorization for clustering,” Proc SIGKDD Int’l Conf. on Knowledge Discovery and Data Mining, 2006.

[67] H. Ning, W. Xu, Y. Chi, Y. Gong, and T. Huang, “Incremental spectral clustering with application to monitoring of evolving blog communities,” SIAM Int. Conf. on Data Mining, 2007.

[68] Y. Chi, X. Song, D. Zhou, K. Hino, and B. L. Tseng, “On evolutionary spectral clustering,” ACM Trans. Knowl. Discov. Data 3, 4 (Nov. 2009), 1-30.

[69] M. B. Berry, M. Browne, A. N. Langville, V. P. Pauca, and R. J. Plemmons, “Algorithms and applications for approximate nonnegative matrix factorization,” Computational Statistics & Data Analysis Volume 52, Issue 1, 15 September 2007, Pages 155-173.

[70] Chih-Jen Lin, “On the Convergence of Multiplicative Update Algorithms for Nonnegative Matrix Factorization,” IEEE Transactions on Neural Networks 18 (6): 1589-1596, 2007.

[71] A. Shashua, and T. Hazan, “Non-negative tensor factorization with applications to statistics and computer vision,” Proceedings of the 22nd international Conference on Machine Learning (ICML '05), Pages 792-799, 2005.

[72] F. L. Hitchcock, The expression of a tensor or a polyadic as a sum of products, J. Math. Phys., 6 (1927), pp. 164-189.

[73] F. L. Hitchcock, Multiple invariants and generalized rank of a p-way matrix or tensor, J. Math. Phys., 7 (1927), pp. 39-79.

[74] L. R. Tucker, Implications of factor analysis of three-way matrices for measurement of change, in Problems in Measuring Change, C. W. Harris, ed., University of Wisconsin Press, 1963, pp. 122-137.

[75] L. R. Tucker, The extension of factor analysis to three-dimensional matrices, in Contributions to Mathematical Psychology, H. Gulliksen and N. Frederiksen, eds., Holt, Rinehardt,& Winston, New York, 1964, pp. 110-127.

[76] L. R. Tucker, Some mathematical notes on three-mode factor analysis, Psychometrika, 31(1966), pp. 279-311.

[77] J. D. Carroll and J. J. Chang, Analysis of individual differences in multidimensional scaling via an N-way generalization of “Eckart-Young” decomposition, Psychometrika, 35(1970), pp. 283–319.

[78] R. A. Harshman, Foundations of the PARAFAC procedure: Models and conditions for an “explanatory” multi-modal factor analysis, UCLA Working Papers in Phonetics, 16 (1970), pp. 1–84.

[79] C. J. Appellof and E. R. Davidson, Strategies for analyzing data from video fluorometric monitoring of liquid chromatographic effluents, Anal. Chem., 53 (1981), pp. 2053–2056.

[80] R. Bro, PARAFAC. Tutorial and applications, Chemometrics and Intelligent Laboratory Systems, 38 (1997), pp. 149–171.

[81] R. Bro, Multi-way Analysis in the Food Industry: Models, Algorithms, and Applications, Ph.D.thesis, University of Amsterdam, 1998.

[82] C. M. Andersen and R. Bro, Practical aspects of PARAFAC modeling of fluorescence excitation-emission data, J. Chemometrics, 17 (2003), pp. 200–215.

[83] C. A. Andersson and R. Henrion, A general algorithm for obtaining simple structure of core arrays in N-way PCA with application to fluorimetric data, Comput. Statist. Data Anal., 31 (1999), pp. 255–278.

[84] D. Bini, The Role of Tensor Rank in the Complexity Analysis of Bilinear Forms, presentation at ICIAM07, Z ̈urich, Switzerland, 2007.

[85] J. B. Kruskal, Three-way arrays: Rank and uniqueness of trilinear decompositions, with application to arithmetic complexity and statistics, Linear Algebra Appl., 18 (1977), pp. 95–138.

[86] L. De Lathauwer and B. De Moor, From matrix to tensor: Multilinear algebra and signal processing, in Mathematics in Signal Processing IV, J. McWhirter and I. Proudler, eds., Clarendon Press, Oxford, 1998, pp. 1–15.

[87] M. A. O. Vasilescu and D. Terzopoulos, Multilinear analysis of image ensembles: TensorFaces, in ECCV 2002:Pr oceedings of the 7th European Conference on Computer Vision, Lecture Notes in Comput. Sci. 2350, Springer, 2002, pp. 447–460.

[88] N. Liu, B. Zhang, J. Yan, Z. Chen, W. Liu, F. Bai, and L. Chien, Text representation: From vector to tensor, in ICDM 2005:Pr oceedings of the 5th IEEE

International Conferenceon Data Mining, IEEE Computer Society Press, 2005, pp. 725–728.

[89] Steffen Rendle, Leandro Balby Marinho, Alexandros Nanopoulos, and Lars Schmidt-Thieme. 2009. Learning optimal ranking with tensor factorization for tag recommendation. In Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD '09). ACM, New York, NY, USA, 727-736.

[90] T. G. Kolda, B. W. Bader, and J. P. Kenny, Higher-order web link analysis using multilinear algebra, in ICDM 2005:Pr oceedings of the 5th IEEE International Conference on Data Mining, IEEE Computer Society Press, 2005, pp. 242–249.

[91] H. A. L. Kiers, Towards a standardized notation and terminology in multiway analysis, J. Chemometrics, 14 (2000), pp. 105–122.

[92] B. W. Bader and T. G. Kolda, Algorithm 862: MATLAB tensor classes for fast algorithm prototyping, ACM Trans. Math. Software, 32 (2006), pp. 635–653.

[93] A. Smilde, R. Bro, and P. Geladi, Multi-Way Analysis: Applications in the Chemical Sciences, Wiley, West Sussex, England, 2004.

[94] J. Levin, Three-Mode Factor Analysis, Ph.D. thesis, University of Illinois, Urbana, 1963.

[95] L. De Lathauwer, B. De Moor, and J. Vandewalle, A multilinear singular value decomposition, SIAM J. Matrix Anal. Appl., 21 (2000), pp. 1253–1278.

[96] N. K. M. Faber, R. Bro, and P. K. Hopke, Recent developments in CANDECOMP/PARAFAC algorithms: A critical review, Chemometrics and Intelligent Laboratory Systems,65 (2003), pp. 119–137.

[97] B. W. Bader and T. G. Kolda, Efficient MATLAB computations with sparse and factored tensors, SIAM J. Sci. Comput., 30 (2007), pp. 205–231.

[98] Mark P. Sears, Brett W. Bader, and Tammy Kolda, Parallel Implementation of Tensor Decompositions for Large Data Analysis, SIAM AN09 Minisymposium on High Performance Computing on Massive Real-World Graphs, 2009.

[99] Qiang Zhang, Michael W. Berry, Brian T. Lamb, and Tabitha Samuel. 2009. A Parallel Nonnegative Tensor Factorization Algorithm for Mining Global Climate Data. In Proceedings of the 9th International Conference on Computational Science (ICCS 2009), Springer-Verlag, Berlin, Heidelberg, 405-415.

[100] Jukka Antikainen, Jiří Havel, Radovan Jošth, Adam Herout, Pavel Zemčík, Markku Hauta-Kasari, "Non-Negative Tensor Factorization Accelerated Using GPGPU," IEEE Transactions on Parallel and Distributed Systems, 08 Nov. 2010.

[101] Zhengguo Sun, Tao Li, and Naphtali Rishe. Large-Scale Matrix Factorization using MapReduce. In Proceedings of the Optimization-based Methods for Emerging Data Mining Workshop associated with ICDM 2010 (OEDM'10), 2010.

VITA

ZHENGGUO SUN

2004	B.E., Software Engineering Zhejiang University Hangzhou, China
2007	M.E., Software Engineering Beihang University Beijing, China
2007-2011	Doctoral Candidate in Computer Science Florida International University Miami, FL, USA

PUBLICATIONS

- Ariel Cary, Zhengguo Sun, Vagelis Hristidis, and Naphtali Rishe. 2009. Experiences on Processing Spatial Data with MapReduce. In Proceedings of the 21st International Conference on Scientific and Statistical Database Management (SSDBM 2009), Marianne Winslett (Ed.). Springer-Verlag, Berlin, Heidelberg, 302-319.

- Zhengguo Sun, Tao Li, Naphtali Rishe, "Large-Scale Matrix Factorization Using MapReduce," pp.1242-1248, 2010 IEEE International Conference on Data Mining Workshops, 2010.