Large-Scale Matrix Completion Using Orthogonal Rank-One Matrix Pursuit, Divide-Factor-Combine, and Apache Spark

by

Brian Krouse

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved April 2014 by the
Graduate Supervisory Committee:

Jieping Ye, Chair
Huan Liu
Hasan Davulcu

ARIZONA STATE UNIVERSITY

May 2014

ABSTRACT

As the size and scope of valuable datasets has exploded across many industries and fields of research in recent years, an increasingly diverse audience has sought out effective tools for their large-scale data analytics needs. Over this period, machine learning researchers have also been very prolific in designing improved algorithms which are capable of finding the hidden structure within these datasets. As consumers of popular Big Data frameworks have sought to apply and benefit from these improved learning algorithms, the problems encountered with the frameworks have motivated a new generation of Big Data tools to address the shortcomings of the previous generation. One important example of this is the improved performance in the newer tools with the large class of machine learning algorithms which are highly iterative in nature.

In this thesis project, I set about to implement a low-rank matrix completion algorithm (as an example of a highly iterative algorithm) within a popular Big Data framework, and to evaluate its performance processing the Netflix Prize dataset. I begin by describing several approaches which I attempted, but which did not perform adequately. These include an implementation of the Singular Value Thresholding (SVT) algorithm within the Apache Mahout framework, which runs on top of the Apache Hadoop MapReduce engine.

I then describe an approach which uses the Divide-Factor-Combine (DFC) algorithmic framework to parallelize the state-of-the-art low-rank completion algorithm Orthogoal Rank-One Matrix Pursuit (OR1MP) within the Apache Spark engine. I describe the results of a series of tests running this implementation with the Netflix dataset on clusters of various sizes, with various degrees of parallelism. For these experiments, I utilized the Amazon Elastic Compute Cloud (EC2) web service.

In the final analysis, I conclude that the Spark DFC + OR1MP implementation does indeed produce competitive results, in both accuracy and performance. In particular, the Spark implementation performs nearly as well as the MATLAB implementation of OR1MP without any parallelism, and improves performance to a significant degree as the parallelism increases. In addition, the experience demonstrates how Spark's flexibile programming model makes it straightforward to implement this parallel and iterative machine learning algorithm.

DEDICATION

I dedicate this thesis to my dear wife Penny, whose willingness to patiently listen to me talk on at such length about the merits of this or that algorithm (and even ask questions) is certain proof that she loves me.

ACKNOWLEDGEMENTS

I would like to thank all those who helped me achieve the requirements for my Master of Science degree, including the many excellent professors who taught my courses, and in particular my advisors for this research project. Through my various stops, starts, and turn-abouts, Dr. Jieping Ye exhibited limitless patience and always successfully helped me navigate a path forward. I am also appreciative of the willingness of Dr. Huan Liu and Dr. Hasan Davulcu to support my efforts as advisors to this project. Last but not least, I would like to express my gratitude to everyone at Arizona State University and the Ira A. Fulton School of Engineering for their hard work in creating such a vibrant and supportive learning environment.

TABLE OF CONTENTS

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

LIST OF LISTINGS

Listing Page

Chapter 1

INTRODUCTION

1.1 Background and Motivation

In recent years, machine learning algorithms have found increasing application in a wide range of business and research contexts. Business applications range from eCommerce-enabled companies seeking to engage more effectively with their customers, to product companies seeking to build products which utilize learning mechanisms or intelligent processing of information. Research applications include any field seeking to discover order or structure in large volumes of data, ranging from medical research, to studies of internet activity, to the physical, bioinformatical, environmental, or sociological sciences.

As increasingly diverse audiences become interested in leveraging machine learning algorithms, it becomes important to consider the frameworks and tools to make these algorithms more easily applied. Many technology companies are currently engaged in building solutions to meet the demands of Big Data. These tools aim to make it easier to process very large datasets which are especially challenging to deal with using more traditional data analysis tools.

Modern machine learning researchers will spend most of their time using tools which are not well-suited for Big Data applications in non-academic environments. An indispensible tool for conducting machine learning research is one of several popular numerical computational packages (e.g. MATLAB [1], R [2], or Octave [3]) running on a desktop PC. Unfortunately, this toolset does not easily scale to handle large datasets, nor does it try to address any of the operational concerns around managing

and processing Big Data which are critical for a business context. Researchers who are dealing with larger datasets will usually incorporate high performance computing (HPC) systems and tools. However, these systems are usually quite expensive and are tailored to academic researchers, making them unlikely to appear in more mainstream technology business contexts.

Several open source frameworks have emerged over the last decade from large internet companies to meet the demand for Big Data tools. By far the most well-known of these frameworks is Apache Hadoop [4], which was originated in 2005 by Doug Cutting, a software engineer working at Yahoo! at the time. This framework was inspired by a seminal paper from Google titled "MapReduce: Simplified Data Processing on Large Clusters" [5], which describes an approach to processing Big Data used by Google. Part of Hadoop's appeal is that it is an open source project, and can be deployed on commodity hardware – even in conjunction with the readily available and low cost cloud computing services, such as Amazon Web Services (AWS) [6] – making it economically and technically accessible to a wide audience. A large ecosystem of related tools has grown up around Hadoop's popularity, and many companies today are forming and evolving their Big Data business strategies around this ecosystem. One such related tool which I will discuss in this paper is Apache Mahout [7], whose stated goal is to build a library of machine learning algorithms on top of Hadoop.

As companies have developed experience with the MapReduce paradigm and Hadoop, a new generation of Big Data tools has emerged to address some of the weak points many users of these tools have encountered. One particular weak point which I will discuss in this paper is the challenges Mahout/Hadoop has with handling the highly iterative aspects of many machine learning algorithms, as well as

some of the more computationally intensive linear algebra operations which are quite common, such as singular value decomposition (SVD).

Apache Spark [8], coming out of UC Berkeley, is one of the newer frameworks to emerge. Spark boasts a 10- to 100-fold performance increase over Hadoop for certain algorithms (primarily by caching active datasets in memory for repeated access throughout the duration of the program), along with a more flexible computational framework that can support MapReduce style programs as well as other approaches, side-by-side.

An example of a class of machine learning algorithms which is challenging to implement in a Big Data context is that of low rank matrix completion. Matrix completion involves reconstructing missing elements of a matrix, based on a set of observed and possibly noisy matrix entries. Matrix completion finds application in a wide variety of contexts, including recommendation engines (e.g. Amazon.com or Netflix recommendation lists) and image reconstuction. Matrix completion algorithms are challenging for Big Data because they are highly iterative, and can involve repeated factorization of large matrices.

1.2 Overview of Research Project

In this paper, I will discuss my various attempts to implement a matrix completion algorithm in a popular Big Data framework. My initial attempt was to implement the Singular Value Thresholding (SVT) algorithm [9] within Apache Mahout. This approach ultimately proved to be much too slow for practical use, primarily as a result of overhead imposed by the underlying Hadoop framework.

My second attempt was to implement a very recent matrix completion algorithm, Orthogonal Rank-One Matrix Pursuit (OR1MP) [10], developed by Zheng Wang and other machine learning researchers at Arizona State University (ASU). This algorithm

extends the orthogonal matching pursuit method used in signal recovery problems from the vector case to the matrix case. OR1MP improves on other matrix completion algorithms by decreasing both time and storage complexity, in large part by relying on a method for producing a matrix factorization incrementally, rather than computing a full SVD outright.

My first attempt at implementing OR1MP was to use the Spark framework, and to distribute my linear algebra operations over the cluster by using the Spark framework's Resilient Distributed Dataset (RDD) operations. While this proved to be much more performant than the Mahout implementation of SVT, it was still much slower than desirable.

My second attempt at implementing OR1MP on Spark leveraged the additional Divide-Factor-Combine (DFC) [11] algorithmic framework, developed by machine learning researchers at UC Berkely. This framework implements large matrix factorization by first partitioning the matrix (e.g. by columns), factoring each of the submatrices, and then combining the results into the final factorization. Thanks to the flexibility of the Spark framework, I was able to implement this non-MapReduce algorithm on Spark, while still leveraging its benefits of cluster management, distributed task management, data caching, operational manageability, etc. In addition, as Spark is designed to work well within a cloud computing environment, I was able to conduct testing at different scales by running Spark on top of AWS.

As the Spark DFC + OR1MP imlementation ultimately proved to be the most successful, I describe the particulars of this implementation in some detail below. Empirical results are provided from test runs over the Netflix Prize dataset [12] on various cluster sizes using AWS. The performance of the Spark OR1MP algorithm proved to perform nearly as well as the native MATLAB implementation on small datasets without any use of parallelization. By leveraging the DFC approach to parallize

the computations, the Spark implementation matched or surpassed MATLAB performance with no parallelization and achieved significant performance improvements through increased parallelization - all while achieving similar prediction accuracy.

Chapter 2

RELATED WORK

2.1 Overview of Related Work

The challenges of Big Data analysis are currently generating tremendous attention, with many new systems and solutions emerging every year, originating from both university research and industry.

For the purposes of comparing this research project to the related work in the field, it may be helpful to consider several categories of related work:

1. Large-scale matrix factorization using Stochastic Gradient Descent
2. Graph-oriented approaches
3. Other work to improve performance with in-memory caching
4. Related work within the Hadoop ecosystem and Berkeley Data Analytics Stack

The following sections describe related work in these categories, and contrast the purposes and approach of this research project to clarify the unique contributions of my work.

2.2 Large-scale Matrix Factorization Using Stochastic Gradient Descent

One approach to large-scale matrix factorization that has received considerable attention over recent years is that of distributed variations of Stochastic Gradient Descent (SGD).

For example, HogWild! [13] was proposed in 2011 as an approach to parallelizing SGD in a shared memory system by removing memory locking to increase performance, while establishing that if the problem is sparse that their approach achieves a good rate of convergence.

DSGD [14] is a method proposed in 2011 for large-scale matrix factorization using a "stratified" SGD variant. The researchers study the performance of their approach on both an R-based cluster, and a Hadoop cluster.

FPSGD [15] is another method proposed in late 2013 for shared memory systems which claims to improve upon HogWild and DSGD.

Sparkler [16] is another effort that attempts to improve upon the performance of distributed SGD by leveraging Spark's benefits. Sparkler's unique contribution involves augmenting Spark to include a new framework construct called "Cluster Maps" in order to achieve it's performance goals. The motivation of the Cluster Map abstraction is to more effeciently store large matrices in the aggregate memory of a cluster, and support the operations supported on them during SGD.

My research project is distinct from these other efforts in the following ways:

1. These other efforts all focus on a parallel version of the SGD algorithm. Here, we focus on implementing a different approach to low-rank matrix completion, namely DFC with OR1MP.

2. Not all of this research aims to try to apply the algorithms in the context of a main stream Big Data framework. Those that do primarily consider only MapReduce and Hadoop, and encounter some of the limitations. This project aims to leverage the more recent Spark project to improve upon the performance of Hadoop for iterative algorithms.

3. While Sparkler is one example that does try to leverage Spark, it does so by augmenting Spark in order to optimize the operations during SGD. This project attempts to implement a different algorithm (DFC + OR1MP) without modifying the core of Spark.

2.3 Graph-Oriented Approaches

Another very popular subtopic within large-scale machine learning is that of graph processing. Many large-scale learning problems in industry today can be expressed most naturally using a graph representation.

Google has written about Pregel [17], which is their proprietary graph processing engine that scales to billions of vertices and edges.

Apache Giraph [18] is an open source graph engine inspired by Pregel which is in use at Facebook. Facebook is purported to have scaled Giraph to more than a trillion edges.

GraphLab [19] is another well-known and mature open source graph processing engine, which originated out of Carnegie Mellon University in 2009.

While graph processing (sometimes called "graph-parallel" computation) is clearly an important and successful subfield of large-scale Big Data processing, most frameworks use a different processing model (i.e. Bulk Synchronous Parallel, or BSP [20]), requiring algorithms to be programmed within that model.

For the purposes of this research project, I will not investigate the area of "graph-parallel" handling of large-scale data, but instead will focus on scaling "data-parallel" models, which is a way to describe the parallelization of the familiar matrix-based operations used by most numerical processing frameworks, such as MATLAB and R.

It should be noted that the Berkeley Data Analytics Stack (BDAS) [21], which includes Spark, also includes GraphX as a higher level of abstraction on top of Spark.

GraphX aims to distinguish itself in the area of graph engines by uniting the "graph-parallel" and the "data-parallel" representations, as well as by integrating with and leveraging the fault-tolerance capabilities of the larger BDAS stack. GraphX is still in development.

2.4 Other Work to Improve Performance via In-Memory Caching

One of the key ways in which Spark is able to signficantly improve upon Hadoop MapReduce when it comes to iterative machine learning algorithms is by its approach to in-memory caching. The resilient distributed dataset (RDD) abstraction allows Spark programs to be written which repeatedly access the working data in memory across multiple iterations, without requiring the data to be re-loaded from disk.

Other systems launched recently in the Big Data area also aim to improve query performance by better leveraging in-memory caching strategies.

For example, Impala [22] is described as a massively parallel processing (MPP) SQL query engine which allows users to interactively query data stored in HDFS or HBase, at 10-100x improved performance over Apache Hive.

Presto [23] is another open source SQL query engine built by Facebook with a similar purpose.

While these systems could provide the foundation for improved performance of iterative machine learning algorithms accessing data within HDFS, I do not focus on these systems in this paper, as they do not currently have any explicit focus on or library support for machine learning algorithms.

2.5 Related Work Within the Hadoop Ecosystem and BDAS

Mahout [7] is one of the more well-known machine learning projects built on top of Hadoop. Mahout has implemented a variety of different algorithms, primarily around

recommendation mining, clustering, and classification. Mahout includes a single-threaded implementation of SGD, but not yet a parallelized version. As mentioned above, I did initially spend considerable effort attempting to use Mahout to implement a low-rank matrix completion algorithm, namely Singular Value Thresholding (SVT) [9]. Mahout has already implemented a couple of SVD implementations which I attempted to modify for use with the SVT algorithm. However, after experiencing challenges with the performance constraints of the underlying Hadoop MapReduce framework, I turned my attention towards Spark.

MLbase [24] is a project within the BDAS that is currently in development which aims to provide a simple-to-use interface for machine learning users, on top of Spark. MLbase consists of several layers of increasing programming abstraction, up to the "ML Optimizer" layer, which aims to automate the task of model selection for users. Currently, the lower-level "MLLib" library is a growing collection of machine learning algorithms written against the Spark runtime. Currently, MLLib does not have an implementation of a low-rank matrix completion algorithm other than SGD, and may be interested in including some of the results of this research project.

Chapter 3

MAPREDUCE, HADOOP AND MAHOUT

3.1 Overview of MapReduce and Apache Hadoop

Over the last decade, MapReduce has been one of the most prominent approaches to processing large-scale data in Internet companies. This processing paradigm was formally described in a paper published by Google Research in 2004 [5].

As described in the Google paper, two distinct phases comprise a MapReduce program: map, and reduce. Input data is sent to the user-provided map routine as a list of key/value pairs. The map phase transforms the key/value input into a list of intermediate key/value pairs. This intermediate data is sorted and aggregated, such that for every unique key there is a sorted list of unique values, and is then fed into the user-provided reduce routine, which in turn outputs another list of values. The conceptual type signatures of the map and reduce phase are as follows:

$$\begin{aligned} map\,(k1,\, v1) &\longrightarrow list\,(k2,\, v2) \\ reduce\,(k2,\, list\,(v2)) &\longrightarrow list\,(v3) \end{aligned}$$

MapReduce programs can consist of a single map and reduce pass over the data, or a chain of programs taking multiple passes over the data.

A MapReduce execution engine will do the job of feeding different chunks (aka "splits") of the input data to multiple instances of the user-provided map routine in parallel, usually distributed across multiple networked servers. The engine will then sort and aggregate the mappers' output, and feed this intermediate data into one or more instances of the user-provided reduce routine. In addition, the execution

engine will handle operational concerns, such as ensuring that slow or failed map or reduce instances are dealt with or restarted. Figure 3.1 shows the high-level flow of the system.

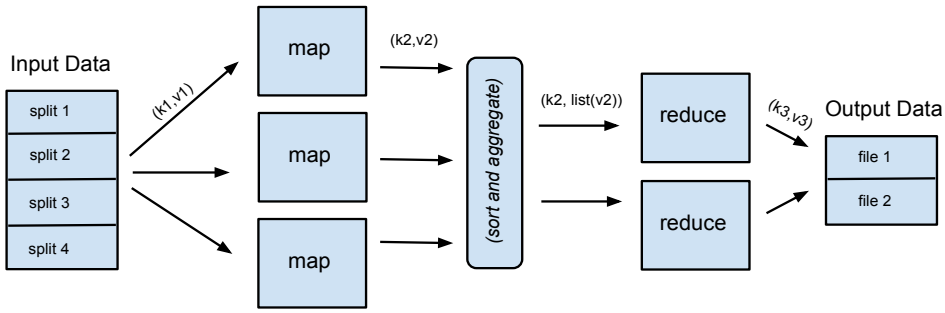


Figure 3.1: MapReduce Flow

While the semantics of MapReduce are quite simple, many data processing algorithms can be expressed as a series of MapReduce programs. Since there is no interaction between the different map instances, a MapReduce program can be easily scaled out to process very large datasets with a high degree of parallelism. In addition, the "shared nothing" approach to parallelism makes it possible to run MapReduce on a collection of networked commodity servers, which can be a very cost-effective alternative to traditional HPC clusters with high-speed interconnects and a shared clustered file system, or expensive "scaled-up" RDBMS servers.

Inspired by the Google MapReduce paper, the Hadoop project [4] was created by Doug Cutting in 2005 while at Yahoo!. Originally, Hadoop consisted of two main elements: Hadoop Distributed File System (HDFS), a distributed filesystem inspired by another paper by Google on their Google File System (GFS) [25], and a MapReduce engine. In recent years, Hadoop has grown into an ecosystem with many related projects, including Apache Hive [26] (a SQL-like interface for data warehouse capabilities), and Apache HBase [27] (a NoSQL database that runs on top of HDFS).

With MapReduce at it's core, Hadoop is best suited for a batch-oriented processing model. A typical workflow for a Hadoop job is to identify splits in input data stored in HDFS (or another data source), instantiate mappers on servers close to the data, feed the data into the mappers, write the output of the map tasks to local storage (potentially multiple times, if the map output is large and needs to be buffered to disk), sort the map output across the network to the server where the reduce tasks are running, and finally write the reduce task output back into HDFS. In the case that a data analysis routine consists of multiple MapReduce jobs chained together, this same pattern – including the repeated disk access – is repeated for every MapReduce job, since there is by default no mechanism to manage in-memory caching of data across multiple jobs. Note that this design is integral to the framework's approach to its "embarassingly parallel" method of scaling-out, and is also key to it's operational resiliency, in that this allows for restarting individual map or reduce tasks without having to restart the entire job. However, as is discussed more thoroughly below, these aspects of its design are also some of the reasons as to why Hadoop has struggled to perform iterative machine learning algorithms with adequate performance.

Partially in response to some of these concerns, Hadoop has recently attempted to re-position itself as a more generic framework for managing distributed computing. For example, in addition to the traditional MapReduce model, Hadoop now includes a more generic scheduling component (YARN) [28] which can coordinate both MapReduce and non-MapReduce applications on the same cluster (e.g. sharing the same data stored in HDFS or HBase). In particular, this allows both MapReduce and Spark applications to run side-by-side on the same cluster; Spark is discussed more in the next chapter.

3.2 Implementing Singular Value Thresholding with Apache Mahout

Apache Mahout [7] is a project built on top of Hadoop which aims to provide a library of machine learning algorithms suitable for large-scale applications. For example, Mahout includes algorithms for clustering, classification and collaborative filtering. While Mahout does include some algorithms which are not distrubuted, it's *raison d'etre* is to perform distributed machine learning algorithms using large-scale data stored in HDFS (or another HDFS-compatible data source) via a series of map-reduce jobs.

For the purpose of this research project, namely to implement a large-scale version of a matrix completion algorithm on top of a popular Big Data framework, Apache Mahout initially appeared to fit the bill perfectly. I selected Singular Value Thresholding (SVT) [9] as the algorithm to implement within the Mahout framework, as SVT is one of the most well-known methods for performing matrix completion. The SVT algorithm is shown in pseudo-code in Algorithm 3.1.

While there has been considerable research on the method of Stochastic Gradient Descent in a large-scale data context [13–16] , I have not seen much, if any discussion about implementing an algorithm like SVT at scale. This is likely because there is a common recognition that the Singular Value Decomposition (SVD) that occurs in step 6 is very slow and computationally expensive for large matrices. However, while implementing a distrubuted SVD from scratch would be a major endeavor in itself, Mahout fortunately already has two implementations for a distributed SVD as a part of its library of algorithms: a Distributed Lanczos version, and a stochastic approach called SSVD. In addition, only a truncated SVD is required by SVT, rather than a full SVD. Therefore, I determined to evaluate whether use of these distributed SVD implementations could support a workable version of large-scale SVT.

Algorithm 3.1 Singular Value Thresholding (SVT)

Input: sampled set Ω and sampled entries $P_\Omega(\boldsymbol{M})$, step size δ, tolerance ϵ, parameter τ, increment l, and maximum iteration count k_{max}
Output: $\boldsymbol{X}^{\text{opt}}$
Description: Recover a low-rank matrix $\boldsymbol{M}$ from a subset of sampled entries

Set $\boldsymbol{Y}^0 = k_0\,\delta\,P_\Omega(\boldsymbol{M})$
Set $r_0 = 0$
for $k = 1$ **to** k_{max} **do**
  Set $s_k = r_{k-1} + 1$
  repeat
    Compute $[\boldsymbol{U}^{k-1}, \boldsymbol{\Sigma}^{k-1}, \boldsymbol{V}^{k-1}]_{s_k}$
    Set $s_k = s_k + l$
  until $\sigma^{k-1}_{s_k-l} \leq \tau$
  Set $r_k = \max\{j : \sigma^{k-1}_j > \tau\}$
  Set $\boldsymbol{X}^k = \sum_{j=1}^{r_k}(\sigma^{k-1}_j - \tau)\,\boldsymbol{u}^{k-1}_j\,\boldsymbol{v}^{k-1}_j$
  if $\left\|P_\Omega(\boldsymbol{X}^k - \boldsymbol{M})\right\|_F / \left\|P_\Omega(\boldsymbol{M})\right\|_F \leq \epsilon$ **then**
    break
  end if
end for
Set $\boldsymbol{X}^{\text{opt}} = \boldsymbol{X}^k$

After implementing the full SVT algorithm within Mahout, I tested it on the large scale Netflix dataset [12]. I used the Cloudera distribution of Hadoop [29], which makes it straightforward to deploy Hadoop and Mahout on top of a cluster of AWS servers. Unforunately, the performance of this implementation was at least an order of magnitude too slow. While the SVT algorithm might expect to converge within tens to hundreds of iterations on the Netflix dataset, each iteration of my Mahout-based SVT would execute in about 45 minutes. At this rate, it would require days or longer to run the entire SVT algorithm to completion. Even after many attempts at trying to improve the performance by tweaking the code, trying both SVD implementations, tuning the Hadoop cluster, and throwing hardware at the problem (i.e. adding servers to the cluster, using larger servers, etc.). I was not able to significantly improve the performance.

Through performance tracing and debugging, it because clear that the performance of the Mahout-based SVT implementation was bound by the performance of the underlying MapReduce engine. With each iteration of SVT, multiple Hadoop jobs were required, each of which had to take a separate pass over the entire dataset. Even though an increased degree of parallelism could speed this up, each pass over the data required multiple rounds of reading and writing the data (e.g. when buffering the map output to disk, prior to sorting it for input to the reducers) to disk. These performance constraints were seemingly insurmountable within the current Hadoop MapReduce paradigm.

As a result of these challenges, I ultimately decided to back up and consider another approach. Fortunately, others trying to use Hadoop and MapReduce for iterative machine learning algorithms had encountered the same problem, and had begun work on alternative frameworks. SVT was perhaps not a good fit for MapReduce and Mahout, but might still work well within a different Big Data framework.

Chapter 4

SPARK AND OR1MP

4.1 Apache Spark

Apache Spark [8] is a newer open-source Big Data framework originally developed at UC Berkely. Over the last few years, Spark has developed from a fledgling research project in the AMPLab at UC Berkeley [30] to being used in production environments by large and well-known technology companies, including Yahoo! and Intel. In early 2014, Spark graduated to a top-level Apache project. Spark is a core component of the Berkeley Data Analytics Stack (BDAS) [21], which includes a variety of other related projects in various stages of development: Shark (a SQL-like API) [31], GraphX (Graph computation) [32], MLbase (a library of machine learning algorithms) [24], Tachyon (an in-memory file system) [33], and Mesos (a scheduler) [34] to name a few. A diagram of the BDAS ecosystem from their website is shown in Figure 4.1.

Spark is nicely compatible with the Hadoop ecosystem, in that it integrates closely with HDFS. While it can run stand-alone, it also works well side-by-side with Hadoop MapReduce programs via the newer Hadoop scheduler YARN (although BDAS also includes a competing cluster scheduler, Mesos). Like Hadoop, Spark is built to work well with clusters of commodity servers with inexpensive locally attached storage. This makes it possible for companies with existing Hadoop installations to try out Spark quite easily, which may significantly increase the rate of Spark adoption.

Spark presents a programmatic interface that is able to support the same MapReduce, "embarassingly parallel" approach that Hadoop supports, but also supports other non-MapReduce semantics, while preserving the same benefits of scalability

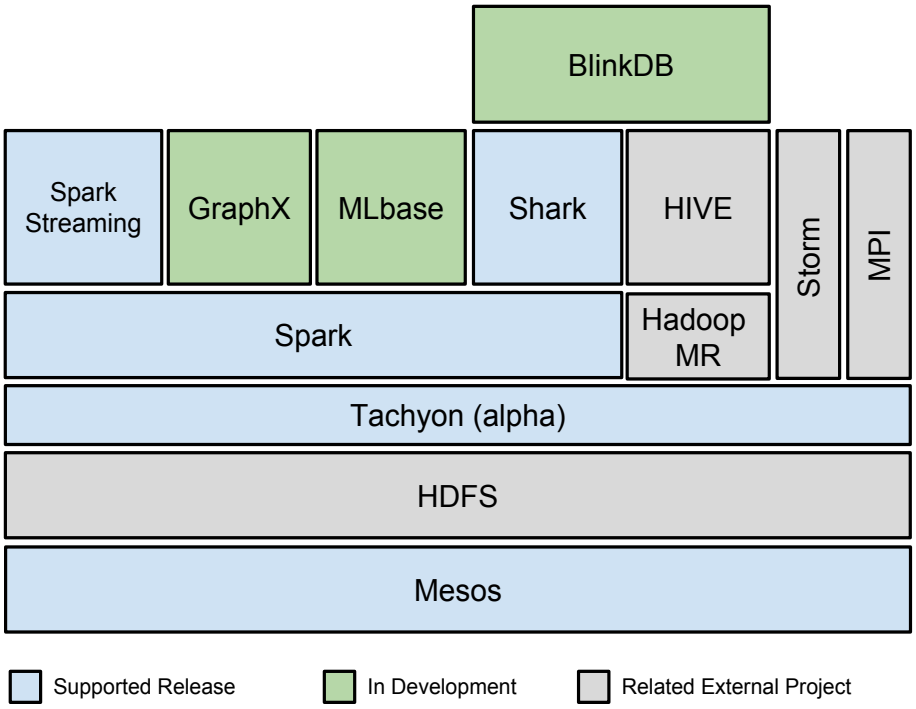


Figure 4.1: Berkeley Data Analytics Stack

and fault-tolerance. It's central abstraction is called a Resilient Distributed Dataset (RDD) [35]. To the programmer writing a Spark application, an RDD looks like a simple collection of objects, to which operations can be applied to create a resultant RDD. A subset of the RDD operations currently available in Spark is decribed in Table 4.1.

The Spark framework is responsible for partitioning the elements of the RDD across the compute cluster, where they typically reside in active memory, and for managing the execution of the operations applied to the RDDs. Spark manages fault-tolerance by tracking the lineage of each RDD created in the course of a Spark program. In the case that one of the machines fail, the lineage information is sufficient to recreate elsewhere the RDD partitions which were resident on that machine.

Operation	Description
$map(f : T \Rightarrow U)$	Transforms RDD of type T (i.e. RDD[T]) to RDD[U]
$filter(f : T \Rightarrow Bool)$	Filters RDD[T] to RDD[T] based on function f
$groupByKey()$	Acts on RDD[(K, V)] to produce RDD[$(K, Seq[V])$]
$join()$	RDD[(K, V)] joins RDD[(K, W)] $\Rightarrow$ RDD[$(K, (V, W))$]
$count()$	Returns count of records in RDD to the driver client
$collect()$	Collects values in RDD of type T to driver into Seq[T]

Table 4.1: Spark RDD Operations

A typical Spark program will begin by creating a first RDD based on some data source, e.g. data within HDFS. In terms of the execution environment, the main Spark "driver" process will at this point maintain a reference to the various RDD partitions now distributed in memory across the machines in the cluster. As the Spark program continues, it will execute a series of operations, each of which acts on the existing RDDs, and produces resultant RDDs. Some of these operations (e.g. map, reduce) will operate within the context of each machine, manipulating the RDD partition on that machine to produce a newer partition of the resultant RDD, still resident in memory of the same machine. Others of these operations (e.g. join, group-by) result in cross-network interactions, to combine portions of RDDs across machines. Still other operations (e.g. collect, broadcast) involve the "driver" process collecting data from the RDDs back to itself, or sending some data out to the RDDs for use in an operation.

Note that because the RDDs remain resident in the collective memory across the cluster (as much as possible), it is possible to write a Spark program to implement an iterative algorithm which repeatedly accesses and manipulates the working data

Algorithm 4.1 Orthogonal Rank-One Matrix Pursuit (OR1MP)

Input: $\boldsymbol{Y}_\Omega$ and stopping criterion.
Initialize: Set $\boldsymbol{X}_0 = 0, \boldsymbol{\theta}^0 = 0$ and $k = 1$.
  repeat
    Step 1: Find a pair of top left and right singular vectors $(\boldsymbol{u}_k, \boldsymbol{v}_k)$ of the observed residual matrix $\boldsymbol{R}_k = \boldsymbol{Y}_\Omega - \boldsymbol{X}_{k-1}$ and set $\boldsymbol{M}_k = \boldsymbol{u}_k(\boldsymbol{v}_k)^T$.
    Step 2: Compute the weight $\boldsymbol{\theta}^k$ using the closed form least squares solution $\boldsymbol{\theta}^k = (\bar{\boldsymbol{M}}_k^T \bar{\boldsymbol{M}}_k)^{-1} \bar{\boldsymbol{M}}_k^T \dot{\boldsymbol{y}}$.
    Step 3: Set $\boldsymbol{X}_k = \sum_{i=1}^k \theta_i^k (\boldsymbol{M}_i)_\Omega$ and $k \leftarrow k + 1$.
  until stopping criterion is satisfied
Output: Constructed matrix $\hat{\boldsymbol{Y}} = \sum_{i=1}^k \theta_i^k \boldsymbol{M}_i$.

set strictly within memory, and avoids the Hadoop penalty of needing to re-load data from disk with every pass over the data. Primarily as a result of this design, it has been reported that iterative algorithms can perform 10-100x faster when implemented on Spark vs Hadoop.

4.2 Orthogonal Rank-One Pursuit

As I was experiencing first-hand the performance issues Hadoop has with iterative machine learning algorithms, I came across the Spark framework as a more promising approach and decided to pursue using it rather than Mahout. At about the same time, I learned of another state-of-the-art matrix completion algorithm called Orthogonal Rank-One Pursuit being developed at Arizona State University [10]. The OR1MP algorithm is depicted in pseudo-code in Algorithm 4.1.

Unlike SVT, OR1MP does not need to iteratively compute a truncated SVD. Instead, it takes two main steps in each iteration: it computes the top singular vector pair (e.g. via power method), and then it refines a series of weights for combining these singular vector pairs by using a closed form least squares solution. The OR1MP paper

shows results of tests against the Netflix and MovieLens datasets which demonstrate that the method is one of, if not the, best performing method in the literature.

Spark and OR1MP appear to be an excellent combination: whereas Spark is establishing itself as the best-of-breed framework for implementing machine learning algorithms at large-scale, OR1MP is state-of-the-art amongst matrix completion algorithms in terms of efficiency and performance. Therefore, I determined that I would also switch gears away from implementing SVT on Hadoop and instead pursue implementing OR1MP on Spark.

4.3 Implementing OR1MP with Spark RDDs

My first approach to implementing OR1MP on Spark was to represent my input matrix as an RDD, loaded into distributed memory across the cluster, and to then leverage the RDD abstraction to implement the linear algebra operations in OR1MP.

One of the implementation approaches I tried was to store both a column and a row representation of my input matrix as RDDs. To see why, Listing 4.1 shows a snippet of the MATLAB implementation of OR1MP:

```
function [u, s, v] = topsvd(A, round)
stopeps = 1e-3;
[m, n] = size(A);
u = ones(m,1);
vo = 0;
for i=1:round
    v = u'*A/(norm(u))^2;
    u = A*v'/(norm(v))^2;
    if norm(v-vo) < stopeps break end
    vo = v;
end
u = u/norm(u);
v = v'/norm(v);
s = norm(u)*norm(v);
```

Listing 4.1: MATLAB topsvd() Function

To efficiently implement this as a distributed routine within Spark, where the input matrix is distributed across the cluster as an RDD, I needed to consider how to implement the iterative vector-to-matrix multiplication. On one hand, if I represented the input matrix `A` in row format – such that each RDD partition contained a set of rows – then the calculation in row 7 would require significant cross-network traffic to multiply `u` by each column of `A`. On the other hand, if I represented the input matrix in column format – such that each RDD partition contained a set of columns – then the same would be true for the calculation in row 8.

My solution was to store both representations. Listing 4.2's Scala code snippet corresponds to line 7 of the MATLAB version:

```
1  ...
2        //setup u and u_norm_sqrd for broadcast
3        val u_Br = sc.broadcast(u)
4
5        //multiply u' and matrix to get v, and collect to driver
6        val v_pairs : Array[(Int,Double)] = Acols.map(c =>
7        {
8          var sum = 0.0
9          for (e <- c._2.iterateNonZero()) {
10           sum += u_Br.value(e.index())*e.get()
11         }
12         (c._1, sum / pow(norm(u_Br.value),2))
13       }).collect()
14 ...
```

Listing 4.2: Scala/Spark topsvd() Function Snippet

Note here that Spark's "broadcast" mechanism is being used, in order to send the vector `u` within the `map()` call in line 7 to every partition of the RDD `Acols` in the cluster. The resultant vector is collected to the driver, and then the result can be used to multiply by the RDD `Arows` (not shown here).

A diagram of what is happening during this routine within Spark in terms of the network traffic and execution environment is shown below in Figure 4.2.

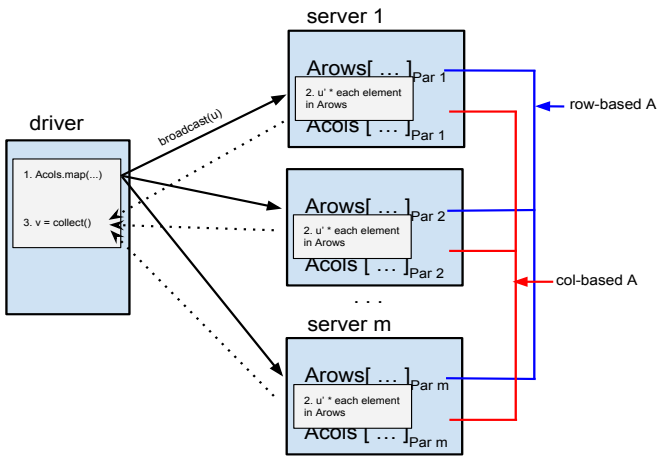


Figure 4.2: Spark OR1MP Using RDDs - Flow Over Network

After I confirmed that my implementation produced numerically accurate results, I ran it for multiple iterations using the Netflix dataset on a cluster of 3 AWS servers, each with 8GB of RAM. In this test, the program completed 2 iterations in approximately 11.4 minutes. With added parallelism, this improved such that across 10 AWS servers, the program completed four iterations in about the same time.

At this point, this implementation of OR1MP using Spark RDDs was 7-8 times better than the Mahout implementation of SVT. While this was encouraging, before I invested more time into trying to optimize and improve this approach, I first decided to investigate one other approach, which I describe in the next chapter.

Chapter 5

DFC + OR1MP ON SPARK

5.1 Divide-Factor-Combine

Mackey, Talwalkar and Jordan published a paper in NIPS in 2011 which describes an algorithmic approach to large-scale matrix factorization they name Divide-Factor-Combine (DFC) [11]. They apply the technique to both the problem of matrix completion, as well as robust matrix factorization. As the name suggests, the first step of the DFC algorithm is to "divide" the input matrix into smaller subproblems. Their paper examines two approaches to the divide step, based on either column projection (DFC-Proj) or the generalized Nyström method (DFC-Nys). With the input matrix subdivided into a set of smaller matrices using column sampling, the "factor" step uses any "base" matrix factorization technique to factor the smaller matrices in parallel. Finally, the factored matrices are "combined" using one of several approaches to produce the solution. Note that the DFC method is an algorithmic framework rather than a complete algorithm, since it must be used in conjunction with a base algorithm to perform the actual factorizations. This variant of the DFC algorithm I used, DFC-Proj, is presented in pseudocode in Algorithm 5.1.

In their research, the authors implemented the DFC framework using MATLAB for the DFC algorithm steps and used a variety of existing MATLAB programs which implemented the base algorithms. In addition, they implemented the parallel executions of the base algorithm using the MATLAB qsub command – which submits jobs to a MATLAB cluster. This method assumes the availability of a typical HPC environment – i.e. a pre-configured HPC cluster, complete with MATLAB licenses.

Algorithm 5.1 DFC-Proj

Input: $P_\Omega(\boldsymbol{M}), t$
  $\{P_\Omega(\boldsymbol{C}_i)\}_{1 \le i \le t} = \textsc{SampCol}(P_\Omega(\boldsymbol{M}), t)$
  do in parallel
    $\hat{\boldsymbol{C}}_1 = \textsc{Base-MF-Alg}(P_\Omega(\boldsymbol{C}_1))$
    $\vdots$
    $\hat{\boldsymbol{C}}_t = \textsc{Base-MF-Alg}(P_\Omega(\boldsymbol{C}_t))$
  end do
  $\hat{\boldsymbol{L}}^{proj} = \textsc{ColProjection}(\hat{\boldsymbol{C}}_1, \ldots, \hat{\boldsymbol{C}}_t)$

5.2 Implementing DFC on Spark

Contrasted with the implementation I describe in the previous chapter, the DFC algorithmic framewok presents an alternate approach worth investigating. In particular, it may be the case that since the base algorithm itself is not distributed, that the DFC approach might offer improved performance.

Since the DFC researchers used a MATLAB + HPC based implementation and execution environment, I first re-implemented the DFC algorithm within Spark. This proved to be relatively straightforward, due to Spark's flexible programming model. For example, the code snippet in Listing 5.1 displays how to execute the base algorithm in parallel across the Spark cluster. Note here that the `parallelize()` call in line 3, which is made available as part of the `SparkContext` object in Spark, is performing the task of creating an RDD with `numPars` number of splits. Then, the `mapPartitionsWithIndex()` call in line 4 is used to execute the containing code for each partition. This, in effect, launches the parallel jobs on which to run the base algorithm (OR1MP in this case). The `collectPartitions()` method in line 12 then serializes the results back to the driver, for the subsequent "combine" step (not shown).

```
...
     //Step 2: Calculating CHat for each matrix in parallel
    val cHatPars = sc.parallelize(parSplits, numPars).
                    mapPartitionsWithIndex[(Int, SVD)]
    ((partitionIdx, iter) =>
      {
        val splitName = iter.next()
        val C = MatrixUtils.readMatrix(...)
        val cHat : SVD = SparkOR1MP.run(C, splitName, rank)
        List((partitionIdx, cHat)).iterator
      }
    ).collectPartitions()

 ...
```

Listing 5.1: Scala/Spark DFC-Proj Function Snippet

As mentioned above, the DFC researchers examined a few variations of the "divide" and the "combine" steps within their paper. For the purposes of this experiment, I implemented the variations which proved to be simplest, without losing significant performance or accuracy. In particular, I chose to implement the column projection method of dividing the matrix. Also, I chose to use the simple column projection method which uses the first matrix partition's factorization as an orthonormal basis, and projects each of the subsequent submatrix factorizations against that basis. In the DFC paper, their results show that the more sophisticated "ensemble" or "random SVD" methods of combining the base factorizations do not lead to any significant performance or accuracy gains with the Netflix dataset.

5.3 Re-Implementing OR1MP with Breeze

With the DFC algorithmic framework in place, it was also necessary to re-implement the OR1MP algorithm without leveraging the distributed constructs of the Spark RDD, such that it would execute as efficiently as possible within the memory of each partition of the RDD. To this end, I implemented OR1MP using the Breeze [36] nu-

merical computing library for Scala. Rather than manipulate vectors and matrices using Scala types (e.g. Array[Double], or a sparse equivalent), I chose Breeze for it's underlying use of the netlib-java project [37], which in turn makes use of highly optimized implementations of the LAPACK/BLAS interfaces [38, 39].

LAPACK (Linear Algebra PACKage) is a library for numerical linear algebra operations (including solving systems of linear operations, and matrix factorizations). LAPACK uses BLAS (Basic Linear Algebra Subprograms), which provides a set of low-level linear algebra operations (including dense vector and matrix operations). Highly optimized implementations of BLAS include ATLAS [40], Intel MKL [41], cuBLAS (for CPU cards) [42], and OpenBLAS [43], as well as F2J [44], a Java bytecode native version of the Fortran library. Since there are benchmarks showing a wide range of performance between these different BLAS implementations (see links within the netlib-java "Performance" section [37]), I included some simple performance comparisons before selecting a library for use in my experiments.

When I compared the initial performance of DFC + OR1MP to OR1MP with RDDs, it became clear that the former was much faster, completing 5 iterations of OR1MP alone on a single desktop PC within about 3.5 minutes. When parallelizing the execution of DFC + OR1MP across multiple servers, the program was able to complete 10 iterations in as quick as 25 seconds. Thus, with the promising performance results from this approach, I pursued the full round of experiments using DFC + OR1MP, and left the topic of optimizing OR1MP with RDDs for future work.

Chapter 6

EXPERIMENTAL RESULTS

6.1 Overview of Experiments

In this chapter, I describe the experimental results obtained with the Scala/Spark implementations of OR1MP and DFC. The following sections each describe one of the tests performed:

1. MATLAB version vs Scala version of OR1MP in desktop environment
2. Optimal BLAS/LAPACK library configuration
3. Spark implementation of DFC + OR1MP with Netflix on EC2

The dataset used in these experiments is the Netflix dataset. The Netflix dataset has 10^8 ratings of 17,700 movies by 480,189 Netflix customers. Each movie rating is an integer from 1 to 5.

In these experiments, a specified "sampling percentage" (e.g. 5%, 10%, 50%) of the full dataset is separated out, such that the remaining data is the training data and the sampled data is testing data. The resulting matrix factorization of the training data is used to predict the testing data.

The running time in the experiments is calculated based on the actual running time of the algorithm itself, leaving apart any time spent on pre-processing the data to prepare it for input into the programs. The accuracy is measured in terms of root-mean-square error (RMSE). The RMSE is calculated based on the difference between the predicted and actual values of the testing data.

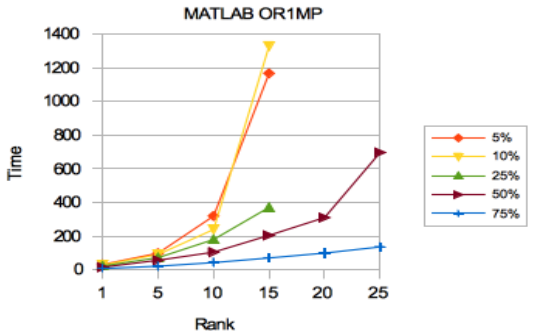
6.2 MATLAB vs Scala OR1MP in Desktop Environment

This section describes a comparison between the performance and accuracy of the Scala implementation of OR1MP and the MATLAB implementation (which code was provided by the primary author, Zheng) when running on a typical desktop system. For the tests, I used a MacBook Pro with a 2.7 GHz Intel Core i7 and 16GB of 1600 MHz DDR3 memory with a SSD hard drive. I ran these tests using the full Netflix dataset, for as many iterations as possible with the available memory. Even though Netflix is usually referred to in the literature as a large scale dataset, it is still possible to run a small number of iterations of our algorithms on the full dataset on a single PC if using a sparse format.

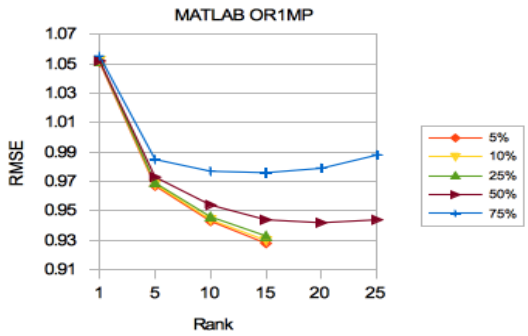
Figure 6.1 shows the results of these tests. In terms of accuracy, both the Scala and the MATLAB implementations are identical. This is a useful result, in that it helps to confirm the numerical correctness of the Scala implementation. Note that for the larger sizes of the input matrix (e.g. 5% sampling), the rank could only be calculated up to 15 before running out of memory, whereas for the smaller sizes of the input matrix (e.g. 75% sampling), we could calculate rank up to 25. This is expected, based on the storage requirements of the OR1MP algorithm.

In terms of performance, the Scala OR1MP implementation is slower than the MATLAB implementation by about 1.5x to 3x, depending on the size of the problem. Table 6.1 shows the values at rank=10. This is not surprising, since MATLAB is tuned specifically for numerical computing such as this. In fact, this performance difference is relatively small, given the overheads expected with the Scala type system and JVM. It may be possible to optimize the Scala OR1MP implementation further, but this was not pursued since the purpose of this research project is to investigate the relative performance when parallelizing the algorithm.

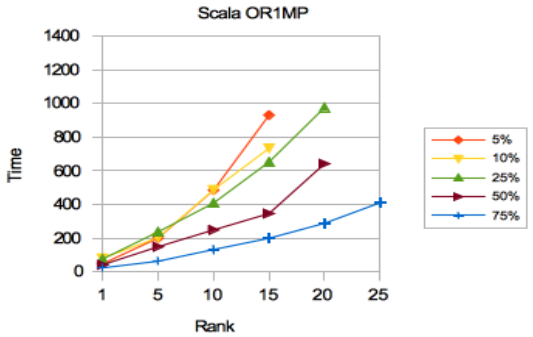
In both the MATLAB and the Scala implementations, the results show that the performance increases non-linearly with the rank. Each of the charts display "elbows", where performance starts to decrease rapidly. This was the point at which memory started to become scarce on the system, and a certain amount of slowdown was caused due to memory management overhead (e.g. paging to disk).



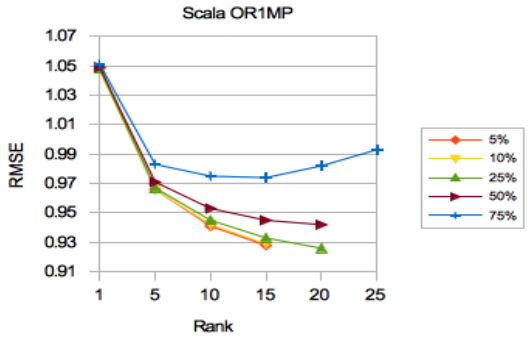
(a) MATLAB: Rank vs Time



(b) MATLAB: Rank vs RMSE



(c) Scala: Rank vs Time



(d) Scala: Rank vs RMSE

Figure 6.1: MATLAB OR1MP vs Scala OR1MP on Desktop

Test	Time (s)	RMSE
MATLAB-5%	319.8	0.943
MATLAB-10%	239.7	0.944
MATLAB-25%	180.3	0.946
MATLAB-50%	105	0.954
MATLAB-75%	43.1	0.977
Scala-5%	485.1	0.941
Scala-10%	487.3	0.942
Scala-25%	408.6	0.945
Scala-50%	248.8	0.953
Scala-75%	132.6	0.975

Table 6.1: MATLAB OR1MP vs Scala OR1MP on Desktop (at rank=10)

6.3 Optimal LAPACK/BLAS Library

This section discusses a simple performance test comparing a few different choices of LAPACK/BLAS libraries. As mentioned above, my Scala implementation of OR1MP uses a Scala library called Breeze for linear algebra operations, which, via the netlib-java project, can be configured to use a variety of different BLAS implementations at runtime.

The library which comes with Apple OS X (veclib framework) turns out to be one of the better performing LAPACK/BLAS libraries. Therefore, I did not try out different configurations for the tests I ran on my desktop PC.

However, prior to running my experiments on AWS, I ran a test to determine which LAPACK/BLAS library performed best. Since Spark's EC2 support is built

upon a Linux AMI from AWS, I selected a few well-known varieties to test which are readily available for this distribution of Linux. (Note that Intel MKL requires a license. For these tests, I used an evaluation license.)

The test consisted of running the Scala OR1MP program against the Netflix dataset with 10% sampling and rank = 10. The test was run on a single AWS (m2.4xlarge) server instance. Table 6.2 shows the results of these tests.

Library	Time (s)
F2J (JVM bytecode)	602
atlas (generic)	594
atlas-sse3 (SSE3 optimized)	593
Intel MKL	587

Table 6.2: BLAS/LAPACK Library Performance with Scala OR1MP on EC2

As can be seen from the results, for this program, only relatively minor performance differences were observed for the different BLAS/LAPACK libraries. Based on these results, even though Intel MKL performed slightly better, I used the ATLAS package (atlas-sse3) provided by AWS for use with the Linux AMI for ease of configuration.

6.4 Spark Implementation of DFC + OR1MP with Netflix on EC2

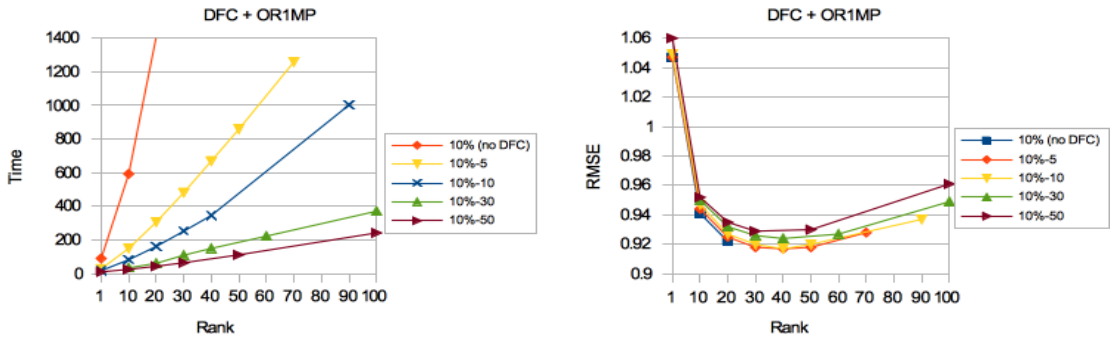
This section shows performance and accuracy results of running the Spark-based implementation of DFC + OR1MP across various splits of the Netflix dataset, where Spark is configured to execute each split in parallel across a separate AWS server instance.

In this test, I used AWS m2.4xlarge server types, which come with 8 virtual cores and 68GB of memory. While the large amount of memory was not required in many of the test scenarios, I wanted to be sure that I could scale up to larger rank tests without needing to rebuild the cluster. Even though I tested up to 50 splits of the data, I used a cluster of only 20 server instances, due to AWS limitations on the number of Spot Instances allowed of this server type. However, there were enough cores and memory for Spark to run all the data splits in parallel, even though in some cases (e.g. with 30 and 50 splits of the data) multiple splits were processed concurrently on a single server.

I used 10% sampling to generate the dataset for each of the test scenarios. Keeping the dataset constant, I varied the number of splits (e.g. "10%-5" refers to 5 splits, "10%-50" refers to 50 splits, etc.). As a baseline comparison, I also ran the Scala OR1MP algorithm without DFC for as many iterations as possible on this server type.

Figure 6.2 shows the results of these tests. As can be seen in 6.2b, the accuracy of DFC + OR1MP trails OR1MP alone very closely, even up to splits of 50. Also, note that it was possible to test DFC + OR1MP to much higher rank than possible with OR1MP alone, as the cumulative memory of the entire cluster could be utilized.

The performance benefits of DFC + OR1MP over OR1MP alone are clearly seen in 6.2a. Even with the additional overhead of combining the factorizations from each split of the data, the DFC + OR1MP algorithm surpasses the performance of OR1MP alone with the smallest numer of splits tested (i.e. 5 splits), and continues to improve up through the maximum number of splits tests (i.e. 50 splits). The exact values for the various tests at rank=20 are displayed in Table 6.3.



(a) DFC + OR1MP: Rank vs Time (b) DFC + OR1MP: Rank vs RMSE

Figure 6.2: Scala DFC + OR1MP on Netflix Dataset

Also, note that each of the splits shows near-linear performance with increased rank, which is due to the abundace of memory available – i.e. no slow-down due to memory paging was seen.

Test	Time (s)	RMSE
10% (no DFC)	1410	0.922
10%-5	307	0.925
10%-10	162	0.927
10%-30	62	0.932
10%-50	44	0.935

Table 6.3: DFC + OR1MP vs OR1MP (at rank=20)

Chapter 7

CONCLUSION

7.1 Summary of Findings

The purpose of this research project was to study the effectiveness and ease-of-use of a popular open source Big Data solution for iterative machine learning on large-scale data. More specifically, I set out to implement a low-rank matrix completion algorithm within a Big Data framework and then apply it to the Netflix Prize dataset.

My initial effort, implementing the SVT algorithm on Mahout/Hadoop, did not produce an adequately performant solution. While Mahout is an excellent project which provides users with a library of some useful large-scale machine learning algorithms, it is at present unacceptably constrained by the underlying MapReduce platform when it comes to iterative algorithms, such as SVT.

My second effort, implementing the OR1MP algorithm on Spark by utilizing RDDs to distribute the linear algebra operations, was significantly faster than my first effort, but it also did not produce a sufficiently performant solution. In the case of my implementation, it appears that there was too much cross-machine communication during the routine, and potentially unoptimized implementations of vector-matrix multiplication contributing to the lackluster performance. However, I did not pursue further profiling to identify any opportunities for optimization of this approach.

Instead, my third effort was to implement the DFC algorithm on Spark, and to use the low-rank completion algorithm OR1MP as the DFC "base" algorithm. I found that this implementation produced excellent performance and accuracy numbers, when compared to the MATLAB implementation of OR1MP on a single desktop

PC. In addition, as I increased the degree of parallelism of DFC + OR1MP, the performance of the algorithm increased significantly.

I also found that the implementation of DFC + OR1MP on Spark was relatively straightforward, especially when compared to my experience implementing SVT on Mahout/Hadoop. In the case of Mahout/Hadoop, every algorithm or operation not already expressed in Mahout needed to be translated to a MapReduce-compatible format before it could be implemented. In addition, I found that there was a considerable amount of boilerplate code dedicated to framework-related needs with Mahout/Hadoop. In the case of Spark, however, the programming model is flexible yet powerful enough such that both of my Spark-based implementations were concise and fairly straightforward to program.

7.2 Future Work

The results of this research project sugget a variety of different possible directions for future work, some of which are described below:

The OR1MP algorithm used in this paper is constrained in the number of iterations by the available memory, since the algorithm has to track all pursued bases in each iteration. As the OR1MP paper explains, it demands $O(r|\Omega|)$ storage complexity to obtain a rank-r estimated matrix [see 10, section 4]. The OR1MP paper goes on to propose an economic form of the algorithm called EOR1MP. This would be an interesting alternative to test as the base algorithm for DFC with our Spark-based implementation.

The competitive performance of DFC + OR1MP on Spark suggests it could be worth a more detailed comparison of this approach to methods based on Stochastic Gradient Descent, which is the more popular approach in the literature to low-rank matrix completion problems or collaborative filtering problems. As mentioned above,

there have been a variety of approaches proposed to scaling SGD in the context of large-scale data. It would be interesting to compare the methods in this paper with those algorithms, in the context of the Spark framework.

Finally, the author would be interested in connecting this research to the topic of graph representations. Some of the existing graph processing enginges (e.g. GraphLab) have translated SGD, SVD, and other machine learning algorithms into the semantics of their framework. It would be interesting to study whether OR1MP or DFC + OR1MP could be expressed well within graph semantics, and if so, how the performance characteristics compare to a Spark implementation.

REFERENCES

[1] MATLAB website. http://www.mathworks.com/products/matlab/.

[2] The R Project for Statistical Computing. http://www.r-project.org/.

[3] GNU Octave. https://www.gnu.org/software/octave/.

[4] Apache Hadoop. http://hadoop.apache.org/.

[5] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[6] Amazon Web Services. http://aws.amazon.com/.

[7] Apache Mahout. https://mahout.apache.org/.

[8] Apache Spark. https://spark.apache.org/.

[9] Jian-Feng Cai, Emmanuel J Candès, and Zuowei Shen. A singular value thresholding algorithm for matrix completion. *SIAM Journal on Optimization*, 20(4):1956–1982, 2010.

[10] Zheng Wang, Ming-Jun Lai, Zhaosong Lu, Wei Fan, Hasan Davulcu, and Jieping Ye. Rank-One Matrix Pursuit for Matrix Completion. In *The 31st International Conference on Machine Learning (ICML)*, Beijing, China, 2014.

[11] Lester W Mackey, Michael I Jordan, and Ameet Talwalkar. Divide-and-Conquer Matrix Factorization. In *Advances in Neural Information Processing Systems*, pages 1134–1142, 2011.

[12] James Bennett and Stan Lanning. The Netflix Prize. In *KDD Cup and Workshop in conjunction with KDD*, 2007. http://www.netflixprize.com.

[13] Feng Niu, Benjamin Recht, Christopher Ré, and Stephen J Wright. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. *Advances in Neural Information Processing Systems*, 24:693–701, 2011.

[14] Rainer Gemulla, Erik Nijkamp, Peter J Haas, and Yannis Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 69–77. ACM, 2011.

[15] Yong Zhuang, Wei-Sheng Chin, Yu-Chin Juan, and Chih-Jen Lin. A fast parallel SGD for matrix factorization in shared memory systems. In *Proceedings of the 7th ACM conference on Recommender systems*, pages 249–256. ACM, 2013.

[16] Boduo Li, Sandeep Tata, and Yannis Sismanis. Sparkler: Supporting large-scale matrix factorization. In *Proceedings of the 16th International Conference on Extending Database Technology*, pages 625–636. ACM, 2013.

[17] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A System for Large-scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0032-2. doi: 10.1145/1807167.1807184. http://doi.acm.org/10.1145/1807167.1807184.

[18] Apache Giraph. https://giraph.apache.org/.

[19] GraphLab website, . http://graphlab.org/projects/index.html.

[20] Bulk Synchronous Parallel on Wikipedia. http://en.wikipedia.org/wiki/Bulk_synchronous_parallel.

[21] Berkeley Data Analytics Stack (BDAS). https://amplab.cs.berkeley.edu/software/.

[22] Cloudera Impala. http://www.cloudera.com/content/cloudera/en/products-and-services/cdh/impala.html.

[23] Presto DB website. http://prestodb.io/.

[24] MLbase website. http://www.mlbase.org/.

[25] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 29–43. ACM, 2003.

[26] Apache HIVE. https://hive.apache.org/.

[27] Apache HBase. https://hbase.apache.org/.

[28] Apache Hadoop NextGen MapReduce (YARN). http://hadoop.apache.org/docs/r2.3.0/hadoop-yarn/hadoop-yarn-site/YARN.html.

[29] Cloudera CDH. http://www.cloudera.com/content/cloudera/en/products-and-services/cdh.html.

[30] AMPLab at UC Berkeley website. https://amplab.cs.berkeley.edu/.

[31] Shark website. http://shark.cs.berkeley.edu/.

[32] GraphX website, . https://amplab.cs.berkeley.edu/publication/graphx-grades/.

[33] Tachyon website. http://tachyon-project.org/.

[34] Apache Mesos. http://mesos.apache.org/.

[35] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[36] ScalaNLP website. `http://www.scalanlp.org/`.

[37] netlib-java website. `https://github.com/fommil/netlib-java`.

[38] LAPACK website. `http://www.netlib.org/lapack/`.

[39] BLAS website. `http://www.netlib.org/blas/`.

[40] ATLAS website. `http://math-atlas.sourceforge.net/`.

[41] Intel MKL website. `http://software.intel.com/en-us/intel-mkl`.

[42] cuBLAS website. `https://developer.nvidia.com/cublas`.

[43] OpenBLAS website. `http://www.openblas.net/`.

[44] F2J website. `http://icl.cs.utk.edu/f2j/`.