# TERM PROJECT REVIEW – III REPORT

# COMPARING OPTIMAL COFLOW

**SCHEDULING ALGORITHMS FOR DATA**

**CENTER NETWORKS**

**SUBMITTED BY**

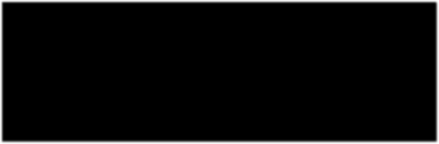
|  |  |  |
| --- | --- | --- |
| **NAME** | **REG NO** | **EMAIL** |
| **RITWIK RANJAN** | **17BCE0488** | **ritwik.ranjan2017@vitstudent.ac.in** |
| **SIMRAN KOUL** | **17BCE2210** | **simran.koul2017@vitstudent.ac.in** |

## **CSE4001: PARALLEL AND DISTRIBUTED COMPUTING**

**PROF. BALAMURUGAN.R**

### SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

**NOVEMBER 2019**



|  |  |
| --- | --- |
| **Title** | **Page Number** |
| Title Pages | i |
| Table of Contents | ii |
| Project Management Plan | 1 |
| Overview | 1 |
| Scope and Objectives | 2 |
| Assignment of Roles and Responsibilities | 3 |
| Project Schedule | 4 |
| Requirements Specifications | 5 |
| Analysis | 7 |
| Design | 8 |
| Implementation | 13 |
| Test Documentation | 23 |
| Appendices | 25 |
| Source Code | 25 |
| References | 26 |

1. Project Management Plan

#### Overview

Due to the need of processing large datasets in the modern world of computing, there is a need of speeding up these processes so that they don’t take large amounts of time to complete. If we use traditional serial computing methods (executing a program or a set of instructions in a sequential manner one by one) would take not only large amounts of time to finish but they will also use quite a bit of precious resources in the form of memory, processor cycles etc.

In order to reduce this time, various ways of developing algorithms which would enable execution of various chunks of programs in a parallel fashion have been proposed. All these algorithms have certain advantages and draw-backs too. Some ensure fast processing speed; others ensure that the time taken for program to terminate and display output is less. However, they also run into certain problems such as data dependency issues, memory issues etc.

Parallel computing is a type of computer running algorithm in which many calculations or the execution of various different kinds of processes are carried out at the same time. Large problems are often divided into smaller ones, which can then be solved at the same time. There are several different types of parallel computing: bit-level, data level, instruction level and task parallelism. Parallelism has traditionally been used in highperformance computing, but it's being used in other paradigms due to the physical constraints on processor size and transistor speed preventing frequency scaling. As power consumption and heat generation by computers has become a concern in recent times, parallel computing has become the dominant field in computer architecture, mainly these problems are solved using multi-core processors.

One major area where parallel programming may be used is in our current data centres. Data centres are large servers which process huge amounts of data. This data may be in Gigabytes or even more than that. It would be greatly beneficial if a good, optimal parallel programming algorithm can be developed for this purpose, which balances the benefits of parallelism with minimum drawbacks.

In current data centres, an algorithm (e.g, MapReduce, Dryad etc.) usually generates a stream of parallel flows to complete a job. These flows comprise a coflow and only completing them all will be beneficial to the application. Hence, minimizing the average Coflow Completion Time (CCT) becomes an important objective of flow scheduling. However, achieving this goal in today’s Data Centre Networks (DCNs) is big challenging, not only because the schedule problem is NP-hard, but also because it is tough to execute practical flow scheduling in large-scale DCNs.

Generally, the term flow is used to indicate a string of packets from a source computer to a destination belonging to the same data transport task, for example when a worker reads the input data from another worker, or replicates the result to a replica node of the cluster file systems.

In this project, we find that minimising the average CCT of a set of coflows is equal to the well-known solution of minimizing the sum of completion times in a concurrent open shop. As there are lots of preexisting solutions for the concurrent open shop, we test a variety of techniques for coflow scheduling. Inspired by what we found out by researching, we derive a comparision between 3 algorithms for coflow scheduling, and further develop an analysis about the coflow scheduling algorithms which avoids the system problems associated with current centralized proposals and also addressing the performance challenges of decentralized suggestions.

#### Scope and Objectives

This project seeks to find out how are the various methods and parameters along which coflow scheduling algorithms works. This means that each algorithm has it’s own advantages and disadvantages and the scope of this project was to compare them with each other and understand which one would be suitable for what situations on DCNs.

Parallel computing is closely related to concurrent computing such that they are more or less used together, and often used interchangeably, though the two are distinctly different: it is possible to have concurrency without parallelism (such as multitasking by time-sharing on a single-core CPU) and it is possible to have parallelism without concurrency (such as bit-level parallelism). In parallel computing, a computational task is typically broken down into many similar subtasks that can be looked upon independently and whose outputs are combined afterwards, when all the tasks individually get completed. In contrast, in concurrent computing, the various processes often do not address related tasks; when they do, as is typical in distributed computing, the separate tasks may have a varied nature and often require some inter-process communication during execution.

One major area where parallel programming may be used is in our current data centres. Data centres are large servers which process huge amounts of data. This data may be in Gigabytes or even more than that. It would be greatly beneficial if a good, optimal parallel programming algorithm can be developed for this purpose, which balances the benefits of parallelism with minimum drawbacks.

As there are lots of preexisting solutions for the concurrent open shop, we test a variety of techniques for coflow scheduling. Inspired by what we found out by researching, we derive a comparison between 3 algorithms for coflow scheduling, and further develop an analysis about the coflow scheduling algorithms which avoids the system problems associated with current centralized proposals and also addressing the performance challenges of decentralized suggestions.

#### Assignment of Roles and Responsibilities

This project involved various challenging activities which would not have been possible to completion without the valuable contribution of all the members of the project team.

Various aspects of the project such as researching of the topic, looking at the base papers and analysing it, looking up existing research papers on this topic area were done collectively

In addition to this, the execution part of the project- collecting datasets and implementing a suitable community detection algorithm on these datasets to analyse diversity and communities was a team effort.

There was a lot of trial and error involved as well such that every member suggested various changes that would make the topic that we were researching better suited to the results that we desired. In this regard there are a lot of contributions which hasn’t been documented per-say but will be recognized nevertheless.

The project involved various stages such as deciding the processes, research, requirements analysis, design as well as the implementation phase and the testing phase.

## 2.Requirements Specification

### Functional Requirements

These are the requirements that need to be fulfilled in order to realize the desired outcome of the project. This means setting down all the things that are required from the research product that we’re building on paper to realize our potential. We then worked towards realizing all the following requirements:

1. Tool to partition input dataset in such a way which would be suitable for processing using MapReduce algorithm
2. Division of large input dataset into subtasks suitable for servers to process parallel
3. Map function to dynamically decide which subtask will be processed by which server
4. Reduce function to collate all output data from every server to produce a final unified output
5. Data analysis display about the time taken, number of processes spawned etc

### Non- functional Requirements

These are the requirements that are for supporting the application. They are the ones which are not directly visible to the user but are required for the correct operation of the application.

1. **Operating systems**: Windows, Linux, Ubuntu etc
2. **Design constraints:** The tool’s success depends on the availability of Facebook, Twitter and LinkedIn datasets. In addition, these platforms also have to allow the download of personal user’s data, an issue which may prove to be a sensitive issue.
3. **Implementation constraints:** Building of the tool depends on the architecture of the machine on which it is being executed. We might have to take permission before publishing the tool also
4. **Dependencies:** The dataset collection tool’s efficiency will depend on the size being downloaded as well. If the dataset is too big the software may run into problems.
5. **User interface:** The user will interact with the tools by means of a computer application. This will be connected to the internet to be able to download the dataset.
6. **Hardware interface:** The application would require a minimum of

#### a. 2 GB RAM

1. 40 MB dataset buffer on ROM
2. 100 MB space for dataset fetcher + graphing tool

7. **Software interface:** We would require Facebook, Twitter platforms for datasets

## 3.Analysis

**MapReduce:**

Hadoop MapReduce is a software framework for easily writing applications which process vast amounts of data (multi-terabyte data-sets) in parallel on large clusters (thousands of nodes) of commodity hardware in a reliable, fault-tolerant manner.

A MapReduce job, for the most part, splits the input data collection into independent lumps, which are handled by the map tasks in a totally parallel way. The framework sorts the yields of the maps, which are then inputted to the reduce tasks. Commonly both the input and the yield of the job are stored in a file-system framework. The framework deals with scheduling tasks, observing them and re-executes the failed assignments.

Typically the compute nodes and the storage nodes are the same, that is, the MapReduce framework and the Hadoop Distributed File System (see HDFS Architecture Guide) are running on the same set of nodes. This configuration allows the framework to effectively schedule tasks on the nodes where data is already present, resulting in very high aggregate bandwidth across the cluster.

The MapReduce framework consists of a single master JobTracker and one slave TaskTracker per cluster node. The master is responsible for scheduling the jobs' component tasks on the slaves, monitoring them and re-executing the failed tasks. The slaves execute the tasks as directed by the master.

Minimally, applications specify the input/output locations and supply map and reduce functions via implementations of appropriate interfaces and/or abstract-classes. These and other job parameters comprise the job configuration. The Hadoop job client then submits the job (jar/executable etc.) and configuration to the JobTracker which then assumes the responsibility of distributing the software/configuration to the slaves, scheduling tasks and monitoring them, providing status and diagnostic information to the job-client.

**Spark:**

Apache Spark is an open-source distributed all-purpose cluster-computing framework. Spark provides an interface for programming entire clusters with inherent data parallelism and fault tolerance.

Spark programming begins with a data set or few, usually remaining in some form of distributed, persistent storage like the Hadoop Distributed File System (HDFS). Writing a Spark program comprises of some related steps:

* Specifying a set of conversions on input data sets.
* Summoning operations that output the transformed data sets to persistent storage or return results to the driver’s local memory.
* Running local computations that operate on the results computed in a distributed manner. These can help you decide what transformations and actions to try next.

Understanding Spark means understanding the intersection between the two sets of abstractions the framework offers: storage and execution. Spark couples these abstractions in a concise fashion that typically concedes any middle step in a data processing pipeline to be cached in memory for later use.

### Deployment Norms for Spark

Spark can be deployed in an old-fashioned on-premises data centre as well as within the cloud. The cloud permits organisations to deploy Spark while not the requirement to accumulate hardware or specific setup experience. Enterprise Strategy Group (esg-global.com) found forty-three per cent of respondents considering cloud as their primary deployment for Spark. The highest reasons customers perceived the cloud as a bonus for Spark are a quicker time to launching, better availability, a lot of frequent feature/functionality updates, more flexibility, a lot of geographic coverage, and prices connected to real utilisation.

## 4. Design

**MapReduce:**

### Input and Output

The MapReduce framework operates exclusively on <key, value> pairs, that is, the framework views the input to the job as a set of <key, value> pairs and produces a set of <key, value> pairs as the output of the job, conceivably of different types.

Input and Output types of a MapReduce job:

(input) <k1, v1> -> map -> <k2, v2> -> combine -> <k2, v2> -> reduce -> <k3, v3> (output)

### Working

This section provides a reasonable amount of detail on every user-facing aspect of the MapReduce framework. Let us first take the Mapper and Reducer interfaces. Applications typically implement them to provide the map and reduce methods. Applications typically implement the Mapper and Reducer interfaces to provide the map and reduce methods. These form the core of the job.

There are other core interfaces like JobConf, JobClient, Partitioner, OutputCollector, Reporter, InputFormat, OutputFormat, OutputCommitter

### Mapper

Mapper maps input key/value pairs to a set of intermediate key/value pairs.

Maps are the individual tasks that transform input records into intermediate records. The transformed intermediate records do not need to be of the same type as the input records. A given input pair may map to zero or many output pairs. The Hadoop MapReduce framework spawns one map task for each InputSplit generated by the InputFormat for the job.

Overall, Mapper implementations are passed the JobConf for the job via the

JobConfigurable.configure(JobConf) method and override it to initialize themselves. The framework then calls map (WritableComparable, Writable, OutputCollector, Reporter) for each key/value pair in the

InputSplit for that task. Applications can then override the Closeable.close() method to perform any required clean up.

Output pairs do not need to be of the same types as input pairs. A given input pair may map to zero or many output pairs. Output pairs are collected with calls to OutputCollector.collect(WritableComparable,Writable).

The Mapper outputs are sorted and then partitioned per Reducer. The total number of partitions is the same as the number of reduce tasks for the job. Users can control which keys (and hence records) go to which Reducer by implementing a custom Partitioner.

**How Many Maps?**

The number of maps is usually driven by the total size of the inputs, that is, the total number of blocks of the input files.

The right level of parallelism for maps seems to be around 10-100 maps per-node, although it has been set up to 300 maps for very CPU-light map tasks. Task setup takes a while, so it is best if the maps take at least a minute to execute.

Thus, if you expect 10TB of input data and have a block size of 128MB, you'll end up with 82,000 maps, unless setNumMapTasks(int) (which only provides a hint to the framework) is used to set it even higher.

### Reducer

Reducer reduces a set of intermediate values, which share a key to a smaller set of values.

The number of reduces for the job is set by the user via JobConf.setNumReduceTasks(int).

Overall, Reducer implementations are passed the JobConf for the job via the

JobConfigurable.configure(JobConf) method and can override it to initialize themselves.

The framework then calls reduce (WritableComparable, Iterator, OutputCollector, Reporter) method for each <key, (list of values)> pair in the grouped inputs. Applications can then override the Closeable.close() method to perform any required clean up.

Reducer has 3 primary phases: shuffle, sort and reduce.

#### Shuffle

The input to the Reducer is the sorted output of the mappers. In this phase, the framework fetches the relevant partition of the output of all the mappers, via HTTP.

#### Sort

The framework groups Reducer inputs by keys (since different mappers may have output the same key) in this stage. The shuffle and sort phases occur simultaneously; while map-outputs are being fetched they are merged.

#### Secondary Sort

If equivalence rules for grouping the intermediate keys are required to be different from those for grouping keys before reduction, then one may specify a Comparator via JobConf.setOutputValueGroupingComparator(Class). Since JobConf.setOutputKeyComparatorClass(Class) can be used to control how intermediate keys are grouped, these can be used in conjunction to simulate secondary sort on values.

#### Reduce

In this phase, the reduce(WritableComparable, Iterator, OutputCollector, Reporter) method is called for each <key, (list of values)> pair in the grouped inputs. The output of the reduce task is typically written to the FileSystem via OutputCollector.collect(WritableComparable, Writable). Applications can use the Reporter to report progress, set application-level status messages and update Counters, or just indicate that they are alive. The output of the Reducer is not sorted.

**How Many Reduces?**

The right number of reduces seems to be 0.95 or 1.75 multiplied by (<*no. of nodes*> \* mapred.tasktracker.reduce.tasks.maximum).

With 0.95 all of the reduces can launch immediately and start transferring map outputs as the maps finish. With 1.75 the faster nodes will finish their first round of reduces and launch a second wave of reduces doing a much better job of load balancing.

Increasing the number of reduces increases the framework overhead, but increases load balancing and lowers the cost of failures.

The scaling factors above are slightly less than whole numbers to reserve a few reduce slots in the framework for speculative-tasks and failed tasks.

**Spark:**

Apache Spark has as its architectural foundation the resilient distributed dataset (RDD), a read-only multi-set of data things distributed over a cluster of machines, that's maintained during a fault-tolerant manner. The RDD technology still underlies the Dataset API.

Spark and its RDDs were developed in 2012 in response to deficiencies within the MapReduce cluster computing paradigm that forces a specific linear data-flow structure on distributed programs: MapReduce programs scan input file from disk, map a function across the data, cut back the results of the map, and store reduction results on disk. Spark's RDDs function as a working set for distributed programs that gives a (deliberately) restricted style of distributed shared memory.

Spark facilitates the implementation of both iterative algorithms that visit their knowledge set multiple times during a loop, and interactive/exploratory data analysis, i.e., the repeated database-style querying of knowledge. The latency of such applications is also reduced by many orders of magnitude compared to a MapReduce implementation.

### Spark Core

Spark Core is that the foundation of the comprehensive project. It provides distributed task dispatching, scheduling, and basic I/O functionalities, exposed through an application programming interface (for Java, Python, Scala, and R) focused on the RDD abstraction.

A typical example of RDD-centric functional programming is that the following Scala program that computes the frequencies of all words occurring during a set of text files and prints the most prevalent ones. Each map, flatMap (a variant of map) and reduceByKey takes an anonymous function that performs an easy operation on one knowledge item (or a pair of items) and applies its argument to remodel an RDD into a brand new RDD.

### Spark SQL

Spark SQL could be a part of Spark Core that introduced a data abstraction known as DataFrames, that gives assistance for structured and semi-structured knowledge. Spark SQL provides a domain-specific language (DSL) to manage DataFrames in Scala, Java, or Python. It additionally provides SQL language support, with command-line interfaces and ODBC/JDBC server. Although DataFrames lack the compile-time type checking yielded by RDDs, as of Spark 2.0, the powerfully written DataSet is totally supported by Spark SQL further.

### Spark Streaming

Spark Streaming uses Spark Core's quick scheduling aptitude to perform streaming analytics. It ingests knowledge in mini-batches and performs RDD transformations on those mini-batches of knowledge. This style permits the same set of application code written for batch analytics to be employed in streaming analytics, therefore facilitating simple implementation of lambda architecture. However, this convenience comes with the penalty of latency equal to the mini-batch length.

### MLib Machine Learning Library

Spark MLlib could be a distributed machine-learning framework on top of Spark Core that, due in massive part to the distributed memory-based Spark design, is the as much as nine fold as quick because of the disk-based implementation employed by Apache driver.

Many common machine learning and applied math algorithms are enforced and are shipped with MLlib that simplifies large-scale machine learning pipelines, including:

* summary statistics, correlations, stratified sampling, hypothesis testing, random data generation
* classification and regression: support vector machines, logistical regression, linear regression, decision trees, naive Bayes classification
* collaborative filtering techniques together with alternating least squares (ALS)
* cluster analysis ways together with k-means, and latent Dirichlet allocation (LDA)
* dimensional reduction techniques like singular value decomposition (SVD), and principal component analysis (PCA)
* feature extraction and transformation functions
* optimisation algorithms like stochastic gradient descent, limited-memory BFGS (L-BFGS)

### GraphX

GraphX is a distributed graph-processing framework on prime of Apache Spark. As a result of it's supported RDDs, which are changeless, graphs are changeless and therefore GraphX is unsuitable for graphs that require to be updated, let alone in a transactional way like a graph database.

**Data Set**

We’re going to use [an example data set](http://bit.ly/1Aoywaq) from the UC Irvine Machine Learning Repository, which is a wonderful reference for a variety of interesting (and free) data sets. The dataset we’ll be investigating was curated from a record linkage study that was performed at a German hospital in 2010, and it contains several million pairs of patient records that were matched according to various diverse criteria, such as the patient’s name (first and last), address and birthday. Each matching field was allocated a numerical score from 0.0 to 1.0 based on how comparable the strings were, and the data was then hand-labelled to identify which pairs described the same person and which did not. The underlying values of the fields themselves that were used to create the data set were excluded to preserve the privacy of the patients, and numerical identifiers, the match scores for the fields, and the label for each pair (match versus nonmatch) were published for use in record linkage research.

## 

5.Implementation

**MapReduce:**

The MapReduce algorithm is mainly split into 2 main parts. The first specifies the function to be performed on various distributed servers and the second specifies how each server’s data will be combined to give final output.

MapReduce algorithm is best used on large datasets that require a lot of processing power to interpret data. The large dataset can take advantage of MapReduce’s ability to run parallelly on distributed servers, and later combine all the data to finally interpret it.

For eg, if we have a 100 GB text file that is to be processed for some data, the file's contents can be divided across five 20 GB servers (MAP stage). Later the main server can take all the outputs and produce the final result (REDUCE stage).

The steps implemented by us in implementing the Hadoop MapReduce algorithm are as follows:

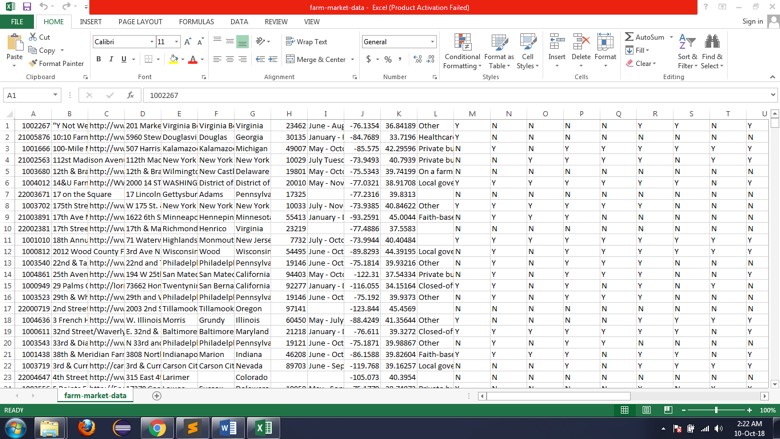
1. Since we wanted to test the MapReduce algorithm on a big data sets, we looked for freely available datasets on the internet that can we analysed.

The US govt maintains a directly of datasets of various things in its country such as ZIP code data, precipitation data, crime data, farmers’ data etc

We chose the following dataset to conduct our test:

[https://catalog.data.gov/dataset/Agriculture/Farmers-Markets-Geographic-Data/ wfna-38ey](https://catalog.data.gov/dataset/Agriculture/Farmers-Markets-Geographic-Data/wfna-38ey)

The raw datasets looked like this:



1. It has over 7000 rows of data and total size is of 10 MB. It contains information about farmer markets in the USA. It displays the following information:
   1. Locations in USA (street, city, county, state)
   2. Zip
   3. Market Name
   4. Latitude-Longitude details
   5. Classification of various aspects on the market and whether they exist or not (indicated as Y for yes and N for no). This runs from columns M to column AK
2. Devised the Map method to separate each row of the dataset. Each row’s rating can be calculated by an individual server. And the values can be returned to the Reducer.

City and state are taken as the Key values.

The rating is calculated based on the facilities available at the market from columns 11-31.

If a facility is available, it’s rating count increases.

Finally, the overall rating is normalized based on the formula:

rating=(count\*100)/21

as there are 21 columns

The Map function in Java is as follows:

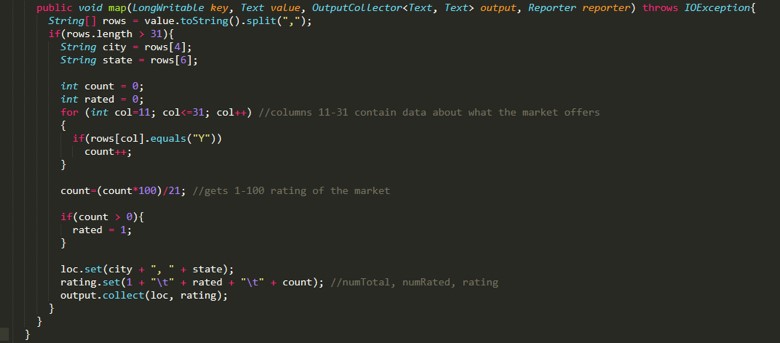
1. Then the Reduce function is defined. It takes data from all the distributed Map functions and combines them

Here, it counts the number of times a town and state has occurred (key values) and combines all of their ratings to give a final value out of 100.

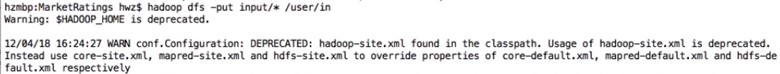
The function definition is as follows:

1. Define the main function that will make MapReduce work.
2. Compile Java program using Hadoop directive

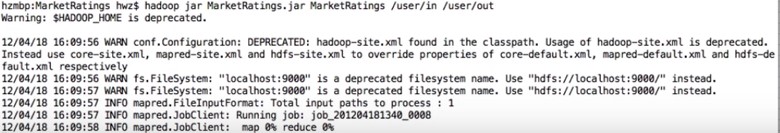




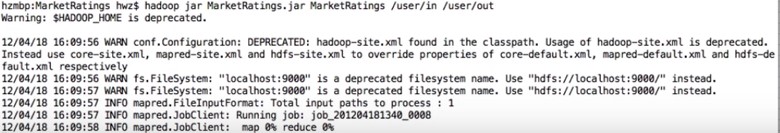
1. Make distributed file system using Hadoop's dfs to run programs on distributed servers.

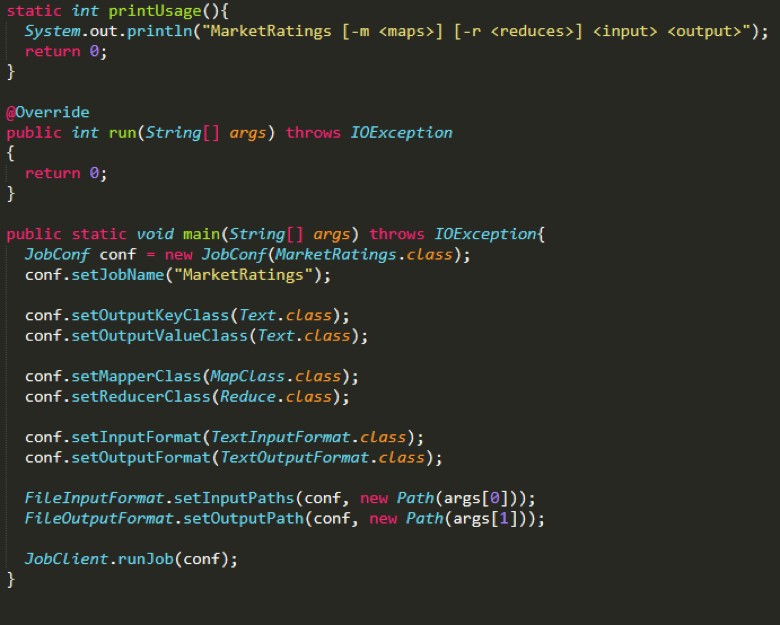


1. Copy the input file into Hadoop’s distributed file system



1. Run the Java file

Wait for both Map and Reduce functions to execute



1. See the summary of how much data was processed:
2. See the output file:

Here we can see that the output file consists of the following:

* 1. City, State
  2. Whether rating was given or not
  3. Number of Markets
  4. Rating of all markets out of 100

**Spark:**

Here we use Apache Spark to train the the algorithm with the data set and find that the training error to test the fault tolerance.

### Resilient Distributed Datasets

An RDD is laid out across the cluster of machines as a collection of partitions, each including a subset of the data. Partitions define the unit of parallelism in Spark. The framework processes the objects within a partition in sequence, and processes multiple partitions in parallel. One of the simplest ways to create an RDD is to use the parallelise method on SparkContext with a local collection of objects:

val rdd = sc.parallelize(Array(1, 2, 2, 4), 4)

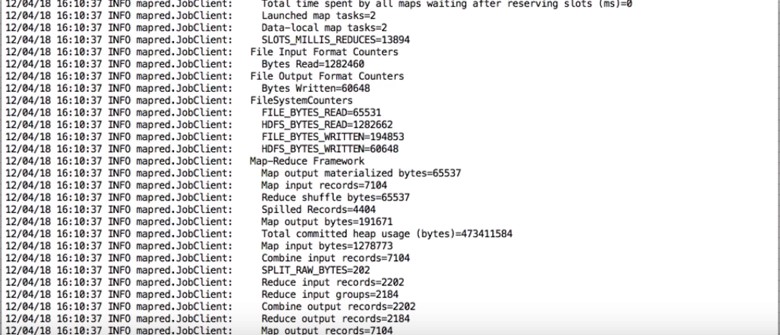
... rdd: org.apache.spark.rdd.RDD[Int] = ...

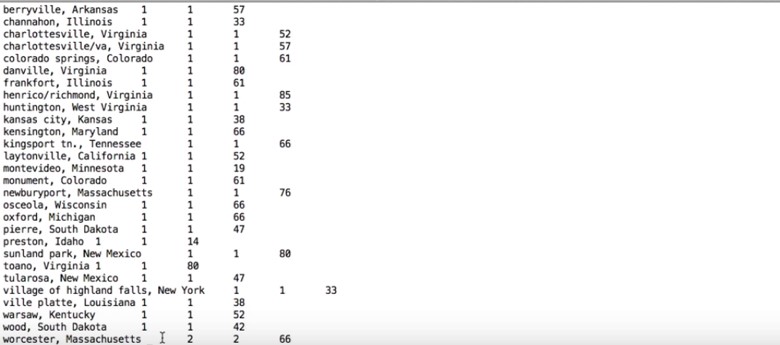
The first argument is the collection of objects to parallelise. The second is the number of partitions. When the time comes to compute the objects within a partition, Spark fetches a subset of the collection from the driver process.

To create an RDD from a text file or directory of text files residing in a distributed filesystem like HDFS, we can pass the name of the file or directory to the textFile method:

val rdd2 = sc.textFile("hdfs:///some/path.txt")

... rdd2: org.apache.spark.rdd.RDD[String] = ...





When you’re running Spark in local mode, the textFile method can access paths that reside on the local filesystem. If Spark is given a directory instead of an individual file, it will consider all of the files in that directory as part of the given RDD. Finally, note that no actual data has been read by Spark or loaded into memory yet, either on our client machine or the cluster. When the time comes to compute the objects within a partition, Spark reads a section (also known as a split) of the input file, and then applies any subsequent transformations (filtering, aggregation, etc.) that we defined via other RDDs.

### Actions trigger execution and return a non-RDD result

Command:

rdd.first() Result:

|  |  |
| --- | --- |
| u'"id\_1","id\_2","cmp\_fname\_c1","cmp\_fname\_c2","cmp\_lname\_c1","cmp\_lname\_c2","cmp\_sex","cmp\_bd | |
| ","cmp\_bm","cmp\_by","cmp\_plz","is\_match"' |  |

Command:

rdd.take(10) Result:

|  |  |  |  |
| --- | --- | --- | --- |
| [u'"id\_1","id\_2","cmp\_fname\_c1","cmp\_fname\_c2","cmp\_lname\_c1","cmp\_lname\_c2","cmp\_sex","cmp\_bd | | | |
| ","cmp\_bm","cmp\_by","cmp\_plz","is\_match"', | |  | |
| u'37291,53113,0.833333333333333,?,1,?,1,1,1,1,0,TRUE', | | |  |
| u'39086,47614,1,?,1,?,1,1,1,1,1,TRUE', u'70031,70237,1,?,1,?,1,1,1,1,1,TRUE', u'84795,97439,1,?,1,?,1,1,1,1,1,TRUE', |  | |
| u'36950,42116,1,?,1,1,1,1,1,1,1,TRUE', |
| u'42413,48491,1,?,1,?,1,1,1,1,1,TRUE', u'25965,64753,1,?,1,?,1,1,1,1,1,TRUE', u'49451,90407,1,?,1,?,1,1,1,1,0,TRUE', |
| u'39932,40902,1,?,1,?,1,1,1,1,1,TRUE'] |

Command:

|  |  |
| --- | --- |
| **def** is\_header(line): |  |
| **return** "id\_1" **in** line | |

### Transforms return an RDD and are lazy

Command:

vals = rdd.filter(**lambda** x: **not** is\_header(x)) vals

PythonRDD[4] at RDD at PythonRDD.scala:42 vals.count()

Output:

5749132

### Now it is evaluated

Command:

vals.take(10) Result:

|  |  |
| --- | --- |
| [u'37291,53113,0.833333333333333,?,1,?,1,1,1,1,0,TRUE', | |
| u'39086,47614,1,?,1,?,1,1,1,1,1,TRUE', u'70031,70237,1,?,1,?,1,1,1,1,1,TRUE', u'84795,97439,1,?,1,?,1,1,1,1,1,TRUE', |  |
| u'36950,42116,1,?,1,1,1,1,1,1,1,TRUE', |
| u'42413,48491,1,?,1,?,1,1,1,1,1,TRUE', u'25965,64753,1,?,1,?,1,1,1,1,1,TRUE', u'49451,90407,1,?,1,?,1,1,1,1,0,TRUE', u'39932,40902,1,?,1,?,1,1,1,1,1,TRUE', |
| u'46626,47940,1,?,1,?,1,1,1,1,1,TRUE'] |

Each time we access vals, it is *reconstructed* from the original sources. Spark maintains a DAG of how each RDD was constructed so that data sets can be reconstructed- hence *resilient distributed datasets*. However, this is inefficient.

#### "vals" is reconstructed again

Command:

vals.first()

Result: u'37291,53113,0.833333333333333,?,1,?,1,1,1,1,0,TRUE'

#### Spark allows us to persist RDDs that we will be re-using

Command:

Output:

Command:

Result:

Command:

Result:

#### Parse lines and work on them

Commands: **def** parse(line):

pieces = line.strip().split(',') id1, id2 = map(int, pieces[:2])

scores = [np.nan **if** p=='?' **else** float(p) **for** p **in** pieces[2:11]] matched = True **if** pieces[11] == 'TRUE' **else** False **return** [id1, id2, scores, matched] mds = vals.map(**lambda** x: parse(x)) mds.cache()

Output:

PythonRDD[10] at RDD at PythonRDD.scala:42

Commands:

match\_counts = mds.map(**lambda** x: x[-1]).countByValue()

|  |  |
| --- | --- |
| **for** cls **in** match\_counts: |  |
| **print** cls, match\_counts[cls] | |

Result:

|  |  |
| --- | --- |
| False 5728201 | |
| True 20931 |  |

## 6. Test Documentation

**MapReduce:**

The MapReduce algorithm was run on varying lengths of datasets. We did this to test the effects of MapReduce's parallel algorithm and how effective it is on various large datasets in terms of time it takes to execute the program.

|  |  |  |  |
| --- | --- | --- | --- |
| **Sr. No** |  | **Lines of text in an input file** | **Time taken to run**  **MapReduce program** |
|  | 1 | 5 | 0.1s |
|  | 2 | 500 | 0.7s |
|  | 3 | 1000 | 1.5s |
|  | 4 | 2000 | 3.2s |
|  | 5 | 5000 | 9s |
|  | 6 | 20000 | 23s |
|  | 7 | 50000 | 52s |

We can see that for a very high number of lines in the input dataset, due to Map function allocation overheads, the time taken to execute the program becomes very high. This is because of the costs to allocate chunks of the dataset to various servers (distributed file system architecture of Hadoop MapReduce)

However, still, when compared to serial execution, parallel execution takes much lesser time.

|  |  |  |  |
| --- | --- | --- | --- |
| **Sr. No** |  | **Lines of text in the input file** | **Time taken to run a Serial program** |
|  | 1 | 5 | 0.1s |
|  | 2 | 500 | 0.7s |
|  | 3 | 1000 | 1.5s |
|  | 4 | 2000 | 4.5s |
|  | 5 | 5000 | 15s |
|  | 6 | 20000 | 30s |
|  | 7 | 50000 | 100s |

**Spark:**

### Summary Statistics

Command:

mds.map(**lambda** x: x[2][0]).stats()

Result:

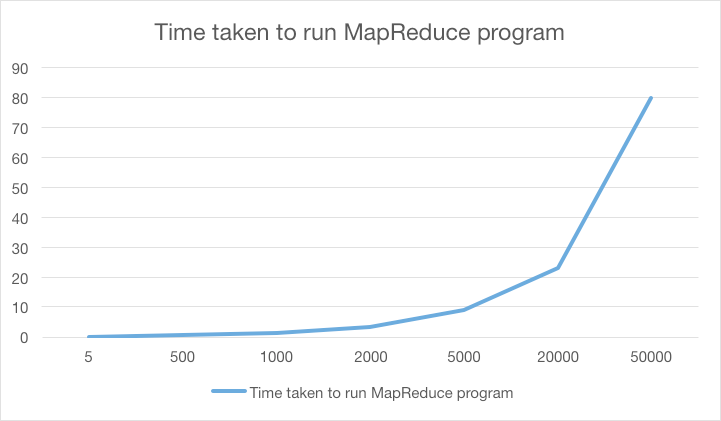
(count: 5749132, mean: nan, stdev: nan, max: nan, min: nan)

Command:

mds.filter(**lambda** x: np.isfinite(x[2][0])).map(**lambda** x: x[2][0]).stats()

Result:

(count: 5748125, mean: 0.712902470443, stdev: 0.3887583258, max: 1.0, min: 0.0)



### Using MLib for Regression

Results:

[0, 2, 4, 5, 6, 7, 8]

5160175 574313

5734488 5160175 574313

Training Error = 0.00356774093569

We trained the the algorithm (given in appendix) with the data set and found that the training error with this data set is very low (given above). This proves that Apache Spark is a very reliable framework for data parallelism and fault tolerance.

## 7. Appendix

**Spark Source Code:**

|  |  |  |  |
| --- | --- | --- | --- |
| **from** **pyspark.mllib.classification** **import** LogisticRegressionWithSGD | | | |
| **from** **pyspark.mllib.regression** **import** LabeledPoint | | |  |
| **def** parsePoint(md): |  |
| **return** LabeledPoint(md[-1], md[2]) | |

full\_count = mds.count()

|  |  |  |
| --- | --- | --- |
| *# Only use columns with less than 20% missing as features* | |  |
| idxs = [i **for** i **in** range(9) **if** |  |
| mds.filter(**lambda** p: np.isfinite(p[2][i])).count() > 0.8\*full\_count] | | |

data = mds.filter(**lambda** p: np.all(np.isfinite(np.array(p[2])[idxs]))).map(**lambda** p: parsePoint(p)) train\_data, predict\_data = data.randomSplit([0.9, 0.1]) model = LogisticRegressionWithSGD.train(train\_data)

|  |  |
| --- | --- |
| labelsAndPreds = predict\_data.map(**lambda** p: (p.label, model.predict(p.features))) |  |
| trainErr = labelsAndPreds.filter(**lambda** (v, p): v != p).count() / float(predict\_data.count()) | |

**print** "Training Error = " + str(trainErr)

**References:**

1. Mining diversity on social media networks: Lu Liu · Feida Zhu · Meng Jiang · Jiawei Han· Lifeng Sun · Shiqiang Yang
2. Emergence of communities and diversity in social networks: Xiao Han, Shinan Cao, Zhesi Shen, Boyu Zhang, Wen-Xu Wang, Ross Cressman, and H. Eugene Stanley
3. Diversity in social support by role relations: A typology: FilipAgneessensHansWaegeJohnLievens
4. Race, Opportunity, And Diversity Of Social Circles In Managerial Networks: Herminia Ibarra