View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Digital Commons@Wayne State University

Wayne State University

Wayne State University Theses

1-1-2015

Performance Comparison Of Two Data Mining Algorithms On Big Data Platforms

Md Rajiur Rahman Raju
Wayne State University,

Follow this and additional works at: https://digitalcommons.wayne.edu/oa_theses

Part of the Computer Sciences Commons

Recommended Citation

Raju, Md Rajiur Rahman, "Performance Comparison Of Two Data Mining Algorithms On Big Data Platforms" (2015). *Wayne State University Theses*. 476.
https://digitalcommons.wayne.edu/oa_theses/476

This Open Access Thesis is brought to you for free and open access by DigitalCommons@WayneState. It has been accepted for inclusion in Wayne State University Theses by an authorized administrator of DigitalCommons@WayneState.

PERFORMANCE COMPARISON OF TWO DATA MINING ALGORITHMS ON BIG DATA PLATFORMS

by

MD RAJIUR RAHMAN RAJU

THESIS

Submitted to the Graduate School

of Wayne State University,

Detroit, Michigan

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

2015

MAJOR: COMPUTER SCIENCE

Approved By:

Advisor Date

DEDICATION

To my father Md Siddiqur Rahmnan, who is the biggest source of my inspiration.

ACKNOWLEDGEMENTS

I take this opportunity to offer my sincere gratitude to my advisor Dr Chandan K. Reddy for his guidance and continued support. His immense knowledge and advice helped me throughout my Masters study. Without his direction, this work would not be possible. I am really grateful for the kind opportunity he gave me. I am confident that the skills and knowledge that I gained by being part of Dr Reddy's research team will be helpful for the rest of my life.

Besides, I would like to thank Dr Zaki Malik and Dr Zichun Zhong for being part of my thesis committee.

I would also like to thank my fellow labmates for providing valuable opinions. I really appreciate their friendly behaviour during my days in the lab.

Finally, I would like to thank my family members for supporting me spiritually all the time.

TABLE OF CONTENTS

Dedication ii

Acknowledgements iii

List of Tables vi

List of Figures vi

Chapter 1: Introduction 1

Chapter 2: Background 4

2.1 Hadoop 4

2.1.1 Programming Model 6

2.1.2 HDFS 6

2.1.3 MapReduce 9

2.1.4 YARN 11

2.2 Spark 13

2.2.1 RDDs 14

2.2.2 Parallel Operations 15

2.3 GPU 16

2.3.1 GPU and CPU Interaction 17

2.3.2 GPU Thread Organization 17

2.3.3 GPU Memory Hierarchy 19

2.4 Multicore CPU 23

Chapter 3: Data Mining Algorithms and their Implementations 26

3.1 K-means clustering 26

3.1.1 K-means on MapReduce 27

3.1.2 K-means on Spark 29

3.1.3 K-means on GPU 30

3.1.4 K-means on Multicore CPU 32

3.2 K-nearest neighbor classification 34

3.2.1 K-nn on MapReduce 35
3.2.2 K-nn on Spark 36
3.2.3 K-nn on GPU 37
3.2.4 K-nn on Multicore 38
Chapter 4: Results and Discussion 40
4.1 Dataset Used 40
4.2 Experimental Setup 41
4.3 Results and Discussion 42
Chapter 5: Conclusion 46
References 47
Abstract 50
Autobiographical Statement 52

LIST OF TABLES

Table. 2.1 MapReduce pipeline. 9

Table. 2.2 Configurations of a standard CPU and GPU. 16

Table. 2.3 GPU memories in different compute capabilities 22

Table. 2.4 Properties of GPU memory types 23

Table. 4.1 Description of the AWS Instances used in our experiments. 41

Table. 4.2 Description of the AWS Instances used for Scalability Test. 42

LIST OF FIGURES

Figure. 2.1 Components of Hadoop. Figure taken from [2] 6

Figure. 2.2 HDFS Architecture. Figure taken from [8]. 7

Figure. 2.3 Flowchart of writing a file in HDFS. Figure taken from [25]. . . . 8

Figure. 2.4 MapReduce ecosystem. 10

Figure. 2.5 Hadoop YARN architecture. Figure taken from [5]. 12

Figure. 2.6 Spark cluster overview. 14

Figure. 2.7 An illustration of interconnection between GPU and CPU. . . . 18

Figure. 2.8 Memory hierarchy in GPU. Figure taken from [4] 21

Figure. 2.9 Multicore CPU architecture. 25

Figure. 3.1 K-means flow of execution on MapReduce. 28

Figure. 3.2 K-means flow of execution on Spark. 30

Figure. 3.3 K-means flow of execution on GPU. 31

Figure. 3.4 K-means flow of execution on Multicore CPU. 33

Figure. 3.5 K-nn flow of execution on MapReduce. 35

Figure. 3.6 K-nn flow of execution on Spark. 37

Figure. 3.7 K-nn flow of execution on GPU. 37

Figure. 3.8 K-nn flow of execution on Multicore CPU. 39

Figure. 4.1 Comparison of running time of K-means algorithm on five different inds of platforms (including single core machine) with varying number of clusters k and number of features. (a) varying the number of features with k=10, (b) varying number of clusters k with 70K features. 43

Figure. 4.2 Running time of K-menas algorithm with large dataset for scalability test. 44

Figure. 4.3 Comparision of running time for K-nn algorithm on five platforms. 45

Chapter 1: Introduction

In recent years, we have seen a growing interest in big data analysis. Different sectors like retail business, medical informatics and imaging, web and social media, manufacturing industries, government agencies collect tremendous amounts of data every year. International Data Corporation (IDC) reported that in the year 2011, total created and copied data had a size of approximately 1.8 zettabytes ($\approx 10^{21}B$) - which is a growth by nearly nine folds [12]. Cheaper and sophisticated data storage media have also helped this data explosion. To efficiently process and extract knowledge from this enormous wealth of digital information, we need scalable big data technologies and platforms. Data mining algorithms are also becoming more and more data driven. Effective technologies can enable us taking better decisions faster.

With the rise of big data, the traditional tools for data processing have not been able to keep up the pace. Researchers around the world are trying to build the right hardware and software platforms harder than ever now. There are several big data platforms that exist now a days with different characteristics. They deal with the volume, velocity and variety of the data differently.

The "right platform" depends on the problem at hand and choosing it requires a great deal of knowledge [7]. Typically, the two main concerns regarding platform choice are:

- How quickly can we process the data?
- How big is the data to be processed?

The platforms are mainly of two kinds, horizontal scaling and vertical scaling [22]. In horizontal scaling, the workload is distributed across multiple machines or nodes. This is also know as "scale out" method. Some examples of these horizontal scaling

platforms include Apache Hadoop, Spark, Peer-to-peer networks, etc. The primary advantage of horizontal scaling is that we can employ our regular commodity machines in the big data paradigm. We can scale out the system as much as needed and the financial cost is relatively low. The software in this case is a major drawback since it has to handle the distributed scenario and the programming requires special expertise. The available software options are also limited in number. On the other hand, vertical scaling involves adding more processors and memory into a single machine. This is also known as "scale up" and examples of this category are Graphics Processing Unit (GPU), Field Programmable Gate Arrays (FPGA), High Performance Cluster (HPC), Multicore CPUs etc. Its easy to manage and install more hardware in a single machine. The system has to be powerful enough to accommodate the extra hardware and it generally requires substantial investments. However, in vertical scaling it is not possible to add more hardware after a certain limit.

To the best of our knowledge there is no comprehensive research which compares all these platforms. In this work, we attempt to compare the performances of four big platforms, namely, Hadoop, Spark, GPU, and Multicore CPU and one regular single core machine without leveraging any parallelization technique.

We systematically compare these platforms using the Amazon Web Service (AWS) cloud computing platform[1] so that the cost per hour needed to run the algorithms on the platforms remains similar across all the platforms. It is important to have this requirement as a constraint on these platforms since this will enable us to compare and evaluate the platforms in the most optimal manners. This thesis aims at understanding *what will be the most optimal choice of platform for running the big data computations under certain budget limitations and a particular problem at hand.*

We discuss different aspects including strengths and weaknesses of all the four platforms chosen for our analysis. We compare these platforms using two prominent

[1]http://aws.amazon.com/

data mining algorithms, namely, K-means clustering and K-nearest neighbor classification and discuss specific details of these algorithms' implementation on each of these platforms along with high-level pseudocodes. We also ensure that all the platforms are performing the same tasks and producing the same result. We provide several insights into the best possible implementations of these algorithms and systematically compare the benefits and drawbacks of each of these platforms. Since many other data mining algorithms either use these two methods during pre-processing or as an integral component, we hope that our analysis will have impact in many other applications and algorithms beyond the ones that are being reported in this thesis. We analyze the performances in terms of running time with varying data size and parameters using popular text datasets.

In summary, our contributions are as follows:

- Describe working principle and flow of execution of K-means and K-nearest neighbor algorithms on four big data platforms.
- Compare the four big data platforms in terms of running time.
- Experimentally show Horizontal scaling platforms can handle much bigger datasets compared to the vertical scaling platforms while vertical scaling platforms take much lesser time for relatively smaller datasets.
- Demonstrate that for both iterative and non-iterative jobs involving in-memory computations, Spark outperforms Hadoop.

The rest of the thesis is organized as follows. In Section 2, we discuss the basic characteristics of all the four big data platforms along with their key strengths and drawbacks. We describe basic K-means and K-nearest neighbor algorithms, along with their implementation details on these platforms in Section 3. The experimental results on the performance comparison are shown in detail in Section 4. Finally, Section 5 concludes our discussion.

Chapter 2: Background

In this section, we discuss details including the working principles, advantages and disadvantages about the big data platforms. We also discuss about the programming model and important concepts and components of these platforms.

2.1 Hadoop

The Apache Hadoop has become a dominant platform in recent times. The main advantage of Hadoop is that anyone can run it on regular commodity hardware instead of expensive supercomputer or high-end computing machines. Its highly fault tolerant. Unlike regular programming paradigm, it brings code to data, not data to code. It abstracts low-level and complex parallelization and distributed operations and presents a simple interface to the end-user. For example, a developer doesn't need to be concerned about system level steps like the distributed file system, data partitioning, process pipelining, job scheduling, etc. Hadoop also takes care of advanced problems like data starvation, race condition etc. A developer can spend more time on application development and business logic [1].

It is created by Doug Cutting and available for free usage. The name "Hadoop" comes from Doug Cutting's son's toy, a yellow stuffed elephant. The concept of what we see today as Hadoop started with the publication of Google's GFS (Google File System) in 2003 [13] and MapReduce in 2004 [11]. These were parts of Google's Nutch project which was started in 2002 to handle millions of web pages and billions of searches. Building an efficient web search engine can be challenging. Mike Cafarella and Doug Cutting calculated that supporting a one billion page index would require half a million dollars for hardware, and monthly running cost of 30K dollars[1] if they need to maintain a dedicated system. The authors of Nutch realized that a distributed

[1]https://queue.acm.org/detail.cfm?id=988408

file system like GFS can extend great help to their cause by managing the storage needs. Therefore in 2004, they started to write an open source implementation of GFS, called Nutch Distributed File System (NDFS). By mid-2005, all the major Nutch algorithms were ported to run using NDFS and MapReduce [25].

In 2006, NDFS and MapReduce started together as a project named "Hadoop" under Lucene which was separate from Nutch. Doug Cutting joined Yahoo the same year and along with a dedicated team he concentrated on developing Hadoop into a large web scale. Yahoo declared to run a 10,000-core Hadoop cluster to produce search index in February 2008. Apache made Hadoop a top priority project in the same year. Meanwhile a lot of other big companies like Facebook, Last.fm, and New York Times started using Hadoop for commercial purpose. In April 2008, Hadoop was able to sort one terabyte of data in only 209 seconds beating world record. It used a cluster of 910 nodes. Google announced that they have successfully used Hadoop to sort one terabyte data in 68 seconds in November of same year [10]. In May 2009, a Yahoo team was able to perform the same task in only 62 seconds. In December 2011, Hadoop 1.0 version was released. Hadoop version 2.0.6 was made available in August 2013. Till date, the most recent version is 2.7.1 which was released on July 6, 2015. Now a days, countless number of companies ranging from tech giants to small start-ups use Hadoop to perform computation. For example, Facebook has two major clusters, one consisting 1,100 machines with 8,800 cores and the other one with 300 machines with 2400 cores. Facebook uses Hadoop to deal with the massive user data. Yahoo has the biggest employment of Hadoop clusters. It has a total 100,000 CPUs in more than 40,000 computers running Hadoop. The biggest cluster consists of 4,500 nodes [2].

[2]http://wiki.apache.org/hadoop/PoweredBy

2.1.1 Programming Model

The main parts of Hadoop are the following:

- HDFS
- MapReduce
- YARN

The earlier version of Hadoop consisted of two main parts. From version 2.0 onwards it added one more layer named "YARN" for cluster management. The addition of YARN made Hadoop not limited to MapReduce alone. Using YARN, Hadoop can run different applicaitons.

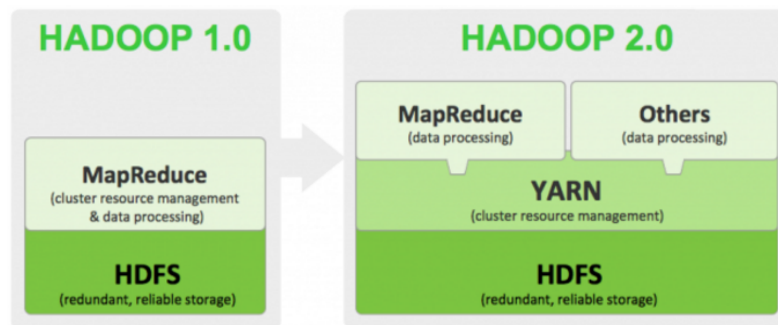


Figure 2.1: Components of Hadoop. Figure taken from [2]

We will discuss about all these components in detail in the following sections.

2.1.2 HDFS

HDFS stands for Hadoop Distributed File System. It is based on GFS and the earlier version was called NDFS. When a dataset grows so large that it cannot be accommodated to a single physical computer, we need distributed storage. We partition the dataset and store it across a number of different machines (maintaining master-slave architecture). HDFS is designed to hold very large datasets and providing access to the datasets. HDFS runs on regular commodity computers, of which

the Hadoop cluster is built from. HDFS creates the illusion of single disk to the programmer. It operates on top of regular file systems (e.g. ext3, ext4, and XFS) in the nodes.

In HDFS, files are stored as blocks. The blocks can be 64MB or 128MB where the default size is 64MB. First, a data is divided into blocks and these blocks are saved in data nodes. Hadoop's fault tolerance scheme is achieved by saving a single data block in several nodes, i.e. replication. It can keep working if the machine or disk goes bad. Default data replication factor is 3. The replication of data blocks is rack aware. The first replica block resides on the local rack. The second replica block is on the local rack but different machine. The third replica block is in a different rack. The user can control the replication factor. Fig. 2.2 shows distribution of blocks.

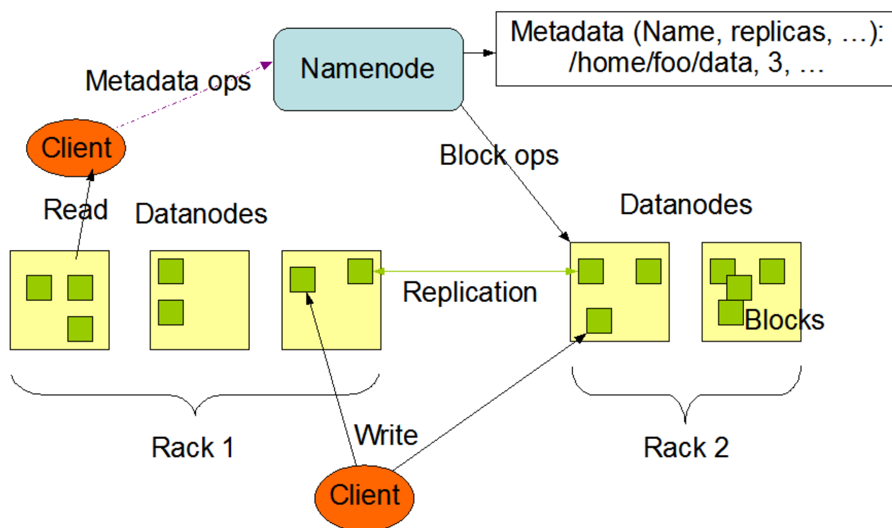


Figure 2.2: HDFS Architecture. Figure taken from [8].

The namenode (master node) saves metadata about these blocks and manages seamless access. Metadata about the blocks include file names, locations of blocks in the data nodes, file attributes, etc. The namenode performs operations like opening, closing, renaming files and directories. The data nodes do block creation, deletion etc. All the data nodes (slave nodes) exchange heartbits with namenode on a regular

basis. A heartbit is a simple signal by exchanging which the namenode learns the condition of data nodes. If heartbit is missing from any datanode then it is considered lost and namenode starts the process of recovery from replicated copies of the lost data blocks. The replication is actively maintained [8].

In a Hadoop file operation, first the namenode determines the location of data block of that particular file. For each block, the namenode returns the addresses of the blocks, the data nodes. Then all these blocks are read and reassembled. For writing, first Hadoop creates an entry for new file in namenode's namespace. Then, it calculates the block topology. The file is then divided into blocks and these blocks are streamed to the correct data node over the network. After writing them in a data node, a Success/Failure acknowledgement is propagated to the client program. Fig. 2.3 shows the workflow for write operation.

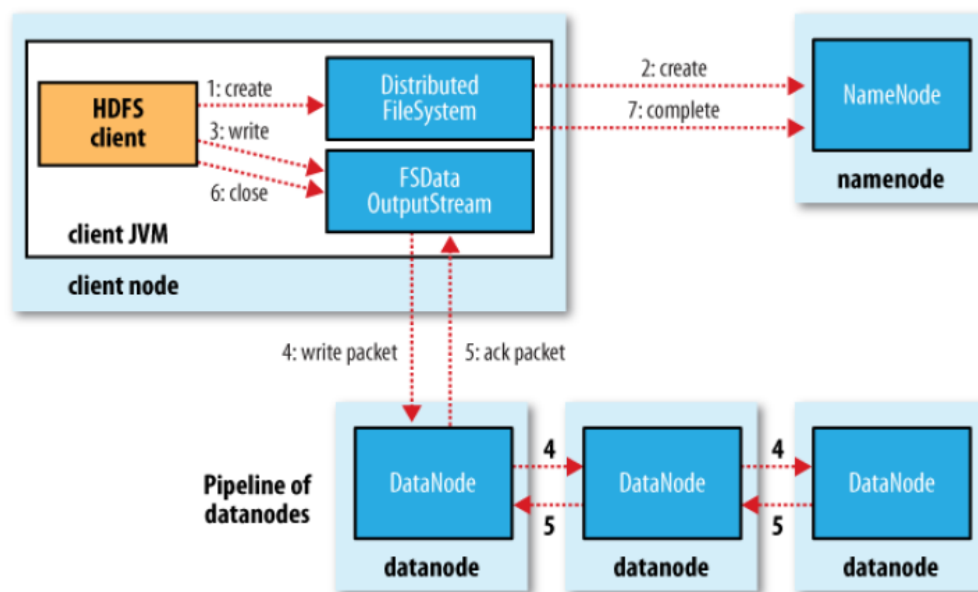


Figure 2.3: Flowchart of writing a file in HDFS. Figure taken from [25].

HDFS does not work well if very low data latency (e.g. tens of milliseconds range) is required or we have a lot of small files. It is due to the overhead operations, the HDFS needs to perform for any kind of file I/O. Some of these operations has to go

Table 2.1: MapReduce pipeline.

Map	Iterate over the data Perform calculation of our interest
Reduce	Shuffle and sort intermediate results
	Aggregate intermediate results Generate final output Write results to disk

over the network as well. For a lot of files with small size the overall operations can be overwhelming, thus hurting the performance. HDFS is also not well suited for cases of multiple writers or arbitrary modifications of files. It is designed assuming write-once-read-many-times model.

2.1.3 MapReduce

MapReduce is a programming model which distributes a task across multiple datanodes of the cluster and process the data [11]. As the name suggests, it has two main phases, map and reduce. The elementary functions of MapReduce are file reader and writer, mapper function, combiner function, partition function, compare function and a reducer function. It's programming paradigm fits well for analyzing a whole dataset in a batch fashion. The main components of MapReduce include an input reader, a map function, a combiner function, a partition function, a compare function, a reduce function, and an output writer. A more broad MapReduce pipeline is shown in 2.1.

MapReduce engine consists of JobTracker and TaskTrackers. To start with client programs submit jobs to the cluster. There is usually one JobTracker which resides in the namenode. A job consists of codes for mapping, codes for reducing, codes for shuffling and partitioning, configuration parameter etc. After job submission, JobTracker program retrieves the location of data blocks from the namenode and locates TaskTracker slots at data node. Each job is broken into smaller tasks. TaskTracker

programs which reside at data nodes accept mapping and reducing jobs from Job-Trackers. Then the TaskTracker initiates a JVM processes to actually perform the job. Each map task works with a fraction of the dataset from HDFS and generates intermediate results. Each reduce task works with the intermediate results and after processing them, finally writes them back to HDFS. MapReduce does not follow a greedy approach i.e. reduce operation cannot start until all the jobs in mapping is finished. There is a scheduler in MapReduce which takes care of scheduling jobs and resources to different nodes. The concept of scheduler is similar to a standard queue. An ecosystem of MapReduce is shown in Fig. 2.4.

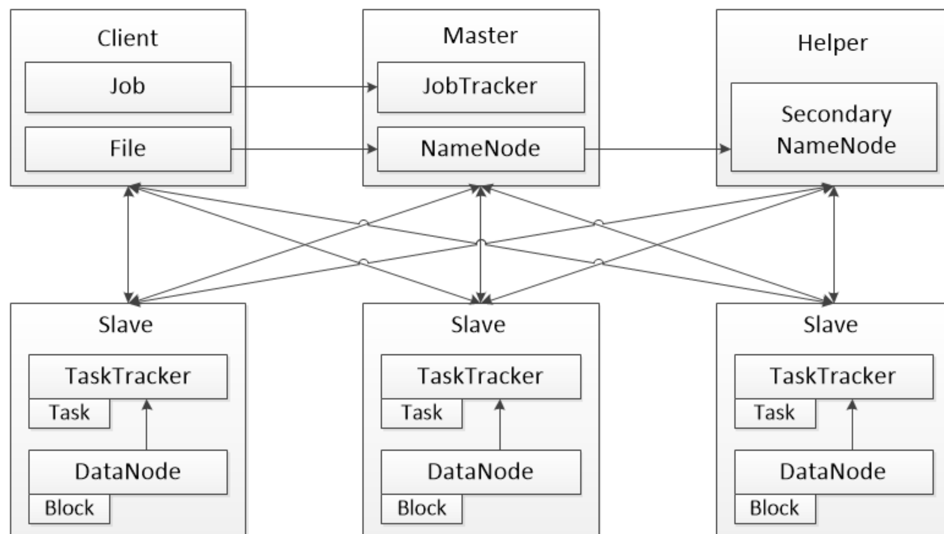


Figure 2.4: MapReduce ecosystem.

The TaskTracker monitors the progress of the job and exchange heartbits with JobTracker. If a TaskTracker at a particular data node fails then JobTracker program resubmits the job elsewhere assuring fault tolerance. But if the JobTracker program fails then all the running jobs are halted [25].

MapReduce suits well with summing, grouping, filtering, and joining kinds of tasks. It is suitable for batch jobs with massive datasets and iterative jobs where each iteration requires file input/output operations. But scenarios where the algorithm is

iterative and does not require any file I/O, Hadoop can suffer from unnecessary file I/O operations. MapReduce is not suitable for jobs that involve shared state or require interstate communication and coordination. For a job with small dataset, it may not be an optimal choice due to the system overhead. It is not well suited for programs like finding individual records. Since no in-memory operations are allowed in Hadoop where different iterations can take advantage of, after each iteration MapReduce has to write the output to HDFS to pass it to next iteration, which may take significant time for a lot of iterations.

2.1.4 YARN

YARN is the cluster resource manager for Hadoop which was added on version 2.0 [24]. YARN extends the power of Hadoop by adding the capability to work with different technologies other than MapReduce. It provides interaction with data in mutltiple ways beyond batch servicing. Other than resource manager, YARN also has one node manager per node and one application manager. As the name suggests, node manager manages node resources and application manages the scheduling and life cycle of an application. All the wrappers of MapReduce works based on YARN.

The resource manager (RM) in YARN runs as a daemon on master node. It is the central authority of arbitrating with resources in a cluster. It enforces properties like fairness, capacity, and locality across nodes while allocating resources [24] depending on the priority, demand, etc. Resources are allocated as a bundle called containers (e.g. 1GB RAM, 1 CPU).The similar thing to task tracker in YARN is node manager (NM). One NM runs on each slave node and RM communicates with them for assigning the tasks. When a job is submitted to RM, first it validates various operational, security and administrative aspects of the job. Then if the job is accepted, an application manager (AM) is created and the job is passed on to the scheduler. Once the scheduler acquires enough resources, the job status is changed from 'accepted' to

'running'. A snapshot of accepted job is written to HDFS for the cases of RM failure or restart. The AM manages a job's flow of execution. An AM program can be written in various languages. When AM require resources (e.g. CPUs, RAM, disks, etc.) for the job to run, it requests the RM. After acquiring resources, AM presents the resources to NM as lease. All these communications are done in the form of heartbit exchanges similar to MapReduce. Fig. 2.5 presents the architecture of YARN.

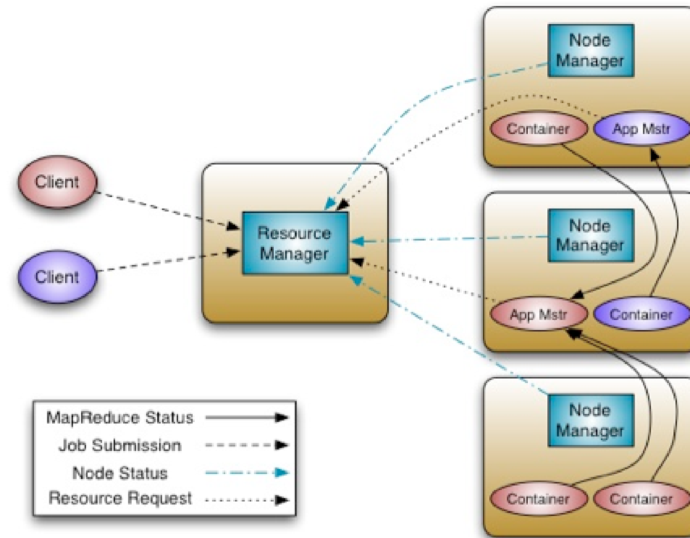


Figure 2.5: Hadoop YARN architecture. Figure taken from [5].

Hadoop is generally good for implementing parallel algorithms. It has wrappers like (1) Apache Pig [18], which is a SQL-like environment developed at Yahoo and (2) Hive [23], which is a wrapper that facilitates managing and querying large distributed datasets. Microsoft Research developed DryadLINQ [27] which is a C# environment that provides users with more flexibility over MapReduce. Apache Mahout [19] provides free implementations of scalable machine learning algorithms. These kinds of software make development easier to the programmers.

2.2 Spark

Spark is a next generation big data platform first created at UC Berkely AMPlab in 2009[3]. It is now available as an open source platform. According to the authors of the Spark project, it can be ten times faster than Hadoop for iterative machine learning tasks [29]. It utilizes Hadoop's HDFS for data storage. But in contrast, Spark allows in memory calculation of primitives and caching of results on memory. Although it is not as mature as Hadoop, it has comfortable APIs for Java, Scala, and Python. From Spark version 1.5.1 onward, it supports computation in R through SparkR package[4]. The APIs are data centered and yield easier programming paradigm than hadoop. It broadens the scope of computing workloads that Hadoop can handle. Spark comes equipped with a library MlLib which has a lot of well known machine learning algorithms for example basic statistics, regression, classification, clustering, collaborative filtering, and optimization etc. are already implemented. It allows the users to seamlessly intermix different types of jobs like batch processing, streaming, and interactive queries [30]. Spark is implemented in Scala. A big difference of Spark with Hadoop is Spark programs act as their own scheduler as flow of parallel operations unlike standard MapReduce schedulers. Finally, Spark suffers from problems similar to Hadoop when it comes to algorithms involving shared state. At the same time, if there is a slave node failure, Hadoop can continue working where it left of but Spark has to start over from the beginning. Its an active open source project under development. Fig. 2.6 presents a short overview of Spark cluster.

Spark has two main components, namely Resilient Distributed Datasets (RDDs) and Parallel Operations [29]. To utilize Spark, the user needs to write a program that defines the high-level control flows involving RDDs and parallel actions with them. In the following subsections we discuss about them in more detail.

[3]https://amplab.cs.berkeley.edu/
[4]https://spark.apache.org/docs/latest/sparkr.html

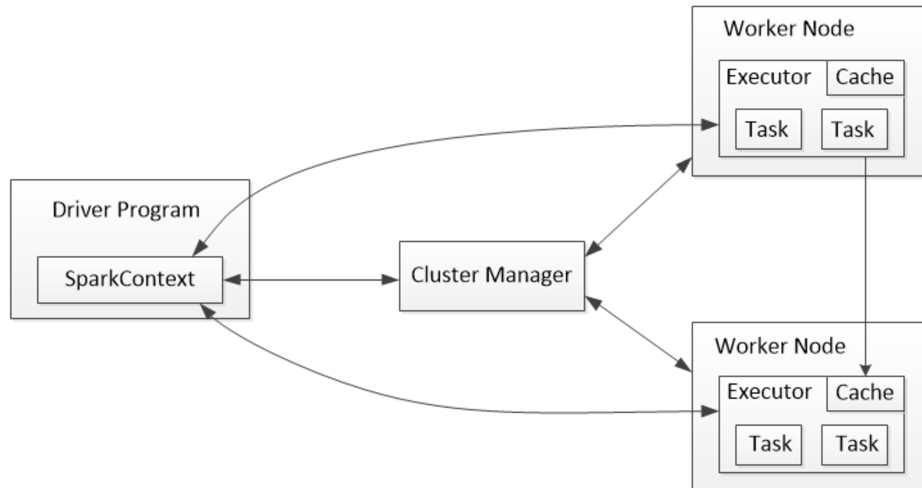


Figure 2.6: Spark cluster overview.

2.2.1 RDDs

The RDD is the main abstraction of Spark. RDD is a partitioned collection of read-only data. User can control the partitioning of the RDD variable. The RDD abstraction is similar to the Apache Crunch's PCollection[5] (Cunch is a Java library that provides framework for writing MapReduce programs). Apache Crunch can be used on top of Spark. A RDD variable is represented by a Scala object. There are four ways to create a RDD variable

- By reading a file from a shared file system (e.g., HDFS)
- By parallelizing a Scala object. For example, dividing an array into a number of nodes.
- By transformation of an existing RDD. For example, if we have a RDD named A, we can perform our operation of interest on it and save it as another RDD named B.
- By altering the persistence of any existing RDD. Spark follows lazy creation

[5]http://crunch.apache.org/apidocs/0.9.0/org/apache/crunch/PCollection.html

of RDDs, i.e., the variable is materialized on demand. A user can change the persistence by *cache* action or *save* action.

Unlike Hadoop, Spark allows to cache variables. But if the cluster machines do not have enough memory to cache all the partitions of the data, then the variable is recomputed when it needs to be used. It is different from Hadoop's way of dealing with the data in a number of ways. The RDDs allow the programmer to explicitly use either memory or disk to hold intermediate results. It is possible to make an RDD persist on memory as well which can be a practically useful thing considering the reuse. There is no synchronization barrier for these models which these allows for in-memory computation. RDDs are also equipped with fault tolerance but in a manner that is completely different from Hadoop. If a partition of RDD is lost, the RDD has enough information to retrieve the lost partition. It achieves fault tolerance through a notion of lineage by remembering how one RDD is generated from other RDDs. This saves the costly replication of the data which greatly reduces the network I/O and storage of data that is already big in size [28].

2.2.2 Parallel Operations

We can perform a wide range of actions like count, collect, save etc. and transformations like map, sort, filter, join, union, cross-product, group by, reduce etc. with the APIs. For iterative models Hadoop has to write back the intermediate results on HDFS at the end of every iteration and read it again at the start of the next iteration. But the memory utilization of Hadoop saves a lot of disk I/O. Overall RDDs offer a much faster general execution model compare to disk-oriented Hadoop framework.

For writing a Spark code, developer manipulates the RDDs by invoking different parallel operations by passing closures (functions). These closures are regular variables following their scope of creation. However, Spark also allows developers to create two restricted type shared variables. First, the Broadcast variables, the

purpose of which is to make sure that only one RDD is created only once. This is particularly helpful when one RDD is required in multiple parallel operations. Second type is the Accumulators. This type of variables can be read only by the driver and worker nodes can only add them for associative operations.

2.3 GPU

The Graphics Processing Units (GPUs) first introduced almost 15 years back have now become an integral part of big data computing systems. They provide great scale up solution to increase the computational power. Although in the initial times the primary purpose of GPUs was to enhance processing of graphics pixels for a better display output [20], scientists have started to explore the use of GPU to accelerate scientific computations. In 2006, Nvidia introduced CUDA (Compute Unified Device Architecture) API [16] which allows executing programs written in C programming language on GPU devices. The basic idea of GPU is to employ a lot of processors with less computing power instead of one high configuration CPU and achieve efficiency through parallelization. In 2012, Nvidia presented OpenACC [26], a parallel computing programming standard. CUDA abstracts the parallelization details from the programmer. All the codes are forward compatibile i.e. a piece of code written in a version will still be working with any newer GPU device. It gives flexibility to the programmer and vendor in terms of upgrading.

Table 2.2 lists some configurations of a standard GPU and a standard CPU.

Table 2.2: Configurations of a standard CPU and GPU.

CPU: Intel Xeon X5650	GPU: NVIDIA Tesla M2070
Clock speed: 2.66GHz 4 instructions per cycle	Core clock: 1.15GHz Single instruction per cycle
CPU cores: 6	CPU cores: 448
Bandwidth: 32 GB/sec	Bandwidth: 150 GB/sec
Total GFlops: 2.66 x 4 x 6 = 63.84	Total GFlops: 1.15 x 1 x 448 = 515

GPU offers support for different programming languages. For C language it has

CUDA and OpenACC, for C++ it has Thrust and CUDA C++, for python it has PyCUDA and Copperhead. It also supports Matlab. It has several libraries like Nvidia's CUDA BLAS, FFT and many more third party libraries. GPU is called massively parallel because of its lightening fast speed of executing huge number of operations. However, GPU comes with physical limitations. Due to its fixed memory size, it is not possible to implement algorithms that require memory higher than GPU's limit. In following sections we will discuss the details of programming model of GPU and CUDA.

2.3.1 GPU and CPU Interaction

In the GPU and CPU interaction, CPU is called the *host* and GPU is called the *device*. The are connected to each other via the PCIE Bus cable. The bus cable has a transfer speed ranging from 8GB/s to 16 GB/s. Both the host and device has their own memory, but one cannot access the memory of the other. Therefore, before running any program, the user needs to copy the data from the host memory to the device memory by allocating additional memory. The device has different kinds of memory which will be discussed in later sections. In GPU, all the multiprocessors can perform the same tasks independently. Parallelism is achieved by running Kernel functions. After memory allocation and transfer on device, the programmer has to create a specific number of Kernels and its functionalities. Then the Kernel functions process the data according to business logic. To view the results of computation, it has to be transferred back to host again. Fig. 2.7 presents an overview of GPU and CPU interaction.

2.3.2 GPU Thread Organization

GPU executes its parallelization program as kernel. Its a simple C program that runs the same code in all the multiprocessors in parallel. Some of the functions

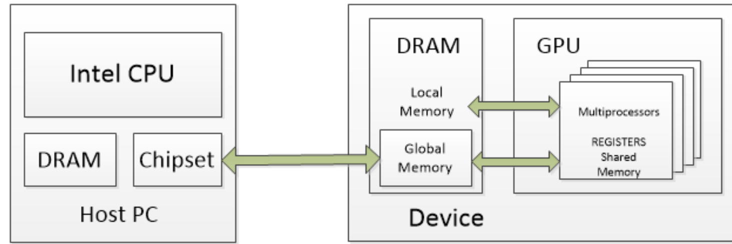


Figure 2.7: An illustration of interconnection between GPU and CPU.

include constraints like it must have a return type void, it should not contain any static variable and it cannot be recursive in nature. On a GPU device, only one kernel program can run at the same time and it cannot access the CPU memory. The Kernel program is launched as a grid. The grid is a collection of thread blocks. A thread is the unit of execution. A single kernel program can lunch at most 65,536 blocks. A thread can be referenced by (x,y,z) value. Several blocks can reside in a multiprocessor based on GPU resources. The workload per thread is very lightweight [14]. A grid is a one or two dimensional collection of blocks (one grid per device).

The programming language CUDA is often referred to as SIMT (Single Instruction Multiple Thread) architecture. Each thread has unique identification number and is used to access memory addresses and take control decisions. A GPU unit has three kinds of memory, namely, global memory, shared memory, and local memory. A thread can read and write from and to all these three memories whereas CPU can access only the global memory. Global memory can be accessed by all the threads of all the blocks. Shared memory is block-specific, i.e., can be accessed by threads of a particular block. Local memory is private to each thread. CUDA multithreading hides the latency and helps in maximizing the utilization of the device. It is very difficult to reach a dead-lock situation in GPU given there is no message passing. The entire framework is very powerful as it can handle 100K+ threads at the same time. However, to fully appreciate the power of CUDA parallelization, the programmer has

to utilize the full potential of multithreading. Programmer has to design the code in a way that can leverage the power of CUDA massive parallelization. Although the registers and local memory in the device is very fast, but they are limited in number. Programmer has to work around the limited on-chip memory.

The number of threads one block can launch varies based on the compute capability of the GPU device. For example, in compute capability 1.x the number is 512, for compute capability 2.0 it is 1024. The threads can be launched in one dimension, two dimensions or three dimensions.

The concept of thread synchronization is very important. For the threads, CUDA provides synchronization barrier by a method **__syncthreads()**. Thread synchronization is possible across block only, not across the entire grid. It is due to the fact that CUDA executes multiple block of threads on multiprocessors in any order simultaneously. When a kernel call is made, CUDA distributes blocks across multiprocessors on the device runtime. As long as enough resources, e.g., registers, shared memory are there, CUDA distributes a maximum of 8 blocks per multiprocessor. The total number of blocks handled simultaneously depends on the device. For example, in Fermi architecture, a total of 16 multiprocessors can simultaneously handle 128 blocks of threads.

2.3.3 GPU Memory Hierarchy

There are different kinds of memory available in a GPU device. Based on the characteristics of each type of memory, the programmer has to optimize his code to get best performance. The device has three kinds of memory. They are following

- Global memory
- Shared memory
- Registers

Each of these memories has different characteristics, access time, latency, address space, scope, and lifetime. The host can only access the global memory of the device. Global memory is the largest among all the memories. Global memory has a large address space but the access time is high. Shared memory on the other hand has low access time but its size is very much limited. All the elements of all the threads in the grid have access to the global memory. But shared memory is accessed only block wise. There is one shared memory space for each block of threads. The scope is most limited for registers. Registers are accessed per thread. One block of thread cannot gain the access of shared memory for another block of threads. Similarly, one thread can access its registers only.

Global memory is also considered *off-chip* memory and shared memory and registers are considered *on-chip* memory. The latter two combined together will act as the cache memory of the device. Access time of the off-chip memory can be $100x$ slower than on-chip memory. On the devices with version $2.0x$ computation capability, there is another concept of L1-cache which is stored along each multiprocessor. Fig. 2.8 shows the hierarchy of device memory.

Registers are the fastest among all the types of memory. If any variable is declared inside the scope of a kernel function and it does not have any special property, then generally its stored in registers. Being fast, the number of registers is limited. An array declared in a kernel function will be stored in the registers if the elements are accessed with constant indices (i.e., the identifier to access the array element is not a variable, rather a constant value). Register variables are private for threads. Once a kernel is finished running, the register variable cannot be accessed after that. Each time we invoke a kernel function, it must initialize the variables. Read and write operations for the register memory do not need to be synchronized.

In a thread, the kernel allows it to spill over to local memory when any variable that is not able to fit into the registers. Local memory accessing speed is similar to

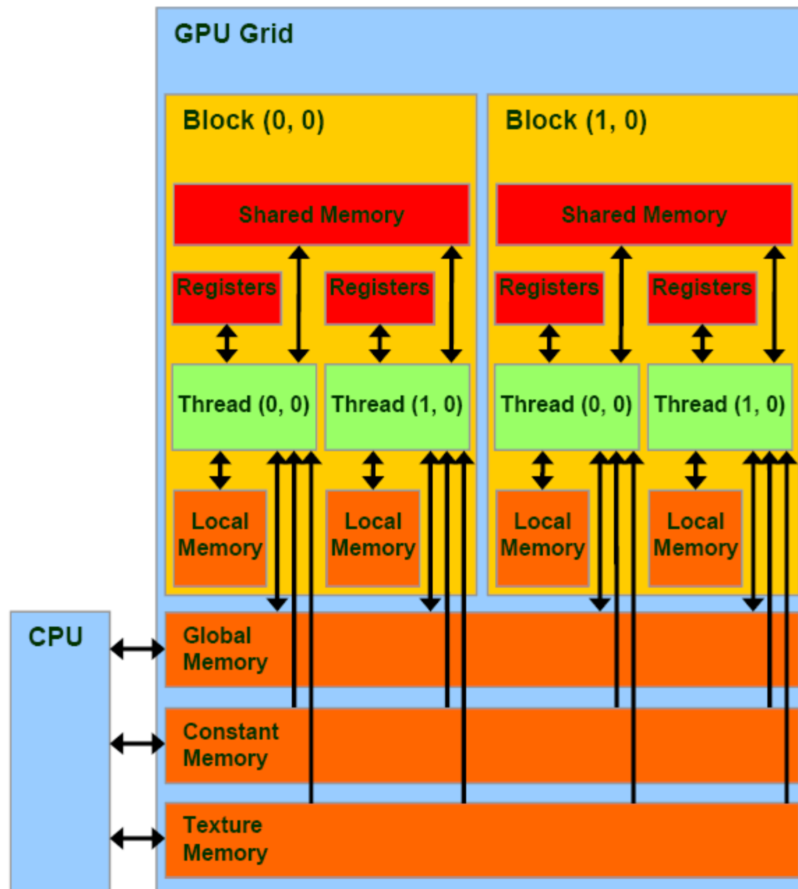


Figure 2.8: Memory hierarchy in GPU. Figure taken from [4]

the global memory in terms of access time. The L1 caching scheme from compute capability version in $2.x$, saves the values from local variable. Similar to registers, local memory is also private to threads. Once the thread is finished running, variables in local memory cannot be accessed any longer. Programmers cannot decorate a variable declaration but the compiler will make the variable declaration in local memory under the following conditions:

- Arrays that have accessing indices as not constant i.e., the indices are calculated runtime, compiler cannot decide the indices during the compilation time. If compiler can, the variable goes to registers.
- A large structure of variable that could take too much register space.

- A variable that exceeds the number of registers allocated for a kernel.

Variables which are decorated with the keyword **__shared__** are called shared variables. Each multiprocessor has limited amount of shared memory. The access time for them is very fast. These kinds of variables are declared in a kernel but it has a lifetime per block, i.e., even if the kernel finishes running, any other kernel from the same block can still access it. Any other thread in same block will get the access to shared memory variables. When a block finishes running, it cannot be accessed any more. Given that a lot of threads can be present in a block and they can try to modify the shared memory variables at the same time, it is necessary to synchronize them. The keyword **__syncthreads()** is used for this purpose. Since it is faster than global memory, based on the application in hand it may be a good idea to copy values from global memory to shared memory. For some applications, this can greatly reduce the memory access time, thus increasing the overall performance.

Table 2.3: GPU memories in different compute capabilities

Technical Specifications	**1.0**	**1.1**	**1.2**	**1.3**	**2.0**
Number of 32-bit registers per MP	8 K		16 K		32 K
Maximum amount of shared memory per MP	16 K				48 KB
Amount of local memory per thread	16 K				512 KB
Constant memory size	64 KB				

Variables which are declared with the keyword **__device__** are called global variables. They are declared outside the kernel scope, for example in the main function of the host program. We already know that the global memory is slow but the total amount of global memory is large. For different devices, it can be upto 6GB. Unlike previously mentioned memories, global memory is allocated first from the main function using the function called *cudaMemcpy*. The function is same for both copying from and to the global memory. A parameter defines the direction of copy. After the end of the memory usage, the function *cudaFree* is used to free the memory space. Global memory has a lifetime of accessibility to all the blocks and threads. Pointers

to global memory variables are passed to kernel function as parameters.

Variables which are decorated with the keyword **__constant__** are constant variables. Similar to global variables, they have to be declared in the global scope which is outside the kernel function. They share the same memory block but the total size of the constant memory is only 64KB across all the compute capabilities. But the constant memory is considerably faster than the global memory. Because constant memory cannot be written from the kernel space, it can be cached due to the guarantee that the values will not be changed. Similar to global memory, it has a lifetime of existence in the application.

The amount of available memory to CUDA mostly depends on the compute capability of the device. Each compute capability has different sizes for different kinds of memory. Table 2.3 summarizes the relationship between compute capability and memory types [17].

Table 2.4 summarizes different memory types and their properties.

Table 2.4: Properties of GPU memory types

Memory	**Located**	**Cached**	**Access**	**Scope**	**Lifetime**
Register	Cache	N/A	Host: None Kernel: R/W	Thread	Thread
Local	Device	1.x: No 2.x: Yes	Host: None Kernel: R/W	Thread	Thread
Shared	Cache	N/A	Host: None Kernel: R/W	Block	Block
Global	Device	1.x: No 2.x: Yes	Host: R/W Kernel: R/W	Application	Application
Constant	Device	Yes	Host: R/W Kernel: R/W	Application	Application

2.4 Multicore CPU

The performances of processors have been increased 1000x over the past 20 years. Transistor count in processors approximately doubles in every two years. A recent pro-

cessor (Intel Haswell-EP Xeon E5-2699 v3) released in 2014 has 18 cores, 36 threads per chip and 45MB shared chache memory in only $662mm^2$ space. In a single CPU with multiple cores, we can perform tasks in parallel. Multicore CPUs are regular CPUs having multiple processing cores. Generally, these machines share a single memory and disk. In recent designs, the number of cores per chip has increased significantly, thus allowing CPUs to gain parallel computing capabilities.

The parallelism is achieved by multi-threaded programming. The overall task is broken down into various threads where each thread can execute in parallel in different cores. Although sharing the same memory, threads work with different parts of the large dataset. There are dedicated libraries available for thread programming in most of the well-known languages. The advantage of multicore CPUs is the simplicity of their usage. Parallel programs can be written with threads with minimal effort. The main drawback is their limited number of processing cores. System memory for such machines is also limited. This severely limits the data size that can be handled by multicore CPUs. GPU has a lot more processing units where each unit has lower processing capability. GPU can spawn a lot more threads compared to CPU thus making it more appealing for parallelization. Fig. 2.9 presents the organization of the threads.

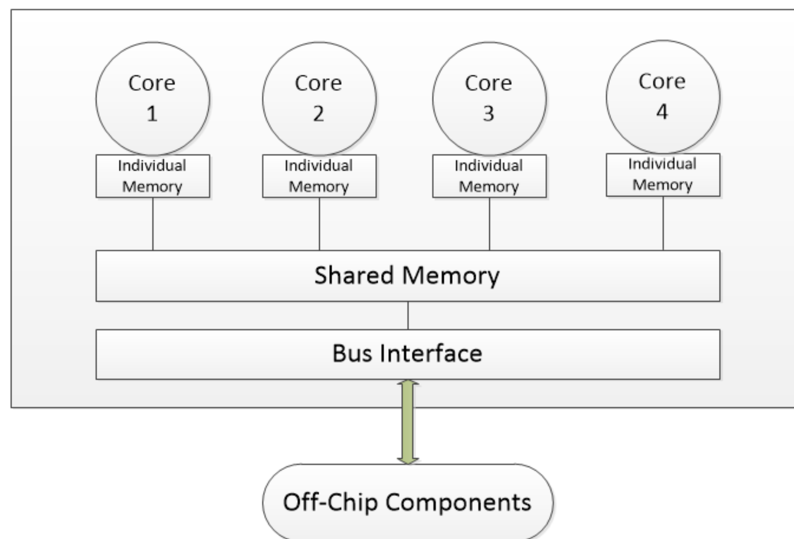


Figure 2.9: Multicore CPU architecture.

Chapter 3: Data Mining Algorithms and their Implementations

In this section, we will describe the two data mining algorithms that are extensively studied for this thesis, namely, K-means clustering and K-nn classification, along with their implementation details on the four big data platforms.

3.1 K-means clustering

The goal of clustering is to find natural groupings in a set of data points [6]. K-means is the most popular and widely used clustering technique [15, 21]. Given a dataset D with N instances and M features, the algorithm aims to partition D into K groups, where K is the number of clusters which is a user parameter. The algorithm starts with randomly initializing k centroids. Then, in every iteration it associates each data point to it's nearest centroid and using the averages of the data points belonging to a cluster, the new centroids are calculated.

This process continues until there is no change in the cluster membership of the data points or it reaches a specific pre-defined number of iterations. In our implementation we limit the iteration number to 20 and take the first K data points as the centroids instead of initializing randomly so that all the platforms can start with the same seed centroids and yield same resulting cluster membership. Algorithm 1 presents the pseudocode of the standard K-means algorithm.

The most expensive part of K-means is the distance calculation between the data-points and the centroids. If we are to perform Euclidean distance calculation between a dataset with N rows and M features and centroids with K rows and D features, we need to perform total $\mathcal{O}(KMN)$ computations. For a dataset with very high number of features or instances, this step might take a lot of time to complete.

Algorithm 1 K-means Clustering

Inputs: Dataset D, Number of clusters k
Output: Data points with cluster memberships
1: Initialize random k training data points as *centroids*
2: **do**
3: Compute the distance between each point in D and each point in *centroids*
4: Associate data points to their closest centroid
5: Recompute the *centroids*
6: **while** No changes in cluster membership

The next two steps which involve finding the closest centroid (with minimum distance) and recalculating the centroids are comparatively less time consuming. So, if we run K-means for I iterations, the overall time complexity becomes $\mathcal{O}(IKMN)$. In all the big data platforms, we parallelize the most costly part, the distance calculation where multiple worker nodes/cores work with separate parts of the data and calculate the distances in parallel. Once this computation is done, a central node/core calculates the nearest neighbor and recalculates the centroids.

3.1.1 K-means on MapReduce

Algorithm 2 presents the pseudocode for the process and Fig. 3.1 shows the flow of execution. For implementing the K-means algorithm in MapReduce, we start by uploading two files in HDFS, one for dataset and another for centroids. Here, centroids are the first k rows of the dataset. At the beginning of each iteration, the *Map* function will read these two files from HDFS and calculate Euclidean distances from each data point to each cluster centroid. The total number of map functions depend on the data size and the number of slave nodes in the cluster. Each map function will output the *key-value* pairs where the key is the id of the data point and the value will hold another tuple of centroid id and the distances between them. These pairs are buffered in memory and stored on the local map workers local disk on a periodic basis. In the next step, these *key-value* pairs are partitioned into regions.

After all the map workers finish their jobs, a shuffle and sort operation is performed based on the keys. Together the partition and shuffle operation groups all the pairs based on the key values and decide which reduce worker will work on which key [31]. Next, the master node notifies the reduce worker to start. For each unique key the reduce worker calls a reduce function. Reduce function will associate data points with the closest centroid and then recomputes the centroids by taking the average of the data points. Finally, when all the reduce functions finish their job, it creates a new folder in HDFS according to the iteration number and the resultant centroids are written to a file. The entire mapping and reducing process is repeated twenty times.

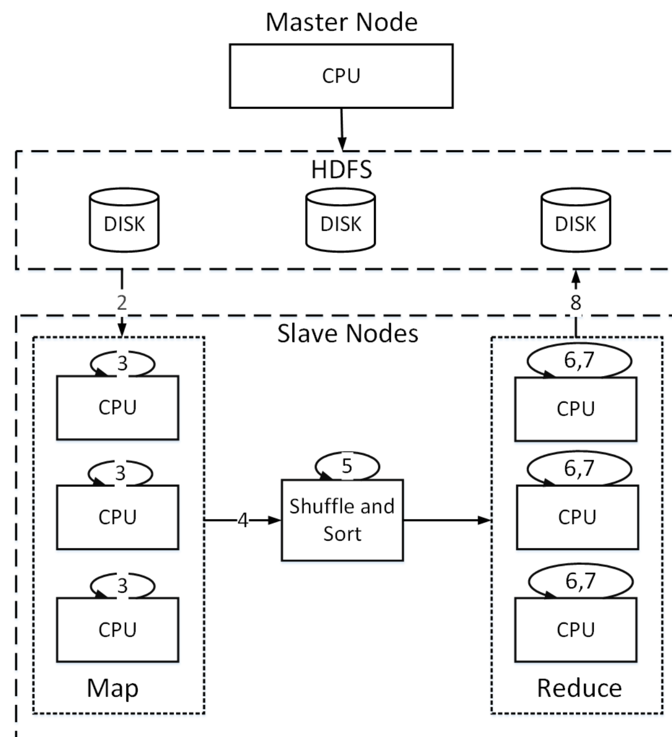


Figure 3.1: K-means flow of execution on MapReduce.

Hadoop is very resilient to failure. MapReduce is kind of complex to use in practice. It is not always possible to implement each and everything in MapReduce. But many companies and startups are inventing tools that can ease the burden. The primary advantage of Hadoop is scalability. Its possible to process gigantic files which is not possible in vertical scaling platforms. It has to perform file I/O for

each iteration, which is time consuming, for algorithms that do not involve iteration, Hadoop is comparable to the best platform in terms of running time.

Algorithm 2 K-means Clustering on MapReduce

Input: Dataset D, *centroids*, Number of clusters k
Output: Data points with cluster memberships
1: **for** $iteration = 1$ to $MaxIterations$ **do**
2: Mapper: Read D and *centroids* from HDFS
3: Mapper: Compute the distance between each point in D and each point in *centroids*
4: Mapper Output: Key-value pairs with key as data point id and value as a tuple of centroid id and distance between them
5: Shuffle and Sort: Aggregate for each key
6: Reducer: Associate data points to their closest centroid
7: Reducer: Recompute the *centroids*
8: Reducer Output: Write *centroids* to HDFS
9: **end for**

3.1.2 K-means on Spark

Algorithm 3 describes the pseudocode of the process and Fig. 3.2 presents the workflow of the algorithm. Unlike Hadoop, Spark does not have separate *Map* and *Reduce* phases. One does not require to write separate functions for mapping and reducing. The implementation is similar to the one that is done in standard Java except the fact that all operations are done using a distributed variable. For the K-means implementation, first we need to copy the data file to HDFS. At the beginning of the Spark program, it reads the datafile from HDFS as a RDD variable. The first k rows are selected as centroids. In the next step, a map function over the dataset is called and the Euclidean distances from each data point to each centroid are calculated which is then saved as another RDD. The centroid with minimum distances is assigned as their nearest centroid. After this, the average of all the data points in a cluster is taken and the centroids are updated. This is continued for twenty iterations.

In contrast to Hadoop, caching of variables in memory is allowed in Spark. We cache the variable so that accessing them becomes more efficient. In Spark, we can utilize the same variables in all the iterations, therefore, there is no need to perform file I/O in every iteration, thus resulting in better performance compare to Hadoop. Spark does not have dedicated partition and shuffle operations. These operations have to be done manually on RDD variables.

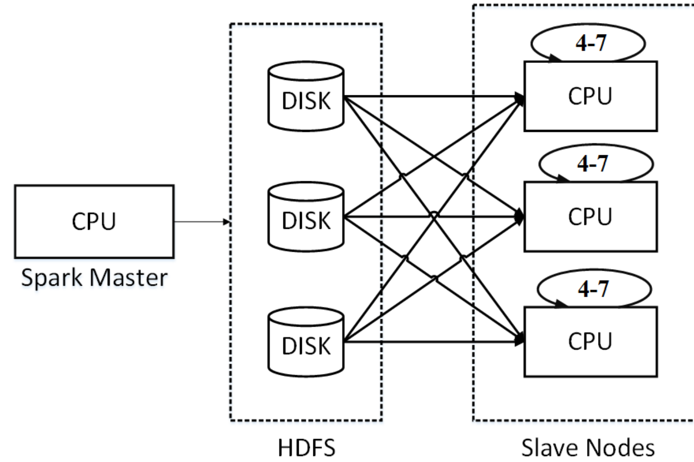


Figure 3.2: K-means flow of execution on Spark.

Algorithm 3 K-means Clustering on Spark

Input: Dataset D, Number of clusters k
Output: Data points with cluster memberships
1: Read D from HDFS as RDD
2: Initialize first k data points as *centroids*
3: **for** $iteration = 1$ to $MaxIterations$ **do**
4: Compute distance between each point in D and each point in *centroids*
5: For each data point group distances
6: Associate data points to their closest centroid
7: Recompute the *centroids*
8: **end for**

3.1.3 K-means on GPU

Algorithm 4 presents the pseudocode for the procedure and Fig. 3.3 explains the execution flow. The GPU device has a lot of multiprocessors stacked up. While

implementing the K-means algorithm, the data is first read and the centroids are created from the the first k data rows. In each iteration, these two are copied into the shared memory of GPU device and a Kernel function is invoked. The kernel function forks a lot of threads. Each thread will access a set of data rows decided by it's own thread ID. The number of threads created depends on the total number of data points. Each thread calculates the distances from the centroids to a particular data row which is identified by the thread ID. After the end of the kernel call, the distance variable is byte wise copied to CPU. Then, in the CPU, the index of the nearest centroid is computed. After that the centroids are recomputed by taking average of the data points belonging to each cluster. This is done for twenty iterations.

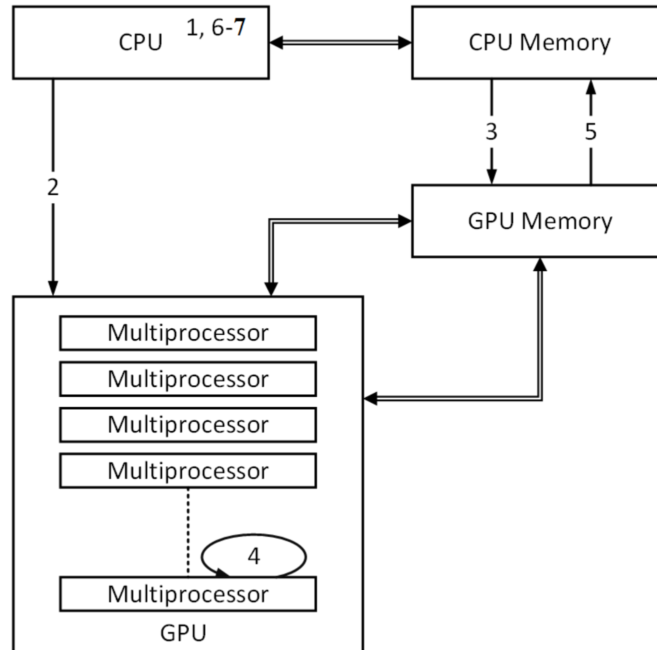


Figure 3.3: K-means flow of execution on GPU.

GPU is rapid in terms of the algorithm running time. The main strength of the GPU is the number of processing cores. It creates high number of threads which can perform the distance calculations, which is the most time consuming component of K-means algorithm, on the entire dataset. However, it must be noted that the

GPU memory is limited. It is not possible to upload very big data into the memory. Special care is needed to work around alternate solutions for large-scale data that does not fit into the memory. For algorithms that has state to state transitions, GPU is not a favourable platform. Also, a single multiprocessor in GPU has very limited power. The threads can handle only light jobs. If it is required to perform massive computations in a single thread, GPU may not be an appropriate platform.

Algorithm 4 K-means Clustering on GPU

Input: Dataset D, Number of clusters k
Output: Data points with cluster memberships
1: Initialize first k data points as *centroids*
2: **for** $iteration = 1$ to $MaxIterations$ **do**
3: Copy D and *centroids* to GPU shared memory. Split D into threads
4: Kernel process: Compute distance between point in D and each point in *centroids*
5: Send the distances to CPU
6: CPU process: Associate data points to their closest centroid
7: CPU process: Recompute the *centroids*
8: **end for**

3.1.4 K-means on Multicore CPU

Algorithm 5 describes the pseudo code for the implementation and Fig. 3.4 shows the process flow for running K-means on Multicore. Similar to other platforms, we start with reading the data and selecting first k rows as centroids. Then several threads which work in parallel to calculate the distances are created. For this job all the threads share the same CPU memory but access different data rows. After the end of the distance calculations, all the distances are sent to a core where each data point is assigned to it's nearest centroid. Then the centroids are updated by taking the average of the data points belonging to a cluster. This process is done for a pre-defined number of iterations (twenty in this case).

Multicore has an advantage of holding all the variables in a single memory. There is no need to copy the variables from one memory to another. Passing simple parameters

to thread is sufficient. Therefore, there is no data transfer latency. It does not have any dedicated distributed file system as well. This makes the programming a lot easier in this platform compared to that on other platforms. However, it severely suffers from physical limitations. The CPU memory and the number of cores in the CPU is limited. It cannot read a dataset bigger than a certain size.

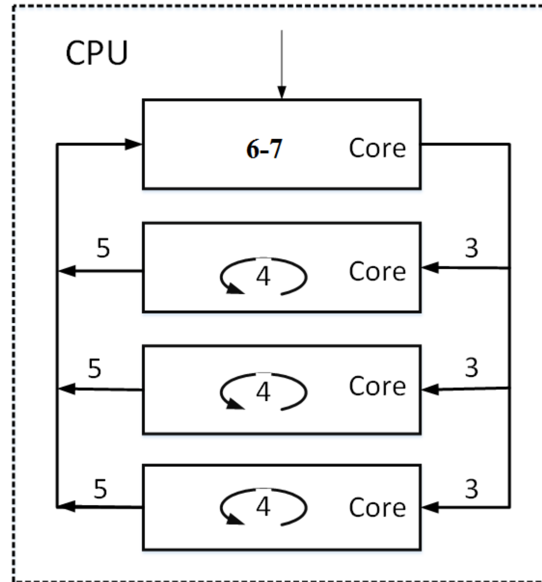


Figure 3.4: K-means flow of execution on Multicore CPU.

Algorithm 5 K-means Clustering on Multicore CPUs.

Inputs: Dataset D, Number of clusters k
Output: Data points with cluster memberships
1: Initialize first k training data points as *centroids*
2: **for** $iteration = 1$ to $MaxIterations$ **do**
3: Split D into multiple cores
4: Compute distance between each point in D and each point in *centroids*
5: Send distances to central core
6: Associate data points to their closest centroid
7: Recompute the *centroids*
8: **end for**

3.2 K-nearest neighbor classification

The K-nearest neighbor (K-nn) is a prominent classification algorithm widely used in many applications [9]. In this method, the class labels of the test instances are calculated using it's nearest neighbors class labels from the training instances. In the naive implementation, the distances between each train and test instance are calculated at first. For each test instance, k instances from the training data with minimum distance to it are taken as the nearest neighbors. With a majority voting of the class labels of the nearest neighbors, the test instance is classified.

In the training phase, the Euclidean distances between the multidimensional vectors are calculated. When the number of training instances, test instances and the dimensionality increase, the distance calculation step can take considerably long time. The nearest neighbor calculation can also take significant amount of time to complete. If our training dataset has N rows, test dataset has M rows and each vector has D features, the distance calculation has a time complexity of $\mathcal{O}(NMD)$. Finding a single nearest neighbor takes $\mathcal{O}(N)$. Therefore, finding K nearest neighbors for total M test instances takes $\mathcal{O}(NMK)$. We parallelize both of these costly operations on the big data platforms. Algorithm 6 describes the pseudocode. It should be noted that for comparing different big data platforms, we use the naive version of the K-nn algorithm and do not use other efficient techniques such as locality sensitive hashing or KD tree representation. Using such representations will show bias in the performance of certain platforms and will not provide a genuine comparison. Hence, we show the performance results of the naive implementation of K-nn algorithm.

Algorithm 6 K-nn classification

Inputs: Train Data D, Test Data X, Number of nearest neighbors k
Output: Predicted class labels of X

1: Calculate the distance between each $x_i \in X$ and each $d_j \in D$
2: Find the indices of k smallest distances for each x_i as nearest neighbors
3: Assign the most frequent class label from nearest neighbors as predicted class label.

3.2.1 K-nn on MapReduce

Algorithm 7 presents the pseudocode for the process and Fig. 3.5 shows the flow of execution. In MapReduce, the train and test datasets are first uploaded into HDFS. In the map phase, the Euclidean distances between the train and test instances are calculated. The mapper outputs a key-value pair where the key is test instance ID and the value is another tuple consisting of train instance ID and the distance between them. The next step of shuffle and sort aggregates all the key-value pairs and groups them based on the key. All the tuples with the same key value (test instance ID) are passed to a reducer function. The first k instances with the minimum distance value are taken as nearest neighbors. With a majority voting of class labels of the nearest neighbors, the class label of the test instance is defined.

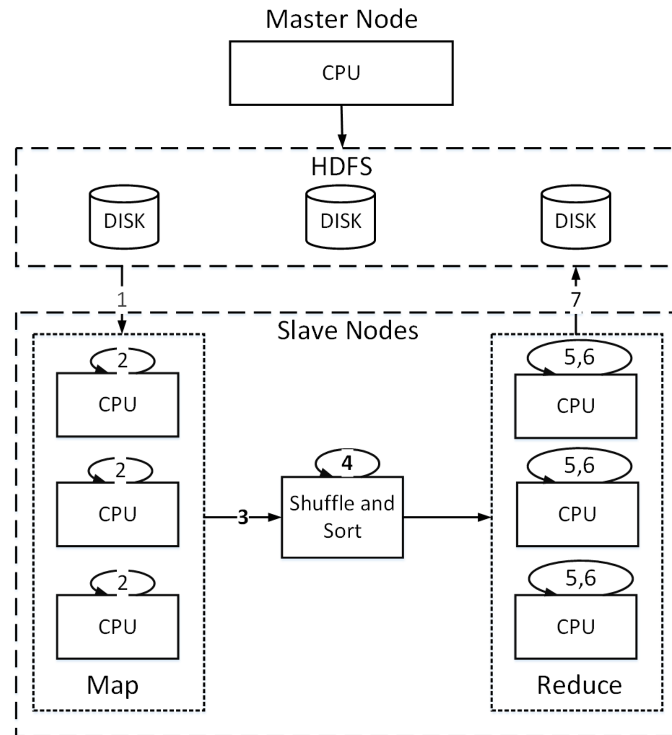


Figure 3.5: K-nn flow of execution on MapReduce.

Algorithm 7 K-nn on MapReduce

Inputs: Train Data D, Test Data X, Number of nearest neighbors k
Output: Predicted class labels of X

1: Mapper: Read D and X from HDFS
2: Mapper: Compute the distance between each $x_i \in X$ and each $d_j \in D$
3: Mapper Output: Key-value pairs with key as test instance ID and value as a tuple of train instance ID and the distance between them
4: Shuffle and Sort: Aggregate for each key (test instance)
5: Reducer: Find the indices of k smallest distances for each x_i as nearest neighbors
6: Reducer: Take majority voting of class labels of nearest neighbors
7: Reducer Output: Class labels of test instances

3.2.2 K-nn on Spark

Fig. 3.6 describes the flow of execution and Algorithm 8 shows the pseudocode. In Spark implementation of K-nn, first we read the test data as a RDD variable and read train data as a two-dimensional float variable. We utilize the advanced Spark feature of 'broadcasting' whose purpose is to efficiently pass a large data variable to the worker nodes via a P2P protocol. This is particularly helpful in cases where the data is immutable and read-only. Broadcasting also makes sure the data is sent only once irrespective the number of partitions. We broadcast our train dataset over the network. We create a new distance RDD variable where we calculate the Euclidean distances between each test data to all the train data points. We further utilize this distance variable to find the indices of k smallest distances as the nearest neighbors. Finally, with a majority voting of the class labels from nearest neighbors, we find the predicted class label and write our prediction to HDFS.

Algorithm 8 K-nn on Spark

Inputs: Train Data D, Test Data X, Number of nearest neighbors k
Output: Predicted class labels of X

1: Read X as RDD_X and D from HDFS
2: Broadcast D to all the worker nodes
3: Calculate the distance between each point in RDD_X and D as $RDD_{distance}$
4: Find the indices of k smallest distances for each x_i as nearest neighbors
5: Assign most frequent class label from nearest neighbors as predicted class label
6: Write predicted class labels to HDFS

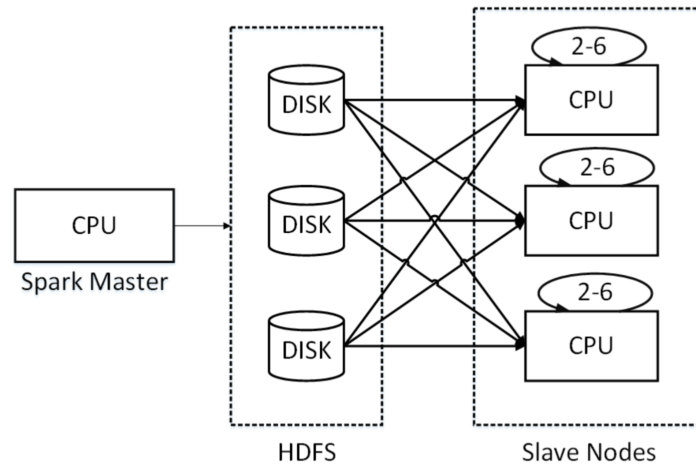


Figure 3.6: K-nn flow of execution on Spark.

3.2.3 K-nn on GPU

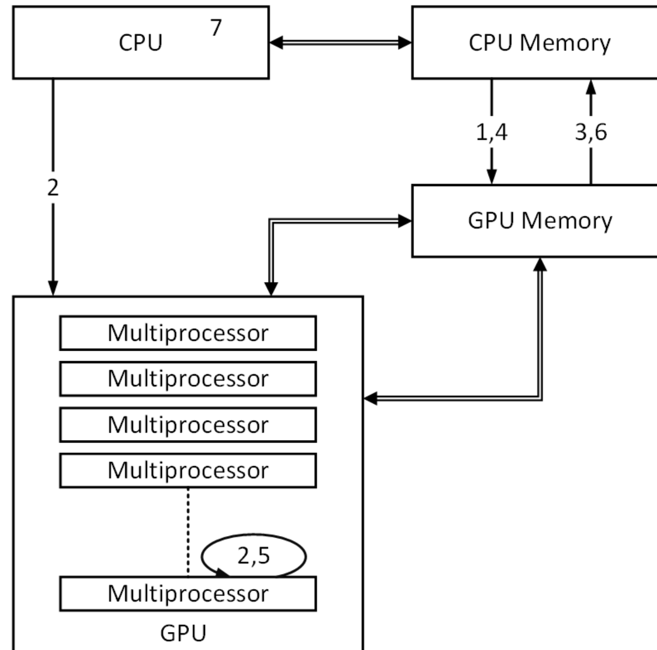


Figure 3.7: K-nn flow of execution on GPU.

Algorithm 9 presents the pseudocode for the process and Fig. 3.7 shows the flow of execution. In GPU implementation, the train and test data are first read and copied into the GPU shared memory. After that, a kernel function which creates threads for parallel operation is called. The number of threads depend on the train

data size and each thread accesses a single row identified by the thread ID. Each thread calculates the Euclidean distances between the train instance and all the test instances. When the kernel finishes working, all the distances are copied into the CPU memory. After that a second kernel function is called for finding the indices of nearest k neighbors. The distance matrix is copied to the GPU shared memory and similar to the first kernel, the threads of the second kernel access a single row which contains the distances from a test row to the train data. The indices of k minimum distances i.e., nearest neighbors are found and returned to the CPU. We do not need to sort the whole row for this purpose. After that, in the CPU a majority voting of the class labels of nearest neighbor decides the class label of the test instances.

Algorithm 9 K-nn on GPU

Inputs: Train Data D, Test Data X, Number of nearest neighbors k
Output: Predicted class labels of X

1: Copy D and X to the GPU shared memory. Split D into threads
2: Kernel 1: Calculate the distance between each $x_i \in X$ and each $d_j \in D$
3: Send the distances to CPU
4: Copy distances to GPU shared memory, split into threads
5: Kernel 2: Find the indices of k smallest distances for each x_i as nearest neighbors
6: Send indices of k nearest neighbors to CPU
7: CPU process: Assign most frequent class label from nearest neighbors as predicted class label

3.2.4 K-nn on Multicore

Algorithm 10 lists the steps in the pseudocode and Fig. 3.8 describes the flow of execution. In multicore CPU, we used Java thread programming to manage the threads. After reading the train and test data, several threads which work in parallel to compute the distances are created. All the threads share the same memory space but work with different train instances. After the distance calculation is done, again another set of threads are created for finding the k indices of nearest neighbors. After that we take a majority voting of the class labels of the nearest neighbors from train

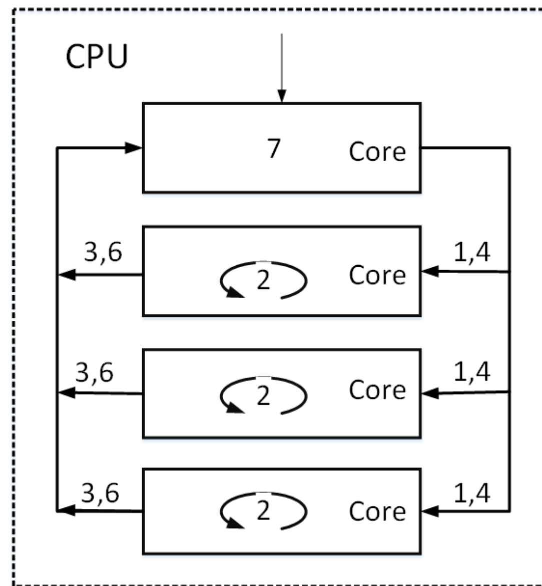


Figure 3.8: K-nn flow of execution on Multicore CPU.

data to assign the predicted class label to the test instance.

Algorithm 10 K-nn on Multicore

Inputs: Train Data D, Test Data X, Number of nearest neighbors k
Output: Predicted class labels of X

1: Split D into multiple cores
2: Calculate the distance between each $x_i \in X$ and each $d_j \in D$
3: Send distances to central core
4: Split the distances into multiple cores
5: Find the indices of k smallest distances for each x_i as nearest neighbors
6: Send indices of k nearest neighbors to central core
7: Assign most frequent class label from nearest neighbors as predicted class label

---

Chapter 4: Results and Discussion

In this section, we will explain the details of the datasets that are being used in our experiments along with the data generation and some properties. After that, we discuss about the setup for experimentation and provide other relevant details. Finally, we present the results of our experiments that were conducted on four big data platforms and discuss about them.

4.1 Dataset Used

In this work, we use the NewsGroup dataset[1] and LibSVM news20 binary classification data[2]. In the NewsGroup collection, there are a total of 19,998 text documents which are grouped by 20 topics. It is a popular dataset for conducting machine learning and text mining related experiments. For K-means, we generated 10 datasets using the NewsGroup corpus by varying the number of features from 30K to 130K, all having 19,998 rows. In matrix representation, the smallest dataset with 30K features has a size of 1.12 Gigabytes and the largest dataset with 130K features has a size of 3.97 Gigabytes. We have also used news20 data for running K-means to show scalability performance of horizontal platforms. The news20 data has 19,996 rows and 1.355 million features. The matrix representation of this data has a size of 50.4 Gigabytes. MapReduce uses sparse representation of the data. The rest of the platforms use regular matrix as input.

For K-nn, we used NewsGroup corpus to create a classification data with 10K rows and 10K features. We created a test dataset by randomly selecting 2,000 rows. The rest 8,000 rows are used to create the training data.

[1]http://qwone.com/ jason/20Newsgroups/
[2]http://www.csie.ntu.edu.tw/ cjlin/libsvmtools/datasets/binary.html

4.2 Experimental Setup

The EC2 (Elastic Computing) instances on AWS utilized for our experiments are listed in Table 4.1. We used the same setup for both K-means and K-nn experiments. In order to be unbiased to all the four platforms, our aim was to carry out experiments in a way that all the platforms cost similar amount of dollars on AWS. In the K-means experiments, we limit the total iterations to 20 for NewsGroup data, and 10 for the large news20 dataset. We vary the number of clusters k parameter as $5, 10, 15, and 20$. For K-nn we conduct experiments by setting the number of nearest neighbours 5 and 10. In all our experiments, the results obtained on all the four platforms are the same and only the time taken to obtain the results is being compared in our analysis.

Table 4.1: Description of the AWS Instances used in our experiments.

Platform Name	Instance Type	Total Instances	Cost Per Hour
Hadoop	m3.large	5	0.65
Spark	r3.large	4	0.7
GPU	g2.2xlarge	1	0.65
Multicore	c3.4xlarge	1	0.84
Single core	c3.4xlarge	1	0.84

The m3.large instance used for Hadoop is a general purpose instance and it has 7.5 GiB (1 GiB $\approx$ 1.074 GB) of memory, processor with 2 CPU cores, 32 GB of SSD storage. We used memory optimized instances for Spark. The r3.large instances has 15.25 GiB of memory, processor with 2 cores, and 32 GB of SSD storage. For GPU, we used g2.2xlarge instance which consists of one NVIDIA GRID GPU (Kepler GK104) which has 1536 CUDA cores. Along with it, the instance has 15 GiB of memory and 60 GB of local instance storage. For multicore platform we use compute optimized instance c3.4xlarge. This instance is based on the Intel Xeon E5-2666 v3 (“Haswell”) processor which is designed to deliver the highest level of computing power. It has a total 30 GiB of memory, a 16 core processor and 320 GB of SSD based storage. Also

for single core experimentation we have used c3.4xlarge instance.

For testing the scalability of horizontal platforms, we ran K-means on Hadoop and Spark clusters with news20 dataset. For this purpose we have created clusters with higher end EC2 instances. For Hadoop m3.2xlarge instance has a 8 core processor, 30 GiB memory and two 80 GB SSD storage disks. For Spark, m3.2xlarge instance has 4 cores in the processor, 30.5 GiB memory, and 80 GB SSD storage. The details are listed in Table 4.2.

Table 4.2: Description of the AWS Instances used for Scalability Test.

Platform Name	Instance Type	Total Instances	Cost Per Hour
Hadoop	m3.2xlarge	8	2.128
Spark	r3.xlarge	6	2.10

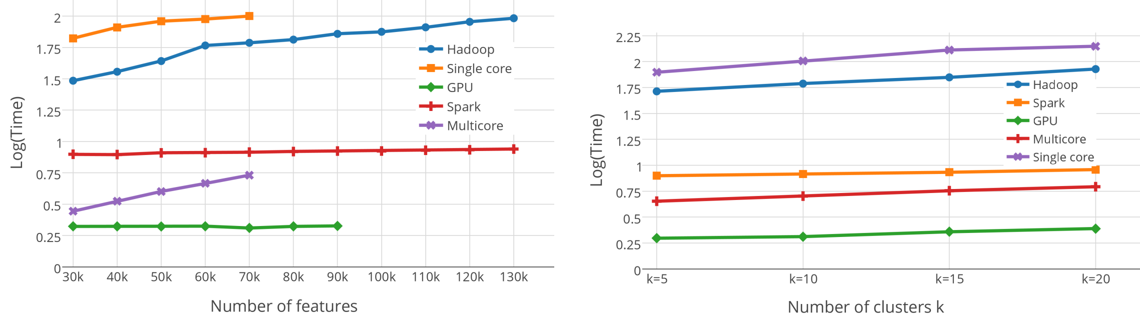
For K-means, we used the MapReduce implementation from this link[3]. We use the GPU implementation by Serban Giuroiu [3]. We wrote the Spark code using Java API. We implemented Multicore program with Java multi-threading. The single core implementation of K-means is also done with standard Java without invoking any parallelization of computation. For K-nn, we used the Mapreduce implementation available at this link[4]. We have implemented naive K-nn on the rest of the platforms. All the running times are reported in minutes.

4.3 Results and Discussion

For K-means, we observe that both the vertical scaling platforms, GPU and Multicore CPU can obtain results better than the horizontal scaling platforms, Hadoop and Spark. Single core implementation is the slowest as expected and GPU is the fastest. It is because the distance calculation, the most time consuming operation of K-means, is performed in GPU by invoking massive parallelization. For the multicore

[3]http://cmj4.web.rice.edu/MapRedKMeans.html
[4]http://cmj4.web.rice.edu/KNN.html

CPU implementation, we run our code on a 15 core machine which obviously has less parallelization capability. Still, it performs considerably well in terms of overall running time. We also observe the physical limitation of vertical scaling platforms. We used the same instance for multicore and single core which has limited memory. It cannot load a data containing more than 70K features, hence it is not able to perform K-means clustering. GPU device also has limited memory as well. While calculating distances, each thread needs to copy the centroids to block shared memory which is allocated per thread. If the data dimension is too big or if k is too large, then it is no longer possible to hold the centroids in the block shared memory. We see that the GPU implementation cannot perform K-means on data containing more than 90K features. Fig. 4.1 shows the comparison results for K-means clustering. The horizontal scaling platforms use distributed storage of data which resides on hard disks of several machines. Vertical scaling platforms uses the same memory to hold the data files, thus reducing file transfer and network latency. This also contributes to a better performance.



(a) Running time for k=10

(b) Running time for number of features = 70K

Figure 4.1: Comparison of running time of K-means algorithm on five different inds of platforms (including single core machine) with varying number of clusters k and number of features. (a) varying the number of features with k=10, (b) varying number of clusters k with 70K features.

Despite of being slower, the horizontal scaling platforms can easily scale with

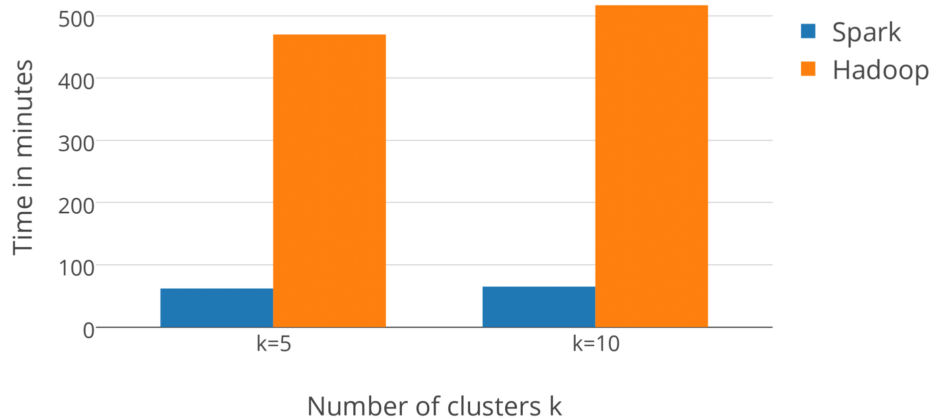


Figure 4.2: Running time of K-menas algorithm with large dataset for scalability test.

bigger datasets. Instead of memory, they use distributed data storage which can hold very big datasets. Fig. 4.2 presents the time comparison on large news20 dataset. In our experiments, we observe that GPU cannot handle a dataset with more than 90K features, and both multicore and single core cannot process a dataset containing more than 70K features. But Hadoop and Spark can comfortably handle a data with even 1.35 million features.

We observe that Spark produces results in orders of magnitude faster than Hadoop with all datasets for K-means. The reason being Spark's capability of holding data and cluster variables in the memory by caching and performing in-memory computations. On the other hand, MapReduce has to read the data and centroids from HDFS at the beginning of every iteration and need to write them back at the end. One iteration cannot simply pass anything to the next iteration. This adds sizeable amount of extra file I/O tasks thus making the program run slower.

We also observe that vertical scaling platforms performing better for K-nn as well. With higher number of threads, GPU produces resutls better than multicore CPU. GPU does massive parallelization of the processes and Multicore offers more active computation units compared to both the Spark and MapReduce paradigm. It also enjoys the advantage of handling everything in the same physical memory unit.

GPU is extremely fast taking only 0.68 minutes to complete K-nn algorithm with the number of nearest neighbours set to 5. With the same data and parameter setting, Hadoop takes 4.174 minutes and Spark takes 3.79 minutes to do the job. But for the cases of vertical scaling, speed comes with scalability limitation. In a single machine, the memory is limited. Therefore, the amount of data that can be loaded each time is also limited. With traditional implementation, it is not possible to handle large datasets.

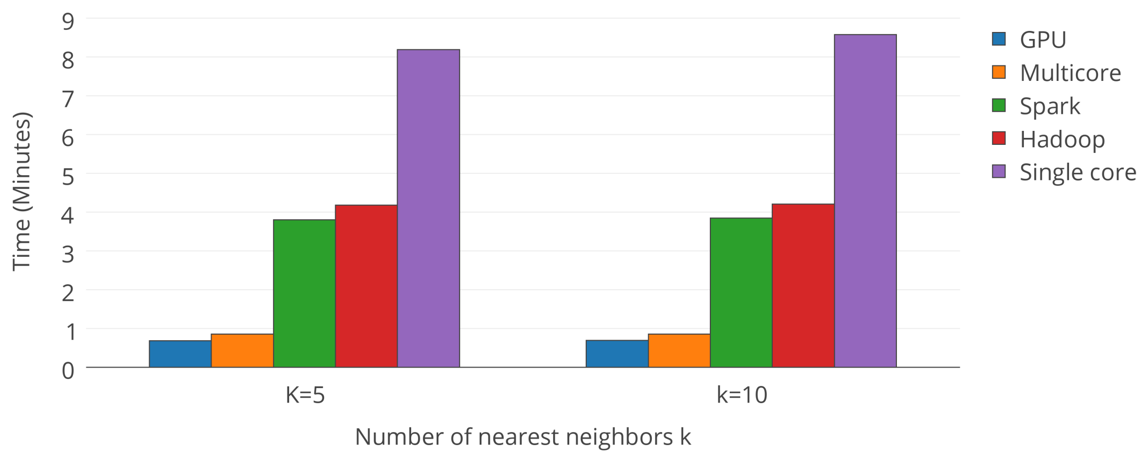


Figure 4.3: Comparision of running time for K-nn algorithm on five platforms.

Chapter 5: Conclusion

The need for big data technologies is evident. Efficient processing of big data can help us gain various critical insights about the data. As the volume and complexity of the data is increasing rapidly, we need scalable platforms which can handle the parallelization of the computations. Traditional platforms neither can scale with the size, nor be able to deal with the computational workload. In this thesis, we first reviewed four popular big data platforms along with their characteristics and technical details. We then discussed the implementation of two prominent data mining algorithms, namely, K-means clustering and K-nn classification, on these platforms along with the strengths and weaknesses of each implementation. We compared the platforms in terms of the running time of the algorithms by varying data sizes parameters such as the and number of clusters (or neighbors).

We observed that vertical scaling platforms are able to produce same results as the horizontal scaling platforms with much lower running time. Here, due to enormous amount of parallelization, GPU performs better than Multicore CPU despite having data transfer latency between host and device. But the vertical scaling platforms lack scalability due to the physical hardware limitations. They cannot process data larger than a certain size. Horizontal scaling methods on the other hand can scale with bigger datasets easily. In this category comparatively newer platform Spark performs better than Hadoop for both K-means and K-nn due the reduced number of disk I/O operations and in-memory computation capabilities.

REFERENCES

[1] Apache hadoop. `https://www.mapr.com/products/apache-hadoop`.

[2] Components of hadoop. `http://bigdataanalyticsnews.com/hadoop-2-0-yarn-architecture`.

[3] Cuda k-means clustering. `http://serban.org/software/kmeans/`, author=Giuroiu, Serban.

[4] Gpu memory hierarchy. `http://www.3dgep.com/cuda-memory-model/`.

[5] Yarn architecture. `http://hortonworks.com/blog/apache-hadoop-yarn-background-and-an-overview/`.

[6] Aggarwal, C. C., and Reddy, C. K. *Data clustering: algorithms and applications.* CRC Press, 2013.

[7] Agneeswaran, V. S., Tonpay, P., and Tiwary, J. Paradigms for realizing machine learning algorithms. *Big Data 1*, 4 (2013), 207–214.

[8] Borthakur, D. Hdfs architecture guide. *HADOOP APACHE PROJECT http://hadoop. apache. org/common/docs/current/hdfs design. pdf* (2008).

[9] Cover, T. M., and Hart, P. E. Nearest neighbor pattern classification. *Information Theory, IEEE Transactions on 13*, 1 (1967), 21–27.

[10] Czajkowski, G. Sorting 1pb with mapreduce. *Google, Blog. Available at googleblog. blogspot. com/2008/11/sorting-1pb-with-mapreduce. html. Last accessed on January 7* (2008), 2012.

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

[12] Gantz, J., and Reinsel, D. Extracting value from chaos. *IDC iview*, 1142 (2011), 9–10.

[13] Ghemawat, S., Gobioff, H., and Leung, S.-T. The google file system. In *ACM SIGOPS operating systems review* (2003), vol. 37, ACM, pp. 29–43.

[14] Lindholm, E., Nickolls, J., Oberman, S., and Montrym, J. Nvidia tesla: A unified graphics and computing architecture. *IEEE micro*, 2 (2008), 39–55.

[15] Lloyd, S. P. Least squares quantization in pcm. *Information Theory, IEEE Transactions on 28*, 2 (1982), 129–137.

[16] Nvidia, C. Programming guide, 2008.

[17] Nvidia, C. Nvidia cuda c programming guide. *NVIDIA Corporation 120* (2011), 18.

[18] Olston, C., Reed, B., Srivastava, U., Kumar, R., and Tomkins, A. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data* (2008), ACM, pp. 1099–1110.

[19] Owen, S., Anil, R., Dunning, T., and Friedman, E. *Mahout in action*. Manning Shelter Island, 2011.

[20] Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E., and Phillips, J. C. Gpu computing. *Proceedings of the IEEE 96*, 5 (2008), 879–899.

[21] Reddy, C. K., and Vinzamuri, B. A survey of partitional and hierarchical clustering algorithms. *Data Clustering: Algorithms and Applications 87* (2013).

[22] Singh, D., and Reddy, C. K. A survey on platforms for big data analytics. *Journal of Big Data 2*, 1 (2015), 1–20.

[23] Thusoo, A., Sarma, J. S., Jain, N., Shao, Z., Chakka, P., Anthony, S., Liu, H., Wyckoff, P., and Murthy, R. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment 2*, 2 (2009), 1626–1629.

[24] Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing* (2013), ACM, p. 5.

[25] White, T. *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.

[26] Wienke, S., Springer, P., Terboven, C., and an Mey, D. Openaccfirst experiences with real-world applications. In *Euro-Par 2012 Parallel Processing*. Springer, 2012, pp. 859–870.

[27] Yu, Y., Isard, M., Fetterly, D., Budiu, M., Erlingsson, Ú., Gunda, P. K., and Currey, J. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI* (2008), vol. 8, pp. 1–14.

[28] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I. 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* (2012), USENIX Association, pp. 2–2.

[29] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing* (2010), vol. 10, p. 10.

[30] Zaharia, M., Das, T., Li, H., Shenker, S., and Stoica, I. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing* (2012), USENIX Association, pp. 10–10.

[31] Zhao, J., and Pjesivac-Grbovic, J. Mapreduce: The programming model and practice. In *Tutorials of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)* (2009), p. 105.

ABSTRACT

PERFORMANCE COMPARISON OF TWO DATA MINING ALGORITHMS ON BIG DATA PLATFORMS

by

MD RAJIUR RAHMAN RAJU

December 2015

Advisor: Dr. Chandan K. Reddy

Major: Computer Science

Degree: Master of Science

In this Big data era, the need for performing large-scale computations is evident. A better understanding of the most suitable platforms which can efficiently run these computations is needed. In this thesis, we attempt to compare four such big data platforms, namely Hadoop, Spark, GPU, and Multicore CPU. We compare these platforms using two prominent data mining algorithms, namely, K-means clustering and K-nearest neighbour classification and discuss specific implementation-level details. We provide several insights into the best possible implementations of these algorithms and systematically compare the benefits and drawbacks of each of these platforms. We conduct experiments by varying data size and parameters to obtain runtime and scalability performances of these platforms. Our experiments show that GPU and Multicore CPU are faster but have certain limitations. On the other hand, Hadoop and Spark are able to handle large scale datasets. We also observe that Spark performs better than Hadoop for both iterative and non-iterative jobs. In summary, we have examined different characteristics of four big data platforms and provided comparative analysis for the cases of two algorithms. Since many other data mining algorithms either use these two methods during pre-processing or as an integral component, we hope that our analysis will have impact in many other applications and

algorithms beyond the ones that are being reported in this thesis.

AUTOBIOGRAPHICAL STATEMENT

MD RAJIUR RAHMAN RAJU

EDUCATION

- Master of Science (Computer Science), December 2015
  Wayne State University, Detroit, MI, USA

- Bachelor of Science in Computer Science and Engineering, February 2011
  Bangladesh University of Engineering and Technology, Dhaka-1000, Bangladesh

PUBLICATIONS

1. RAJIUR RAHMAN AND CHANDAN K. REDDY Electronic Health Records: A Survey. In *Healthcare Data Analytics, Chandan K. Reddy and Charu C. Agarwal (eds.), CRC Press* (2014).

2. [SUBMITTED] RAJIUR RAHMAN AND CHANDAN K. REDDY Performance Comparison of Two Data Mining Algorithms on Big Data Platforms. In *BioMed Central Big Data Analytics* (2015).