

Towards Platforms for Improved Recommender Systems at Social Media Scale



Christina Diedhiou Sowinski
University of Portsmouth

The thesis is submitted in partial fulfilment of the requirements
for the award of the degree of

Doctor of Philosophy

September 2019

This thesis is dedicated to my family and my husband
for their strong support through my career.

Declaration

Whilst registered as a candidate for the above degree, I have not been registered for any other research award. The results and conclusions embodied in this thesis are the work of the named candidate and have not been submitted for any other academic award.

Word count: 27215

Christina Diedhiou Sowinski
September 2019

Acknowledgements

I would like to express my deepest gratitude to Dr Bryan Carpenter, my main supervisor, not only for believing in me and giving me the opportunity to start this research but also for his endless support and constant involvement in my doctorate journey. I have learned so much from him, from analyzing challenges that come along the research to tackle them while being as accurate as possible and ensuring that the satisfactory solutions provided are irrevocable. He has been a great inspiration and his willingness to always give his time to guide me has been highly appreciated.

The advice and encouragement given by my second and third supervision: Dr Mohamed Bader and Dr Mo Adda as well as Dr Mihaela Cocea, my assessor during my major and annual reviews, have been a great help along with my research. I wish to particularly thank Dr Mohamed Bader for giving me the opportunity to present my work at the conference Women in Data Science.

I would like to offer my special thanks to my husband, Dawid Sowinski for always being by my side to motivate and encourage me. Even at my most stressful moment, I could never feel alone. His unconditional love and support have been a constant reminder of what I am fighting for.

My Special thanks are extended to my parents and three amazing sisters: Cecile, Ingrid and Raissa for their endless love and support both moral and financial. I would not have been able to start this journey if it were not for them neither would I had the strength to complete this doctorate without their encouragement. They have taught me how to never give up however the difficulty of the situation. Being geographically so far from my family and yet so close with all the technology tools and social media was such a comfort.

I am particularly grateful to my dearest colleagues at university and at work as well as my closest friends including the club "darlings" (Cecile, Muriel, Sarah and Ana), for all their moral support, lovely humour and for being so understandable for all the time I could not socialize as I wish I could due to the fact that I needed to dedicate myself more to my studies. I am specially grateful to my best friend Zubilla for being such a great advisor in my life and for being like a sister to me to whom I can share anything and everything.

I would like to express my very great appreciation to all my research collaborators. Their knowledge and expertise have contributed to the success of this research. I would like to extend my great appreciation to my previous supervisor Mr St Clair Cordice who recently passed away, for always having the right words to motivate me when I needed it the most and for teaching me how to be a good soldier during my studies. He taught me all the best practices and I am forever grateful for his support and kindness. I hope to make you proud.

Finally, I would like to thank the organization Funds for Women Graduate for their financial help during my third year. The grant I have received from this trust has enabled to cover the shortfall of my tuition fees. This has taken a lot of pressure off me and as a result, I was able to focus more on my research.

Abstract

One of the challenges our society faces is the ever increasing amount of data, requiring systems to analyze large data sets without compromising their performances and humans to navigate through a deluge of irrelevant material. Among existing platforms that address the system requirements, Hadoop is a framework widely used to store and analyze Big Data. On the human side, one of the aids to finding the things people really want is recommendation systems. This thesis evaluates approaches to highly scalable parallel algorithms for recommendation systems with application to very large datasets. A particular goal is to evaluate an open source Java message passing library for parallel computing called MPJ Express, which has been integrated with Hadoop. We also use MapReduce a core component of Hadoop to partition our data and implement a parallel distribution of our datasets. These last have been acquired from well-known recommender systems such as MovieLens and Yahoo Music. Based on these datasets we generate our own synthetic dataset aiming for a larger size in order to test the scalability of the model. We name that dataset “SyntheD”.

As a demonstration we use MPJ Express to implement collaborative filtering on various datasets using the algorithm ALSWR (Alternating-Least-Squares with Weighted- λ -Regularization). We benchmark the performance and demonstrate parallel speedup on Movielens, Yahoo Music and SyntheD datasets. We then compare our results with other frameworks such as: Mahout, Spark and Giraph then measure the accuracy of the program with a suitable error metric. Our results indicate that MPJ Express implementation of ALSWR has a very competitive performance and scalability in comparison with the other frameworks we evaluated.

Contents

| | | |
|----------|---|-----------|
| 1 | Introduction | 1 |
| 1.1 | Motivation and Contribution | 1 |
| 1.2 | Background | 3 |
| 1.3 | Methodology | 4 |
| 1.3.1 | Research Design and Methodology | 4 |
| 1.3.2 | Description of Data and Test Environment | 7 |
| 1.4 | Outline of Thesis | 8 |
| | | |
| 2 | Literature Review | 10 |
| 2.1 | Introduction | 10 |
| 2.2 | Overview of Recommender Systems | 10 |
| 2.3 | Synthetic Datasets | 12 |
| 2.3.1 | GenerateData.com | 14 |
| 2.3.2 | FreeDataGenerator.com | 14 |
| 2.3.3 | DataGenerator | 14 |
| 2.4 | Recommender System Techniques | 15 |
| 2.4.1 | Content Based Filtering Techniques | 15 |
| 2.4.2 | Collaborative Filtering Techniques | 16 |
| 2.4.3 | Hybrid Techniques | 17 |
| 2.5 | Challenges within Recommender Systems | 17 |
| 2.6 | Parallel Computing | 19 |
| 2.6.1 | Memory Architectures of Parallel Computer | 19 |
| 2.6.2 | Programming Models | 20 |
| 2.7 | Evaluation Metrics | 22 |
| 2.8 | Related Work | 23 |
| 2.9 | Summary | 25 |

| | | |
|----------|--|-----------|
| 3 | Overview of Existing Recommender Systems Platforms | 26 |
| 3.1 | Introduction | 26 |
| 3.2 | Overview of Hadoop | 27 |
| 3.3 | Apache Mahout | 28 |
| 3.4 | Apache Spark | 31 |
| 3.5 | Apache Giraph | 33 |
| 3.6 | Conclusion | 34 |
| 4 | Implementation of ALSWR with MPJ Express | 36 |
| 4.1 | Introduction | 36 |
| 4.2 | MPJ Express | 37 |
| 4.3 | Alternating Least Squares with Weighted Lambda Regularization | 39 |
| 4.4 | Implementation of ALSWR | 41 |
| 4.5 | Partitioning of Dataset | 44 |
| 4.5.1 | Partitioning of Ratings Dataset using a Serial Java Appli- cation | 46 |
| 4.5.2 | Partitioning of Ratings Dataset with MapReduce | 47 |
| 4.6 | Partitioning Data within the SPMD Program | 50 |
| 4.6.1 | Background on CARI APIs | 51 |
| 4.6.2 | Use of CARI in the present work | 54 |
| 4.7 | Conclusion | 57 |
| 5 | Performance Comparison across Hadoop-based Frameworks | 59 |
| 5.1 | Introduction | 59 |
| 5.2 | Environment Set Up | 60 |
| 5.2.1 | Prerequisites | 60 |
| 5.2.2 | Setting Up the Nodes | 61 |
| 5.2.3 | Bashrc File | 62 |
| 5.3 | MovieLens 20M Ratings Experiments | 63 |
| 5.4 | Yahoo Music 700M Ratings Experiments | 65 |
| 5.5 | Root Mean Square Error (RMSE) Implementation | 68 |
| 5.6 | Analysis of the results | 68 |
| 5.7 | Conclusion | 71 |

| | | |
|----------|--|------------|
| 6 | SPMD Processing of Hadoop Data | 73 |
| 6.1 | Introduction | 73 |
| 6.2 | Statement of Problem | 74 |
| 6.3 | Heuristics to solve Local Block allocation to nodes | 76 |
| 6.4 | Evaluation on Randomly Generated Replica Distributions | 83 |
| 6.5 | Practical Application | 84 |
| 6.6 | Conclusion | 88 |
| 7 | Toward Social Media Scale | 89 |
| 7.1 | Introduction | 89 |
| 7.2 | Example of Social Media Recommendation Techniques | 90 |
| 7.2.1 | Facebook | 90 |
| 7.2.2 | Instagram | 91 |
| 7.2.3 | Linkedin | 91 |
| 7.3 | Growth of Data within Social Media | 93 |
| 7.4 | SyntheD: A Solution to Measure Scalability | 94 |
| 7.4.1 | Prerequisites | 94 |
| 7.4.2 | Implementation | 95 |
| 7.4.3 | Data Creation | 96 |
| 7.5 | SyntheD 10B Ratings Experiments | 97 |
| 7.5.1 | Prospects for Processing Larger Datasets | 99 |
| 7.6 | Conclusion | 100 |
| 8 | Conclusion | 101 |
| 8.1 | Introduction | 101 |
| 8.2 | Summary of the Research | 101 |
| 8.3 | Contributions | 102 |
| 8.4 | Future Works | 103 |
| | Bibliography | 105 |
| A | Ethical Review Certificate | 116 |
| B | UPR16 Form | 118 |

List of Figures

| | | |
|------|---|----|
| 1.1 | Research Road Map | 5 |
| 1.2 | Recommender Systems Techniques | 6 |
| 2.1 | Amazon Recommendation | 13 |
| 2.2 | Recommendation Evolution | 13 |
| 2.3 | Single Core VS Dual Core | 19 |
| 2.4 | Shared Memory | 20 |
| 2.5 | Distributed Memory | 21 |
| 2.6 | Hybrid Shared & Distributed Memory | 21 |
| 2.7 | Rotational Hybrid Approach [52] | 25 |
| 3.1 | Hadoop 2 Architecture | 27 |
| 3.2 | YARN Architecture [98] | 28 |
| 3.3 | Apache Mahout Architecture | 29 |
| 3.4 | Apache Spark Components | 32 |
| 4.1 | MPJ Express Configuration | 37 |
| 4.2 | MPJ Express Integrated in YARN | 39 |
| 4.3 | Visualization of an iteration of distributed ALSWR algorithm | 42 |
| 4.4 | Sparse data structure to represent locally held ratings | 43 |
| 4.5 | Outline Java code for step 1 of the ALSWR update (Equation 4.2) using sparse data structures of Figure 4.4 | 45 |
| 4.6 | MapReduce/MPJ Express Processing Pipeline, mediated by HDFS | 48 |
| 4.7 | Map Reduce Partitioning Class Diagram | 49 |
| 4.8 | Schematic C++ API for CARI | 53 |
| 4.9 | Actual Java API for CARI subset (various class members used only as part of implementation are omitted) | 55 |
| 4.10 | Partioning code using CARI | 56 |

| | | |
|-----|---|----|
| 5.1 | Frameworks Performance Comparison MPJ Express with MovieLens dataset | 64 |
| 5.2 | Parallel Speedup MPJ Express vs Spark with MovieLens dataset . | 64 |
| 5.3 | Performance Comparison MPJ Express vs Spark with Yahoo dataset on Max four Nodes | 65 |
| 5.4 | Parallel Speedup MPJ Express vs Spark with Yahoo dataset . . . | 66 |
| 5.5 | Comparison MPJ Express vs Spark Time Performance with Yahoo dataset on 12 Nodes Cluster (SPMD Model) | 66 |
| 5.6 | Parallel Speedup MPJ Express vs Spark with Yahoo dataset on Max 12 nodes (SPMD model) | 67 |
| 5.7 | Outline Java code for the RMSE calculation (Equation 2.1). . . . | 69 |
| 5.8 | RMSE Results with Different Lambda Parameters | 70 |
| 6.1 | Simple example of allocation of block reads. In this example the SPMD world covers the entire cluster of four hosts. | 76 |
| 6.2 | A non-uniform distribution of replicas over four hosts, with the results of various allocation heuristics. | 80 |
| 6.3 | Comparison of “parallel read times”. Blue plots are for “Round Robin heuristic” and red plots for “Improved Heuristic”. | 85 |
| 6.4 | Example use of HDFS API for extracting metadata about blocks | 86 |
| 6.5 | Example use of MapReduce API for processing blocks in SPMD program. | 87 |
| 7.1 | Users 'Growth of Facebook, Instagram and LinkedIn | 93 |
| 7.2 | Synthetic Dataset Creation Process | 96 |
| 7.3 | MPJ Express Time Performance with SyntheD dataset on 12 Nodes | 98 |
| 7.4 | Parallel Speedup with threads per node - MPJ Express with SyntheD on 12 nodes | 98 |

List of Tables

| | | |
|-----|--|----|
| 4.1 | Time Performance in Seconds of Datasets Partitioning | 50 |
| 4.2 | MapReduce Partitioning on 12 Nodes Cluster in Seconds | 50 |
| 4.3 | CARI Partitioning on 12 Nodes Cluster in Seconds | 57 |
| 5.1 | Performance MPJ Express vs Spark in minutes | 65 |
| 6.1 | Times in seconds for reading Yahoo Training set with and without local block read heuristic | 88 |
| 7.1 | Synthetic Datasets Generation Time | 97 |

Abbreviations

ALS Alternating Least Squares.

ALSWR Alternating Least Squares with Weighted Lambda Regularization.

AM Application Master.

CARI Collective Asynchronous Remote Invocation.

CDD++ Cyclic Coordinate Descent ++.

DAAL Intel Data Analytic Acceleration Library.

HDFS Hadoop Distributed File System.

HPC High-Performance Computing.

MAE Mean Absolute Error.

MPI Message Passing Interface.

MPJ Java Message Passing Interface.

NMs Node Managers.

RDD Resilient Distributed Datasets.

RM Resource Managers.

RMSE Root Mean Square Error.

SGD Stochastic Gradient Descent.

SPMD Single Program, Multiple Data.

YARN Yet Another Resource Negotiator.

Chapter 1

Introduction

1.1 Motivation and Contribution

Our daily activities are led by choices and decisions that constantly have to be made, whether it is about some news to read, music to listen to, movies to watch, clothes to wear and so many others. From a business perception, the main questions remain how to increase sales, who to target and what to auction and more importantly how to retain existing customers and attract new ones. The innovation of technology and the availability of the Internet have lead to data overload. An individual looking for a particular product online, for instance, could end up spending a large amount of time due to the variety of items available. One of the challenges our society faces is the ever-increasing amount of data, requiring systems to analyze large data sets without compromising their performances, and humans to navigate through a deluge of irrelevant material. An alternative solution to assist people in finding what they want is through recommendation systems. Recommender systems are software tools and techniques providing suggestion to users in order to help them find items that match their requirements. In this dissertation, the term user will be used to determine a person interested in some goods or services and the term item as a product or service recommended by a system.

Among existing platforms that address the system requirements, Hadoop [2] is a framework widely used to store and analyze Big Data. Some of the papers reviewed have discussed the application of High-Performance Computing (HPC) technologies like Message Passing Interface (MPI) to recommender systems problems. Our particular interest is in employing platforms inspired by HPC technologies specifically an open source Java message passing library for

parallel computing called MPJ Express [60] that can also interface and integrate naturally with commercially important Big Data frameworks like Hadoop and showing how the synthesis of the two can facilitate the exploitation of both for the problems of interest. This approach has received little attention so far. The envisaged platforms and software should appeal to commercial entities already using Hadoop and its ecosystems. They should also leverage the performance of HPC environment allowing practical solutions of problems on the scale envisaged by Facebook [52] and others. Our contribution to the area therefore includes:

- The development of suitable approaches for highly scalable recommender systems with application to the scale of data characteristic of social media. This has been covered in chapter 4.
 - To demonstrate the benefits of integrating MPJ Express platform into the Hadoop ecosystem, including the possibility of using MPJ Express as a phase in a processing pipeline and including traditional Hadoop platforms like MapReduce (achieved in section 4.5.2).
 - To evaluate our approach to HPC inspired parallel recommender algorithms in Hadoop in comparison with other frameworks like Apache Mahout, Spark and Giraph and to show the benefit of our approach. This has been covered in chapter 5.
 - To devise and test strategies for efficiently reading HDFS data into Single Program, Multiple Data (SPMD) parallel programs including MPI or MPJ as explained in chapter 6.
 - The implementation of a method to generate synthetic datasets that can reach beyond billions of ratings and evaluate the performance of our approach on a large synthetic dataset. This has been covered in chapter 7.
 - To demonstrate a high impact on HPC technologies with MPJ Express by achieving good parallel speedup and demonstrating a good error metric as compared to other well-known frameworks. The results of our work are published in the following conference papers.
- i. Christina Diedhiou, Bryan Carpenter, Ramazan Esmeli: Comparison of Platforms for Recommender Algorithm on Large Datasets. ICCSW 2018: 4:1-4:10

- ii. C. Diedhiou, B. Carpenter, A. Shafi, S. Sarkar, R. Esmeli and R. Gadsdon, "Performance Comparison of a Parallel Recommender Algorithm Across Three Hadoop-Based Frameworks," 2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), Lyon, France, 2018, pp. 380-387. doi:10.1109/CAHPC.2018.8645926

1.2 Background

Over the last decade, Apache Hadoop has established itself as a pillar in the ecosystem of software frameworks for Big Data processing. As an open source, mostly Java-based Apache project with many industrial contributors, it retains a commanding position in its field.

When first released Hadoop was a platform primarily supporting the MapReduce programming model and other projects built on top of MapReduce. Around 2014 with the release of Hadoop 2.0 the platform was refactored into a separate YARN (Yet Another Resource Negotiator) resource allocation manager, with MapReduce now just one of the multiple possible distributed computation frameworks that could be supported on top of YARN. Several other major big data projects rapidly migrated to allow execution on the Hadoop YARN platform (for example Apache Spark [100], Apache Giraph [1], Apache Tez [86], and Microsoft Dryad [50]). Around the same time, Zafar, Khan, Carpenter, Shafi and Malik envisaged adding the existing MPJ Express framework for MPI-like computation in Java to that distinguished group and developed a version of the software that could also run under Hadoop YARN [98].

MPJ Express is a relatively conservative port of the standard MPI 1.2 parallel programming interface to Java and is provided with both "pure Java" implementations (based on Java sockets and threads) and "native" implementations exploiting specific interconnect interfaces or implementations on top of standard MPI. The vision was thus to support MPJ as one computational framework among many largely Java-based or JVM-based frameworks that could be mixed and matched for different stages of complex big data processing, with Hadoop and Hadoop Distributed File System (HDFS) as the "glue" between stages.

Our goal in this thesis is to provide evidence that such a scenario can be realized and that it may be advantageous. We concentrate on one particular computationally intensive Big Data problem - generating product recommendations through the collaborative filtering algorithm Alternating Least Squares

with Weighted Lambda Regularization (ALSWR). A version of this algorithm is developed and evaluated using MPJ running under Hadoop. We show in passing how our implementation can usefully be decomposed into two stages using different YARN-based computational frameworks—MapReduce for the partitioning of data followed by MPJ for the parallel iterative computation. We then go on to compare our implementation with three existing implementations of ALSWR that can run under Hadoop—one taken from the Apache Mahout project using MapReduce, another using Apache Spark and a last using Apache Giraph. Figure 1.1 presents a road map for this research.

1.3 Methodology

This section details the methodology adopted for this research specifying the types of data used and the recommender system approaches as well as the hardware, software and libraries required for the implementation of this project. We furthermore justify the decision made to implement the chosen methods. We discuss the design of the research and methods employed including the frameworks that are mostly used for large datasets and the facilities that are available to us for the overall implementation of our study and describe the data used for the experiments along with our test environment and resources available.

1.3.1 Research Design and Methodology

The aim of this research is to develop and evaluate approaches to scalable, highly parallel platforms and algorithms for recommendation problems with application to online commerce and social media. Facebook, for example, have recently described their needs to recommend products to more than a billion users and publicized parallel algorithms they are evaluating to achieve their goal [52]. The authors also claimed that Facebook average dataset for collaborative filtering has 100 billion ratings. The size of this dataset is much larger than the one from the well-known Netflix Prize recommender competition with 100 million ratings. Therefore new updated techniques are required to deal with such large data. Facebook favour one particular framework called Apache Giraph. It aims to iteratively perform graph processing on very large data. Sakr [87] describes Giraph as a distributed and fault-tolerant system that adopts the Bulk Synchronous Parallel programming model to run parallel algorithms for processing large-scale graph data. We evaluate different frameworks including especially our own MPJ

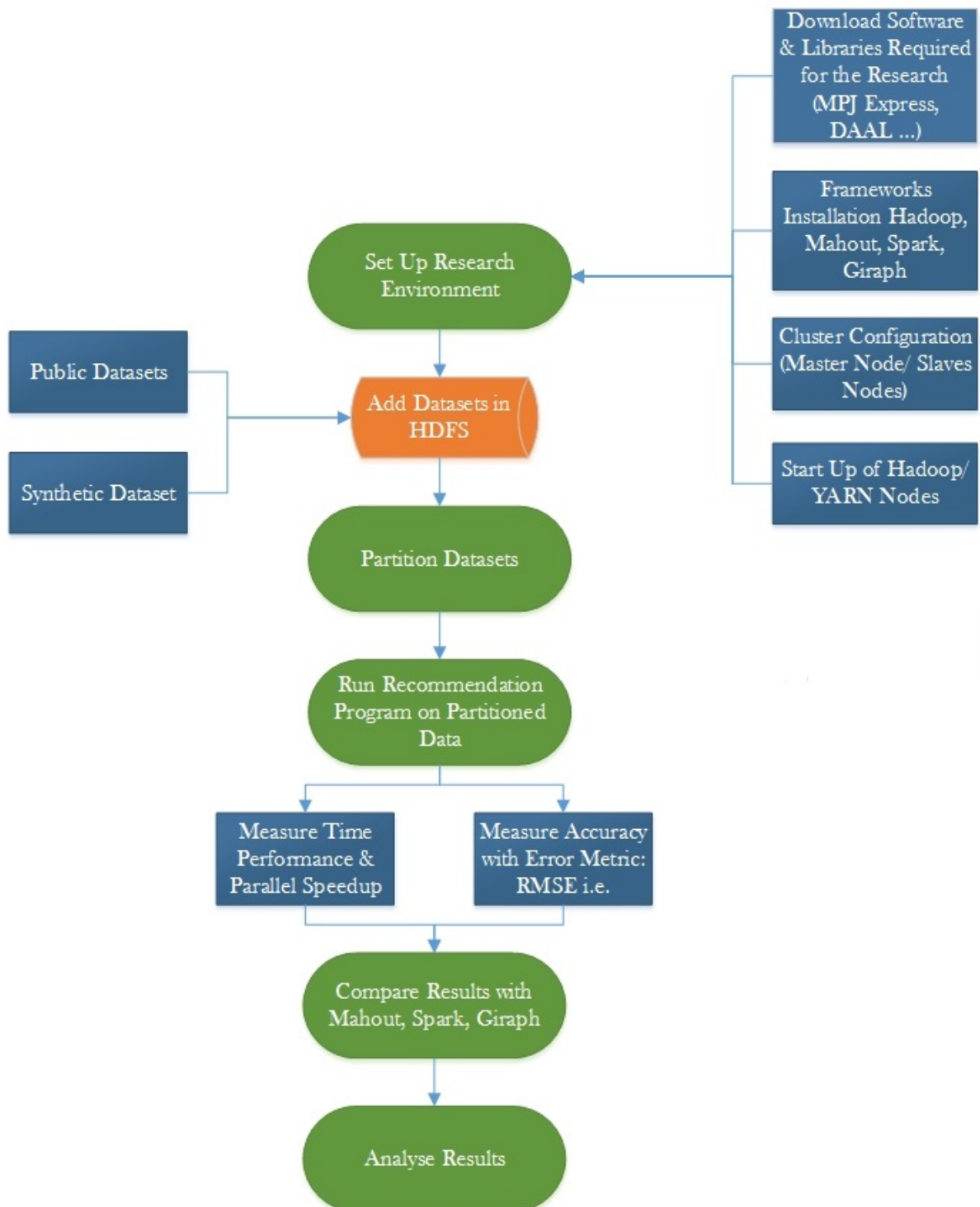


Figure 1.1: Research Road Map

Express that can run under Apache Hadoop YARN. We discuss MPJ Express in chapter 4. We also assess frameworks such as Hadoop, MPJ Express, Apache Mahout, Apache Spark and Apache Giraph as well as the importance of Java as a de facto standard for much commercial computing, versus traditional HPC languages with MPI. A good example of work using MPI for large datasets is the experimental results of Yu, Hsieh, Si, and Dhillon [97]. They have used coordinate descent approaches such as Stochastic Gradient Descent (SGD), to parallel matrix factorization. We intend to consider different underlying algorithms including ALSWR described in chapter 4. Figure 1.2 represents the hierarchy of the approaches we use. Some of the recommender system techniques are discussed in section 2.4.

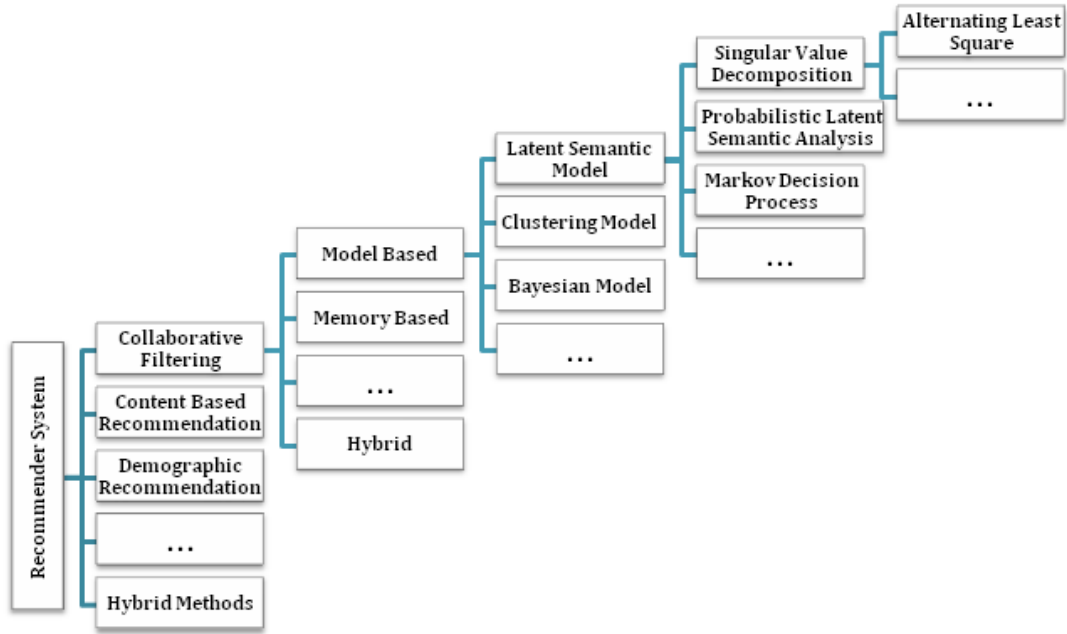


Figure 1.2: Recommender Systems Techniques

We opt for iterative approaches because we want to apply some regularization to our data in order to prevent overfitting which is a common challenge when dealing with matrices. Once the model is built we evaluate the error metric of the predictions by calculating their Root Mean Square Error (RMSE). In section 2.7 we distinguish between two suitable evaluation metrics: MAE and RMSE. MAE and RMSE are both popular and widely used. The size of the datasets used and their complexity have put RMSE in favour as we aim for an approach more sensitive to errors and RMSE penalises more large errors through the least-square technique with the aim to improve the performance of the model

[19]. The aspiration to compare our results with those from the paper [101] has furthermore been a motivation to choose RMSE approach. We also measure the speed of execution on highly parallel platforms called parallel speedup. This involves extensive benchmarking using suitable protocols. Hoeffler and Belli [46] discuss some benchmarking applicable to parallel computing. The Resources available for the pursuit of our research include a cluster in the Big Data lab in the school of computing composed of four nodes and a cluster located in the school of engineering consisting of 13 nodes. These should be configured or configurable to provide Hadoop services. It is important because we typically find that some phases of processing certain datasets are best done using MapReduce interspersed with iterative algorithms using other computational frameworks that can also exploit Hadoop, MPJ Express, Spark and so on. Another reason why it is important to have these clusters configured with Hadoop is that this last is commercially a popular platform for distributed computing.

1.3.2 Description of Data and Test Environment

For the purpose of performance evaluation, we acquired our datasets from public domains. These consist of anonymous user ratings from two different sources: MovieLens and Yahoo Music. The dataset obtained from MovieLens contains 20,000,263 ratings for 27,278 movies, created by 138,493 users [37]. The dataset from Yahoo Music—that is much larger—contains over 717 millions ratings for 136 thousand songs rated by 1.8 million users [96]. The data from Yahoo has been separated into training and test datasets. The size of the dataset is extended by generating a synthetic dataset in order to verify the scalability of MPJ Express integrated with Hadoop and Yarn. Some data generator tools and libraries are reviewed in section 2.3. We have decided to create our own data generator program instead of using one of the generator tools described in section 2.3 because we wanted the code to be more flexible, to easily adapt it to our changing requirements and to have it personalised for future usage. When generating synthetic data it is important that the data looks as real as possible and for that reason, we produce data based on the existing models already obtained. This is achieved by an algorithm which creates data while following the same patterns of the designated dataset. We target to reach a dataset size ranging between billions to trillions of ratings. In chapter 7 we further discuss the implementation of the synthetic dataset: SyntheD.

Our initial test environment comprised a small Linux cluster composed of

two nodes having six cores each and two nodes with four cores, giving us in total 20 cores. In experiments on this cluster we generally limited ourselves to using 16 cores in order to get better load balancing across nodes. Our final test environment consists of a cluster of 13 nodes having each 12 cores. In total, we have at our disposal 156 cores for the experiments with the awareness that using more processes than the number of cores available could lead to a lower performance of the program more particularly in terms of time and parallel speedup results. The software used for the tests consist of:

- Java 1.7
- Apache ant 1.6.2
- Hadoop-2.9.2
- MPJ Express (version 0.44), Mahout (version 0.12.2), and Spark (version 2.2.0)
- Intel Data Analytic Acceleration Library (DAAL) 2019

1.4 Outline of Thesis

This introductory chapter has given an insight into the research area. We have defined the problem and state our contribution to the field. The rest of the thesis is organized as follows.

- Chapter 2 assesses and evaluates relevant tools, technologies and models that can be applied to the research as well as related studies in the similar field. This chapter brings a firm background to our study and presents the options available in term of methods suitable for the accomplishment of this thesis. Chapter 2 also describes the approaches followed to implement our programs and determines the hardware, software and libraries we use to achieve our goals. We additionally justify the choice of the methods followed and outline the various datasets used for the research.
- Chapter 3 gives an overview of the frameworks reviewed for this research and on which some experiments have been carried out. These frameworks consist of Apache Hadoop, Mahout, Spark, and Giraph. As Hadoop is the chosen framework for this research, we provide details of the architecture of Hadoop 2 which is the version that includes the resource manager YARN.

- Chapter 4 focuses on the implementation of the Alternating Least Squares with Weighted Lambda Regularization (ALSWR) and collective communication which is the backbone of how our accomplishment of parallel computing. We review MPJ Express, its configuration and the mechanism behind its integration with YARN (Yet Another Resource Manager). We then describe the different methods that have been adopted to partition the datasets, measure their performances and compare them against each other.
- Chapter 5 compares the results obtained from MPJ Express with those from Apache Spark, Mahout and Giraph by measuring their performance and more particularly the time, the parallel speedup and the accuracy of the recommendation. We break down the environment set up for the testing and describe the variables used for the tests. The comparison between the frameworks is followed by a thorough analysis of their results.
- Chapter 6 makes some general observations about loading data from an HDFS file system into a Single Program, Multiple Data (SPMD) program. The chapter discusses how the block reads can be allocated to the nodes in order to allow records to be read locally from HDFS into a parallel program. This approach is evaluated on block distributions that have been randomly generated. The parallel reading time for Yahoo dataset is then benchmarked using this approach.
- Chapter 7 discusses the steps we take to move toward the social media scale and provides details on the implementation of our synthetic dataset: SyntheD. We go through the steps followed that has enabled us to generate the desired size of data and determine the software as well as libraries required for its development.
- Chapter 8 summarizes our research and provides guidance for future works. The chapter reviews the contributions made to the field and outlines the limits of the research while suggesting how these drawbacks could be overcome.

Chapter 2

Literature Review

2.1 Introduction

With the aim to establish a firm background of this thesis, this chapter assesses and critically evaluates the literature relating to recommender systems and more particularly collaborative filtering techniques. We evaluate works that have been done so far in system recommendation and provide an understanding of the field. This enables to position our thesis within the context and determine the contribution brought to this area of study. Therefore the literature review chapter focuses on existing techniques and previous work applied to recommender systems, their efficiency when dealing with a large amount of data and challenges faced. This chapter is structured in four sections. Section 2.2 gives an insight into recommender systems background and current approaches. Section 2.3 analyzes some data generator tools that could be used to generate data for our experiments on top of the datasets already acquired. Section 2.4 describes some of the recommender system techniques. Section 2.5 outlines some of the challenges faced in the area of recommender systems. We explain the concept of parallel computing in section 2.6 and in section 2.8 we review similar studies while assessing their methodologies, their effectiveness and issues.

2.2 Overview of Recommender Systems

The ability to build systems recommendations emerged at an early stage in the history of computing. Rich [84] argued that a major technique people use to build models of other people very quickly is the evocation of stereotypes or clusters of characteristics. Following that thought, Grundy [84], an automatic system act-

ing as a librarian was implemented to provide books recommendations to users based on a stereotype model. The rationale behind the system is to simulate as closely as possible the behaviour of a librarian by asking a few questions to the users before suggesting them a book in which they might be interested. Although the system was very primitive it has been identified as one of the major starting points in the recommender systems era. The beginning of the nineties saw the inception of the term collaborative filtering. Back then, collaborative filtering was identified as an ideal approach to tackle information overload. It was claimed that information filtering can be more effective when humans are involved in the filtering process [34]. In consequence, Tapestry [34], an experimental mail system has been developed at the Xerox Palo Alto Research Centre designed with the aim to support both content-based filtering and collaborative filtering. The main idea was to deal with electronic mails received and act as a mail filter and repository at the same time. The terminology collaborative filtering is employed in the sense that users of the Tapestry system collaborate to help each other by keeping a record of their feedbacks such as whether a given document was interesting or not. These feedbacks could then be reviewed at any moment whenever desired. By the mid-nineties, recommender systems had evolved into automated collaborative systems. Only ratings anonymously made by users were required. It was not necessary anymore for a particular user to be aware of the interest of another user and the items or users contained in the system in order to get a suggestion. GroupLens [37], a research lab in Computer Science and Engineering adopted the same technique. Konstan et al [55], demonstrate how the GroupLens system applies collaborative filtering to Usenet news and how the value of this latter can be restored by sharing their judgements of articles while protecting their identities with pseudonyms. Automatic collaborative filtering was a big step in the history of recommender systems from the stage where users were responsible for entering their own reactions to a particular item and looking up previous reactions, to a stage where the connection is already automatically established through a system following some rules to recommend items without revealing the identity of other users. Following this trend, many recommender systems and collaborative filtering techniques have been developed in various domains such as computer science, medicine, research and development, academia, e-commerce and so many others. Sivapalan, Sadeghian, Rahnama, and Madni [90] argued that the movement toward E-commerce has allowed companies to provide customers with more options which are causing businesses to increase the amount

of information that customers must process before they are able to select their desired items. They further state that one solution to this information overload problem is the use of recommender systems. Surely this last has proven its ability to increase not only sales but also the consumers' satisfaction on the service received. Amazon.com which is a popular website widely use for various services has a recommender system built in. Suggestions to users on products they might be interested in are provided by the system and depend on their purchases and browsing history. Figure 2.1 illustrates a type of recommendation sent by email to Amazon customers based on previous purchases. Beyond collaborative filtering, several techniques have been implemented such as content-based methods mostly focusing on relevance feedback, genetic algorithms, neural networks, and the Bayesian classifier which are among the learning techniques for evaluating users' profile. Another approach to designing recommender systems is through hybrid methods. This last can combine collaborative filtering and content-based, for instance, targeting a better performance and effectiveness of the technologies. In 2006, research in algorithms for recommender systems gained significant importance as Netflix introduced the Netflix prize an open competition intending to improve the conception of movies recommendations. In order to win the competition candidates had to come up with their best collaborative filtering solutions that could beat Netflix own internal CineMatch algorithm by 10 per cent on its test set. Figure 2.2 graphically represents the main points marking the evolution of recommender systems.

2.3 Synthetic Datasets

Many datasets are already publicly available; however, to reach a certain scale sometimes a larger dataset might be required for testing purposes. Synthetic datasets are defined by the production of data in a random way while following some criteria such as the type of data needed and its value. To experiment with our programs we gradually increase the size of the dataset and measure the performance. The code is constantly tested against its scalability which means that the fact of increasing the size of the data should not affect the performance of the MPJ Express program in terms of time and accuracy.

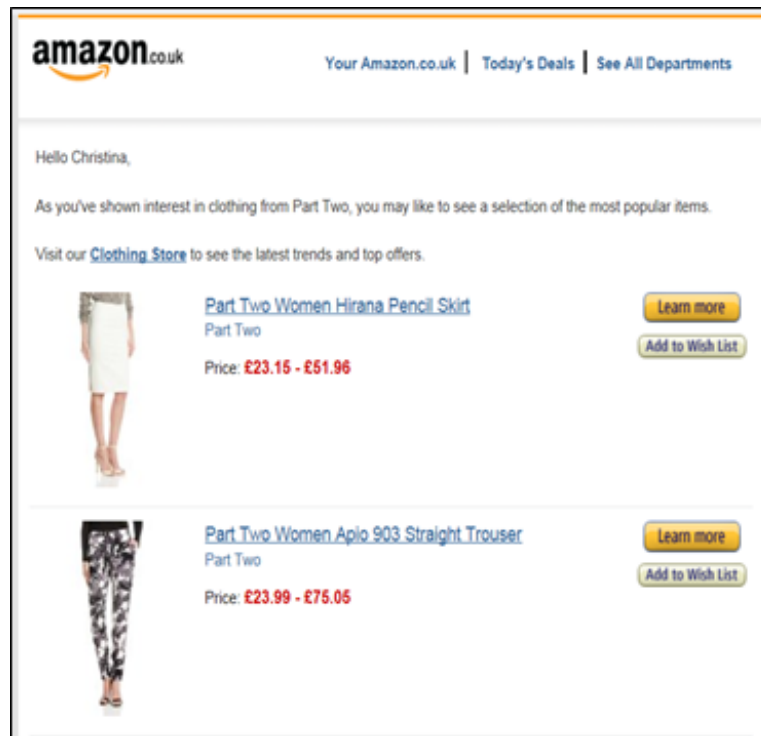


Figure 2.1: Amazon Recommendation

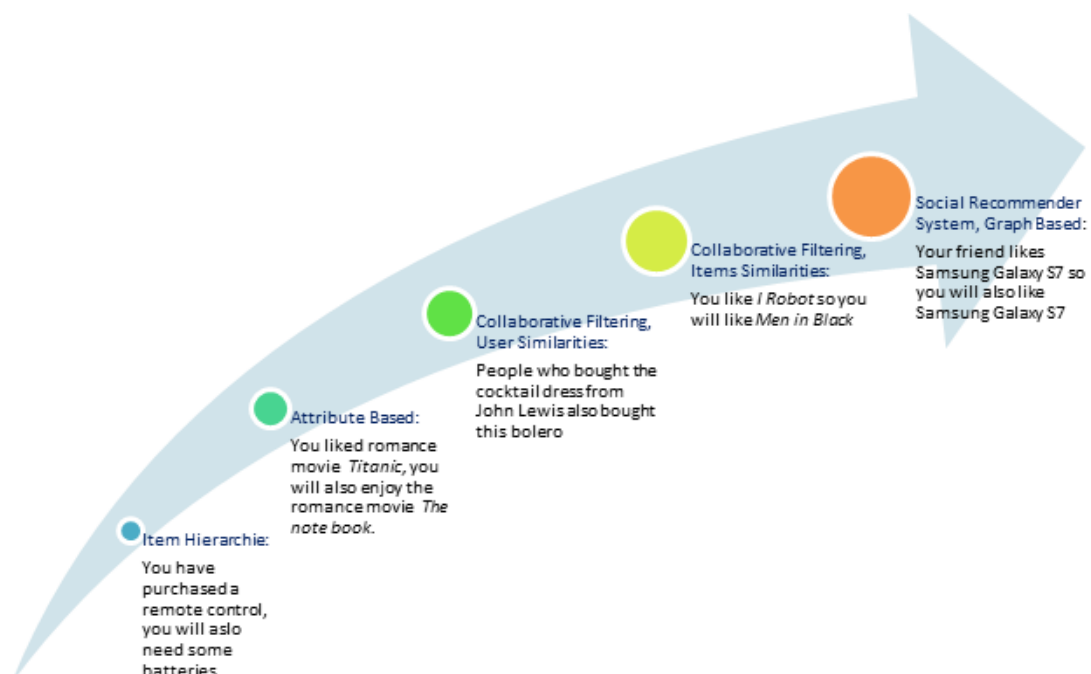


Figure 2.2: Recommendation Evolution

2.3.1 GenerateData.com

GenerateData.com is an open source tool written in JavaScript, PHP and MySQL [54]. The online service enables data to be randomly generated for free up to a maximum of 100 rows. Users requiring more data have to make a donation otherwise there is a possibility to download the software and modify the codes. The codes are written in PHP and JavaScript. Developers can contribute to this software on Github. Once the data has been generated it can be saved on different formats including CSV, Excel, HTML, JSON, LDIF, Programming Language, SQL and XML. One downside is the fact that it allows only the generation of 100 rows at a time, it can be time-consuming if the target is to generate billions of rows and the user is unable to alter the code.

2.3.2 FreeDataGenerator.com

FreeDataGenerator.com is an online tool free of charge which allows users to generate data to a maximum of 1000 rows at a time. At the moment only a beta version is available [61]. The formats supported are CSV, Excel, XML, JSON and SQL. Various types of data can be created. Users are suggested to log in to their account to protect their data. On a downside, this project has not been open sourced yet and there is currently no possibility for developers to adjust the code to their needs. Additionally, this tool could work well for small datasets but similar to GenerateData.com could become time-consuming for users wishing to produce billions of data.

2.3.3 DataGenerator

DataGenerator is an open source library enabling data to be created based on a model thus making the data as real as possible. The library is claimed to be able to produce terabyte of data within minutes [47]. Users are required to write a java code where they can add the DataGenerator library. DataGenerator uses SCXML (State Chart Extensible Markup Language) which is based on XML. This library can also be integrated with Hadoop and MapReduce. Some examples of code, quick start guide and tutorial videos are provided to assist programmers. The samples of code are originally written in Java. Any contribution to the project can be made via the Google group of the project team or GitHub. This tool is suitable for experienced programmers as they must know their way around the tool to write their own code and integrate the required libraries. However,

for less experienced programmers, more time will have to be spent to understand the architecture and to write a code which can implement the DataGenerator tool.

2.4 Recommender System Techniques

Resnick and Varian [83] distinguished two different ways in which a recommendation problem may be formulated: the prediction version of the problem and the ranking version of the problem. The prediction version of the problem consists of predicting the value of ratings obtained in order to establish a user-item correlation. A training data anonymously revealing user preferences for certain items is therefore required. As explained by Zhou, Wilkinson, Schreiber and Pan [101], assuming $A = \{r_{ij}\}_{n_u \times n_m}$ denotes the user-movie matrix, r_{ij} represents the rating score of movie j given by a particular user i , n_u indicates the number of users and n_m the number of movies. The main task to solve the problem is then to estimate some of the missing values in the matrix A depending on the known values. The ranking version of the problem, otherwise called the top-k recommendation problem, involves recommending the top-k items to a particular user. The aim is to find a few specific items which are the most interesting to the user. Although the prediction version is more general due to the fact that the solutions of the ranking problem can be derived from it, the method of the ranking version is more straightforward to design. The information on users that is necessary in order to recommend items to them can be obtained in two different ways. It can be acquired explicitly which means that users' ratings are collected then processed or implicitly, a technique which monitors users' behaviour for instance by keeping track on the websites visited, songs listened to, software downloaded and so on. Many techniques related to recommender systems have been developed along the years since it has first been used. We describe the main ones in the following paragraphs; however, our focus will later remain on collaborative filtering.

2.4.1 Content Based Filtering Techniques

The content-based approach provides recommendations based on an evaluation on the profile of a user and by analyzing the content of some items purchased in the past. A great advantage of this method as discussed in [51] is that if the user profile changes, this technique still has the potential to adjust its rec-

ommendations within a very short period of time. This technique is also able to provide new suggestions to users and the lack of ratings will not affect the accuracy of these suggestions. Furthermore, the fact that users are not required to share their profile keeps their privacy more secured as more detailed information on users increase the number of threats on their privacy. Unfortunately, this technique does not cover the notion of serendipity—an important goal for recommender systems—as all recommendations made must correspond to items which are already determined in the user’s profile. Additionally, this technique has the ability to recommend new items to users but is not efficient when recommending items to new users as they do not have a sufficient list of past ratings.

2.4.2 Collaborative Filtering Techniques

Collaborative filtering systems are based on users’ purchases or decisions histories. Assuming two individuals share the same opinion on an item, they are also more likely to have a similar taste on another item. Collaborative filtering systems are categorized either as memory based or model-based methods. Memory-based methods are characterized by the fact that they used the entire database to make predictions as opposed to model-based methods which only require the user database to make some estimations. Memory-based methods attempt to find users that are similar to the one targeted for predictions, and uses their preferences to predict ratings for the concerned user [10]. K Nearest Neighbours (KNN) which is a memory based approach is arguably the most widely used algorithm for collaborative filtering [8]. The design and usability of this method are straightforward, but its performance tends to decrease when data get sparse thus affecting its scalability and its aptitude to deal with large datasets. Furthermore, new data entry adds some complexity to the mechanism. As for Model-based methods, they use past ratings and machine learning algorithms to find some patterns necessary for recommendations. The advantages of model-based approaches compared to memory based are their ability to better handle data sparsity and their scalability enabling this approach to better deal with large datasets [11]. Some of the main model-based algorithms include Bayesian network, clustering models and Latent semantic models. In our experiments in chapter 5 we have opted for a model-based approach and specifically use Alternating-Least-Squares with Weighted- λ -Regularization (ALSWR) algorithm. The Alternating-Least-Squares method (ALS) can be categorized as a matrix factorization approach otherwise called matrix decomposition. As sug-

gested by its name, the matrix factorization aims to decompose a matrix into smaller components easier to manage. Matrix Factorization aims to add a penalty function to deal with missing ratings which is the main cause of data sparsity. In section 2.5 we go through some of the challenges faced by recommender systems techniques.

2.4.3 Hybrid Techniques

Hybrid techniques are a mix of two or more approaches applied to a model in order to come up with a method aiming to cover the limits of each of the chosen technique. Collaborative filtering and content-based have been often used in hybrid techniques. Hybrid techniques are claimed to have better accuracy than other methods in [33], however, many factors must be taken into consideration such as the type of data and the compatibility between the methods combined. Examples of Hybrid methods can be found in [20] combining item-based collaborative filtering with sequential pattern mining to improve e-learning applications and [26] which attempts to solve the cold start and data sparsity problems by merging deep presentation learning with matrix factorization.

2.5 Challenges within Recommender Systems

Recommender systems have been proven to enhance the manipulation of data within businesses and to better understand the behaviour of customers; however, even though the ability to mine data has been one of the best innovation in computing for the last decade and has been used as a key tool by many organizations to emancipate their business, it still encounters some challenges. As companies grow, knowledge becomes more complex and technologies have to be elevated along with the progress. We next list some of the conflicts encountered in recommender systems. It is to be noted that the following list is not exhaustive and not in any particular order.

- High-performance computation: The speed to load and process information is crucial with the constantly increasing competition in the market, requiring businesses to make fast decisions. An essential feature of high-performance computing (HPC) is parallel computing. Can a program be effectively parallel without losing too much time on the communication between the nodes?

- Predicting the trends in an accurate and efficient manner can be challenging. While industries require high-speed software, they also need all the information collected to be as accurate as possible in order to adopt the best approach and maximize their potential. Shilling attacks—the manipulation of some recommendation scores by users with fake profiles—can occur and systems must be built to be able to detect falsified information. On another side, each model that is built for prediction must have a method to measure the accuracy of its forecasts prior to its utilization in business.
- Mining complex knowledge due to complex data: data nowadays come in different forms and shapes requiring systems to be able to deal with these new types of data.
- Data privacy: it is important to put in place some security means to prevent the privacy of consumers to be breached. Trust is imperative between an organization and its clients. During the whole process of data mining personal data must be kept confidential and under no circumstance disclosed to a third party without authorization.
- Necessity to tackle data sparsity: each user rates only a few items leaving a large number of elements as zero in the matrix. When half of the matrix or more hold the value 0, this matrix is said to be sparse. Data sparsity can affect the quality of recommendation systems.
- Cold start: otherwise called “new user or new items problems”. This issue occurs when there is not enough information gathered on a user or item due to their novelty. The preferences of the user cannot be forecast and this can lead to less accurate results. Although a solution via demographic attributes has been proposed in [85], more research is required in this field for better results.
- Scalability: It is a constant challenge as the size of data within organizations keep increasing and new systems need to accommodate new volumes of data as well as these of the next generations. Therefore a scalable recommender system should perform well on a large dataset the same way it will perform on a smaller dataset.

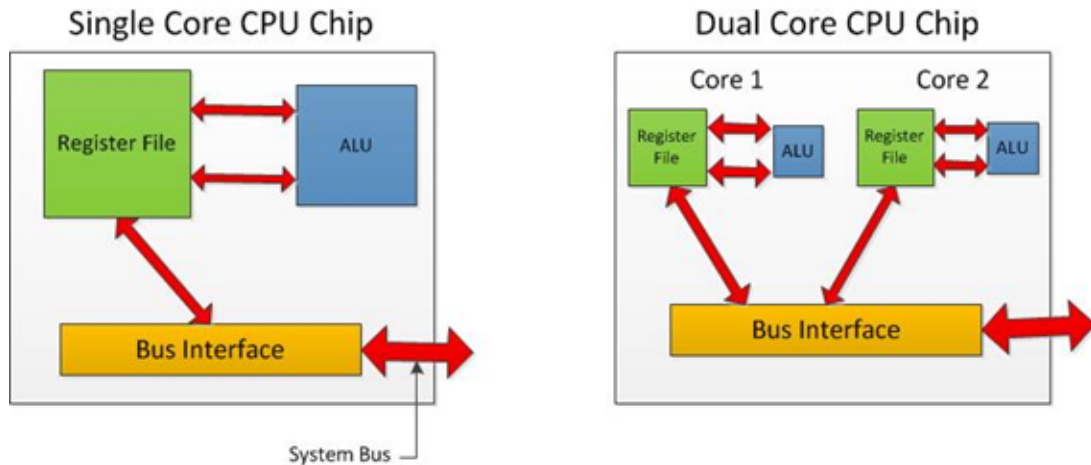


Figure 2.3: Single Core VS Dual Core

2.6 Parallel Computing

Software programs were traditionally written for serial computation. This involves a breakdown of the program into a series of instructions which are sequentially carried out on a single processor. Conversely, parallel computing simultaneously executes its series of instructions on many cores or different processors. Modern computers and laptops already have parallel architecture due to their multiple processors/cores. Domingo in [25] stated that multicore CPUs first appeared on the desktop and laptop PC platforms around 2005. The past few years have seen an evolution on laptops and desktops toward quad-core, hexa-core and octa-core processor. The main advantage of having a multicore CPU is its ability to perform many tasks at the same time. Figure 2.3 illustrates single core versus multi-core CPU chip. The end goal of parallel programming is to be able to deal with complex applications effectively, reliably and in a faster way. Additionally, shortening the completion time of a project by using parallel computing enables cost savings. The following sections discuss some of the features of parallel computers such as their architectures, programming models and what to consider when designing a parallel program.

2.6.1 Memory Architectures of Parallel Computer

The architecture of parallel computers generally consists of shared memory, distributed memory or hybrid shared-distributed memory. Shared memory parallel computer is characterized by the ability of all processors to access and share the same memory although they can operate autonomously. Thus all processors can

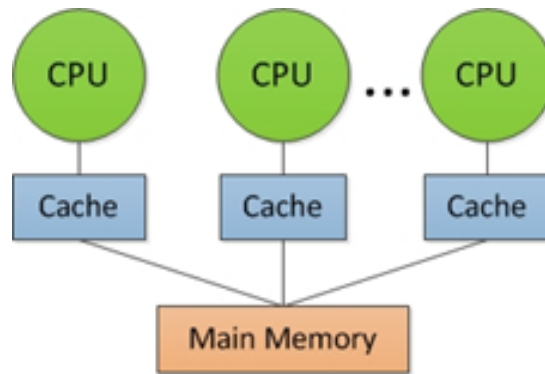


Figure 2.4: Shared Memory

view modifications that are made by another processor in the memory location. Figure 2.4 illustrates the architecture of a typical shared memory parallel machine. One of the advantages of having this architecture is the data distribution speed which is relatively fast due to the closeness between the processors and the memory. On the downside, they are not very scalable and the fact of adding more processors make the exchanges more complex between them. Distributed memory systems, unlike the shared memory architecture, require a network to connect the memory of each processor to enable communication as shown in figure 2.5. Because each CPU has its own memory, modifications of a processor to its memory do not affect the memory of other processors. One of the advantages of having systems built this way is good scalability as there is an excellent balance between the processors and the size of the memory. Nevertheless, programmers have to ensure that all the elements generating communication between the processors are specified. An alternative solution to cover part of the disadvantages of shared and distributed memory systems is a Hybrid Distributed-Shared Memory system (HDSM). HDSM systems can be defined as a group of shared memory systems connected through a network as in figure 2.6. They are generally used by the fastest supercomputers.

2.6.2 Programming Models

There are various programming models used for parallel systems although we will only describe those which are more relevant to the implementation of our experiments in chapter 4. These consist of threads, message passing and Single Program, Multiple Data (SPMD). As explained by Barney [7], models are not specific to a particular type of machine or memory architecture, they can be implemented on any underlying hardware. For instance, threads which are

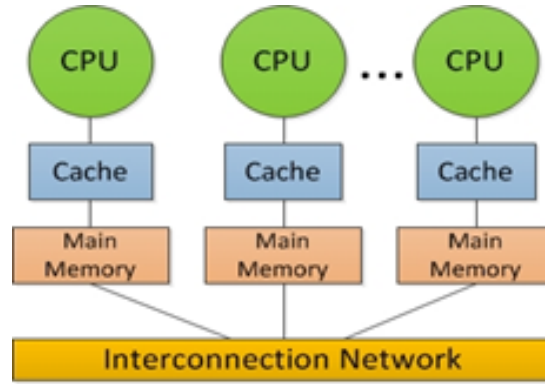


Figure 2.5: Distributed Memory

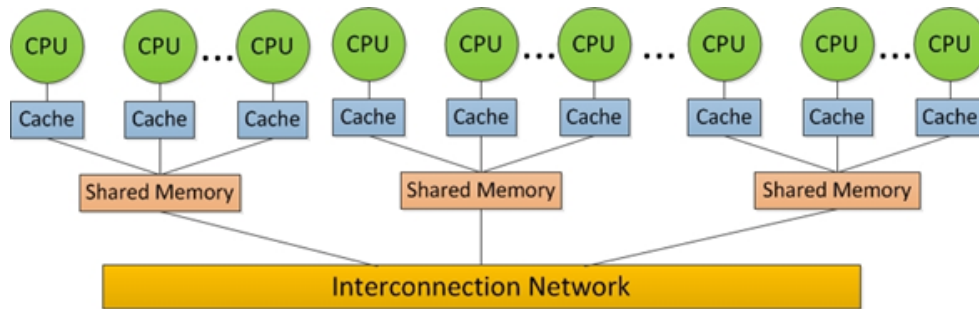


Figure 2.6: Hybrid Shared & Distributed Memory

typically shared memory programming model can also be used on a distributed memory machine.

A Thread is an execution of a single sequence of instruction in a program. The concept of multithreading is to enable the CPU to switch between the threads which then provides an effect of parallel execution. The coder is in charge of determining the parallelism although libraries are used to manage their implementations. OpenMP (Open Multi-Processing) programming has achieved a lot of success in the High-Performance Computing community due to its emphasis on structured parallel programming [94]. There are two common ways of executing threads: through POSIX (Portable Operating System Interface) Threads and through OpenMP.

Message Passing is a programming model for distributed systems. It enables processes to exchange messages generally by means of signals or function with each other. Gropp, Lusk, Doss and Skjellum in [36] stated that originally syntax and precise semantics of each message-passing library implementation was different from each other. In 1992, the process of creating standard means to enable the portability of message-passing applications was initiated. It spawned the Message Passing Interface (MPI). MPI was developed for software using C

or Fortran as the programming language. This later led to Message Passing Interface for Java (MPJ) as this programming language was gaining significant importance. We discuss MPJ further in chapter 4.

Single Program, Multiple Data (SPMD) is a programming method that allows a program to have separate instances of a single program that can be run loosely synchronously on many processors, thus achieving a parallel processing distribution. This method was proposed by Darema in 1984, as a way to enable parallel execution of applications on multiprocessors [22], [23]. The MPI model is widely used as a framework to implement the SPMD paradigm. In MPI this can be achieved by specifying the size of the world, the rank and the MPI communicator: `MPI.COMM_WORLD`. In this context we define the size of the world as the number of processes used to run a program and the rank as the position of a running process. The SPMD model is used extensively in the thesis.

2.7 Evaluation Metrics

Error metrics are some means to measure the quality of a prediction model. They essentially enable to distinguish the difference between the results obtained by a program and some actual values observed. Having some tools to measure the quality of prediction facilitates the comparison between estimators which can be the base of a decision making of an organization wishing to implement a recommender system. The most important aspect with error metrics is their ability to determine the accuracy of recommender systems. There are many evaluation metrics available. The choice of the most appropriate metrics depends on the type of data and the model which is built. Among those which are the most used for the regression model are Root Mean Square Error (RMSE) and Mean Absolute Error (MAE).

The RMSE otherwise called Root Mean Square Deviation (RMSD) is a calculation that aims to provide the standard deviation of a prediction model by computing the square root of the variance between the predicted values and the one which is observed. The value of the RMSE is always a positive number. The lower the results achieved by an estimator, the more accurate the model is. However, the value zero is almost impossible to attain as it would mean that the model is a perfect fit and that all predictions are 100 per cent accurate. The

RMSE is defined by the following equation:

$$RMSE = \sqrt{\frac{\sum_{r=1}^{T_n} (Predicted_r - Actual_r)^2}{T_n}} \quad (2.1)$$

Assuming the RMSE evaluates a model predicting ratings given by some customers. In principle, the dataset is split in training and testing sets to enable the accuracy to be measured after the model is built. In equation 2.1, T_n represents the total number of ratings and r the index of a rating.

The Mean Absolute Error (MAE) is a calculation which produces the average difference of absolute values taken from the predicted results of an estimator and assessed against some actual values. Similarly to the RMSE, lower values of the MAE are favourable and only positive numbers are provided as it takes the absolute modulus of the variance between two variables. Considering the same case of datasets as with the RMSE, the MAE is defined by the following equation:

$$MAE = \frac{1}{T_n} \sum_{r=1}^{T_n} |Predicted_r - Actual_r| \quad (2.2)$$

With the MAE the weight of all variances is equal to each other hence making it a straight forward approach to apply while RMSE has a heavier penalization feature. The RMSE was claimed to be a better fit with model performance and more particularly if the error distribution is expected to be Gaussian [18].

2.8 Related Work

Recommender systems have been a subject of tremendous interest lately. Our interest is however focused on collaborative filtering applied to very large datasets, leading to a need for parallel processing.

For the Netflix Prize, [101] proposed an approach called ALSWR. In order to implement their algorithm in parallel authors used Matlab [43]. The result of the experiments showed a better performance of the ALSWR as the number of features and iterations increased. The metrics used for their experiments are the parallel speedup and Root Mean Squared Error (RMSE) on which they obtained the score 0.8985. This was accomplished with the Netflix dataset consisting of 100 million ratings acquired in 2006. The current state-of-the-art dataset comprises

of billions of ratings [52] and processing this requires scalable methods.

Yu, Hsieh, Si and Dhillon compared ALS methods to Stochastic Gradient Descent methods and Coordinate Descent methods [97]. The study introduced the Cyclic Coordinate Descent++ algorithm (CCD++). The experimental platform is an MPI library and datasets are from MovieLens, Netflix, Yahoo-music and a synthetic dataset. The performance metrics reported in the study are the parallel speedup and the RMSE. From their work, they demonstrate that CCD++ is faster than ALS and SGD. Although ALS seems to have a better parallel speed up than CDD++.

Makari et al [58] compared SGD with Asynchronous SGD (ASGD), Distributed SGD for processing environment with MapReduce (DSGD-MR), Distributed SGD on clusters (DSGD++) and SGD on a single powerful machine with multiple shared-memory processors (CSGD). From their experiments, they proved that CSGD is more scalable than the other algorithms but that DSGD++ had a better overall performance due to its memory consumption and runtime. In their experiment, they used the MPICH2 library. Two datasets were used for the tests including the Netflix dataset and the KDD dataset [45]. For the performance metric, they opted for the parallel speedup.

Various other researchers have realized similar studies such as Karydi and Margaritis [53] (2014) who have also used the Netflix dataset but with a different approach called Bregman Co-clustering. There is another interesting research geared towards very large datasets in [88] that used Alternating Least Squares (ALS) as the algorithm on Hadoop MapReduce and JBlas as the framework. The datasets used for the experiments were from Netflix, Yahoo Music and a synthetic dataset they called Bigflick. Kabiljo and Ilic explained how ALS and SGD algorithms are implemented with Apache Giraph to process an average of 100 billion ratings from Facebook [52] using the rotational hybrid approach. In this approach illustrated in figure 2.7, the vertices correspond to users and the workers are the items. The idea was to accomplish a clockwise rotation following each sequence of iterations also called supersteps.

In the context of MPJ Express, previous work [98] focused on integrating the software with YARN allowing end-users to execute Java MPI programs on Hadoop clusters. As part of this effort, a new YARN-based runtime system was added to the MPJ Express library. The paper demonstrated a reasonable comparative performance of YARN-based runtime against the existing runtime. Also, the K-Means algorithm was parallelized using MPJ Express and MapRe-

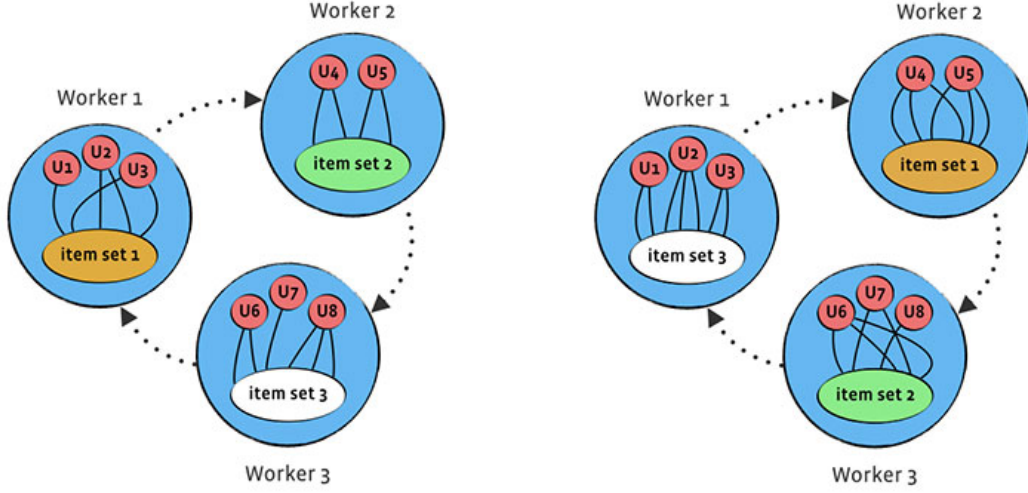


Figure 2.7: Rotational Hybrid Approach [52]

duce and executed on Hadoop clusters and Amazon Elastic Compute Cloud. It was shown that the Java MPI version of K-Means outperformed the MapReduce version. This study did not compare the performance of YARN-based MPJ Express library against some of the newer technologies including Apache Spark.

2.9 Summary

In this chapter, we have studied relevant tools, techniques and approaches in recommender systems required for the research. We have described the origin of recommender systems, the type of resources that can be useful and the main challenges to tackle when building a system. Recommender systems are a broad field, thus comparing the methods available has allowed us to narrow the area in which we wish to evolve. Reviewing related works has furthermore enabled to position the thesis in the research area and to set our targets.

Chapter 3

Overview of Existing Recommender Systems Platforms

3.1 Introduction

This chapter reviews the main third party platforms for recommendation systems evaluated in this research. We evaluate Apache Hadoop, Mahout, Spark and Giraph. The choices of the platforms are based on their popularity, existing reviews on their performance and aptitude to solve some common and newly arising challenges in high-performance computing. While each framework has its strengths and weaknesses, some of them perform better when combined together and more particularly with Hadoop as the backbone. In each section, we describe the main features of the corresponding framework, its components and the method used to implement the ALS algorithm for our testing purposes. Section 3.2 discusses Apache Hadoop and YARN, its resource manager then outlines both their components. Section 3.3 provides information on Apache Mahout as well as its architecture within the Hadoop ecosystem. Section 3.4 Reviews Apache Spark and explains its Resilient Distributed Datasets (RDD) feature. Section 3.5 describes the features of Apache Giraph including its input method.

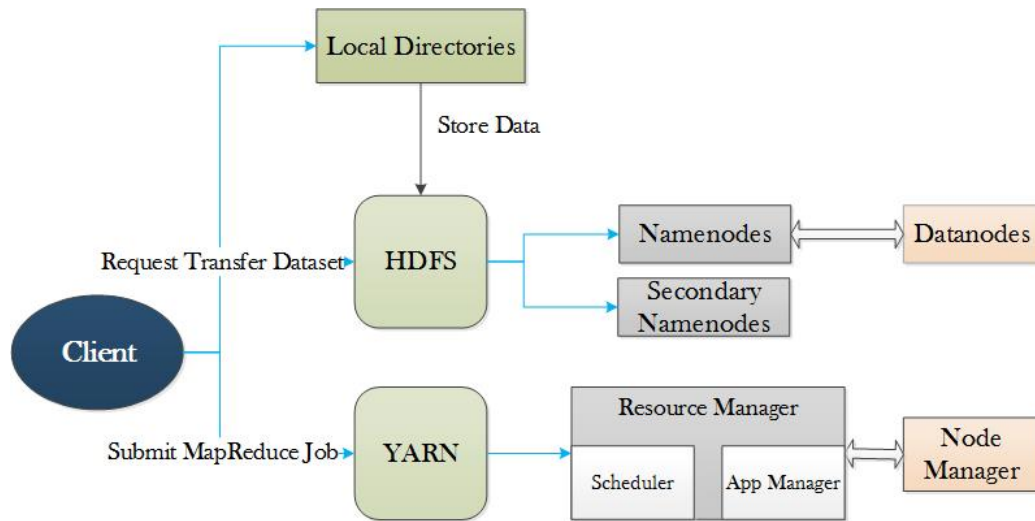


Figure 3.1: Hadoop 2 Architecture

3.2 Overview of Hadoop

Hadoop is a framework that stores and processes voluminous amounts of data. The resource management component was originally in the first version of MapReduce. It was composed of a job tracker and task trackers but is now deprecated in the second version of Hadoop. The job tracker also known as the master node, managed all tasks and resources while task trackers executed map and reduce tasks. These responsibilities have been shifted to YARN. Two important tasks define the implementation of MapReduce: a map phase and a reduce phase that are both specified by the programmer. The map phase has for input a set of data which is divided in tuples made of key and value pairs. The reduce phase uses the output of the map phase as input to a set of reduce tasks also running in parallel. All completed jobs can be monitored thanks to the Job history.

As illustrated in figure 3.1, the Hadoop Distributed File System (HDFS) has two types of node: a namenode which represents the master node, and datanodes also called slave nodes. There is only one namenode per HDFS cluster and its role is to manage the filesystem namespace by storing the metadata of all the files and their directory tree within the filesystem. The namenode keeps track of the location of each block that is stored on the datanodes. Datanodes follow instructions from the namenode to store and retrieve data then report back to it once the job is executed.

Since Hadoop 2, YARN (Yet Another Resource Negotiator) has been integrated into the infrastructure as the resource manager, enabling many other

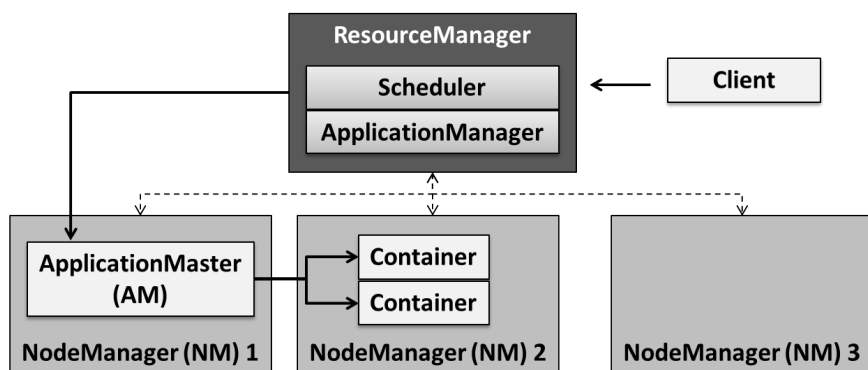


Figure 3.2: YARN Architecture [98]

distributed frameworks besides MapReduce to process their data on the Hadoop cluster. YARN depends on three main components to complete a task: a Resource Manager (RM), Node Managers (NMs), and an Application Master (AM). The RM is responsible for managing and allocating the resources across the cluster. There is only one RM per cluster. NMs run on all nodes available in a cluster and report all the tasks to the RM such as the number of cores and memory space. Each job that is started has an AM specific to the processing framework that manages operation within containers and ensures there are sufficient containers for the task. The term container is used here to describe the environment where applications such as MapReduce tasks are processed—essentially containers correspond to operating system processes running individual tasks. Containers require a certain amount of resources, CPU allocation for instance. Figure 3.2 shows the communication between the RM, the NMs and the AM following a client’s request. The communication between the master nodes and slave nodes is achieved through the Heart Beat Mechanism [38].

3.3 Apache Mahout

Apache Mahout is a distributed linear algebra framework which uses mathematically expressive Scala Domain Specific Languages(DSL) designed to enable users to implement their own algorithms [3]. Mahout is widely used for its distributed implementation on Apache Hadoop. This essentially means that datasets are stored in HDFS and various machine learning algorithms such as collaborative filtering can be applied to the data. Figure 3.3 illustrates the basic structure of Mahout with Hadoop 2. Mahout’s core algorithms include clustering, recommendation including collaborative filtering and classification. Its API can be

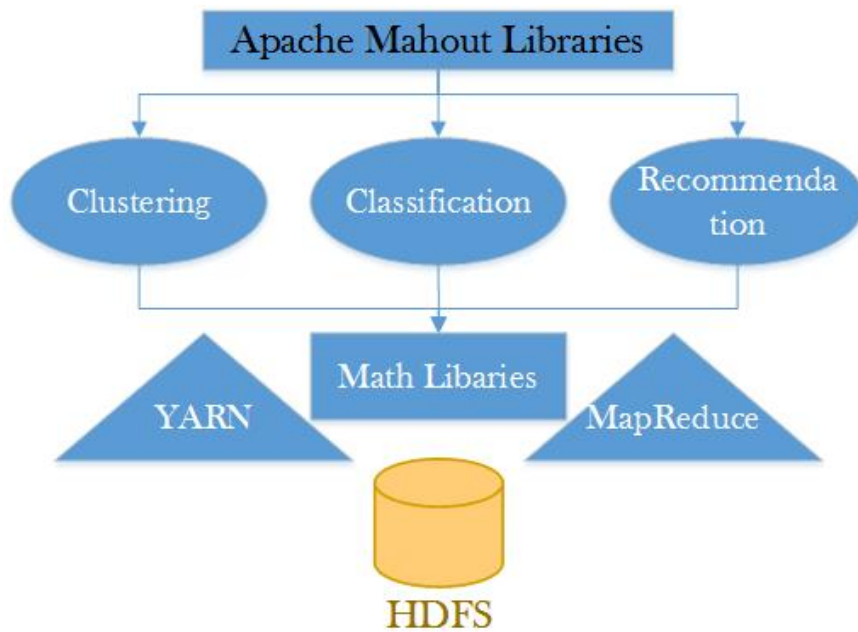


Figure 3.3: Apache Mahout Architecture

implemented in either Java or Scala. Mahout also uses MapReduce for its parallel processing and is consequently only disk based which means that the data is read/written from/to local disks. Apache Mahout has many distributed engine bindings consisting of Spark, Flink and H2O.

The ALSWR implementation with Apache Mahout is done through its machine learning library and more specifically the map-reduce implementation of ALS. This last consists of two stages: a parallel matrix factorization phase followed up by some recommendations. Both phases are detailed in [57]. For instance, to run the ALS Parallel algorithm we use the following command:

```
$ mahout parallelALS --input /user/hdfs/ratings.csv --output
  MahoutTest--lambda 0.1 --implicitFeedback false --alpha 0.8
--numFeatures 15 --numIterations 10 --numThreadsPerSolver 1
--tempDir tmp
```

The file given as input in this example is the ratings from Movielens; the outputs are written in a file we name MahoutTest. We set the argument *implicitFeedback* to false as we are dealing with explicit data.

To make some recommendations we use the following command taking the result of the parallelALS algorithm as input:

```
$ mahout recommendfactorized --input MahoutTest/userRatings
```

```
--userFeatures MahoutTest/U/ --itemFeatures MahoutTest/M/  
--numRecommendations 2 --output Recommendations --maxRating 5
```

We then print out the results of the recommendation in a file called Recommendations where 2 recommendations are provided to each user on a scale ranging from 1 to 5.

The prerequisites in order to build Mahout jobs are Java, Maven and Hadoop. Before running a Mahout application, JAVA_HOME must be set in the bashrc file; we explain the purpose of the bashrc file in section 5.2.3. Hadoop must be well configured and all nodes (namenodes/ datanodes) already started. Additionally, some libraries such as mahout-mr-0.12.2.jar, commons-cli-2.0-mahout.jar and commons-math3-3.0.jar must have their classpath exported along with the latest version of Apache Mahout core. The following are some commands used in the bashrc file to export the required classpaths to compile Mahout jobs:

```
export MAHOUT_HOME=/opt/apache-mahout-distribution-0.12.2  
export PATH=$PATH:$MAHOUT_HOME/bin  
unset MAHOUT_LOCAL  
export  
CLASSPATH=/home/christina/mahout-core-0.7-sources.jar:$CLASSPATH  
export  
CLASSPATH=$CLASSPATH:/home/christina/commons-cli-2.0-mahout.jar  
export CLASSPATH=/home/christina/mahout-mr-0.12.2.jar:$CLASSPATH  
export CLASSPATH=/home/christina/commons-math3-3.0.jar:$CLASSPATH
```

One of the greatest advantages of Apache Mahout integrated with Hadoop is its ability to either run in local mode or in MapReduce mode. It is to be noted that in our case it is important to unset Mahout from local, as we do not wish for Mahout jobs to be run locally but rather in Hadoop file system. As a result, we are exploiting all the benefits of HDFS given that our datasets are relatively large.

Many organizations have adopted Apache Mahout as their analytic tool to study the behaviour of their consumers. Among these organizations, Amazon greatly contributes to Mahout with its project: Apache Mahout on Amazon Elastic MapReduce (EMR). EMR was launched in 2009 and is developed by Amazon Web Service platform (AWS). More information on their projects are available in [62] and [89].

3.4 Apache Spark

Apache Spark is an open-source cluster-computing framework suitable for large scale data processing. It has been claimed to be able to run 100 times faster than Hadoop MapReduce in memory and 10 times faster on disk [4]. Since Hadoop 2, Spark has been integrated with Hadoop allowing its programs to run on YARN. Spark differs from other frameworks integrated with Hadoop in the fact that it does not use MapReduce for parallel processing. While MapReduce is fully disk-based, Spark can use memory and disk processing. Programmers can choose between three languages for the API: Java, Scala or Python. the key feature of Spark is its Resilient Distributed Datasets (RDD). RDD is a data structure enabling fast and efficient data processing by storing intermediate results in distributed memory instead of a disk [5]. As explained in [99], the default is to keep the RDD in memory; when there is no more space in the RAM, Spark stores the rest on disk. Although Spark is an extremely fast framework it is an expensive approach as it requires a considerable large amount of RAM in order to fully exploit its potential when dealing with massive datasets such as billion ratings. To allow a processor to share data with other processors Spark provides shared variables. These are usually copied to all the machines available; however, Sparks also allows programmers to generate two restricted types of variable: broadcast variables and accumulators. The parallel operations that can be executed on an RDD include **reduce**, **collect** and **foreach**. There are two different types of operation: **transformation** which generate new RDDs and **action** which executes the **transformation** operation. Shared variables and parallel operations available in Spark are detailed in [100] and [24]. We have implemented ALS on Spark through its standard machine learning library (MLib). Figure 3.4 shows the components of Spark including the MLib. MLib is further grouped into four different categories of algorithms: Clustering, Regression, Classification and Recommendation, which contains the ALS algorithm. It is to be noted that Apache Spark can be deployed as standalone or by using YARN for its deployment. In the first case, Spark runs on the top of Hadoop whereby only the HDFS is used as a data storage. On the second case Spark not only uses the HDFS to store datasets but Spark jobs are additionally run on YARN hence allowing other components to be combined to the structure.

For the prerequisites, Spark and PySpark must be installed as well as either Java, Python or Scala. The Spark environment may be set up in the bashrc file by commands as:

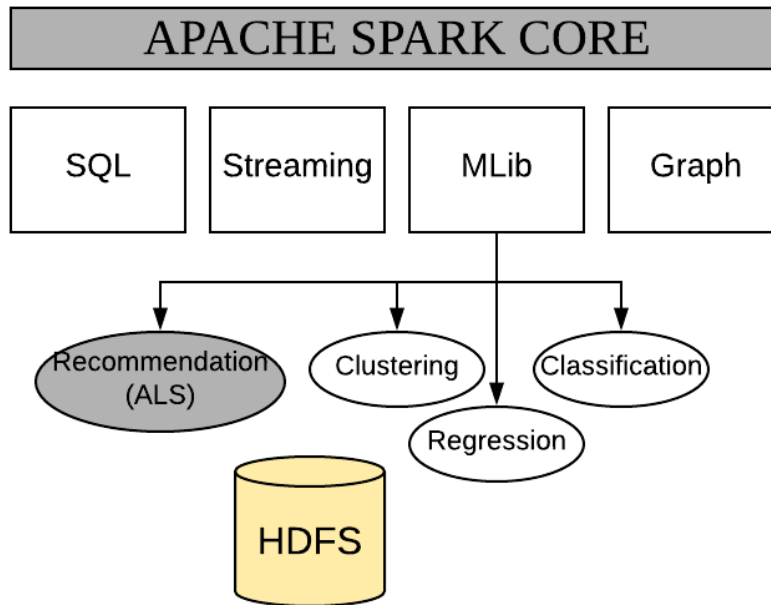


Figure 3.4: Apache Spark Components

```
export SPARK_HOME=/home/christina/
export PATH=$PATH:$SPARK_HOME/bin
```

Spark provides for interactive work with its shell, which implements a Read, Evaluate and Print Loop environment for the users' input, without interrupting the analysis of the data [95]. To start writing an application the developer must open a **Spark Session** and generate an RDD. RDDs are created either by using the **parallelize()** function to which data can be passed or by referencing a dataset from HDFS, for instance. In our case, considering the fact that we have already stored our dataset in HDFS, referencing to an external dataset in Java is as simple as the following line:

```
JavaRDD<String> distFile = sc.textFile("YahooTrainData.txt");
```

Spark does not require the dataset to be partitioned beforehand as the partitioning is done automatically although the number of desired partitions is configurable. The dataset may already be split into training and testing sets or this can also be achieved with the **randomSplit()** function in Spark. With the RDDs created, they are now ready to be used for parallel processing via 2 types of operations: **transformations** such as **map()** and **filter()** functions, for instance and **actions** implemented by methods like **reduce()** and **collect()**. The

next step is to create a model. We chose the ALS algorithm and by applying the `transform()` function we can provide predictions.

Spark is deployed in cluster mode using YARN as the master. By experience as we are dealing with large datasets we use the following configuration parameter.

`spark.executor.memory`

In our case, with 32 Gigabytes per node, we increase the memory to 28 Gigabytes. This prevents us from having Spark errors such as: Max number of executor failures (8) reached.

To solve another following common error due to the number of cores used:

```
Invalid resource request, requested virtual cores < 0,  
or requested virtual cores > max configured,  
requestedVirtualCores=12, maxVirtualCores=4
```

Because we are using 12 cores processors, we add the following property in `yarn-site.xml` located in the `etc` folder in Hadoop :

```
<property>  
  <name>yarn.nodemanager.resource.cpu-vcores</name>  
  <value>12</value>  
</property>
```

We then restart the YARN nodes if necessary.

3.5 Apache Giraph

Apache Giraph is a fast scalable and iterative graph processing open source framework, built on top of Apache Hadoop. Facebook is an active users and developer of Giraph. Facebook has used Apache Giraph to study graphs socially established by its users via their connections with each other [1]. The experiments have been done at the scale of trillions of edges [44]. Giraph takes vertices and edges as data input. The vertices represent the users and items while the edges are the ratings which have been obtained. Before deploying Giraph, Git and Maven 3 or higher must be installed; furthermore, Hadoop must be configured as explained in section 5.2. Then once Apache Giraph has been downloaded

or cloned it can be exported in the `bashrc` file with a command similar to the following:

```
export GIRAPH_HOME=/home/christina/  
export PATH=$PATH:$GIRAPH_HOME/bin
```

It is important to update Hadoop files such as `core-site.xml`, `mapred-site.xml` and `hdfs-site.xml` for better performance optimization. A step-by-step guide on a quick start with Giraph is available on Apache Giraph website [30]. The fact that Apache Giraph was written in Java makes this framework compatible with Hadoop.

Giraph follows the same concept as Pregel which was introduced by Google in 2010 [59]. Pregel enable graphs to be generated from very large datasets and uses the model Bulk Synchronous Parallel. Pregel has not been open sourced, hence the rationale for Apache Giraph. More details on Giraph and Pregel can also be found in [42]. By default, Apache Giraph runs in memory except for input, output and checkpointing when it computes on disk. Some of the few drawbacks that have been claimed are the adding up of a significant overhead due to the fact that Apache Giraph is slow to read/write from disk [29] and the complexity of changes required in Giraph in order to fully exploit its potentiality and obtain more efficient results [21].

The ALS algorithm is implemented on Giraph through Okapi. Okapi is an open source library which can be integrated to Giraph for machine learning and graph mining algorithms. The types of algorithms that are available include collaborative filtering, graph, clustering and Sybil-detection [82]. Sybil attacks can be defined as a malicious users with multiple identities aiming to gain control of a system [27]. The programming languages that can be used with Okapi are Java, Python and JavaScript. The Apache Giraph project itself does not currently have the ALS algorithm available; hence we have opted to use Okapi.

3.6 Conclusion

This chapter has provided a survey about four different platforms for recommendation systems that can be applied to large datasets. Hadoop is mostly known for its distributed file system (HDFS) and its ability to have other frameworks easily integrated into its ecosystem. Apache Mahout, Apache Spark and Apache Giraph have one point in common, they all use HDFS as the main storage for

datasets which is then processed differently according to the features of the framework. We have described the components of each platform, their configurations in the `bashrc` file and the method chosen to implement the collaborative filtering with ALS. The configuration of each of the frameworks mentioned in this chapter plays an important part as most frameworks have some default values already set to enable a quick and easier use. It is nevertheless up to the users to amend these default values to better fit their purposes. In chapter 5, we compare and analyze the different results obtained with these frameworks.

Chapter 4

Implementation of ALSWR with MPJ Express

4.1 Introduction

The implementation of ALSWR with MPJ Express is a crucial part of this thesis. First we explain how MPJ Express is integrated into Hadoop and allows jobs to be run in YARN nodes. We furthermore go through the ALSWR algorithm following the study made in [101] then adapt it to our model by interpreting and outlining each of the required stages. In principle, the data is first partitioned, then the program can read data from the partitioned datasets to implement the ALSWR method. We defer discussion of partitioning to the end of this chapter.

In section 4.2 we discuss the configuration and communication devices of MPJ Express before detailing step by step its integration to YARN. Section 4.3 provides an insight on how the ALSWR is applied to our matrices for prediction purposes. In section 4.4 we describe the implementation of the collaborative filtering technique with ALSWR while using the MPJ Express API as well as its features for parallel computation. We additionally briefly explain the importance of collective communication and indicate the ones that have been used on the program code to achieve parallel computation. The second part of section 4.4 focuses on the Intel Data Analytic Acceleration Library (DAAL) and more importantly on the Cholesky decomposition which is used to solve the symmetric positive definite matrix. Section 4.5.1 discusses about the original method followed to partition our various datasets. Section 4.5.2 describes the MapReduce version of the partitioning and outlines the main phases required for its implementation. In section 4.6 we introduce another alternative approach

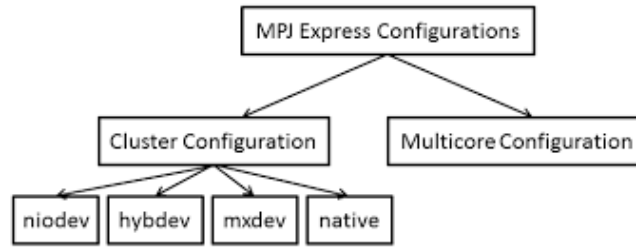


Figure 4.1: MPJ Express Configuration

to MapReduce: the Single Program, Multiple Data (SPMD) and give a background on the application platform interface of Collective Asynchronous Remote Invocation (CARI) and its usage.

4.2 MPJ Express

MPJ Express [60] is an open source Java MPI-like library that allows application developers to write and execute parallel applications on multicore processors and compute clusters. The MPJ Express software can be configured in two ways as depicted in Figure 4.1. The first configuration—called the multicore configuration—allows parallel Java processes to execute on shared memory systems including multicore processors. The second configuration—called cluster configuration—allows execution of parallel Java processes on distributed memory platforms including compute clusters.

Under the cluster configuration, the MPJ Express software provides various communication devices that are suitable for the underlying interconnect of the cluster. Currently, there are four communication devices available under the cluster configuration:

1. `niodev` - uses Java New I/O (NIO) Sockets
2. `mxdev` - uses Myrinet eXpress (MX) library for Myrinet networks
3. `hybdev` - for clusters of multicore processors
4. `native` - uses a native MPI library (like MPICH, MVAPICH, Open MPI)

Since 2015, the MPJ Express software has provided a YARN-based runtime that exploits the `niodev` communication device to execute parallel Java code on Hadoop clusters. Under this setting, HDFS is used as the distributed file system

where application datasets, MPJ Express libraries, and application programs are loaded to allow all processes to access the material. YARN execution is requested by using the `-yarn` switch as an option in the usual `mpjrun` command:

```
mpjrun.sh -yarn -np 2 -dev niodev MPJApp.jar
```

In this command `-yarn` instructs the program to operate with the YARN-based runtime and `-np 2` specifies the number of nodes for the parallel processing—just two nodes in this example.

Figure 4.2 presents the implementation of the MPJ Express library on YARN. In this setting, the Hadoop cluster consists of a client node, where Resource Manager (RM) executes, and two compute nodes, where a Node Manager (NM) executes. The NM process executes on each active node and is responsible for executing assigned tasks.

Figure 4.2 labels various stages of executing a parallel program with the YARN-based MPJ Express runtime.

1. `mpjrun` module starts the `MPJYarnClient`
2. the `MPJYarnClient` enable the execution of AM by requesting to the RM the allocation of a container
3. `MPJAppMaster` generates a CLC and a `mpj-yarn-wrapper.jar` for each container
4. The `mpj-yarn-wrapper` send outputs and error streams of the program to the `MPJYarnClient`

Assuming that the user wants to execute the `MPJApp` Java class. As a prerequisite, the user needs to bundle the `MPJApp` class in a Java Archive (JAR) file that might be placed in the local disk of the client node. Later, the user essentially launches the execution of the parallel program from this node by invoking the `MPJYarnClient` process. In order to pass the user’s application program to YARN, the `MPJYarnClient` makes a request to the RM to allocate a container and enable the execution of the Application Master (AM). The components of YARN are explained in section 3.2. The `MPJYarnClient` is also liable for initializing a Container Launch Context (CLC) and for submitting the number of nodes specified by the command (`-np 2`) as well as the JAR (`MPJApp.jar`) to the AM. The CLC holds all information that is required for a task to run, including

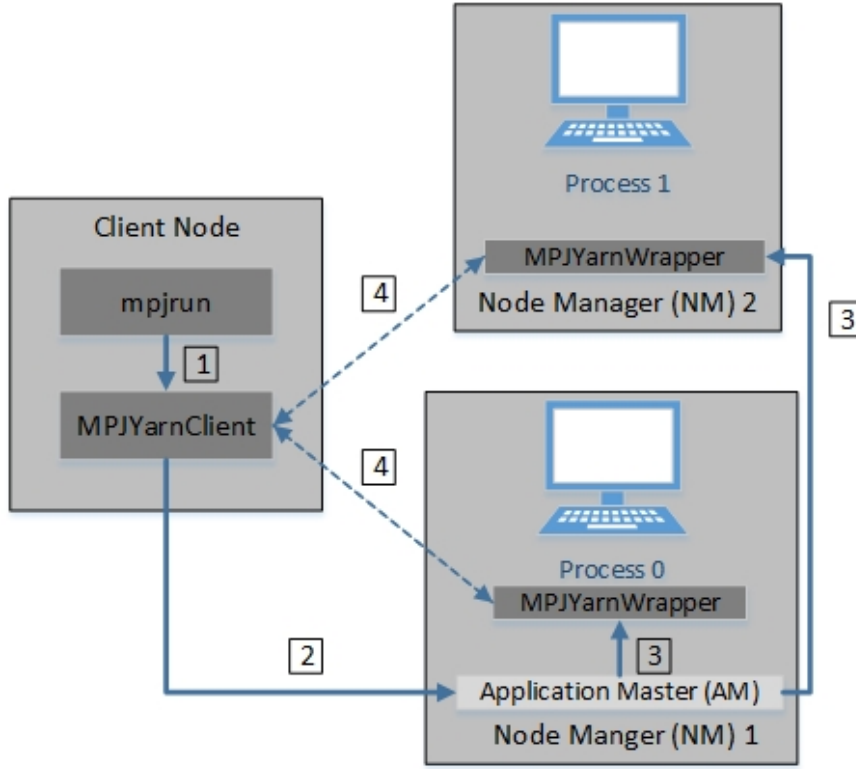


Figure 4.2: MPJ Express Integrated in YARN

the command arguments, environment variables, and authentication tokens. The AM contacts the RM to acknowledge the memory and available processing cores. This information is later used to decide the number of containers required for the execution of the parallel program. Further the allocation of all the containers to the NM, the AM generates a CLC holding the JAR (MPJApp.jar) and a mpj-yarn-wrapper.jar for each of these containers. The mpj-yarn-wrapper JAR file consists of a binary code for the mpj-yarn-wrapper module and is responsible for sending back output and error streams of the program to the MPJYarnClient.

4.3 Alternating Least Squares with Weighted Lambda Regularization

In this section, following Zhou et al. [101], we will often refer to items as “movies”; but of course identical math applies to any kind of item or service.

Assume we have n_u users and n_m movies, and R is the $n_u \times n_m$ matrix of input ratings. Usually, each user can rate only a few movies. Therefore the matrix R will initially have many missing values or loosely speaking it will be sparse. The

problem is to predict the unknown elements of R from the known elements. We model the preferences of users by assuming they have a simple numeric level of preference for each of a number n_f of features to be found in movies; thus the behaviour of user i is modelled by a vector \mathbf{u}_i of length n_f . The features are sometimes called Latent features, because they are implicitly constructed by the algorithm. Similarly, each movie is assumed to have each these features to a simple numeric degree so each movie j is modelled by a vector \mathbf{m}_j of the same size. The predicted preference of user i for movie j is the dot product $\mathbf{u}_i \cdot \mathbf{m}_j$. The vectors are conveniently collected together in matrices U and M of size $n_u \times n_f$ and $n_m \times n_f$ respectively. To fit the model to the known elements of R we use the least squares approach, adding a regularization term parameter λ to the sum of square deviations to prevent the model from overfitting the data. The penalty function we strive to minimize is:

$$f(U, M) = \sum_{i,j} (r_{ij} - \mathbf{u}_i \cdot \mathbf{m}_j)^2 + \lambda \left(\sum_i n_{u_i} \mathbf{u}_i^2 + \sum_j n_{m_j} \mathbf{m}_j^2 \right) \quad (4.1)$$

where the first sum goes over i, j values where the element r_{ij} of R is known in advance, n_{u_i} is the number of items rated by a user i , and n_{m_j} is the number of users who have rated a given movie j .

ALSWR is an iterative algorithm. It shifts between fixing two different matrices. While one is fixed, the other one is updated and then roles switch, eventually solving the matrix factorization problem. The same process goes through a certain number of iterations until a convergence is reached which implies that there is little or no more change on either users and movies matrices. The ALSWR algorithm as explained by Zhou et al [101] is as follows:

Step 1: Initialize matrix M by assigning the average rating movies as the first row, and small random numbers for the remaining entries.

Step 2: Fix M , Solve U by minimizing the objective function (the sum of squared errors);

Step 3: Fix U , Solve M by minimizing the objective function similarly;

Step 4: Repeat Steps 2 and 3 until a stopping criterion is satisfied.

Step 2 is implemented by Equation 4.2 where M_{I_i} is the sub matrix of M , representing the selection of any column j in the set of movies rated by a user i , H

is a unit matrix of rank equal to n_f and $R(i, I_i)$ is the row vector where columns j are chosen

$$\mathbf{u}_i = (M_{I_i} M_{I_i}^T + \lambda n_{u_i} H)^{-1} M_{I_i} R^T(i, I_i) \quad (4.2)$$

Similarly, Step 3 is implemented by Equation 4.3. U_{I_j} is the sub matrix of U , representing the selection of any column i in the set of users who have rated a certain movie j , and R is the vector of known ratings with $R(I_j, j)$ the column vector where rows i are chosen.

$$\mathbf{m}_j = (U_{I_j} U_{I_j}^T + \lambda n_{m_j} H)^{-1} U_{I_j} R^T(I_j, j) \quad (4.3)$$

4.4 Implementation of ALSWR

In this section, we describe our collaborative filtering implementation of ALSWR using the MPJ Express API. HDFS which has been introduced in section 3.2 is the framework adopted to store our dataset.

The basic strategy for distributing the ALSWR algorithm to run in parallel was already described by the original proposers in [101]. All nodes of a cluster contain a certain subset of the large, sparse, recommendations array, R . R is characterized as sparse because in general users rate only a few items. In particular, it is convenient for the R array to be duplicated across the cluster as a whole—divided across nodes both by columns and also by rows (or, equivalently, decomposed two ways, both by users and by items). This is illustrated in figure 4.3, where i is the subscript identifying users and j is the subscript identifying items, and the two different forms of decomposition of R are used in the two different steps. Step 2, as defined in equation 4.2, conveniently uses locally held R decomposed by i to update locally owned elements u_i of the user model. B is a block size for the locally held subset of elements, approximately constant across the cluster for good load balancing.

Because update of u_i potentially involves *any* element of the item model \mathbf{m} , to simplify this step all elements of \mathbf{m} should be stored locally, in globally replicated fashion.

Step 3 has a complementary structure but now update of m_j may require access to any element of \mathbf{u} . So between steps 2 and 3 all the locally computed elements of \mathbf{u} must be gathered together and broadcast to processing nodes. Similarly, between step 3 and step 2 in the *next* iteration of the algorithm, the locally computed elements of \mathbf{m} must be gathered and broadcast.

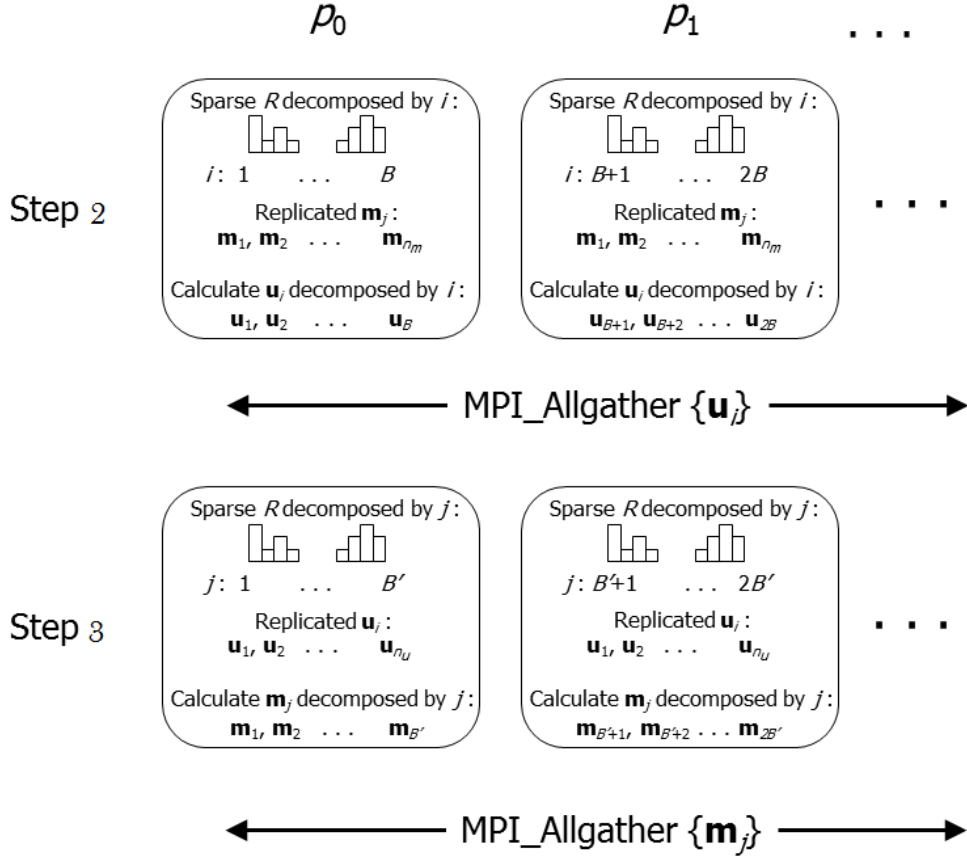


Figure 4.3: Visualization of an iteration of distributed ALSWR algorithm

Due to the fact that we want to implement parallel computing, the nodes of the cluster need to be able to communicate with each other. This is called collective communication. The first point of communication is established with the *world* communicator: `MPI.COMM_WORLD`. With `COMM_WORLD` being a constant of type `Comm` in the Message Passing Interface (MPI) class of MPJ Express. One of the principles of MPI is to enable all processes to work together with the aim to shift data between the memories of the processors. Another important point of communication used in our implementation is `MPI_AllGather`. This last enables many elements to be sent to many processes by gathering all these elements to all processes. An illustration of `MPI_AllGather` is provided in figure 4.3. “Processor space” runs across the page, processes are labelled p_0, p_1, \dots and so on. Time runs down the pages with distributed computational steps labelled as on page 40. Between computational stages there are collective synchronizations in the form of “allgather” operations.

Finally, we use `MPI_Allreduce` to sum over all processes then distribute the results to all of these processes. Collective communication also implies synchro-

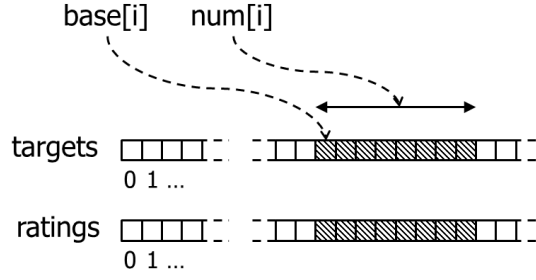


Figure 4.4: Sparse data structure to represent locally held ratings

nization between processes to enable them to wait for each other at a certain point before continuing to the next stage of the program. Synchronization can also be realized through the function: `MPI_Barrier`. This function thus initiates a barrier preventing processes to go beyond that barrier unless they have all send a signal through the function. A major application of this barrier is to synchronise processes before timing in benchmarking the program.

In our program, the data that we used for the implementation of the ALSWR code consists of a sparse matrix of ratings, partitioned by users or by items. Figure 4.4 illustrates the organization of the data where `base[i]` represents the start of entries in big arrays per entity, `num[i]` the number of entries in big arrays per entity, `ratings` are the actual notes given and `targets` the global ID of target of rating. Assuming that the decomposition is by users, then the target is an item otherwise it is a user. For i 'th local entity, X_s represent the corresponding values in “big arrays”. This whole structure is duplicated, once for ratings distributed by user and once for ratings distributed by items. In the “by user” case the size of the `base` and `num` arrays is the total number of locally held users, with `num[i]` being the number of ratings by user i ; `targets` elements hold a *global* index of the rated item (index in the gathered array of item models). In the “by item” case the size of the top arrays is the number of locally held items, with `num` holding the number of ratings per item; a `target` element now holds the global index of the user who *made* the rating.

In order to solve the symmetric positive definite matrix, we use Cholesky decomposition from The Intel Data Analytic Acceleration Library (DAAL). DAAL is an open source library that we use to decompose our matrix into a product of a lower triangular matrix and its conjugate transpose [48]. There are three main steps followed to reuse the structure of DAAL: we first create an algorithm of type Batch from DAAL libraries then set an object as input for the algorithm and compute Cholesky decomposition. Therefore we attempt to solve the linear

equation of type $A\mathbf{x} = \mathbf{b}$, where A represents our full matrix, \mathbf{x} a vector that will contain the result of the equation on exit and \mathbf{b} a vector containing the right side (RHS) of the equation. Cholesky decomposition is computed with the global variable *decomp* which is a $n \times n$ element array that will contain the lower triangular Cholesky factor of A on exit. The following paragraph describes the implementation of the update model

The updated model takes four arguments which are the local model, the local number, the ratings and the complement global model. We implement the update model by using the lower triangular matrix—see figure 4.5. Arrays `base`, `num`, `targets`, `ratings` here are for ratings distributed by user. The arrays `uLoc`, `mGlob` correspond to the distributed version of \mathbf{u} and the replicated version of \mathbf{m} in step 1 of Figure 4.3. In practice, this code is parameterized so it can implement either step 1 or step 2. Various two dimensional arrays with first dimension n_f are “flattened” to one dimensional Java arrays for performance reasons. The code assumes each node holds `numLocal` elements of the distributed user model. Within a node, we run `NUM_THREADS` long-lived threads (they are started at the beginning of the program), where the `NUM_THREADS` parameter will usually be related to the number of cores on the node. The variable `me` identifies a thread within the local node (not to be confused with the MPI *rank* which identifies a node). Threads will be synchronized before MPI collective operations using barriers implemented by `java.util.concurrent.CyclicBarrier`. The MPI operations themselves are only executed by the `me = 0` thread.

In the final version of the code, the rating data for the MPJ code are read from the same HDFS text files as used by the third party implementations of ALS discussed in chapter 3. We use HDFS API to determine the blocks that have replicas on nodes running MPJ processes. A heuristic is used to choose a load balanced set of local replicas to read as described in chapter 6. The locally read ratings are then partitioned to destination nodes using a variant of the CARI communication schedules introduced in [93]. Section 4.5 is an early strategy that was adopted.

4.5 Partitioning of Dataset

The partitioning of a dataset is the process of dividing the data into smaller chunks with each partition holding a part of the dataset that has been split. We partition our datasets for better efficiency of the program as it minimizes the

```

1  // ptr: temporary holding pointer into targets and ratings arrays
2  // u: temporary - vector of length n_f.
3  // A, V: global variables - a matrix of rank n_f and vector of length n_f, both
4  //                                     mapped permanently to DAAL HomogenNumericTable objects.
5  for(int i = me; i < numlocal ; i += NUM_THREADS) {
6      ... initialize A and V to zeroes ...
7      for(int j = 0; j < n_f; j++) {
8          A[n_f * j + j] = LAMBDA * num[i] ;
9      }
10     for(int j = 0; j < num[i]; j++) {
11         ptr = base[i] + j ;
12         target = targets[ptr] ;
13         rating = ratings[ptr] ;
14         for(int k = 0; k < n_f; k++) {
15             for(int l = 0; l <= k) { // lower triangle
16                 A[n_f * k + l] += mGlob[n_f * target + k] * mGlob[n_f * target + l] ;
17             }
18             V[k] += rating * mGlob[n_f * target + k] ;
19         }
20     }
21     solveSymPosDefSystem(u, n_f) ; // use DAAL to solve A u = V
22     for(int j = 0; j < n_f; j++) {
23         uLoc[n_f * i + j] = u [j] ;
24     }
25 }

```

Figure 4.5: Outline Java code for step 1 of the ALSWR update (Equation 4.2) using sparse data structures of Figure 4.4

network traffic between the nodes and enables the program to run in parallel. Having an even distribution of the data across the nodes facilitate the overall computation on the cluster more particularly when dealing with large datasets. The number of partitions is chosen according to the number of nodes available within the cluster. For better optimization, we partition the datasets up to a maximum number equal to the one of the nodes that will be used for the experiments.

The strategies discussed in sections 4.5.1 and 4.5.2 requires data to be partitioned into files before starting the main program. For ALSWR, we partition each dataset two ways—by users and items. This was originally achieved through a serial code whereby the data was accessed from the local file system. The second approach adapted consisted of using MapReduce to partition the data, hence having the datasets stored into HDFS and enabling these datasets to be read from the Hadoop file system. We then measure and compare both approaches in term of the total duration taken to complete a job. The final approach in section 4.6 partitions data during the execution of the program.

4.5.1 Partitioning of Ratings Dataset using a Serial Java Application

As discussed in Section 4 the training data set of ratings is a sparse structure that is stored in memory twice—once partitioned by users and once by recommended items (equivalently it is a sparse matrix that is distributed both by rows and columns).

The partitioning of the ratings by users will induce the partitioned version of the user model, and similarly for the item model (though both these are also stored at various stages of the algorithm in a fully replicated manner). In practice we use a simple partitioning scheme based on a naive hash of the user id or respectively item id—these are typically numerical identifiers stored in the original rating files. This usually gives a sufficiently even partitioning. It is worth noting on passing that the global index in the replicated form of e.g. the user model is that implied by simply concatenating together the segments in the partitioned form of the model because this is what `MPI_AllGather` provides. This global index is what is stored in the `targets` arrays of the internal sparse representation of ratings partitioned by items (Figure 4.4), for example (it is *not* the original numeric identifier in the source data set, and it is necessary to maintain translation tables to convert back to the original identifier when models

or rankings are output at the end of the optimization algorithm.)

In our MPJ Express implementation of ALSWR, an early decision was made to concentrate on the efficiency of the main iteration loop of the algorithm. A corollary of that was the business of *partitioning* would be delegated to a separate Java program that read the data set input file(s) and wrote them to a set of partitioned files— $2P$ in all, because of the two types of partitioning. The parallel ALSWR code would just read these pre-partitioned files, 2 per node, and load the data to its sparse data structures before starting the algorithm (initially all files were stored in standard file systems).

For a production collaborative filtering code this would not necessarily be the preferred strategy, but we were initially focused on the business of efficiently parallelizing the main loop (keeping the partitioning logic out of the main computational code may also be expected to reduce the memory footprint of that code, thus allowing it to handle larger data sets).

Fairly soon it became clear that the overheads of partitioning the data were quite significant compared with the main computational code, once the latter was parallelized. Representative timings for different numbers of nodes are given in Table 4.1a.

4.5.2 Partitioning of Ratings Dataset with MapReduce

We positively interpreted the drawback explained at the end of section 4.5.1 as an opportunity to demonstrate the value of supporting multiple computational frameworks (in our case MapReduce and MPJ Express) on the same Hadoop cluster in an extended processing pipeline as illustrated in figure 4.6. So we wrote a MapReduce version of the partitioning code with input and output files now stored in HDFS. The main ALSWR program now had to be slightly rewritten to take input from HDFS files rather than the local file system. The MapReduce jobs have been implemented through several classes. Because we want each dataset to be partitioned by users and by items, we have defined two mapper classes one for the users and the other for the item; a partitioner class, a reduce class and a driver class. We also aim to prevent poor partitioning by creating custom partitioner classes for both users and items. Poor partitioning is a common issue observed when keys data are unevenly distributed resulting in some reducers having more work than others and by consequence slows down the whole process [91]. Below we describe the role of each class involved in the partitioning process. More details on each class are also presented in the class diagram in

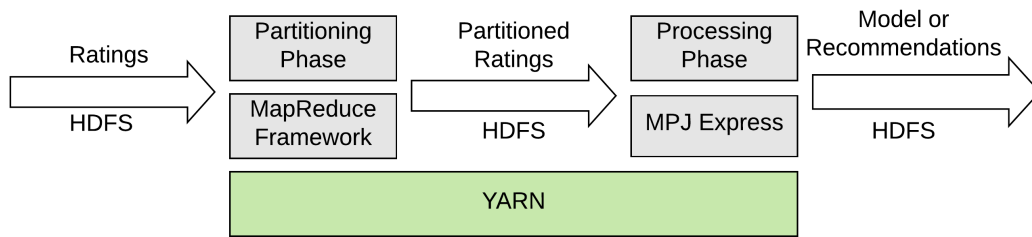


Figure 4.6: MapReduce/MPJ Express Processing Pipeline, mediated by HDFS

figure 4.7.

- The driver class is the main component; it is in this class that the URI of the cluster is defined and all other classes implemented. The role of the driver class is to do all the configuration then to submit the MapReduce job.
- The mapper classes are the first step of the partitioning process. The classes read the data and generate key/value pairs. Their outputs will then be used in the reduce phase.
- The partitioner class manages the partition between the map phase and the reduce phase. It is at this stage that the number of partition is specified. The main job of the partitioner class is to allocate the output of the mapper to the reducer.
- The reduce class takes the outputs from the mappers; it shuffles, sorts and reduces the key/value pairs. The final result of the reducer is then stored in the hdfs ready to be used by our ALSWR program.
- The custom classes enable the results from the mappers to be sorted in different reducers grouped by the key. They enable a better distribution of the data to the reducers.

Before partitioning the data, the dataset must be added to HDFS. This can be done with the next command assuming we wish to add the Yahoo Training dataset to the user christina in HDFS.

```
$ hdfs dfs -put /home/christina/YahooTrainData/* /user/christina
```

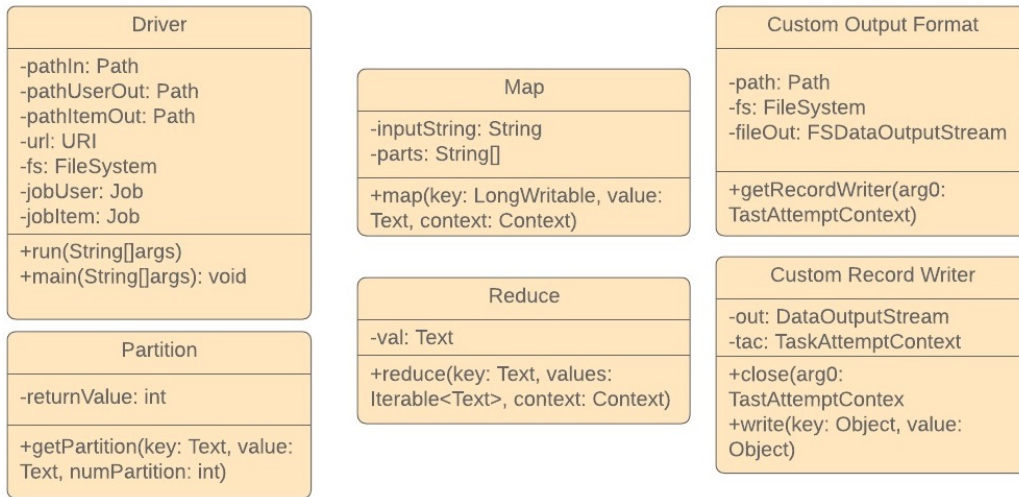


Figure 4.7: Map Reduce Partitioning Class Diagram

The following command illustrates how a YARN job is launched to partition the Webscope dataset (yahoo music data) assuming we want to partition the data in 2 by user and by song. The java code PartitionYahooData.java must have been built into a jar before it can be used.

```
$ yarn jar PartitionYahooData.jar DriverClass 2 YahooTrainData
2YahooPartitionedTrainbyUser 2YahooPartitionedTrainbySong
```

Some errors linked to the Java Heap Space could occur particularly when partitioning a very large dataset. A typical MapReduce Java Heap Space is caused by the value of the percentage of memory designated for the maximum heap size. The default value is 0.9 This value is used for caching values when using the mark-reset functionality [41]. To solve MapReduce Java Heap Space problem we change the value of the mapreduce.reduce.markreset.buffer.percent in the mapred-site.xml and set the value of the attribute to 0.5 as shown in the following property definition:

```
<property>
  <name>mapreduce.reduce.shuffle.input.buffer.percent</name>
  <value>0.5</value>
</property>
```

The much improved performance of partitioning is illustrated in Table 4.1b. For instance we now obtain 37 minutes for the Yahoo training dataset compared to 5 hours with the previous approach. These results were obtained from experiments implemented on a cluster containing 3 nodes. We have run again the same partitioning code with MapReduce but this time on a bigger cluster

made of 12 nodes. We additionally partition our synthetic dataset. The results are displayed in table 4.2. As we increase the number of nodes the the time performance improves until we reach a certain number of processes where there is either no significant improvement or the performance is becoming unstable which means the results obtain are unpredictable and could be better or worse. We interpret this aspect as a communication overhead within the cluster. For the Yahoo testing set data this issue occurs from 8 nodes and with the Yahoo training set from 7 nodes.

Table 4.1: Time Performance in Seconds of Datasets Partitioning

| (a) Serial Java Program | | | | (b) MapReduce | | | |
|-------------------------|-----------|-----------------------|---------------------|---------------|-----------|-----------------------|---------------------|
| # of Procs | MovieLens | Yahoo Testing Dataset | Yahoo Train Dataset | # of Procs | MovieLens | Yahoo Testing Dataset | Yahoo Train Dataset |
| 1 | 910 | 1023 | 7215 | 1 | 121 | 126 | 3771 |
| 2 | 1535 | 1640 | 10859 | 2 | 95 | 97 | 2328 |
| 3 | 2780 | 2957 | 18562 | 3 | 80 | 84 | 2241 |

| # of Nodes | Yahoo Testing Dataset | Yahoo Training Dataset | SyntheD |
|------------|-----------------------|------------------------|---------|
| 1 | 110 | 2979 | 86451 |
| 2 | 88 | 1659 | 44547 |
| 3 | 74 | 1428 | 40954 |
| 4 | 66 | 1061 | 26271 |
| 5 | 65 | 996 | 43539 |
| 6 | 62 | 729 | 28817 |
| 7 | 58 | 908 | 21891 |
| 8 | 61 | 500 | 18512 |
| 9 | 58 | 813 | 19447 |
| 10 | 59 | 540 | 12233 |
| 11 | 56 | 475 | 11243 |
| 12 | 57 | 500 | 8657 |

Table 4.2: MapReduce Partitioning on 12 Nodes Cluster in Seconds

4.6 Partitioning Data within the SPMD Program

The last two sections discussed approaches to partitioning the data before the parallel MPJ program implementing the ALSWR processing algorithm starts.

The partitions are stored to separate HDFS files that the MPJ program then reads.

This approach has some benefits of simplicity, but it can be restrictive and awkward in practice. It is more convenient and, it turns out, more efficient in terms of overall execution time, if the MPJ program reads the original *unpartitioned* ratings files—probably also stored in an HDFS file or an HDFS directory—and does the partitioning of ratings to the appropriate compute node as a phase in the execution of the parallel program.

One advantage that MapReduce has in reading data from the HDFS distributed file system is that it can “move computation to where the data is”. HDFS files are divided into large blocks stored on different data nodes in the Hadoop cluster. Each block may contain a large number of records to be processed by map tasks. Map tasks instantiated on the data nodes that contain the records (the “split”) those tasks will work on. So they only have to read from the *local* file system—they largely avoid *non-local* reads, which are relatively expensive in HDFS.

In Chapter 6 we will discuss a general approach to allocating HDFS block reads to the nodes of a SPMD program such as an MPJ program that affords a similar advantage. We use that approach in the final version of our MPJ ALSWR program. After this phase of reading locally stored ratings records we end up with in-memory ratings that are not necessary in the memories of the nodes that will ultimately process them (they effectively need bucket sorting through the hashing approach mentioned earlier in this chapter, so that in a by-users partition, for example, all ratings made by a particular user are stored on the same compute node).

The required redistribution of data could be done in various ways in an MPI-like implementation, but for simplicity we adopt the Collective Asynchronous Remote Invocation (CARI) approach introduced in [93]. CARI is a class of API that in general supports a kind of remote method invocation in a SPMD program. CARI is designed so that it can efficiently support very large numbers of lightweight invocations, using a bounded amount of buffering for the communications implied.

4.6.1 Background on CARI APIs

A basic pattern for asynchronous remote method invocation in *general distributed programming* is likely to have an implementation class for the body of the remote

method. This class resides on the server side and accesses data there. For illustration, we can simplify it to something like:

```
class myImplementation extends ...base class in the run time... {
    T handleRequest(S arg) {
        T result ;
        ... calculate result in terms of arg and server data ..
        return result
    }
}
```

where `handleRequest` is the method to be invoked remotely, the argument information is embodied in a record of type `S` and the result in a record of type `T`. Asynchronous invocation of the method `handleRequest` from a client could look something like:

```
remoteStub.invoke(actual, resultHandler)
```

where `remoteStub` is some client-side proxy for the server-side remote object, `actual` is an actual argument of type `S`, and `resultHandler` is an object whose class defines the asynchronous handling of the *returned result* (it is a “callback”). The class of this object could be like:

```
class myHandler extends ...another base class in the run time... {
    void handleResponse(T result) {
        ... do something with result ...
    }
}
```

The invocation is *asynchronous* because the `invoke` method returns immediately, but in general the `handleResponse` method will be called later on the client when the result comes back from the server.

CARI APIs adapt this paradigm from distributed programming to run in the context of massively parallel SPMD programs. A possible API, adapted from [93], is given in figure 4.8. Following the original paper, this API is given in C++. C++ has the benefit of powerful template classes, allowing type parameters like *S* and *T* to be primitive types or flat struct types; Java *generic classes* would require these types to be object types, which is not ideal when small entities need to be moved across the network.

The SPMD programmer now extends the single base class `CARIHandler` to define both the method `handleRequest` (called on a “server peer”—all peers generally act as servers as well as clients) and the method `handleResponse` called

```

template <class S, class T>
class CARISchedule {
public:

    CARISchedule(MPI_Comm comm,
                  CARISchedule<S, T> *hdlr,
                  int buffSize, void *buffer);

    void invoke(S *request, int tag, int dest);
    void complete(void);
};

template <class S, class T>
class CARIScheduleHandler{
public:
    virtual void handleRequest(S *request,
                               T *response);

    virtual void handleResponse(T *response,
                                int tag);
};

```

Figure 4.8: Schematic C++ API for CARI

on the invoking peer when the result returns. The `invoke` method called on the “client peer” is adapted to define the target “server” node by its SPMD rank `dest` and also passes in a user defined tag value, as well as the actual argument called `request` here. (The tag value only has local significance on the client—it is passed to the `handleResponse` method when the result asynchronously arrives, and can be use to identify the invocation this response corresponds to).

We won’t give a detailed application here, but in general the main program of an MPI program using CARI may have sections of code like this:

```

CARISchedule<int,int> cari(MPI_COMM_WORLD, myHandler,
                          buffSize, buffer) ;

while(... more to do locally...) {
    ...
    cari.invoke(arg, i, dest) ;
    ...
}

cari.complete() ;

```

In this example the argument and result types are just `int`. At the illustrated point in the main loop the local node remotely invokes the `handleRequest` method on peer node with rank `dest`. It passes in a tag `i`. It is up to the

user how this tag is used in the eventual `handleResponse` method, but a typical example would be where `i` is an index into a local array where the result it to be stored or accumulated. The `complete` method must be called after this phase of local processing is finished, to ensure all outstanding communications and `handleResponse` calls (etc) are completed. After this collective call is completed, we can assume all asynchronous remote invocations are complete across the cluster.

Of course the definition of the `handleRequest` and `handleResponse` methods on `myHandler` are completely under control of the programmer.

A possible implementation will partition the provided buffer, and accumulate arguments of locally made invocations into part of this buffer, until space is exhausted. Then it may sort the buffer by destination and perform a collective all-to-all communication to send arguments to servers. The relevant `handleRequest` calls are made then another collective all-to-all sends results back to clients where `handleResponse` calls are made. Then processing of the main loops continues.

4.6.2 Use of CARI in the present work

The earlier examples were given in variants of C++. Throughout this dissertation we use Java. Java has some limitations on its generic classes and has a different philosophy of memory management, so the Java interfaces implementation will differ in detail.

For the specific partitioning application considered here we don't need the full power of request and response handlers. "Void return" remote invocations are sufficient—in other words we don't need to return results of invocations. Also Java doesn't have struct types that can efficiently pack multiple scalar types in a single argument.

Figure 4.9 lists the finite set of (non-polymorphic) CARI classes we actually implemented.

Here `CARIScheduleInt` is a base class for all CARI communication schedules that invoke void methods with some number of `int` arguments (and thus use communication buffers that are arrays of integers). The only publically used method implemented in this class is the `complete` method.

Then we have two derived classes that are used directly by the programmer. `CARIScheduleI` supports remote invocations that only take a single integer argument and `CARIScheduleIII` supports remote invocations that take three integer arguments. The programmer extends `CARIHandlerI` or `CARIHandlerIII` respec-

```

class CARIScheduleInt {

    public void complete() { ... }
}

class CARIScheduleI extends CARIScheduleInt {

    CARIScheduleI(Intracomm comm, CARIScheduleI handler,
        int buffSize) {...}

    void invoke(int dest, int i1) {...}
}

abstract class CARIScheduleI implements ... {

    abstract void handleRequest(int i1) ;
}

class CARIScheduleIII extends CARIScheduleInt {

    CARIScheduleIII(Intracomm comm, CARIScheduleIII handler,
        int buffSize) {...}

    void invoke(int dest, int i1, int i2, int i3) {...}
}

abstract class CARIScheduleIII implements ... {

    abstract void handleRequest(int i1, int i2, int i3) ;
}

```

Figure 4.9: Actual Java API for CARI subset (various class members used only as part of implementation are omitted)

```

CARIScheduleIII schedule =
    new CARIScheduleIII(MPI.COMM_WORLD, this, 1000000) ;

for(int i ; i < readRatings.size() ; i++) {

    int extID = readIDs.get(i) ;
    int targetExtID = readTargets.get(i) ;
    int rating = readRatings.get(i) ;

    schedule.invoke(extID % p, extID, targetExtID, rating) ;
}
schedule.complete() ;

[...]

void handleRequest(int extID, int targetExtID, int rating) {

    int locID = layout.ext2locIDs.get(extID) ;

    int ptr = base [locID] + (num [locID] ++ ) ;
    targets [ptr] = ext2glbIDs.get(targetExtID) ;
    ratings [ptr] = (short) rating ;
}

```

Figure 4.10: Partitioning code using CARI

tively to define the implementation of these methods on the “server side”, and passes the resulting handler classes to the constructors of the CARI schedule classes (for now only these two pairs of classes were implemented because those were all we needed in our ALSWR program; most of the detailed implementation is in the `CARIScheduleInt` base class and adding new subclasses just involves adding a few short “glue” methods).

No tag is needed in the `invoke` method, because the remote invocations return no results.

Figure 4.10 shows part of the actual code using these CARI classes for partitioning the ratings data. The main loop in this code iterates over three corresponding array lists `readIDs`, `readTargets` and `readRatings` that contain the three fields of the locally read data records. The remote invocation is to the destination processor with rank `extID % p` where `p` is the number of nodes the program is running in. This is the hashing function that determines where a given rating will be stored.

On the remote node, the method `handleRequest` is called. The various arrays `base`, `num`, `targets` and `ratings` were introduced in figure 4.4. The hash tables `ext2locIDs` and `ext2glbIDs` map external IDs stored in files to different kinds of index internal to the program.

| # of Nodes | Yahoo Training Dataset |
|------------|------------------------|
| 1 | 136 |
| 2 | 161 |
| 3 | 139 |
| 4 | 131 |
| 5 | 111 |
| 6 | 99 |
| 7 | 87 |
| 8 | 85 |
| 9 | 82 |
| 10 | 79 |
| 11 | 70 |
| 12 | 70 |

Table 4.3: CARI Partitioning on 12 Nodes Cluster in Seconds

Before this phase of partitioning is run, one previous phase has run that counts the number of ratings that will be stored for a given user (or movie, in the by-movie storage), thus defining the arrays `base` and also `ext2locIDs`, and also computing sizes of `targets` and `ratings`. It is during this earlier phase that our other CARI schedule `CARIScheduleI` is used.

Table 4.3 shows timing results for partitioning the Yahoo Training Set using this CARI-based scheme within the main MPJ program for ALSWR. Data is first read from a single large HDFS file containing the whole training set, using the distributed strategy for reading data from HDFS that will be described in chapter 6.

Comparing table 4.3 with the middle column of table 4.2, the dramatic improvement of this strategy over the MapReduce approach (which itself was a major improvement over using a serial Java program) is clear.

4.7 Conclusion

MPJ Express, an open source Java MPI-like library is the main focus of this research and can be configured on multicore devices or clusters. Since its integration with YARN in 2015, MPJ Express can be executed on YARN nodes while using HDFS as a data storage. We use this potentiality to our advantage and implement a collaborative technique with ALSWR opting for a cluster con-

figuration. In this chapter, we have analyzed the process that follows a typical `mpjrun` command and dissected the ALSWR algorithm by stating the equations that are involved and interpreting the way they are represented in our program.

We have also explained the terminology of dataset partitioning and its importance in parallel computing. It is to be noted that for better performance it is recommended to have a total number of partitions less or equal to the total number of nodes available in a cluster. Our datasets have been partitioned following three different approaches: a Java-based program, partitioning with MapReduce and with the SPMD program. The first approach was limited by its time performance. Therefore with an attempt to solve this drawback MapReduce partitioning was initiated and as a result, we were able to complete partitioning jobs in a significantly better time frame. Nevertheless, there is still room for improvement and more particularly to adapt the program to much larger datasets such as SyntheD where the distribution in 12 processes, for instance, takes over two hours. To improve data loading time and partition the data in the same code implementing ALSWR, we have introduced another approach called SPMD which is further detailed in chapter 6.

Chapter 5

Performance Comparison across Hadoop-based Frameworks

5.1 Introduction

In this chapter, we evaluate performance of our Hadoop MPJ-based collaborative filtering approach and compare it with that of other platforms identified in chapter 3.

We break down the steps followed for the experiments, starting from the environment set up to the analysis of the test outcomes. We run the ALSWR Java code on Hadoop while using the version of MPJ Express which is integrated with YARN; then compare the results obtained with 3 different Hadoop-based frameworks: Apache Mahout, Apache Spark and Apache Giraph. We have described these frameworks in chapter 3. For the experimentation we use datasets that have been partitioned as explained in sections 4.5 or 4.6, consisting of Movielens and Webscope (results for Synthed, our synthetic dataset, are deferred to chapter 7). The performance of the model built is measured on two grounds. Firstly, we measure the time taken to compute the ALSWR code as well as its parallel speedup across the Number of Threads available. Thereafter we measure the accuracy of the recommendation by calculating the RMSE. Evaluation metrics are reviewed in section 2.7.

The rest of the chapter is organized as follow. In section 5.2 we provide details on the configurations required for the experimentation; notably variables in the bashrc file and the preparation of the nodes. We assess the results of the test on data from MovieLens and Webscope respectively in sections 5.3 and 5.4. In section 5.5 we report the outcome of the RMSE calculation followed by an

overall analysis of the comparisons of all the frameworks that have been used on this research in section 5.6.

5.2 Environment Set Up

Implementing the right environment in the cluster is an essential stage to our experimentation as it determines the way resources will be used. It is at this stage that software and frameworks are installed, classpath is defined and libraries required for the program declared.

5.2.1 Prerequisites

Before configuring Hadoop and YARN nodes, Java and Secure Shell (ssh) must be installed on all the nodes of the cluster and Hadoop software must be downloaded. This can be done through Apache mirrors. To install Java on Linux, the software is available for download from Java website [70], preferably the latest version available. A file of type tar.gz should be available. Once the file is downloaded and placed in the desired directory it has to be unpacked then the user should install the Java JDK and the devel. Steps 1, 2 and 3 below respectively illustrate the type of commands that can be input in the terminal to get the Java program ready to run.

1. `tar zxvf name-of-the-file.tar.gz`
2. `yum install java-name.of.version-openjdk`
3. `yum install java-name.of.version-openjdk-devel`

To install ssh on Linux the following commands must be typed on a terminal.

1. `sudo pacman -S openssh`
2. `sudo systemctl enable sshd`
3. `sudo systemctl start sshd`

However, sometimes ssh could be already installed and in that case, only commands 2 and 3 are required to enable the server.

5.2.2 Setting Up the Nodes

The configuration of Hadoop must be done on all the nodes of the cluster. In principle depending on the size of the cluster two machines can be chosen to be the master nodes, these will act as NameNode and others will represent slaves nodes and perform as DataNode. These slave nodes can be registered into the slaves' file available in the configuration folder of Hadoop, therefore indicating the nodes within the cluster that should be used to execute the jobs. Refer to figure 3.1 for details on Hadoop architecture. Configuring Hadoop mainly consist of organizing the layout of its files such as `core-site.xml`, `hdfs-site.xml` and `mapred-site.xml` for instance as well as the environments of Hadoop daemons (NameNode, Datanode and YARN nodes). We briefly describe the role of the main files located in the Hadoop folder which must be amended prior to running any job on the cluster.

- The `core-site.xml` file holds the configuration of the runtime environments of Hadoop. Its role is to keep the Hadoop daemons informed on where the namenode executes. It is on this file that we state which node will represent the namenode and its port number.
- The `hdfs-site.xml` file holds the configuration of the namenode, second namenode and datanodes. We also define on this file the number of block replication. Block replication is enabled to allow fault tolerance within the Hadoop cluster. In case of a datanode failure, the same data is available in another node. The default number for block replication is 3. Having a much higher number may come at some cost to the cluster performance as more copies will need to be saved when blocks are written.
- The `mapred-site.xml` file contains MapReduce configurations. On this file, we determine the name of the framework for MapReduce. As we are using the latest version of Hadoop which includes its resource manager YARN, we set the value of the framework to YARN. By setting the execution framework to YARN we inform Hadoop cluster that all MapReduce jobs must run on YARN nodes.

Starting and stopping hdfs and yarn nodes can be done respectively with the following commands after logging as HDFS user. More details on these commands can be found in [31].

- `HADOOP_PREFIX/sbin/start-dfs.sh`

- `HADOOP_PREFIX/sbin/stop-dfs.sh`
- `HADOOP_PREFIX/sbin/start-yarn.sh`
- `HADOOP_PREFIX/sbin/stop-yarn.sh`

These commands ensure that all the datanode daemons listed on the slave file located within the namenode are launched or interrupted whenever requested. There is a possibility to verify online whether the set up of Hadoop daemons has been successful and that all the nodes are ready to use by entering the following links corresponding to Hadoop web interface:

- `http://name-of-the-host:8088/`
- `http://name-of-the-host:50070/`

The port number 8088 correspond to the namenode default HTTP port and the port number 50070 to the resource manager default HTTP port. We use the host file located within the `/etc` folder to give names to the IP address of the nodes for better usability. For instance the IP address: 148.197.54.18 is named `bdata02.static.port.ac.uk`. Following is an example of how we certify the set up of the nodes of our cluster.

- `http://bdata02.static.port.ac.uk:8088/cluster/nodes`
- `http://bdata02.static.port.ac.uk:50070/dfshealth.html#tab-datanode`

These links provide information such as the number of nodes active, memory allocation, number of containers used but also report all applications that are started on the nodes with their duration. To verify the status of the job history, a similar structure is followed by replacing the port number by 19888.

5.2.3 Bashrc File

Bashrc is an editable shell script file wherein environment variables such as class-path, any type of command, and downloaded libraries can be declared. At each launch, the content of the bashrc file is run and this allows our program to run more efficiently. It is in this file that we have exported the path and classpath of MPJ Express, Java, Intel Data Analytic Acceleration Library (DAAL), Hadoop, MapReduce, Ant and Maven as well as those of other frameworks which are used along with our experimentation such as Apache Mahout, Spark and Giraph. To

amend the bashrc file an editor such as vi, nano or gedit can be used to open the file. The following commands respectively illustrate how to modify the bashrc file then run the updates made:

```
$ nano ~/.bashrc
$ source ~/.bashrc
```

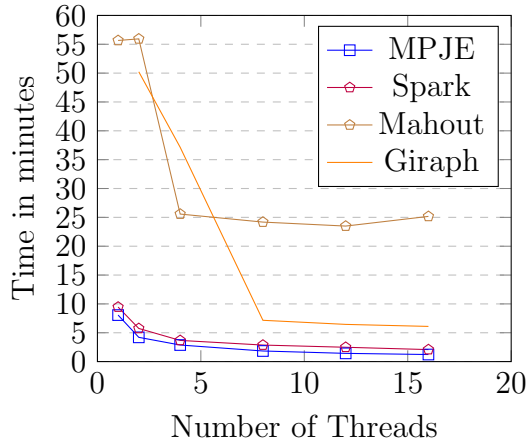
In the next lines, we demonstrate how Java path and Hadoop environment variables have been set up in the bashrc file including the variables for MapReduce and YARN.

```
export PATH=$JAVA_HOME/bin:$PATH
export HADOOP_INSTALL=/opt/hadoop-2.9.2
export PATH=$HADOOP_INSTALL/bin:$PATH
export PATH=$PATH:$HADOOP_INSTALL/sbin
export HADOOP_MAPRED_HOME=$HADOOP_INSTALL
export HADOOP_COMMON_HOME=$HADOOP_INSTALL
export HADOOP_HDFS_HOME=$HADOOP_INSTALL
export YARN_HOME=$HADOOP_INSTALL
export HADOOP_COMMON_LIB_NATIVE_DIR=$HADOOP_INSTALL/lib/native
export HADOOP_OPTS="-Djava.library.path=$HADOOP_INSTALL/lib"
export HADOOP_CONF_DIR=/opt/hadoop-2.9.2/etc/hadoop
```

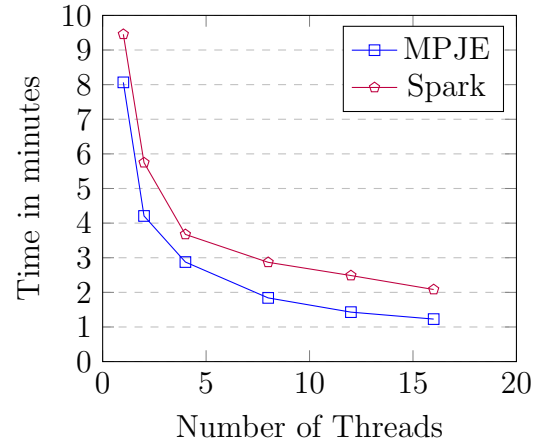
5.3 MovieLens 20M Ratings Experiments

The ALSWR code is tested with 50 features, 10 iterations, 0.01 for the regularization parameter λ and 0.01 for the parameter ϵ that is used in the initial guess for the item model. Figure 5.1a compares the performances between MPJ Express, Spark, Mahout and Giraph on a different Number of Threads. MPJ Express and Spark have both good performance and parallel speedup: as the number of cores increases the time decreases; Mahout does not show many variances from four cores and above. We have obtained the ALS results of Apache Giraph from a research collaborator: Matthew Barker [6]. There was a failure with all attempts to run the ALS with Giraph on 1 thread. From 4 threads to 8 threads, Apache Giraph time decreased from 37.15 to 7.16 minutes which shows a great performance improvement. However, not much improvement was observed from 12 threads onwards.

Figure 5.1b focus on MPJ and Spark to present better visibility on their time completion. MPJ Express has the best performance amongst the four frameworks. It is on average, 13.19 times faster than Apache Mahout, 6.97 times



(a) MPJE vs Spark vs Mahout & Giraph



(b) MPJE vs Spark

Figure 5.1: Frameworks Performance Comparison MPJ Express with MovieLens dataset

faster than Apache Giraph and on average 1.4 faster than Apache Spark. Figure 5.2 represents the parallel speedup of MPJ Express and Spark. With sixteen cores MPJ Express is almost 10 times faster than when it is run sequentially, while Spark is just about 4.5 times faster than its result with one thread.

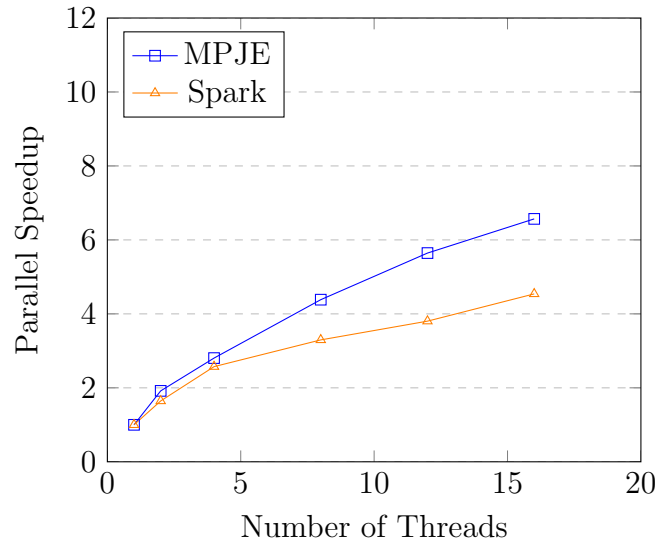


Figure 5.2: Parallel Speedup MPJ Express vs Spark with MovieLens dataset

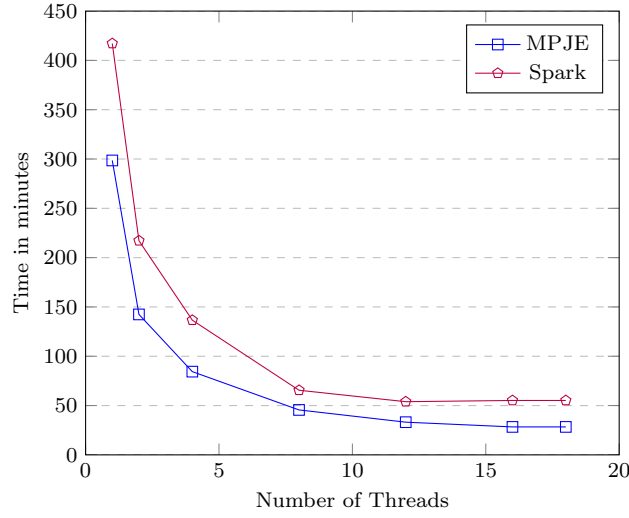


Figure 5.3: Performance Comparison MPJ Express vs Spark with Yahoo dataset on Max four Nodes

5.4 Yahoo Music 700M Ratings Experiments

Apache Mahout and Apache Giraph were unable to cope with the large Yahoo dataset. For this reason, we have evaluated only MPJ Express and Spark versions of the code for this dataset. Figure 5.3 shows a pattern quite similar to figure 5.1b although this time our dataset is about 35 times bigger. Table 5.1 displays the time measurement in minutes of the assessed frameworks. A closer look at figure 5.4 demonstrates a significant parallel speedup improvement of MPJ Express which now runs more than 10.5 times faster on 16 cores than its sequential time. The parallel speedup of Spark has also improved. It implements the ALS on Yahoo dataset 7.5 times faster with 16 cores than when it is run in sequence. However, from 12 cores onwards, the performance of the Spark version starts to decrease.

| # of Threads | MPJ Express | Spark |
|--------------|-------------|-------|
| 1 | 298 | 417 |
| 2 | 142 | 217 |
| 4 | 84.4 | 136 |
| 8 | 45.56 | 65 |
| 12 | 33.15 | 54 |
| 16 | 28.35 | 55 |

Table 5.1: Performance MPJ Express vs Spark in minutes

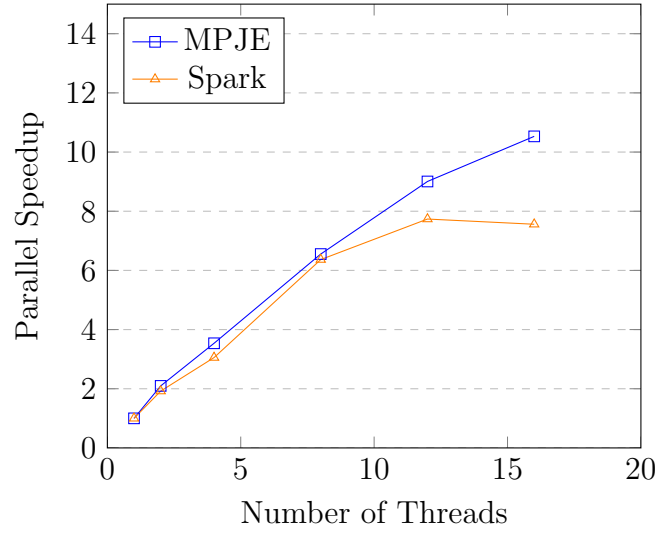


Figure 5.4: Parallel Speedup MPJ Express vs Spark with Yahoo dataset

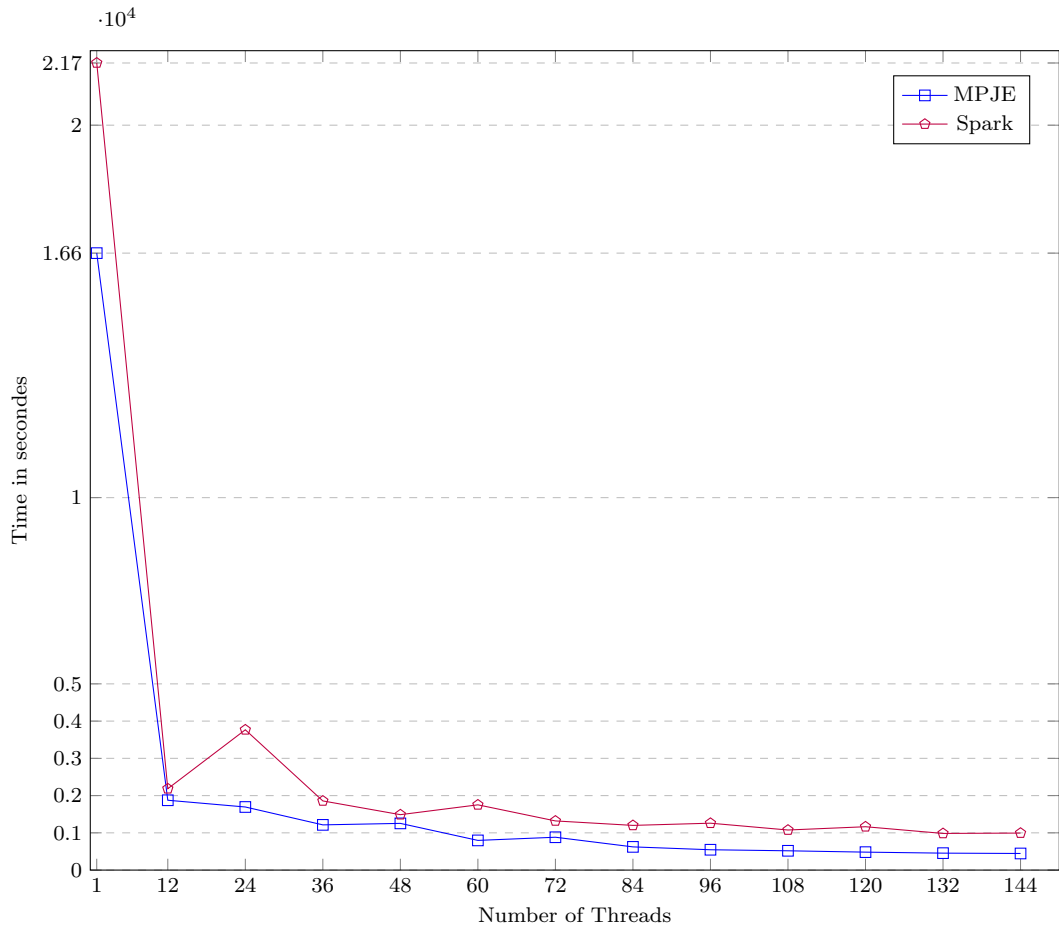


Figure 5.5: Comparison MPJ Express vs Spark Time Performance with Yahoo dataset on 12 Nodes Cluster (SPMD Model)

We now execute more tests using the ALSWR code improved by the SPMD model in chapter 6. The data no longer need to be partitioned beforehand as it is handled directly in the code. The tests are furthermore run on the new cluster using 12 nodes having each 12 cores hence 144 threads used in total. The results in figure 5.5 show a much-improved performance of MPJ Express. With 1 thread, the test is now completed in 276.08 minutes, compared with our previous test without SPMD which was taking 298.54 minutes to load and compute the records while the partitioning which was done separately was taking another 49.65 minutes. With 12 thread MPJ Express was previously taking 33.15 minutes to implement ALSWR and an additional 8.33 minutes to partition the yahoo dataset while now everything is done in 31.25 minutes. Apache Spark has also experienced an improvement with the change of cluster. With the previous cluster, the time performance of Spark was 417 minutes when running on 1 thread and 54 minutes on 12 threads. With the new cluster, Spark now implements the ALS algorithm in 361.2 minutes on 1 thread and 36.25 minutes on 12 threads. The comparison between MPJ Express and Spark shows that for each of the number of thread, MPJ Express had a better performance and from 84 threads MPJ Express performs more than twice faster than Spark. Figure 5.6 shows the parallel speedup of MPJ Express and Spark.

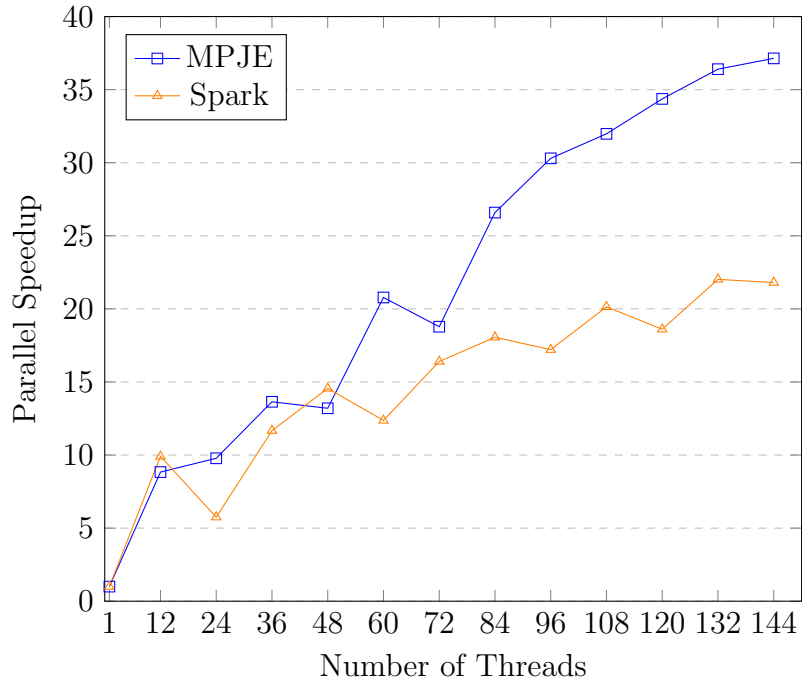


Figure 5.6: Parallel Speedup MPJ Express vs Spark with Yahoo dataset on Max 12 nodes (SPMD model)

Although on 48 and 72 threads, the parallel speedup of MPJ Express went slightly down, it is not without saying that the overall speedup kept increasing. There was still some improvement until 144 threads. On 144 threads MPJ Express was performing 37.14 times faster than its sequential run. On the other side, the parallel speedup of Apache Spark was less stable with a decline on 24, 60, 96, 120 and 144 threads. The best parallel speedup of Spark was on 132 threads with the code running 22.02 faster than when ALS is run on 1 thread.

5.5 Root Mean Square Error (RMSE) Implementation

The error metric rmse is evaluated with 5 different values of lambda consisting of: 0.01, 0.03, 0.05, 0.1 and 0.2. Lambda is our regularization parameter which is used to prevent the overfitting problem. We start by adding to the layout the ratings of Yahoo testing dataset as well as the predictions made with the training dataset. We next calculate and square the residual for each rating which is equal to the difference between the actual rating and the predicted rating. This result enables us to calculate the mean and the squared root of this mean with the function: **Math.sqrt()**. The pseudo-code in figure 5.7 gives more details of the rmse implementation in our java code taking into consideration the MPI **Allreduce()** method to sum over all the processes. Figure 5.8 displays the results of the rmse tuned with different Lambda values on 25 iterations using the Yahoo dataset. We aim for a rate as low as possible because it means that our predictions are more accurate with the full awareness that the rmse cannot be zero. A zero value would imply that the system is perfect which is unlikely due to the fact that human being is hard to predict. The best result was achieved with the lambda parameter 0.05 having a rmse equal to 1.0328.

5.6 Analysis of the results

The Mahout implementation of ALS—not necessarily representative of the wider Mahout project—is based on MapReduce. The performance limitations of MapReduce on iterative algorithms are well documented, see for example [28]. By consequence Mahout takes more time than MPJ Express and Spark to go through each iteration to implement the ALS algorithm. Esteves, Pais and Rong further explain in [28] how the number of iteration can affect the communication

```

1  // ptr: temporary holding pointer into targets and ratings arrays
2  // numLocal: the size of external IDs (ExtIDs) as it appears in the file
3  for(int i = 0 ; i < numLocal ; i++) {
4      // Add in squared residuals
5      for(int j = 0 ; j < testRatings.num [i] ; j++) {
6          int ptr = testRatings.base[i] + j ;
7          int target = testRatings.targets [ptr] ;
8          int rating = testRatings.ratings [ptr] ;
9          double prediction = 0 ;
10         int iExt = testRatings.layout.extIDs.get(i) ;
11         int targetExt = complementTestRatings.globalExtIDs [target] ;
12         int iTrain = trainRatings.layout.ext2locIDs.get(iExt) ;
13         int targetTrain = trainRatings.ext2glbIDs.get(targetExt) ;
14         for(int k = 0 ; k < NUM_FEATURES ; k++) {
15             prediction += localModel [NUM_FEATURES * iTrain + k] *
16                 complementGlobalModel [NUM_FEATURES * targetTrain + k] ;
17         }
18         double residual = rating - prediction ;
19         localSum += residual * residual ;
20     }
21     numLocalRatings += testRatings.num [i] ;
22 }
23 //Sum over all processes Allreduce() MPI method
24 double [] recv = new double [1] ;
25 MPI.COMM_WORLD.Allreduce(new double [] {localSum}, 0,
26                             recv, 0, 1, MPI.DOUBLE, MPI.SUM) ;
27 long [] recvNum = new long [1] ;
28 MPI.COMM_WORLD.Allreduce(new long [] {numLocalRatings}, 0,
29                             recvNum, 0, 1, MPI.LONG, MPI.SUM) ;
30 num = recvNum [0] ;
31 mean = recv [0] / num ;
32 return Math.sqrt(mean);

```

Figure 5.7: Outline Java code for the RMSE calculation (Equation 2.1).

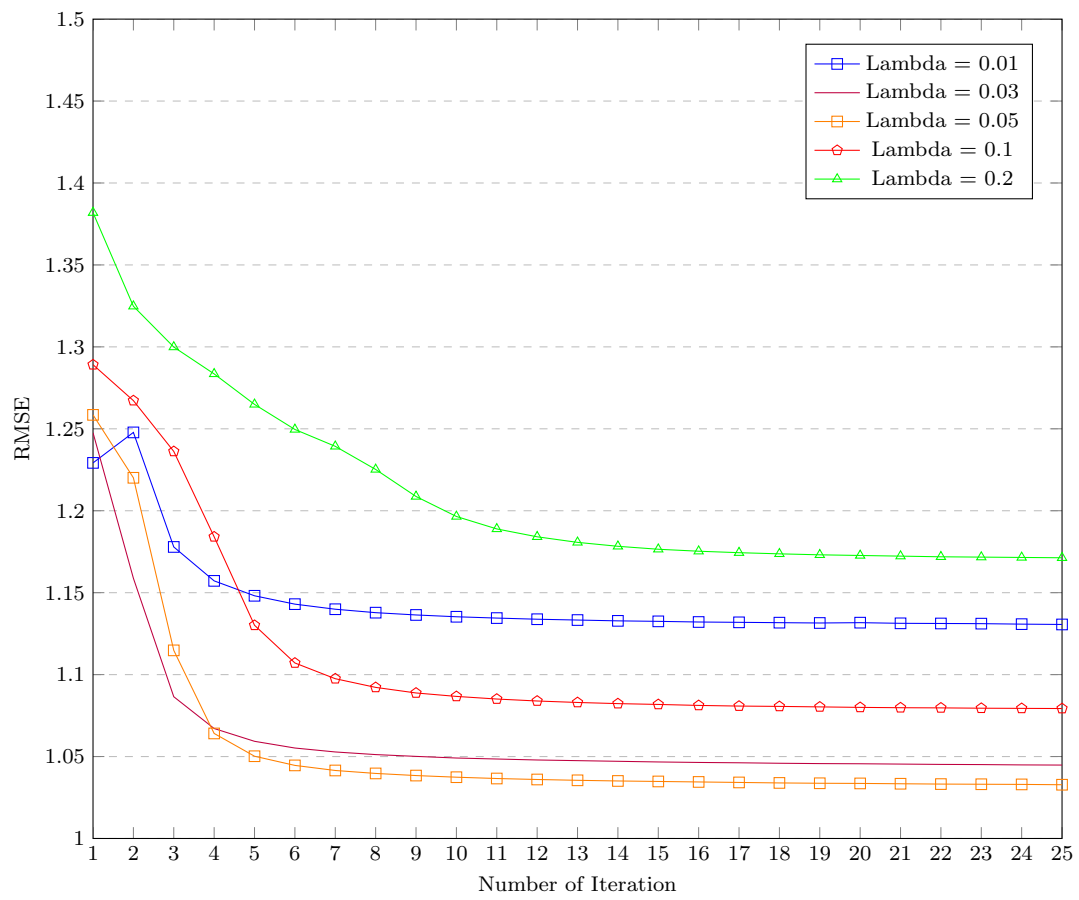


Figure 5.8: RMSE Results with Different Lambda Parameters

between mappers and reducers and as a result adds more network usage during the parallel computation. The overall poor performance of Apache Giraph in the experiments could be due to a lack of optimisation or better configuration put in place. It also seems that Giraph needed a bigger cluster as the tests done by Barker [6] often failed due to insufficient memory. As a solution to improve the performance of Giraph, more time should be spent investigating on the best configuration possible and analysing the output of the log files on Hadoop site. The supersteps should also be observed, particularly the time taken to complete each superstep. Another solution as proposed by Baker [6] could be to optimize the calculation of the symmetrical matrix in the ALS code, to compute half of the matrix instead of the full matrix similar to DAAL calculations [48].

According to pseudocode given in [100], the Spark implementation uses a combination of its *parallelize* and *collect* operations to reproduce the communication operation called *MPI_Allgather* here. We assume that the MPI collective algorithms can implement this pattern more efficiently. There is a discussion of efficient implementations of Allgather in [92] for example. Our original theory regarding Spark been outperformed by MPJ Express was that there may be some degradation of the performance of Spark when there is not enough memory (RAM) as the storage has to be on disk when the program is running out of space. However, with the use of a bigger cluster (from 4 nodes to 12 nodes used) we have demonstrated that MPJ Express was still performing better than Apache Spark.

5.7 Conclusion

In this chapter, we have revealed our experiments in detail starting by the environment set up then running numerous tests on our datasets. The results of the tests were compared with those from other widely used frameworks. Various computational frameworks have been adopted over the last few years for running compute-intensive kernels of recommender algorithms on Hadoop platforms. These include Apache Mahout, Apache Spark and Apache Giraph. We have provided evidence that by adding MPJ Express framework to this list, it can outperform other implementations of the central optimization algorithm. This additional performance certainly comes at some cost in terms of programming complexity. For example, the MPJ programmer has to spend more time concerned with details of data and computation partitioning, as well as orches-

trating communication between Hadoop nodes. It is equally the case that the HPC MPI paradigm does not automatically provide reliability features found in MapReduce or Spark, and achieving resilience may need extra programming effort. Nevertheless, we argue that for some intensive and often used kernels, the extra investment in programming may be justified by the potential performance gains. We see MPI-based processing stages as one more resource in the armoury of big data frameworks that may be used in processing pipelines run on Hadoop clusters. We also suggest that in this setting MPJ Express may be a natural choice of MPI-like platform, given that many other such processing stages will be coded in Java or JVM-based languages.

Chapter 6

SPMD Processing of Hadoop Data

6.1 Introduction

In early stages of development of our SPMD MPJ code for collaborative filtering on Hadoop, the overheads of loading data into the program were unexpectedly large and reduced the performance margin the MPJ approach might have over other Hadoop based approaches (as discussed in chapter 5). In section 4.6 we pointed out that some of these other approaches, like MapReduce, make best use of the properties of the HDFS file system by “moving work to the data”. Distributed file systems like HDFS have the not-surprising property that reads from data blocks stored on the local file system are significantly faster than reads that have to fetch blocks from remote hosts. So if processing of locally stored blocks can be done on the local CPU, significant performance gains are possible.

Our early MPJ implementations of ALSWR read pre-partitioned data from $2P$ HDFS files (ratings data partitioned by user and by item—see section 4.5). It suffered from the overhead that most of the data accesses would have been remote accesses because no attempt was made to localize the partitioned files to the hosts that would load each file. Doing that would have been difficult because at the time files are written we don’t generally know which hosts will later be running particular processes of the SPMD program.

To avoid the overheads of non-local accesses, the final version of our ALSWR program adopted an approach to loading files where blocks of an HDFS file were indeed read by a SPMD process running on the same node of the cluster that held a copy of that block—similar in spirit to how a MapReduce “split” is processed

by a map process running on a host that contains a copy of that split in an HDFS block in its local file system.

In a general SPMD program this strategy means that a record of data will not always be read by the process that ultimately “needs” this item. But our experience with ALSWR was that it works well to do the reads locally then do a permutation of the data within the parallel program to get each value to the process that needs it (see section 4.6 for how we did this permutation in our program). For some types of parallel program, for example embarrassingly parallel programs, such a permutation may not even be needed.

Because this general strategy of reading records locally from an HDFS files system into a parallel program may be useful beyond our ALSWR code for collaborative filtering, we treat it here in a self contained chapter.

6.2 Statement of Problem

We assume that we have a data set stored in HDFS files. The data set therefore consists of a set of N blocks stored in a replicated way across the nodes or hosts of a Hadoop cluster (we will use the terms node and host interchangeably). Each block typically has a small, fixed number of replicas, stored in the file systems of different hosts, those hosts chosen by a policy we will treat as random—though in practice it may not be very uniform. Three replicas of each block is common, but this number is configurable. The size of individual blocks is typically quite large. 128MB is common, but again configurable. So even quite large data sets will have a relatively manageable number of blocks—e.g. thousands of blocks for a terabyte data set.

We want to process this data set using an SPMD program, running across nodes of the Hadoop cluster—with processes on either all or a substantial subset of nodes.

The problem we want to address here is how to divide blocks between node processes of the SPMD program so that:

1. every block is read into exactly one process,
2. the division of blocks between processes is as even as possible, i.e. all nodes of the program read and process similar numbers of blocks, *and*,
3. as far as possible, processes read their blocks from the local host—in other words processes consume blocks that have a replica on the same host of

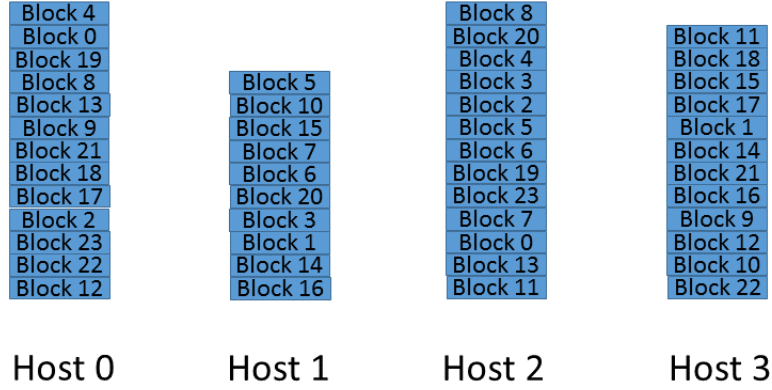
the Hadoop cluster that the process themselves are running on.

For simplicity we will assume there is *at most one SPMD process in the job per host of the cluster*.

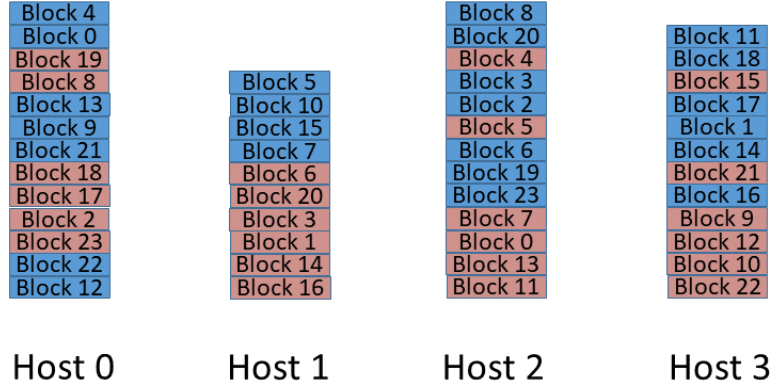
According to requirement 3 above, SPMD processes are supposed to read blocks from their local file system *as far as possible*. We will implement this requirement by a strategy in which we first allocate block reads to processes where it is *possible* to do these reads locally. In general this is possible if and only if the subset of hosts running the SPMD program actually hold a replica of the blocks in question. The likelihood of this depends on how large a fraction this is of the total number of nodes in the cluster, and the replication factor of blocks (we will discuss this further in section 6.4). After all blocks that can be are allocated as local reads—distributed as fairly as possible over the SPMD hosts—the remaining blocks will be read by non-local HDFS reads, probably by less loaded hosts. The remainder of this section is concerned with the first part of the problem—allocating all possible *local* reads.

Figure 6.1 gives a simple example of a possible distribution of replicas in an HDFS cluster, and an ideal allocation of block reads. The distribution in figure 6.1a has been generated by a uniform random processes. The pink shaded replicas in figure 6.1b then represent a possible allocation of block reads to the local host, such that every block is read once, and there are the same number (six) of block reads for every host. For illustration, this figure assumes that SPMD processes from our job run in every node of the Hadoop cluster—most often in practice this will *not* be the case, not least because Hadoop clusters are usually shared resources.

It seems hard to give an algorithm for solving this problem with any proof of optimality. Instead we will consider heuristic approaches, which are observed to work reasonably well in practise. The algorithmic complexity of these approaches are controlled by the number of blocks and the number of nodes, and we expect both these numbers to be small compared with the data size and thus the computational complexity of the processing eventually done by the SPMD program. So it is reasonable for the allocation of blocks to processes to be done centrally at the start of the program—we won't try to make this allocation heuristic parallel or dynamic.



(a) Random distribution of replicas of 24 blocks over four hosts, with replication count two.



(b) Ideal allocation of block reads from local replicas (pink blocks).

Figure 6.1: Simple example of allocation of block reads. In this example the SPMD world covers the entire cluster of four hosts.

6.3 Heuristics to solve Local Block allocation to nodes

Inputs to the local allocation problem include

1. a set of blocks we call **Blocks**:

$$|\mathbf{Blocks}| = N \tag{6.1}$$

2. a set of hosts called **Hosts** comprising the nodes of the Hadoop cluster,
3. a labelled series of subsets of **Blocks** called **Replicas_h**, where $h \in \mathbf{Hosts}$

runs over all hosts of the cluster. **Replicas_h** is the subset of blocks that have a copy stored in the file system of host h . And finally

4. the subset **World** \subset **Hosts** of hosts that the SPMD program will be running on.

The output of the algorithm will be another series of subsets of **Blocks** called **Alloc_h**, where now **Alloc_h** is the set of blocks that will be read and processed by the SPMD process running on host h .

To avoid introducing too many new names, below we will treat the **Alloc_h** as a collection of *variables* that are all initialized to empty sets then iteratively updated to yield their final output value. Another variable that will be used in the algorithm is **Avail**. This is another subset of blocks, namely those that have not yet been allocated to any SPMD process. Its initial value will be just **Blocks**, and the goal is to reduce this to the empty set—i.e. to allocate all blocks.

For all hosts $h \in \mathbf{World}$, we will define:

$$\mathbf{Avail}_h = \mathbf{Avail} \cap \mathbf{Replicas}_h \quad (6.2)$$

which is the subset of blocks that have not been allocated and have replicas on h , or in other words the set of replicas on h that are still available for allocation.

We say that a host h in **World** is *live* if **Avail_h** is not empty. So it is still possible to allocate new blocks to live processes. Formally:

$$\mathbf{Live} = \{h : \mathbf{World} \mid \mathbf{Avail}_h \neq \emptyset\} \quad (6.3)$$

Now we can give our simplest heuristic for allocating blocks to hosts, which is a Round Robin scheme:

Round Robin Heuristic

Avail = **Blocks**

For all hosts g in **World**:

Alloc_g = \emptyset

$h = \text{First}(\mathbf{World})$

Repeat while any host is live:

if $h \in \mathbf{Live}$:

Copy an element of **Avail_h** to **Alloc_h** and remove it from **Avail**

$h = \text{Next}(\mathbf{World}, h)$

Here the function `First()` simply chooses some first element from a set, and the function `Next()` iterates repeatedly through elements of a set in a round robin fashion.

In this pseudo-code it is important to note that the definition of **Avail_h** in equation 6.2 is maintained, so removing the block from the set **Avail** also removes that block from **Avail_h**, *and* removes it from the **Avail_g** set of any other host *g* that has a replica of this block. Mathematically the presentation above is concise, but in practise the way this is implemented is that the object representing a block holds a short list of hosts on which that block is replicated. When a block is “removed from **Avail**”, the actual implementation is to iterate through hosts, *g*, of its replicas, removing the block from a Hash-Set representing **Avail_g** on host *g*.

It is easy to see that the algorithm given above always converges. If time to access Hash-Sets and Hash-Tables is considered constant, and the number or replicas of each block is treated as a small constant factor, typical execution time to allocate all *N* blocks will be $O(N)$. However what is not guaranteed is that this scheme in fact allocates *all* blocks; in other words, when the heuristic completes it is *not* guaranteed that:

$$\bigcup_{h \in \mathbf{World}} \mathbf{Alloc}_h = \mathbf{Blocks} \quad (6.4)$$

No similar scheme can guarantee this, because **World** is a subset of **Hosts** and some blocks may simply not have replicas within **World** so local reads are impossible (this is discussed more quantitatively in 6.4). What this and similar heuristics later in this section *do* guarantee, however, is that all blocks that *do* have a replica inside **World** will have a local block read allocated for them. In other words if we define:

$$\mathbf{Blocks}' = \bigcup_{h \in \mathbf{World}} \mathbf{Replicas}_h \quad (6.5)$$

all blocks in **Blocks'** will get allocated. This is simply because the initial value of

$$\bigcup_{h \in \mathbf{World}} \mathbf{Avail}_h \quad (6.6)$$

is **Blocks'**, and the algorithm doesn't terminate until all **Avail_h** are reduced to the empty set, and a block can only be removed from these sets by allocating it.

So then the main criterion for comparing such heuristics is not how many blocks they successfully allocate, but how evenly the final \mathbf{Alloc}_h are divided across hosts h . For most purposes this can be summarized by the quantity:

$$\max_{h \in \mathbf{World}} |\mathbf{Alloc}_h| \quad (6.7)$$

which we can regard as the parallel read time, where the unit is the time to do a local read of a single block. A more balanced allocation will have a smaller parallel run time.

Later we will present evidence that the simple Round Robin heuristic works quite well provided the SPMD **World** contains a sufficiently large fraction of all **Hosts**, and the distribution of replicas over the hosts is sufficiently uniform. But our experience is that in real HDFS systems this distribution is sometimes not particularly uniform. And experiment suggests even for the randomly generated example of figure 6.1, around half of the time this heuristic does not get the ideal solution—instead it may end up with local allocation sizes varying between 5 and 7 blocks.

An example of a non-uniform distribution of replicas that can be more problematic for the allocation heuristic is given in figure 6.2a.

This distribution has the property that Host 1 holds a replica of every block, and the “second replicas” are scattered over the other three hosts. This may seem contrived, but we have seen similar decompositions arise in practice when the data set was originally written by a single host of the cluster—the first replica of each block goes to the local file system (on Host 1 in the example). In any case, here we just take this as an example of a non-uniform kind of distribution that has been observed in practice.

Figure 6.2b is one experimentally obtained result of applying the Round Robin heuristic to this decomposition. Load balancing is poor, with the number of block reads varying between 4 and 8. Experimentally this is a slightly worse than average result for this basic scenario; but it is also not a very unlikely result.

The problem here is that Host 2 starts with a smaller number of available replicas ($|\mathbf{Avail}_h|$) than other hosts. As other hosts have blocks allocated locally *that are shared with Host 2*, the available replicas on Host 2 are further depleted. Before long, there are no replicas of unallocated blocks remaining to allocate on Host 2. It ceases to be “live” before a fair number of blocks have been allocated here. This is an almost a trivial observation—that allocating replicas on other hosts that share a block gradually depletes replicas available locally. But it

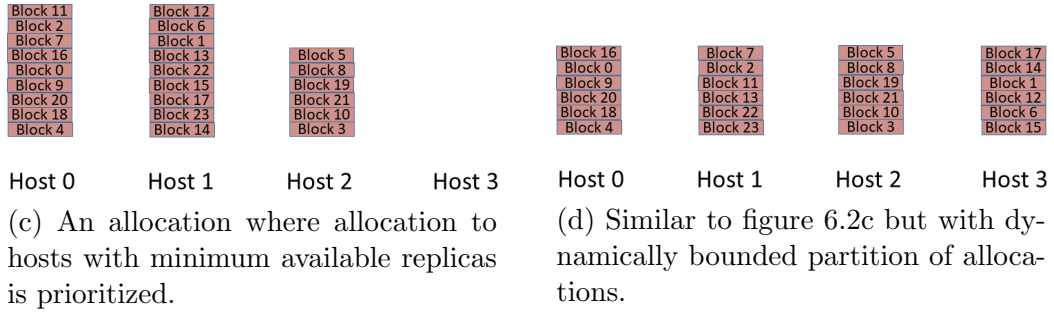
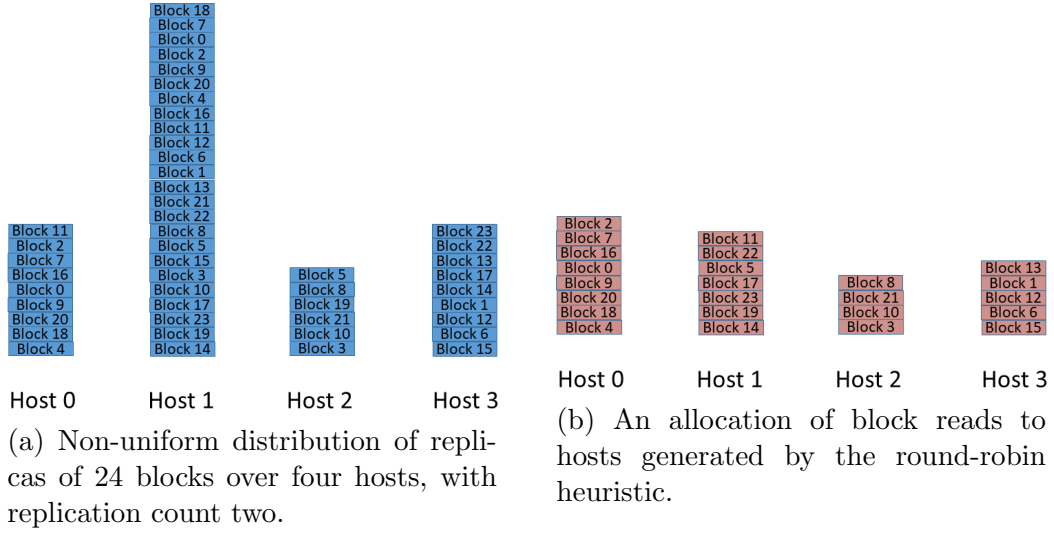


Figure 6.2: A non-uniform distribution of replicas over four hosts, with the results of various allocation heuristics.

motivates the improved heuristics below.

A first attempt to improve the heuristic might simply prioritize the allocation to hosts that have the fewest unallocated replicas available:

Straw Man Heuristic

Avail = **Blocks**

For all hosts g in **World**:

Alloc _{g} = \emptyset

Repeat while any host is live:

Choose a live host h that has the least $|\mathbf{Avail}_h|$

Copy an element of **Avail** _{h} to **Alloc** _{h} and remove it from **Avail**

This contains the germ of an idea, but in itself provides very poor performance. A typical outcome is given in figure 6.2c. Here the number of block reads allocated to hosts varies between nine and zero, with zero reads allocated for the final host,

Host 3.

The problem with this approach is that it loses a natural fairness property of the Round Robin scheme. When we allocate a block read to one of the hosts least endowed with replicas, we also *reduce* the number of available replicas on that host. So typically the next read is also allocated to that *same* host, and so on, until all its local replicas are exhausted. This may be OK for Host 2 in our example, because it starts with only six replicas. But when next allocation moves on to Host 0, all nine replicas there get allocated before moving on, and so on. By the time Host 3 might be processed, all reads have been allocated.

A slightly brute force method of tackling this overly greedy approach is to place a bound on the maximum number of reads that may be allocated on a particular host at a particular time. An obvious first attempt for this bound might be just:

$$B = \lceil N' / |\mathbf{World}| \rceil \quad (6.8)$$

where $N' = \mathbf{Blocks}'$ (see equation 6.5). Then we modify our heuristic as follows:

Interim Heuristic

Avail = **Blocks**

For all hosts g in **World**:

Alloc _{g} = \emptyset

Repeat while any host is live:

Inspect the set of live hosts h with $|\mathbf{Alloc}_h| < B$

Choose a host h from this set that has the least $|\mathbf{Avail}_h|$

Copy an element of **Avail** _{h} to **Alloc** _{h} and remove it from **Avail**

This ensures that no host gets more than B block reads, and increases the chances that all nodes get at least some of the reads. Unfortunately as it stands the static bound in equation 6.8 may lead to divergence of the main loop, because if some hosts run out of available replicas before saturating B , it can become impossible to allocate all the replicas on the remaining live hosts without allocating more than B replicas to any of them.

To fix this, we define the set of *fully loaded* hosts as those in the SPMD **World** that are no longer live. So:

$$\mathbf{Loaded} = \{h : \mathbf{World} | \mathbf{Avail}_h = \emptyset\} = \mathbf{World} - \mathbf{Live} \quad (6.9)$$

There will be no more allocations on the fully loaded hosts, but total number of

blocks that we *hope* to eventually allocate to hosts that are *currently live* is the number of blocks that haven't already been allocated to the fully loaded hosts, viz:

$$N_{live} = N' - \sum_{h \in \mathbf{Loaded}} \mathbf{Alloc}_h \quad (6.10)$$

Note this is *not* the same thing as the number of blocks that have not yet been allocated, because some blocks will already have been allocated on the live hosts. So now at a given stage of processing we can define a less conservative bound on the partition size, B , as:

$$B = \lceil N_{live} / |\mathbf{Live}| \rceil \quad (6.11)$$

This value of B may change as the calculation progresses (it may grow as the live set is reduced), but its effect will still be to suppress over-eager allocation to any single host.

With the definition of B in equation 6.11, it is easy to show that if there is any live host, then there is a live host h with $|\mathbf{Alloc}_h| < B$. This implies that **Interim Heuristic** converges.

For our example of figure 6.2, **Interim Heuristic** together with the definition in equation 6.11 gives the ideal result in figure 6.2c.

One certainly shouldn't be complacent over a heuristic that happens to work for one fairly contrived-looking example. And in fact **Interim Heuristic** is not particularly successful on more uniform initial distributions of replicas—often less so than the original Round Robin heuristic. It still has the undesirable property that predominantly it visits one node and allocates all possible reads on that node, then moves to another node and allocates all possible nodes there, and so on. The original Round Robin approach seems intuitively fairer, and the evidence backs this up.

But now consider the following small modification of **Interim Heuristic**:

Improved Heuristic

Avail = Blocks

For all hosts g in **World**:

Alloc _{g} = \emptyset

Repeat while any host is live:

Inspect the set of live hosts h with $|\mathbf{Alloc}_h| < B$

Choose a host h from this set that has the least $(|\mathbf{Avail}_h| + |\mathbf{Alloc}_h|)$

Copy an element of **Avail** _{h} to **Alloc** _{h} and remove it from **Avail**

The difference from **Interim Heuristic** is that now we prioritize allocation on hosts with least $|\mathbf{Avail}_h| + |\mathbf{Alloc}_h|$. Initially all \mathbf{Alloc}_h are empty so the host with least $|\mathbf{Avail}_h|$ gets priority. But now, as a block is allocated to h , $|\mathbf{Avail}_h|$ is decreased and $|\mathbf{Alloc}_h|$ is increased—so this host’s priority is unchanged. Meanwhile a side effect of the allocation on this host is to reduce $|\mathbf{Avail}_g|$ on other hosts that share the same block, increasing the priority of those hosts. So, either now or shortly, allocation will move on to another host. In fact the eventual behaviour will have the allocation focus jumping from host to host in a way that reproduces the intuitive fairness of Round Robin, whilst also giving priority to hosts with smaller or depleted sets of available replicas.

This is in fact the heuristic that gave rise to the “ideal” allocation in figure 6.1b.

6.4 Evaluation on Randomly Generated Replica Distributions

Assume the SPMD **World** has size wP , where P is the total number of hosts in the cluster and the rational factor w is in the range $0 < w \leq 1.0$. In other words, w is the proportion of hosts in the cluster that are running our current SPMD job. We will also assume that the block replication factor for the HDFS system is integer F , where $F \geq 1$.

Now, assuming replicas are assigned randomly, if $F = 1$, the probability of any given block *not* having a replica in the SPMD **World** is $1 - w$. If $F = 2$, the probability of *both* replicas of a given block *not* being in **World** is $(1 - w)^2$. And so on.

So in general the probability of all replicas of a given block not being in **World** is $(1 - w)^F$, and the expected number of blocks that *do* have a replica in **World** is:

$$N(1 - (1 - w)^F) \tag{6.12}$$

Here we have assumed a very simple random process, and have not taken into account the constraint that there can’t be more than one replica of any given block on any given node. But equation 6.12 is likely to be a reasonable estimate for many purposes (and is in good agreement with our more detailed simulations).

What this tells us in practice is that if F is 3 (the default for an HDFS implementation) and our SPMD job runs on half the nodes of the cluster ($w =$

$1/2$), then the about 88% of blocks can be read by local reads (the remainder will have to be read by non-local HDFS reads). A well balanced set of local reads is then likely to give a useful performance gain. If the SPMD job runs on a quarter of the nodes, about 58% of blocks can be read locally, which *may* still leave good balancing of the local reads beneficial. For smaller jobs there is probably limited gain over just doing all reads by non-local means. Nevertheless we take these results as encouraging for the case of large jobs.

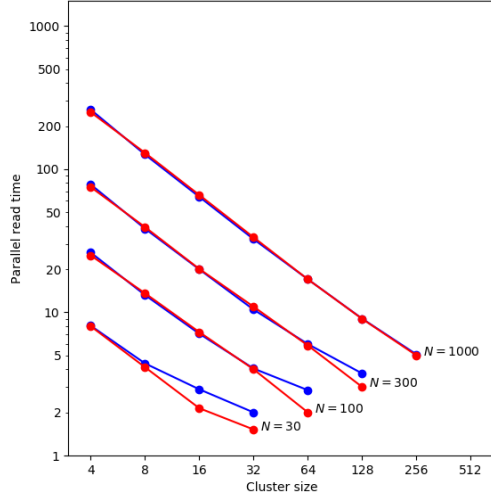
To evaluate our heuristics we have written a program that randomly generates more realistic distributions of blocks over hosts subject to a given replication factor. Some results of running this program are shown in figure 6.3 which compares “parallel read times” (in units of individual block read times) as a function of cluster size, for various numbers of blocks N and values of the ratio w . Blue plots are for the original Round Robin heuristic; red plots for “Improved Heuristic”. Here we have run our simulation for various sized clusters and for a few different values of the ratio w (all for $F = 3$). Vertical axes are labelled as “parallel run time”, but this just means the largest number of local reads allocated to any process in the SPMD program (world). We assume for simplicity that all local reads take the same, unit amount of time. So it is really a measure of load balance. For each value of cluster size and w , 1000 different block distributions are generated, and the plotted value of the parallel read time is the average over these distributions.

Nearly always, our “improved heuristic” is better than the original “round robin” heuristic. Although the differences are often quite small, some advantage of the improved approach seems clear, and we have used this approach in practice.

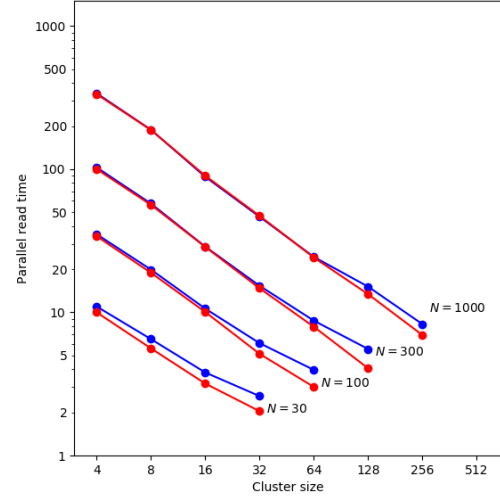
6.5 Practical Application

For reference, figures 6.4 and 6.5 illustrate some essential parts of the Hadoop API used to implement our scheme in practice.

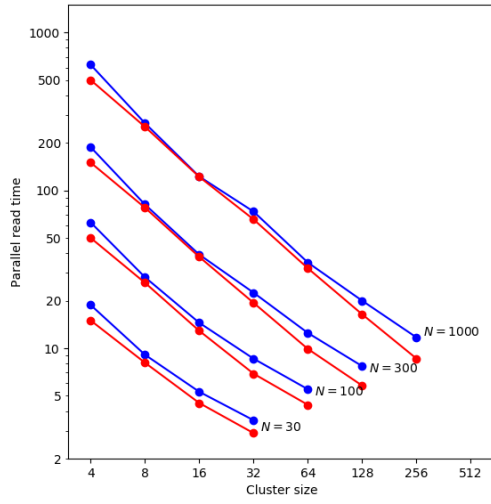
Figure 6.4 shows how metadata about the blocks of a file can be extracted using the HDFS API. Here `fsURI` and `fileName` are inputs to the program that define the URI of the HDFS file system and the name of the input file (in reality our program also allows input of multiple files in the same HDFS directory which requires small changes). The properties of the HDFS `BlockLocations` are extracted and stored in a small serializable object of type `Block` that is part of our program. The relevant properties are the file path, offset of the block from



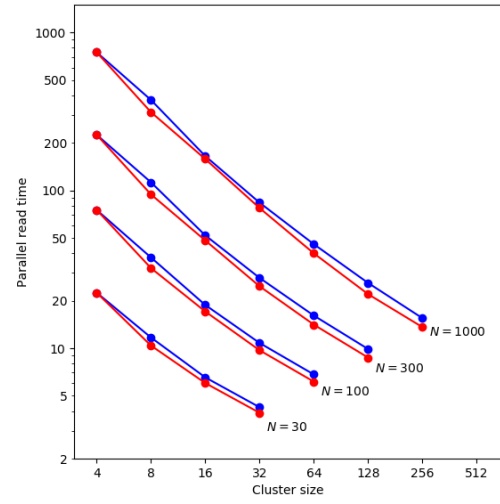
(a) $w = 1.0$



(b) $w = 0.75$



(c) $w = 0.5$



(d) $w = 0.25$

Figure 6.3: Comparison of “parallel read times”. Blue plots are for “Round Robin heuristic” and red plots for “Improved Heuristic”.

```

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.fs.FileStatus;
import org.apache.hadoop.fs.BlockLocation ;
import org.apache.hadoop.fs.CommonConfigurationKeysPublic ;

[...]

Configuration conf = new Configuration() ;
conf.set(CommonConfigurationKeysPublic.FS_DEFAULT_NAME_KEY, fsURI) ;

FileSystem fs = FileSystem.get(conf) ;

[...]

// Extract metadata about blocks of HDFS file:

Path path = new Path(fileName) ;
FileStatus file = fs.getFileStatus(path) ;

long length = file.getLen() ;

BlockLocation[] blkLocations;
blkLocations = fs.getFileBlockLocations(file, 0, length);
for(BlockLocation blkLoc : blkLocations) {
    Block block =
        new Block(file.getPath().toString(),
                  blkLoc.getOffset(),
                  blkLoc.getLength(), blkLoc.getHosts()) ;
    [ ... store block for allocation heuristics ... ]
}

```

Figure 6.4: Example use of HDFS API for extracting metadata about blocks

the start of this file, size of the block in bytes, and the short array of hosts that store replicas of this block. The resulting collection of `Blocks` is an input to our load balancing heuristics. Presently the code in figure 6.4 and the heuristics are run only on the rank 0 process.

After the load balancing heuristics have been run, arrays of local `Blocks` to read are sent to all processes and stored there in a variable `localAllocations`. Then code like figure 6.5 is run to read these blocks and extract individual records (ratings in our case) for local processing. This makes use of a couple of classes from the MapReduce API—`FileSplit` and `LineRecordReader`. There is no guarantee that a block contains an exact number of records—the first and last records in the block may be split over adjacent blocks. But the MapReduce API takes care of any small non-local reads that may be needed to take account of

```

import org.apache.hadoop.mapred.FileSplit ;
import org.apache.hadoop.mapred.LineRecordReader ;

import org.apache.hadoop.io.LongWritable ;
import org.apache.hadoop.io.Text ;

[...]

for(Block block : localAllocations) {

    // Read records in block using MapReduce API:

    FileSplit split =
        new FileSplit(new Path(block.path), block.start,
                      block.length, (String []) null) ;

    LineRecordReader recordReader =
        new LineRecordReader(conf, split, "\n".getBytes()) ;

    LongWritable key = new LongWritable() ;
    Text value = new Text() ;

    while(recordReader.next(key, value)) {

        byte [] record = value.getBytes() ;

        [ ... process fields of record ... ]
    }
}

```

Figure 6.5: Example use of MapReduce API for processing blocks in SPMD program.

this.

Note the `LineRecordReader` class is technically part of the internal implementation of MapReduce and its interface is flagged as “unstable” in the source code. This is not ideal, but it would be tedious to re-implement its functionality from scratch. The API illustrated here is based on Hadoop 2.9.2.

Table 6.1 gives actual benchmarks for the time to simply read all records (without any processing of fields) in our ALSWR, loading the Yahoo Training set. The first column gives times for loading when locality of blocks is ignored, so most reads will be remote reads. The second column gives the timing when applying our heuristic to make most reads local. When the program is running on a number of nodes approaching the cluster size (12 nodes in our case) read times are better by a factor up to about three.

Absolute times for reading data are however relatively small compared with other parts of the computation reported in earlier chapter. It is worth noting

| # of Nodes | Non-local reads | With local block read heuristic |
|------------|-----------------|---------------------------------|
| 1 | 83 | 80 |
| 2 | 50 | 43 |
| 3 | 33.7 | 25.5 |
| 4 | 23.6 | 20.2 |
| 5 | 27.8 | 13.1 |
| 6 | 22.2 | 9.9 |
| 7 | 19.5 | 8.0 |
| 8 | 18.7 | 7.6 |
| 9 | 18.0 | 7.1 |
| 10 | 16.7 | 6.0 |
| 11 | 14.8 | 5.4 |
| 12 | 14.0 | 5.1 |

Table 6.1: Times in seconds for reading Yahoo Training set with and without local block read heuristic

that use of the standard Java APIs for string manipulation and parsing numbers may easily outweigh the costs of actual data loading, and in our program we wrote custom parsing methods to read numbers from byte buffers.

6.6 Conclusion

The data loading overhead that occurred during the implementation of ALSWR has inspired us to try a different approach and eliminate the extra step of partitioning datasets before running the code. The approach described here allows the records to be read locally from HDFS and to be evenly distributed across processes. In this chapter, we have described heuristic approaches which have been applied then improved to allocate local blocks to nodes. Both heuristics (original and improved) have then been evaluated through a program that randomly generates distributions of blocks across the nodes available in the cluster. The results obtained have demonstrated that most cases, the improved heuristic approach performs better than the original one. Using the improved heuristic on the Yahoo dataset, we have benchmarked the time performance to load records and compared the results with non-local data loading. We found that the time taken to load data with the improved heuristic method was almost three times faster.

Chapter 7

Toward Social Media Scale

7.1 Introduction

Social media has taken huge importance in society. The number of subscribers, followers, twitters, bloggers and so on, has massively increased since the emergence of all these various communication tools. In this context, the term social media is used to define any existing platform that is used by some participants to exchange some information by mean of electronic devices. Social media is used for many different ends. It is used for marketing purpose to help a business to grow or to reach out to people and potential stakeholders. It is also used to connect with people without geographic restriction, to share some topics of interest or career opportunities. The fact that the basic usage of the majority of social media platform is mostly free makes it greatly accessible around the world. From a business point of view, connecting with people enables an organisation to assess its customers, to be able to better target the services and products needed and to evaluate its performance based on customers' feedback and comparisons with other competitive brands. To optimise the effect of social media in their business many organisations are already applying some recommendation systems techniques to help analyse their clients and predict their behaviour. The main challenge we want to tackle here is the scalability of recommendation systems and more specifically, for approaches used at present to be usable years ahead despite the rapidity of user growth within businesses.

In section 7.2 we provide some examples of well-known social media and the recommendation techniques they are using. In section 7.3 we evaluate the data growth within the organisations listed as examples in section 7.2. In section 7.4 we propose a means to measure the scalability of recommendation systems and

its usability in future years.

7.2 Example of Social Media Recommendation Techniques

In chapter 2 we have discussed recommender systems and reviewed some of the available techniques. Social recommendation techniques can be defined as a set of algorithms which enables suggestions to be offered to some individuals based on their social connections or similarities with other users on a social network. The suggestions sent by social media platforms are various depending on the type of users' interactions. On a network side, recommendations could be on friends to add when referring to Facebook, people to connect with for LinkedIn, people to follow if referring to Instagram or Twitter for instance. From a business point of view, suggestions are made on products or services used by individuals socially connected and which could influence the interests of the person targeted for the recommendation. In this section, we choose some of the most popular social media platforms in 2019, based on the number of their monthly active users, and analyse their recommendation techniques.

7.2.1 Facebook

Facebook is currently the most popular social media network, by its number of monthly active users which is over 2.3 billion, and its worldwide presence. Various algorithms are used to provide recommendations according to the platform on Facebook. Facebook apply recommendations to the users' timeline on the news feed section, the market place, businesses having a page on Facebook and for friends suggestions. Facebook is well known to offer a simple way to enable participants to increase their network by its famous section: "People You May Know". This suggestion is established from common or mutual friends, individuals frequently sharing the same location or tagged in the same picture or even new added contacts [17]. Facebook has furthermore recently launched Facebook Dating [64]. This new project also requires the implementation of a social recommendation system as it provides suggestion to users who have selected Facebook dating and create their profile. The affinity between the participants is in general based on their common interest.

Regarding news feed, Facebook has put in place a new ranking algorithm since

2018 that calculates the relevance of a post based on people reaction [63]. The number of comments, interaction with the post including likes and posts shared are the main factor for the level of appearance in the Facebook news feed. To improve the posts suggested in the news feed, Facebook launched in March 2019 “Why am I seeing this post?” [65]. This section allows users to acknowledge the reason behind the recommendations they receive. Facebook also adopts graph processing which is implemented in Apache Giraph. This method helps users finding contents such as pages, groups, game and so on. We have discussed this approach in sections 1.3.1 and 3.5. Figure 2.7 shows the process involved in this method.

7.2.2 Instagram

Instagram is a social media network whose main focus is to allow users to connect with each other by sharing photos and videos. Instagram can be used for leisure and business as it is suitable for organisations wishing to advertise content. Instagram was launched in 2010 [13]. The network has experienced a rapid growth; in 2 years they had already reached more than 80 millions users [15]. During the same year in 2012, Instagram decided to merge with Facebook [14], and Instagram is now officially owned by Facebook. There are currently more than a billion monthly active users in Instagram [16]. Similarly to Facebook, Instagram uses ranking feed algorithms to present the most relevant posts. As posts get prioritised on other users feed, it is to be noted that three main factors are considered for the ranking: the interactions with users on previous posts, the interaction of users with similar posts and the novelty of the posts. Some strategies to adapt to the way Instagram posts are displayed are proposed in [35].

The machine learning applied in Instagram is complex and affects different areas. The modelling framework used is Caffe2 which is a deep learning framework suitable for large scale experiments [12]. Gaussian Process is additionally added to the prediction models [9] and Bayesian optimization used to tune normalized values for both Instagram and Facebook [56].

7.2.3 LinkedIn

LinkedIn is a professional social media platform which was officially launched in 2003 [68]. LinkedIn can be compared to an online resume; it exposes the skills

and competence of a candidate and matches the profile that has been set with the skills defined by recruiters. As a result of the affinities between the employers and candidates, jobs prospectus are sent by email to potential candidates showing them how they match a given position. LinkedIn additionally allows its users to expand their network by suggesting people they might know and the place or location they have in common in the section "grow your network". Behind its interface, many techniques are adopted to ensure that the right jobs and candidates are recommended as well as the suggestions sent to connect with other people are relevant. When coming to recommendations, as explained by Guo et al. [39], LinkedIn aims to address three points:

- A model that ensures the interest of either party: the candidate and the recruiter is taken into consideration. This approach is implemented by querying the responses of InMail messages as they can indicate whether a candidate is interested in a job offer. InMails are private messages that can be sent to a user when the connection with that user is beyond the second degree and by consequence would not be able to send a free message on LinkedIn. Further details on the implementation of this method are available in [74].
- Appropriate queries must be written to enable the complexity of the parameters passed in the search engine to be well understood and processed. Parameters can be for example the name of the candidates, the skills defined in their profile or some keywords added to the searching tool. To tackle this challenge, LinkedIn implements a query building and search ranking system [40].
- A model which ensures that personalization is achieved through each recommendation. This is to make sure that the right pool of candidate is also matched by taking into consideration the recruiters' preferences. Among the solutions that were proposed, LinkedIn is using a hybrid method between 2 models: generalized linear mixed (GLMix) and gradient boosted decision tree (GBDT) [73].

From a slow start in 2003 to a quick emergence in 2013 with 225 million users and more than 645 millions users in 2019 [68], LinkedIn has established itself as the most popular professional network. The recommender systems techniques adopted within the company have greatly contributed to its growth.

7.3 Growth of Data within Social Media

In this section we analyse the user growth over the last 5 years of the social media platforms discussed in sections 7.2.1, 7.2.2 and 7.2.3. Instagram monthly active user statistics for the last 5 years are: 400 million in 2015, 500 million in 2016, 700 million in 2017 and more than a billion in 2018 and 2019 [16]. LinkedIn assesses their number of user in general instead of the usual "monthly active users" for the reason that for many users, their main activity on LinkedIn after they have found a job is to update their resume. The number of users in LinkedIn in 2015 is 396 million [66]; in 2016 there is a total of 467 million users [67]. In 2017 LinkedIn has reached 500 million users [32] and in 2018, 590 million users [49]. The current total number of LinkedIn users is 645 million [69]. Facebook in 2015 had 1591 monthly active users [77], in 2016 this number increased to 1860 million [78]. In 2017 the monthly active users was over 2 billion with 2130 million in the fourth quarter [79]; this number increased to 2320 million users in 2018 [80]. The current total number of monthly active users at the end of the second quarter is over 2.7 billion [81]. Figure 7.1 compares the number of users on Facebook, Instagram and LinkedIn for the last 5 years. With the rapidity of users and data growth, recommendation systems must be tested accordingly to ensure that they will be scalable with bigger datasets in the future.

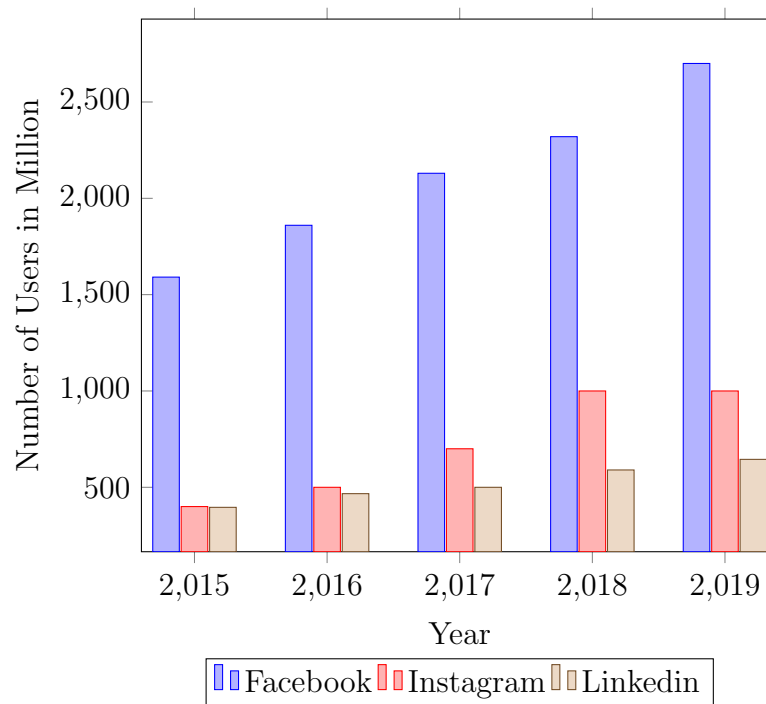


Figure 7.1: Users 'Growth of Facebook, Instagram and LinkedIn

7.4 SyntheD: A Solution to Measure Scalability

SyntheD is a Java-based program that has been written with the purpose to create large synthetic datasets. Our target with this program is to be able to reach billions of ratings that will be used for testing against our MPJ Express model for its scalability. In section 1.3.2 we have stated the acquisition of publicly available datasets such as MovieLens (movies ratings) and Webscope (Yahoo Music ratings). On the other hand, with the ever-increasing amount of data in our society, the size of data in a few years is likely to be much larger than the one which is currently handled. To that end, models that are created for recommender systems must be trained and tested with bigger datasets to proof their scalability. On one of his post on Facebook, Mark Zuckerberg informs his audience that Facebook has already reached a billion users in 2015 [102], the evidence is also displayed on the first quarter 2015 results report [76]. In 2019 the number of users is now over 2.7 billion [81]. This number includes users of Facebook, Messenger, WhatsApp and Instagram; all also referred to as the family of service [81]. Extremely large datasets with billions of ratings are hard if not impossible to find and on another side surveys or interview methods used to collect data is time-consuming. Hence the necessity to create synthetic datasets. Existing data-generating tools are assessed in section 2.3 of the literature review. The program SyntheD has been built in a way to enable a straightforward change of the size of the desired dataset. We first determine the prerequisites to generate datasets with SyntheD then we describe the step followed to implement SyntheD and finally, we illustrate the use of generating tool to create datasets and measure the performance of the data creation experiments.

7.4.1 Prerequisites

The application program SyntheD being Java-based, it primarily requires the software Java (latest version preferably) to be installed and configured on the machines. We use **util.Random** and **io.PrintWriter** libraries from Java to generate and print out the dataset. The size of the cluster required depends on the dataset that the user wants to generate. For instance to generate 10 billion ratings given by 10 million users on 189 thousand items we have used a cluster made of 13 nodes with each node consisting of 12 cores. Having insufficient memory available could result in a Java heap space issue. The Java heap Space is a type of **OutOfMemoryError** exception which occurs when there is not

enough space to allocate an object in the Java heap [72]. The Java heap can be manually configured using the commands: `-Xms or -Xmx` as a parameter when launching a Java job. The maximum heap size that can be used as explained in [71] will depend on the maximum address space available in each process.

7.4.2 Implementation

The method for generating the synthetic dataset borrows its basic idea from the generation of the *Jumbo dataset* described briefly in section 6 of [75]. The authors of [75] state that Jumbo was created as a low rank matrix (rank 10 in their case). We also generate a low rank matrix by the procedure of first randomly generating models for the users and items that have similar structure to the models that will eventually be inferred by the latent features collaborative filtering model (section 4.3). The rank of the full matrix of ratings would thus be n_f which we set to 100 in the generation of the specific dataset considered below. So we create a user model representing the matrix for users and an item model consisting of a matrix for the items.

The individual elements of the user and item models are generated by the formula:

$$a + bu$$

where u is a uniform random variate between 0 and 1, and the constants a and b are tuned so that the mean of the resulting generating rating are in the middle of the range 0 to 5 (for 0 to 5 star ratings) and the standard deviation spans this range realistically.

For as many generated ratings as we desire, user id and item id values are randomly chosen, then the dot product of the corresponding model columns are computed. Following is a pseudo-code of the creation of the ratings:

```
float rating = sum over i
    userModel [user, i] * itemModel [item, i];
```

The output format of SyntheD takes its inspiration from the Webscope dataset (Yahoo dataset) where each rating consists of three elements: the users' ID, the songs' ID and the rates given by the users. We start the implementation by setting the attributes such as the number of user, item and ratings we wish to print in the output file. The program thus outputs a matrix of rows equal to the total number of ratings and three columns to represent the users, the items and the ratings.

7.4.3 Data Creation

To launch the SyntheD program on Linux we type the following command and increase the Java heap to 16 gigabyte to accommodate the size of the new dataset and prevent from the **OutOfMemory** error.

```
java -Xmx16G SyntheD
```

We start our experiments by creating a smaller set with a size similar to the MovieLens data, then increase the data to a size comparable with the Webscope data and finally generate 10 billion ratings which are our current target. The diagram in figure 7.2 is an illustration of the data creation process. At each stage of the experiment, we record the time taken to produce a new dataset. The results are displayed in table 7.1.

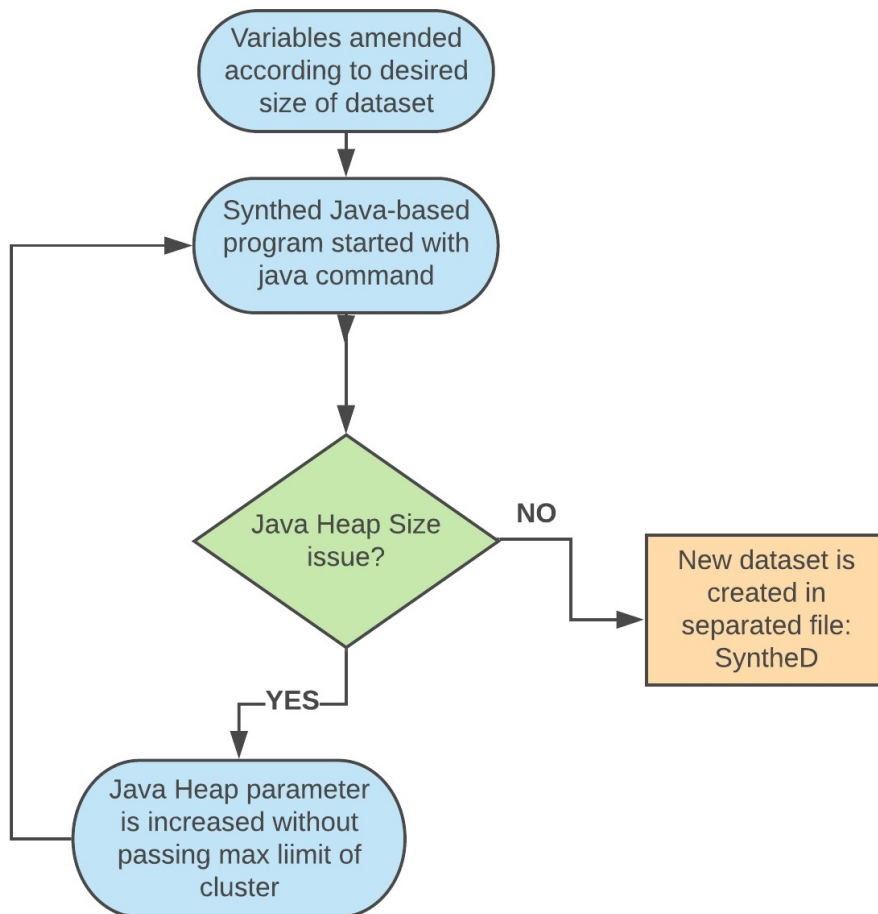


Figure 7.2: Synthetic Dataset Creation Process

| 20 Million | 700 Million | 10 Billion |
|------------|-------------|------------|
| 20 s | 13mn 07s | 2h 54mn |

Table 7.1: Synthetic Datasets Generation Time

7.5 SyntheD 10B Ratings Experiments

Similarly to the experiments done in section 5.3 on the MovieLens dataset and section 5.4 on Yahoo dataset, we benchmarked our ALSWR implementation on SyntheD. The SPMD approach to partitioning data described in chapter 6 has been applied to the code to ensure that we get the best results.

For this dataset it was not possible to run the program on small numbers of nodes. Simply storing the distributed ratings takes over 100GB of memory and our cluster only has 32GB per node. So in figure 7.3, which displays the timings in seconds of the tests by the number of threads, all jobs have been run across all 12 worker nodes of the cluster, with the variable being the number of threads per local node. The overall implementation tends to complete faster when using a higher number for the threads. The program keeps improving until 144 threads even though from 132 threads to 144 there was only a slight difference from 4499 to 4312 seconds. The differences between the performance per thread are reflected in figure 7.4 showing the parallel speedup of the program *relative to the base case of one thread on each of the 12 nodes*. The best time achieved with 12 threads being 15944 seconds, running the code with 144 threads is then 5.49 times faster than with 12 threads. The departure from linear speedup with thread here is mostly accounted for by the times for loading and partitioning data, and performing the MPI collective operations, all of which are presently single threaded on each node (a local version of Amdahl’s law).

We positively interpret the results obtained in two ways. Firstly, the fact that we were able to run some tests on SyntheD certifies that the dataset generated is fit for purpose and can be used to evaluate recommender systems. Secondly, the outputs showing a similar behaviour with previous tests on smaller datasets demonstrate that the implementation of ALSWR with MPJ Express is highly scalable.

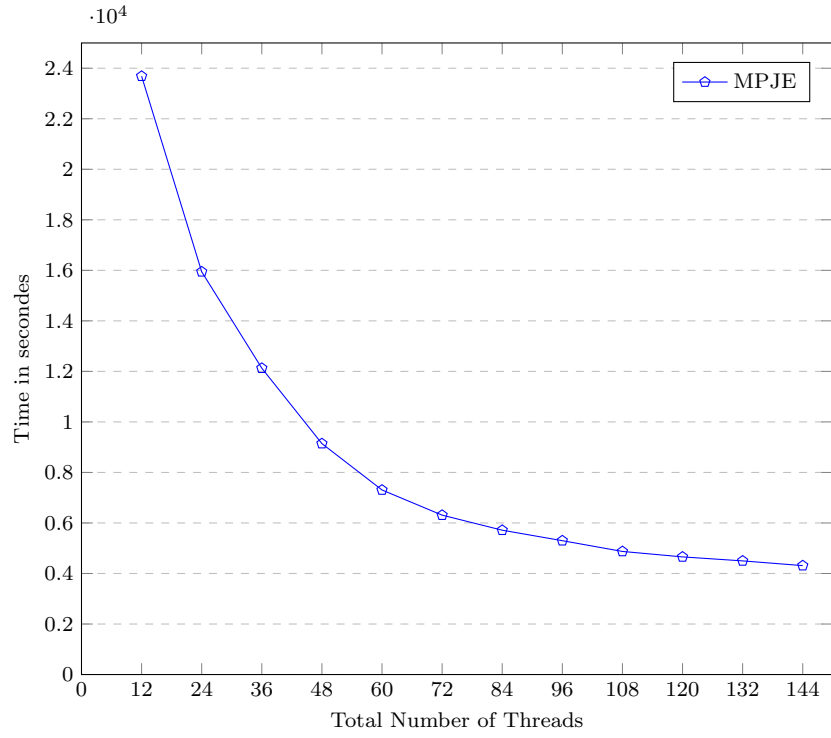


Figure 7.3: MPJ Express Time Performance with SyntheD dataset on 12 Nodes

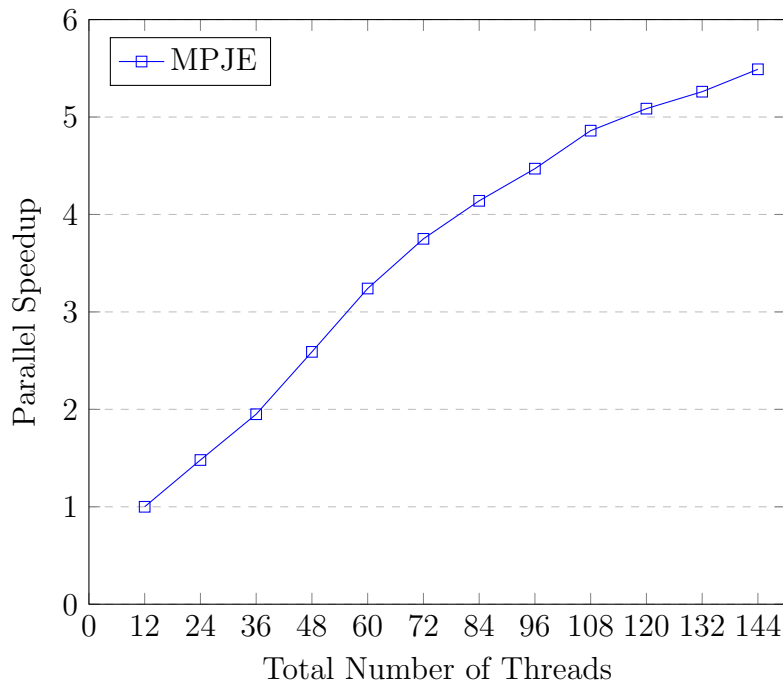


Figure 7.4: Parallel Speedup with threads per node - MPJ Express with SyntheD on 12 nodes

7.5.1 Prospects for Processing Larger Datasets

The parallel execution time of just over one hour on the 10 billion rating dataset is not particularly restrictive, and performance scalability to larger numbers and cores has proven encouraging to date. But with this dataset we were close to the limits of what could be fitted in the memory of our cluster, which has 32GB per node. Several memory optimizations were needed to enable the runs recorded earlier in this section. For example:

- the simplified code for data partitioning in figure 4.10 assumed that all ratings data was initially read into array lists in memory called `readIDs`, `readTargets` and `readRatings`, then routed to the final destination node. The extra memory required by these array lists eventually exceeded our available memory. So the final code reads the ratings data from the file system repeatedly, in different phases of partitioning, rather than store it temporarily in memory. This makes efficient reads from HDFS, as discussed in chapter 6, particularly useful.
- The data structures that ultimately store the partitioned ratings (figure 4.4) involve two “big” arrays, `targets` and `ratings`. For the numbers of ratings stored locally in our datasets, elements of type `int` have proven sufficient for `targets` (and in any case the size limits of Java arrays mean having a larger index type would require more general restructuring of code). The `ratings` element type would most naturally be a floating point type; but to save two bytes of memory per element we actually store a scaled integers in a Java `short` element, on the grounds that user ratings are not in their nature very high-precision quantities.
- The essential calls to `MPIAllgather` in figure 4.3 yield very large arrays holding the replicated copies of the user and item models on every node. Depending on the implementation of the Allgather operation, this may lead to allocation of very large internal *communication buffers*, in addition to the user arrays passed in as arguments. This is the case for the implementation of this operation in the standard distribution of MPJ Express. So the implementation code for this method was in-lined in our ALSWR code, and pipelined so that it only used bounded size communication buffers.

For datasets with significantly larger numbers of users, the overhead of temporarily storing the user model in a globally replicated fashion will become untenable, and we will have to adopt something like the “rotational” approach of

[52] (see figure 2.7 in an earlier chapter). Similar approaches have been used in the context of, say, parallel many body problems for a number of decades.

7.6 Conclusion

In this chapter, we have discussed social recommendation systems and given some example of well-known social media networks actively using recommendation systems on their platforms. By comparing the growth of users for the past 5 years we have identified the importance of having scalable systems put in place. We have proposed an approach that can be used to evaluate the scalability via synthetic datasets. SyntheD is a highly flexible Java-based program as it enables datasets to be created in various ways according to the size of data required by the user. Along with the experiments, we have proven that the program can generate at least if not more than 10 billion ratings which currently represents the largest dataset we are using for this project. The next step following the creation of our synthetic data is to partition the new dataset similarly to the MovieLens and Webscope data; then to use the partitions to train and test the performance of the ALSWR java code.

Our future works on the program SyntheD will consist of improving the time performance taken to produce a new dataset file. Presently to create 10 billion ratings with 10 million users and 189 thousand item, the program takes in total nearly 3 hours. Generating data can be time-consuming. Hence optimizing the program for better results could make a significant difference. Presently the main limit in terms of going to larger dataset is the requirement to keep the user model (for example) in the memory of a single computer. One easy fix would be to distribute the models over the memory of a cluster. Our plan is also to make the program open-source on Github or Bitbucket for instance.

Our current simple strategy of generating a low rank matrix certainly has scope to be made more realistic; for example to give a more realistic distribution of ratings for popular items, or by explicitly modelling clustering in preferences of users.

Chapter 8

Conclusion

8.1 Introduction

This chapter summarises the entire research and outlines the contributions. We go through the steps followed for the completion of our work and discuss our findings as well as limitations. We conclude this chapter by indicating future works that could be carried on.

“There is no real ending. It’s just the place where you stop the story.”

-Frank Herbert

8.2 Summary of the Research

Recommender systems have taken significant importance with time and more particularly with social media networks which are experiencing rapid data growth as their number of users increase. Recommendations systems are expected to be fast and scalable but on another hand, suggestions that are sent out must be relevant to the targeted audience. In our research, we have shifted our focus on high-performance computing methods such as MPJ Express which has been integrated with Hadoop, a framework allowing distributed data processing on large datasets.

Various datasets were used during this thesis. These first consisted of the MovieLens data holding 20 million rating and the Yahoo Webscope with more than 700 Million ratings. Then as we move toward social media scale, we have developed a program allowing to create synthetic datasets: SyntheD. Finding large datasets in the same scale of social media networks can be hard and going

through surveys and questionnaires can be time-consuming. Hence the importance of having a program that can generate data to use for experimentation purpose and to test the scalability feature of recommender systems.

By using MPJ Express and Hadoop we have adapted the collaborative filtering algorithm ALSWR and run the code under YARN nodes. Each dataset was first added to the Hadoop distributed file system (HDFS) then partitioned. The partitioning of the datasets was originally made using a serial java application then by using MapReduce to improve the time performance. However, there were some shortcomings of the initial implementation of ALSWR in MPJ Express. Notably the use of a separate MapReduce phase for partitioning the data, while methodologically interesting as an example of a multistage Hadoop/HDFS pipeline involving MPJ, was not very convenient in practice and introduced some issues when comparing with other implementations that do not require the extra stage. To tackle this challenge, the next version of the MPJ Express code read local blocks of the complete HDFS rating file (in a similar way to how MapReduce map tasks process local blocks) then use internal MPI communications to repartition data in these blocks to appropriate target nodes for parallel processing. As a result of this approach, the implementation of ALSWR became much faster than using a separate MapReduce phase.

The results of the implementation have been benchmarked in two ways. For each dataset, we have first measured the time performance then calculated the parallel speedup across the number of processes available. We have compared the outcome of our tests from the MovieLens and Yahoo datasets with the ALS implementation on the same datasets but using Apache Mahout, Apache Spark and Apache Giraph frameworks. We have demonstrated that by employing ALS with MPJ Express our results outperform the performance of other frameworks and achieve a better parallel speedup. MPJ Express is nevertheless limited by the fact that it involves more complexity on the programming side. We have furthermore measured the accuracy of the predictions by calculating the root mean square error (RMSE). We have run some tests with different values for the regularization parameter (λ). The better output was obtained with the value of λ tuned to 0.05 which gave us a value of RMSE equal to 1.0328.

8.3 Contributions

Our contributions to the research area comprise:

- The development of suitable approaches for highly scalable recommender systems with application to the scale of data characteristic of social media.
- The demonstration of the benefits of integrating MPJ Express platform into the Hadoop ecosystem, with the possibility of using MPJ Express as a phase in a processing pipeline and including traditional Hadoop platforms such as MapReduce.
- The evaluation of our approach to HPC inspired parallel recommender algorithms in Hadoop in comparison with other frameworks like Apache Mahout, Spark and Giraph showing the benefit of our approach.
- Our test strategies for efficiently reading HDFS data into Single Program, Multiple Data (SPMD) parallel programs including MPI or MPJ.
- The implementation of a method to generate synthetic datasets that can reach beyond billions of ratings and the evaluation of the performance of our approach on a large synthetic dataset.
- The demonstration of a high impact on HPC technologies with MPJ Express by achieving good parallel speedup and demonstrating a good error metric as compared to other well-known frameworks.

8.4 Future Works

In term of future work, we need to evaluate alternative parallel organizations of the recommender code, like the rotational hybrid approach described in [52]. Preliminary analysis suggests that implementation of similar schemes in MPI style may benefit from extensions to the standard set of MPI collectives, currently embodied in MPJ Express. Such an extended library could form part of a future data-centric version of MPJ Express that builds on experiences of MPI processing in the Hadoop environment. More prosaically the experiments in this thesis suggest the imminent need to deal with communication buffers significantly larger than those currently supported. In experiments reported in the thesis, we were already increasing these buffer sizes close to their limit.

Future works could also involve improving the accuracy of the predictions. We obtained a RMSE equal to 1.0328 compared to the collaborative filtering for the Netflix prize which accomplished a score of 0.8985 for their RMSE [101]. We believe that more techniques can be adopted to bring down the RMSE and by

consequent to provide better recommendations. Although our primary option was to use the RMSE approach as our error metric, some experimentation with the MAE (mean absolute error) could be done to compare the outcome. Similarly, although we have focused the research on ALSWR, other algorithms such as SGD could be used to test the results of the RMSE and assess its performance against ALSWR.

The SyntheD program could furthermore be extended to improve its time performance when generating datasets. We have mentioned in chapter 7 that currently to generate 10 billion ratings with 10 million users and 189 thousand items, the program takes two 2 hours and 54 minutes to complete. More experimentation could also be done by reproducing bigger datasets with more users toward the scale of social media. Facebook the most popular social media network currently has more than 2.7 billion users. Our next target would be to generate a dataset with a similar number of users. As the objective would be to generate larger data on a shorter period of time, it would be interesting to implement a parallel computation on the synthetic data generator code to speedup the overall data creation process. The data generated could also be tested with other frameworks such as Apache Spark and Apache Giraph to analyse their performance on a larger scale.

To solve some issues we had due to insufficient memory and more particularly for experiments on data like SyntheD, the tests could be reproduced on commercial clusters as they are easily scalable based on the memory capacity required. Surely this comes at a cost; however, one could benefit from a university grant/-funding with the right research prospect. Having access to such clusters would benefit the full exploitation of MPJ Express and other frameworks on datasets synthetically generated. For instance in section 7.5 we mentioned that it was impossible to run the program on a small number of nodes because just by storing the distributed ratings 100GB of memory was used while the cluster has a total of 32GB per node and 13nodes available, therefore not leaving much memory for the computation of MPJ Express program. Commercial clusters would surely help overcome this problem.

Bibliography

- [1] Apache Giraph. <http://giraph.apache.org/>, 2014. [accessed 19-January-2018].
- [2] Apache Hadoop. <http://hadoop.apache.org>, 2010. [accessed 21-September-2017].
- [3] Apache Mahout. <https://mahout.apache.org/>, 2017. [accessed 30-January-2018].
- [4] Apache Spark. <https://spark.apache.org/>, 2014. [accessed 30-January-2018].
- [5] Apache Spark Tutorial. https://www.tutorialspoint.com/apache_spark/apache_spark_rdd.htm, 2017. [accessed 19-January-2018].
- [6] Matthew Barker. Evaluation of apache giraph, using a collaborative filtering algorithm. Master Thesis University of Portsmouth, 2018.
- [7] Blaise Barney. Introduction to parallel computing. https://computing.llnl.gov/tutorials/parallel_comp/. [accessed 01-September-2016].
- [8] Jesús Bobadilla, Fernando Ortega, Antonio Hernando, and Abraham Gutiérrez. Recommender systems survey. *Knowledge-based systems*, 46:109–132, 2013.
- [9] Thomas Bredillet. Lessons learned at instagram stories and feed machine learning. <https://instagram-engineering.com/lessons-learned-at-instagram-stories-and-feed-machine-learning-54f3aaa09e56>, 2018.
- [10] John S Breese, David Heckerman, and Carl Kadie. Empirical analysis of predictive algorithms for collaborative filtering. In *Proceedings of the*

- Fourteenth conference on Uncertainty in artificial intelligence*, pages 43–52. Morgan Kaufmann Publishers Inc., 1998.
- [11] Fidel Cacheda, Víctor Carneiro, Diego Fernández, and Vreixo Formoso. Comparison of collaborative filtering algorithms: Limitations of current techniques and proposals for scalable, high-performance recommender systems. *ACM Transactions on the Web*, 5(1):2:1–2:33, February 2011.
 - [12] INFO CENTER. What is caffe2? <https://caffe2.ai/docs/caffe-migration.html>. [accessed 20-September-2019].
 - [13] INFO CENTER. Instagram launches. <https://instagram-press.com/blog/2010/10/06/instagram-launches-2/>, 2010.
 - [14] INFO CENTER. Instagram + facebook. <https://instagram-press.com/blog/2012/04/09/instagram-facebook/>, 2012.
 - [15] INFO CENTER. The instagram community hits 80 million users. <https://instagram-press.com/blog/2012/07/26/the-instagram-community-hits-80-million-users/>, 2012.
 - [16] INFO CENTER. A quick walk through our history as a company. <https://instagram-press.com/our-story/>, 2019.
 - [17] Facebook Help Centre. <https://www.facebook.com/help/163810437015615>. [accessed 12-September-2019].
 - [18] T. Chai and R. R. Draxler. Root mean square error (rmse) or mean absolute error (mae)? – arguments against avoiding rmse in the literature. *Geoscientific Model Development*, 7(3):1247–1250, 2014.
 - [19] Tianfeng Chai and Roland R Draxler. Root mean square error (rmse) or mean absolute error (mae)? – arguments against avoiding rmse in the literature. *Geoscientific model development*, 7(3):1247–1250, 2014.
 - [20] Wei Chen, Zhendong Niu, Xiangyu Zhao, and Yi Li. A hybrid recommendation algorithm adapted in e-learning environments. *World Wide Web*, 17(2):271–284, 2014.
 - [21] Avery Ching. Scaling Apache Giraph to a trillion edges. <https://www.facebook.com/notes/facebook-engineering/scaling-apache-giraph-to-a-trillion-edges/10151617006153920>, 2013.

- [22] Frederica Darema. The spmd model: Past, present and future. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pages 1–1. Springer, 2001.
- [23] Frederica Darema. *SPMD Computational Model*, pages 1933–1943. Springer US, Boston, MA, 2011.
- [24] Spark rdd operations-transformation & action with example. <https://dataflair.training/blogs/spark-rdd-operations-transformations-actions/>. [accessed 11-June-2018].
- [25] Santo Domingo, Joel. Comparing dual-core vs. quad-core cpus. *Laptop Computers & Notebook Reviews*, 2012. [accessed 05-September-2016].
- [26] Xin Dong, Lei Yu, Zhonghuo Wu, Yuxia Sun, Lingfeng Yuan, and Fangxi Zhang. A hybrid collaborative filtering model with deep structure for recommender systems. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.
- [27] John R Douceur. The sybil attack. In *International workshop on peer-to-peer systems*, pages 251–260. Springer, 2002.
- [28] Rui Maximo Esteves, Rui Pais, and Chunming Rong. K-means clustering in the cloud—a mahout test. In *Advanced Information Networking and Applications (WAINA), 2011 IEEE Workshops of International Conference on*, pages 514–519. IEEE, 2011.
- [29] Sinziana Maria Filip. *A scalable graph pattern matching engine on top of Apache Giraph*. PhD thesis, Master’s thesis, VU University Amsterdam, 2014.
- [30] The Apache Software Foundation. Apache giraph guide. http://giraph.apache.org/quick_start.html. [accessed 02-October-2018].
- [31] The Apache Software Foundation. Hadoop cluster setup. <https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/>, 2018. [updated 25-January-2019].
- [32] Josh Gallant. 48 eye-opening linkedin statistics for b2b marketers in 2019. <https://foundationinc.co/lab/b2b-marketing-linkedin-stats/>, 2019.

- [33] G Geetha, M Safa, C Fancy, and D Saranya. A hybrid approach using collaborative filtering and content based filtering for recommender system. In *Journal of Physics: Conference Series*, volume 1000, page 012101. IOP Publishing, 2018.
- [34] David Goldberg, David Nichols, Brian M Oki, and Douglas Terry. Using collaborative filtering to weave an information tapestry. *Communications of the ACM*, 35(12):61–70, 1992.
- [35] Ana Gotter. How the instagram algorithm works (and where your strategy needs to shift). <https://www.shopify.com/blog/instagram-algorithm>, 2019.
- [36] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel computing*, 22(6):789–828, 1996.
- [37] Datasets — GroupLens. <http://grouplens.org/datasets/>, 2015. [accessed 14-December-2016].
- [38] Rong Gu, Xiaoliang Yang, Jinshuang Yan, Yuanhao Sun, Bing Wang, Chunfeng Yuan, and Yihua Huang. Shadoop: Improving mapreduce performance by optimizing job execution mechanism in hadoop clusters. *Journal of parallel and distributed computing*, 74(3):2166–2179, 2014.
- [39] Qi Guo, C Geyik, Sahim, Cagri Ozcaglar, Ketan Thakka, Nadeem Anjum, and Krishnaram Kenthapadi. The ai behind linkedin recruiter search and recommendation systems. <https://engineering.linkedin.com/blog/2019/04/ai-behind-linkedin-recruiter-search-and-recommendation-systems>, 2019.
- [40] Viet Ha-Thuc, Ye Xu, Satya Pradeep Kanduri, Xianren Wu, Vijay Dialani, Yan Yan, Abhishek Gupta, and Shakti Sinha. Search by ideal candidates: Next generation of talent search at linkedin. In *Proceedings of the 25th International Conference Companion on World Wide Web*, pages 195–198. International World Wide Web Conferences Steering Committee, 2016.
- [41] Apache Hadoop. Tuning the java heap. <http://hadoop.apache.org/docs/r2.4.1/hadoop-mapreduce-client/hadoop-mapreduce-client-core/mapred-default.xml>, 2018. [accessed 11-April-2019].

- [42] Minyang Han and Khuzaima Daudjee. Giraph unchained: barrierless asynchronous parallel execution in pregel-like graph processing systems. *Proceedings of the VLDB Endowment*, 8(9):950–961, 2015.
- [43] Duane Hanselman and BC Littlefield. Mastering MATLAB 5: A comprehensive tutorial and reference prentice hall. *Upper Saddle River, NJ, USA*, 1997.
- [44] Derrick Harris. Facebook’s trillion-edge, Hadoop-based and open source graph-processing engine. *GigaOm*, 2013.
- [45] S. Hettich and S. D Bay. Uci kdd archive. <http://kdd.ics.uci.edu>, 1999. [accessed 28 September 2017].
- [46] Torsten Hoefler and Roberto Belli. Scientific benchmarking of parallel computing systems: twelve ways to tell the masses when reporting performance results. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*, page 73. ACM, 2015.
- [47] Mohammed Ibrahim, Stephen Mele, Daniel Pulitano, Nilkamal Weerasinghe, Moon Kim, Marshall Peters, Bryan Robbins, and Yuriy Yankop. Datagenerator. <https://finraos.github.io/DataGenerator/>. [accessed 18-October-2018].
- [48] Data analytics acceleration library. <https://software.intel.com/en-us/Intel-daal>, 2017. [accessed 21-October-2017].
- [49] Mansoor Iqbal. Linkedin usage and revenue statistics (2018). <https://www.businessofapps.com/data/linkedin-statistics/>, 2019.
- [50] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys ’07, pages 59–72, New York, NY, USA, 2007. ACM.
- [51] FO Isinkaye, YO Folajimi, and BA Ojokoh. Recommendation systems: Principles, methods and evaluation. *Egyptian Informatics Journal*, 16(3):261–273, 2015.

- [52] Maja Kabiljo and Aleksandar Ilic. Recommending items to more than a billion people. <https://code.facebook.com/posts/861999383875667/recommending-items-to-more-than-a-billion-people/>, 2015. [accessed 30-December-2017].
- [53] Efthalia Karydi and Konstantinos Margaritis. Parallel and distributed collaborative filtering: A survey. *ACM Computing Surveys (CSUR)*, 49(2):37, 2016.
- [54] Ben Ken. Generatedata.com. <https://www.generatedata.com/>. [accessed 16-October-2018].
- [55] Joseph A Konstan, Bradley N Miller, David Maltz, Jonathan L Herlocker, Lee R Gordon, and John Riedl. Grouplens: applying collaborative filtering to usenet news. *Communications of the ACM*, 40(3):77–87, 1997.
- [56] Benjamin Letham, Brian Karrer, Guilherme Ottoni, and Eytan Bakshy. Constrained bayesian optimization with noisy experiments. *Bayesian Anal.*, 14(2):495–519, 06 2019.
- [57] Introduction to als recommendations with hadoop. <https://mahout.apache.org/users/recommender/intro-als-hadoop.html>. [accessed 22-June-2018].
- [58] Faraz Makari, Christina Teflioudi, Rainer Gemulla, Peter Haas, and Yannis Sismanis. Shared-memory and shared-nothing stochastic gradient descent algorithms for matrix completion. *Knowledge and Information Systems*, 42(3):493–523, 2015.
- [59] Grzegorz Malewicz, Matthew H Austern, Aart JC 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*, pages 135–146. ACM, 2010.
- [60] MPJ Express. <http://mpjexpress.org>, 2015. [accessed 18-January-2018].
- [61] MrForms. Freedatagenerator. <http://www.freedatagenerator.com/>, 2014.

- [62] Andrew Musselman. Building a recommender with apache mahout on amazon elastic mapreduce (emr). <https://aws.amazon.com/blogs/big-data/building-a-recommender-with-apache-mahout-on-amazon-elastic-mapreduce-emr/>, 2014. [accessed 30-January-2019].
- [63] Facebook Newsroom. Bringing people closer together. <https://newsroom.fb.com/news/2018/01/news-feed-fyi-bringing-people-closer-together/>, 2018.
- [64] Facebook Newsroom. <https://newsroom.fb.com/news/2019/09/facebook-dating/>, 2019.
- [65] Facebook Newsroom. Why am i seeing this? we have an answer for you. <https://newsroom.fb.com/news/2019/03/why-am-i-seeing-this/>, 2019.
- [66] Linkedin Newsroom. Linkedin announces third quarter 2015 results. <https://news.linkedin.com/2015/linkedin-announces-third-quarter-2015-results>, 2015.
- [67] Linkedin Newsroom. Linkedin announces third quarter 2016 results. <https://news.linkedin.com/2016/linkedin-announces-third-quarter-2016-results>, 2016.
- [68] Linkedin Newsroom. About linkedin. <https://news.linkedin.com/about-us>, n.d. [accessed 15-09-2019].
- [69] Linkedin Newsroom. Statistics. <https://news.linkedin.com/about-us#statistics>, n.d. [accessed 21-09-2019].
- [70] Oracle. Java. https://java.com/en/download/help/linux_x64_install.xml. [accessed 12-November-2018].
- [71] Oracle. Tuning the java heap. <https://docs.oracle.com/cd/E19159-01/819-3681/abeii/index.html>, 2010. [accessed 12-March-2019].
- [72] Oracle. Understand the outofmemoryerror exception. <https://docs.oracle.com/javase/8/docs/technotes/guides/troubleshoot/memleaks002.html>, 2018. [accessed 12-March-2019].
- [73] Cagri Ozcaglar, Sahin Geyik, Brian Schmitz, Prakhar Sharma, Alex Shelkovnykov, Yiming Ma, and Erik Buchanan. Entity personalized tal-

- ent search models with tree interaction features. In *The World Wide Web Conference*, pages 3116–3122. ACM, 2019.
- [74] Rohan Ramanath, Hakan Inan, Gungor Polatkan, Bo Hu, Qi Guo, Cagri Ozcaglar, Xianren Wu, Krishnaram Kenthapadi, and Sahin Cem Geyik. Towards deep and representation learning for talent search at linkedin. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*, pages 2253–2261. ACM, 2018.
- [75] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 24*, pages 693–701. Curran Associates, Inc., 2011.
- [76] Investor Relations. Facebook reports first quarter 2015 results. <https://investor.fb.com/investor-news/press-release-details/2015/Facebook-Reports-First-Quarter-2015-Results/default.aspx>, 2015. [accessed 05-September-2019].
- [77] Investor Relations. Facebook reports fourth quarter and full year 2015 results. <https://investor.fb.com/investor-news/press-release-details/2016/Facebook-Reports-Fourth-Quarter-and-Full-Year-2015-Results/default.aspx>, 2015.
- [78] Investor Relations. Facebook reports fourth quarter and full year 2016 results. <https://investor.fb.com/investor-news/press-release-details/2017/facebook-Reports-Fourth-Quarter-and-Full-Year-2016-Results/default.aspx>, 2016.
- [79] Investor Relations. Facebook reports fourth quarter and full year 2017 results. <https://investor.fb.com/investor-news/press-release-details/2018/facebook-reports-fourth-quarter-and-full-year-2017-results/default.aspx>, 2017.
- [80] Investor Relations. Facebook reports fourth quarter and full year 2018 results. <https://investor.fb.com/investor-news/press-release-details/2019/Facebook-Reports-Fourth-Quarter-and-Full-Year-2018-Results/default.aspx>, 2018.

- [81] Investor Relations. <https://investor.fb.com/investor-news/press-release-details/2019/Facebook-Reports-Second-Quarter-2019-Results/default.aspx>, 2019. [accessed 05-September-2019].
- [82] Telefonica Research. Okapi. <https://github.com/grafos-ml/okapi>. [accessed 06-October-2018].
- [83] Paul Resnick and Hal R Varian. Recommender systems. *Communications of the ACM*, 40, 1997.
- [84] Elaine Rich. User modeling via stereotypes. *Cognitive science*, 3(4):329–354, 1979.
- [85] Laila Safoury and Akram Salah. Exploiting user demographic attributes for solving cold-start problem in recommender system. *Lecture Notes on Software Engineering*, 1(3):303–307, 2013.
- [86] Bikas Saha, Hitesh Shah, Siddharth Seth, Gopal Vijayaraghavan, Arun Murthy, and Carlo Curino. Apache tez: A unifying framework for modeling and building data processing applications. In *Proceedings of the 2015 ACM SIGMOD international conference on Management of Data*, pages 1357–1369. ACM, 2015.
- [87] Sherif Sakr. Processing large-scale graph data: A guide to current technology. *IBM Developerworks*, page 15, 2013.
- [88] Sebastian Schelter, Christoph Boden, Martin Schenck, Alexander Alexandrov, and Volker Markl. Distributed matrix factorization with mapreduce using a series of broadcast-joins. In *Proceedings of the 7th ACM Conference on Recommender Systems*, pages 281–284. ACM, 2013.
- [89] Amazon Web Services. Apache mahout. <https://docs.aws.amazon.com/emr/latest/ReleaseGuide/emr-mahout.html>, 2019. [accessed 30-January-2019].
- [90] Sanjeeran Sivapalan, Alireza Sadeghian, Hossein Rahnema, and Asad M Madni. Recommender systems in e-commerce. *Communications of the ACM*, pages 179–184, 2014.
- [91] DATAFLAIR TEAM. Hadoop partitioner – internals of mapreduce partitioner. <https://data-flair.training/blogs/hadoop-partitioner-tutorial/>, 2017. [updated 21-November-2018].

- [92] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in mpich. *The International Journal of High Performance Computing Applications*, 19(1):49–66, 2005.
- [93] Bryan Carpenter Wakeel Ahmad and Aamir Shafi. Collective asynchronous remote invocation (cari): A high-level and efficient communication api for irregular applications. *Procedia Computer Science*, 4:26 – 35, 2011. International Conference On Computational Science, ICCS 2011.
- [94] Weidong Wang, Chunhua Liao, Liqiang Wang, Daniel J Quinlan, and Wei Lu. Hbtm: A heartbeat-based behavior detection mechanism for posix threads and openmp applications. *arXiv preprint arXiv:1512.00665*, 2015.
- [95] Karlijn Willems. Apache spark tutorial: ML with pyspark. <https://www.datacamp.com/community/tutorials/apache-spark-tutorial-machine-learning>, 2017. [accessed 31-January-2019].
- [96] Webscope. <https://research.yahoo.com/>, 2006. [accessed 14-December-2016].
- [97] Hsiang-Fu Yu, Cho-Jui Hsieh, Si Si, and Inderjit Dhillon. Scalable coordinate descent approaches to parallel matrix factorization for recommender systems. In *Data Mining (ICDM), 2012 IEEE 12th International Conference on*, pages 765–774. IEEE, 2012.
- [98] Hamza Zafar, Farrukh Aftab Khan, Bryan Carpenter, Aamir Shafi, and Asad Waqar Malik. Mpj express meets yarn: Towards java hpc on hadoop systems. *Procedia Computer Science*, 51:2678 – 2682, 2015. International Conference On Computational Science, ICCS 2015.
- [99] 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.
- [100] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing, HotCloud’10*, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

- [101] Yunhong Zhou, Dennis Wilkinson, Robert Schreiber, and Rong Pan. Large-scale parallel collaborative filtering for the netflix prize. *Lecture Notes in Computer Science*, 5034:337–348, 2008.
- [102] Mark Zuckerberg. <https://www.facebook.com/zuck/posts/10102329188394581?pnref=story>, 2015. [accessed 11-March-2019].

Appendix A

Ethical Review Certificate



Certificate of Ethics Review

| | |
|-------------------|---|
| Project Title: | Scalable Recommender System for very Large Dataset with Application to Social Media |
| User ID: | 790357 |
| Name: | Christina Pierrette Abilali Diedhiou |
| Application Date: | 12/09/2016 15:53:09 |

You must download your certificate, print a copy and keep it as a record of this review.

It is your responsibility to adhere to the University Ethics Policy and any Department/School or professional guidelines in the conduct of your study including relevant guidelines regarding health and safety of researchers and University Health and Safety Policy.

It is also your responsibility to follow University guidance on Data Protection Policy:

- General guidance for all data protection issues
- University Data Protection Policy

You are reminded that as a University of Portsmouth Researcher you are bound by the UKRIO Code of Practice for Research; any breach of this code could lead to action being taken following the University's Procedure for the Investigation of Allegations of Misconduct in Research.

Any changes in the answers to the questions reflecting the design, management or conduct of the research over the course of the project must be notified to the Faculty Ethics Committee. Any changes that affect the answers given in the questionnaire, not reported to the Faculty Ethics Committee, will invalidate this certificate.

This ethical review should not be used to infer any comment on the academic merits or methodology of the project. If you have not already done so, you are advised to develop a clear protocol/proposal and ensure that it is independently reviewed by peers or others of appropriate standing. A favourable ethical opinion should not be perceived as permission to proceed with the research; there might be other matters of governance which require further consideration including the agreement of any organisation hosting the research.

GovernanceChecklist

A1-BriefDescriptionOfProject: The increasing interest in Java as a programming language for parallel computing has brought many innovations in the multi-core era, notably in High Performance Computing (HPC). A leading programming model remains Message Passing in Java (MPJ). While MPJ has various benefits such as high portability, large scale support and possibility to execute programs either on shared or distributed memory it still encounters some issues regarding communications overhead. This causes the speed of data loading to slow down as the scale of data increases. An approach to resolve this concern is to integrate MPJ with Hadoop and Map Reduce in order to obtain

Certificate Code: 40C8-9C32-6D44-399F-8FD4-1C4A-0631-2279 Page 1

Appendix B

UPR16 Form

FORM UPR16

Research Ethics Review Checklist

Please include this completed form as an appendix to your thesis (see the Research Degrees Operational Handbook for more information)



| | | | |
|---|---|---|--|
| Postgraduate Research Student (PGRS) Information | | Student ID: | UP790357 |
| PGRS Name: | Christina Pierrette Abilali Diedhiou Sowinski | | |
| Department: | Computing | First Supervisor: | Dr Bryan Carpenter |
| Start Date: (or progression date for Prof Doc students) | 01/10/2015 | | |
| Study Mode and Route: | Part-time <input type="checkbox"/> Full-time <input checked="" type="checkbox"/> | MPhil <input type="checkbox"/> PhD <input checked="" type="checkbox"/> | MD <input type="checkbox"/> Professional Doctorate <input type="checkbox"/> |
| Title of Thesis: | Towards Platforms for Improved Recommender Systems at Social Media Scale | | |
| Thesis Word Count: (excluding ancillary data) | 27215 | | |
| <p>If you are unsure about any of the following, please contact the local representative on your Faculty Ethics Committee for advice. Please note that it is your responsibility to follow the University's Ethics Policy and any relevant University, academic or professional guidelines in the conduct of your study</p> <p>Although the Ethics Committee may have given your study a favourable opinion, the final responsibility for the ethical conduct of this work lies with the researcher(s).</p> | | | |
| UKRIO Finished Research Checklist: (If you would like to know more about the checklist, please see your Faculty or Departmental Ethics Committee rep or see the online version of the full checklist at: http://www.ukrio.org/what-we-do/code-of-practice-for-research/) | | | |
| a) Have all of your research and findings been reported accurately, honestly and within a reasonable time frame? | YES | <input checked="" type="checkbox"/> | |
| | NO | <input type="checkbox"/> | |
| b) Have all contributions to knowledge been acknowledged? | YES | <input checked="" type="checkbox"/> | |
| | NO | <input type="checkbox"/> | |
| c) Have you complied with all agreements relating to intellectual property, publication and authorship? | YES | <input checked="" type="checkbox"/> | |
| | NO | <input type="checkbox"/> | |
| d) Has your research data been retained in a secure and accessible form and will it remain so for the required duration? | YES | <input checked="" type="checkbox"/> | |
| | NO | <input type="checkbox"/> | |
| e) Does your research comply with all legal, ethical, and contractual requirements? | YES | <input checked="" type="checkbox"/> | |
| | NO | <input type="checkbox"/> | |
| Candidate Statement: | | | |
| I have considered the ethical dimensions of the above named research project, and have successfully obtained the necessary ethical approval(s) | | | |
| Ethical review number(s) from Faculty Ethics Committee (or from NRES/SCREC): | 40C8-9C32-6D44-399F-8FD4-1C4A-0631-2279 | | |
| If you have <i>not</i> submitted your work for ethical review, and/or you have answered 'No' to one or more of questions a) to e), please explain below why this is so: | | | |
| <div style="border: 1px solid black; height: 20px; width: 100%;"></div> | | | |
| Signed (PGRS): | | | Date: 29/09/2019 |