Development of Application Programs Oriented to Agent-Based Data Analysis

Chang Liu

A whitepaper

submitted in partial fulfillment of the

requirements for the degree of

Master of Science in Computer Science & Software Engineering

University of Washington

2020

Project Committee:

Munehiro Fukuda, Committee Chair

Robert Dimpsey, Committee Member

Michael Stiber, Committee Member

Program Authorized to Offer Degree:

Computer Science & Software Engineering

Contents

University of Washington

Abstract

Development of Application Programs Oriented to Agent-Based Data Analysis

Chang Liu

Chair of the Supervisory Committee:

Professor Munehiro Fukuda

Computing and Software Systems

Different software has been developed in the last two decades to accelerate big data parallelism processes, such as MapReduce, Spark, and Storm. Besides these data-streaming tools, agent-based modelling (ABM) provides an alternative approach for data discovery and gain recognition on its programmability and intuition of coding. As a tool of ABM, the Multi-Agent Spatial Simulation library (MASS) is developed by Distributed Systems Laboratory (DSL) at the University of Washington, Bothell and strives to parallelize agent-based models over a cluster system. This project is to extend the initial achievement in agent-based data analysis by exploring more data science applications and identifying more data science applications that would be the better fit to agent-based parallelization.

This project presents the algorithm designs and implementation process for four applications of Top K, Markov Chain, Connected Components, and Matrix Multiplication on MapReduce, Spark, and MASS. In addition, this project compares the programmability and execution performance of four applications on three frameworks. This study further discussed the previous research on the MASS and the strengths and weaknesses of MASS when it is used to implement data applications.

1. Introduction

1.1 Background

Traditional software applications normally could store and process data with the help of a centralized server. However, when it comes to the large dataset, the data analysis is not just about scale and volume, but also involves velocity, variety, and complexity. Different software has been developed in the last two decades to accelerate big data computing, such as MapReduce, Spark, and Storm. These software tools provide their standard program paradigm to facilitate parallelization on cluster computing. The concept behind these tools is data streaming through batch processing. Batch processing is an automated job that "can run without end-user interaction, or can be scheduled to run as resources permit" [1]. These tools perform data transformation and computation on batches of data in parallel to maximize performance (Spark by mini-batches, and Storm process event by event). However, these applications are not the only solutions for large dataset. There are still alternative approaches being developed such as agent-based data discovery. Researchers and engineers have been developing and testing these frameworks for data processes. This project tends to join previous effort by exploring tools and algorithms for agent-based data discovery.

1.2 Motivation

MapReduce and Spark are two major software tools to facilitate big data analysis with a cluster system or even in the cloud. Nevertheless, not all data science applications fit into their computation models. For instance, consider one of the graph problems: identifying triangles in a social network, which shows the friendship degree of the network. The corresponding MapReduce and Spark programs need to separately enumerate all combinations of two connected edges; to narrow them down to those that have another edge to bridge the first and the last vertices, (which makes a triangle); and to remove duplicated triangles. Contrary to their exhaustive and repetitive data transformations, the same program can be much more intuitively and smoothly described by walking an execution entity called “agent” along with a series of three graph edges from each vertex and checking the third edge’s reachability back to the original edge.

Using lightweight reactive agents, this approach allows frequent agent migration and fine-grained data analysis in data discovery. This approach focuses on discovering the attribute of dataset by observing emergent group behavior of many agents, such as walking, dissemination, swarming, and overriding. Based on this approach, Distributed Systems Laboratory (DSL) at the University of Washington, Bothell, developed the Multi-Agent Spatial Simulation library (MASS). The MASS library has been used for simulating transportation, social-network and biology problems. It has also been used for analyzing large datasets [2], and paralleling data-science applications to extend the use case of MASS in the data analysis area.

1.3 Project Goals

In previous research, DSL lab has already parallelized six data-science applications with MASS and compared their programmability and execution performance with MapReduce and Spark.

This project aims to extend these initial achievements in agent-based data analysis by exploring more data science applications and identifying which data science applications would be the best fit to agent-based parallelization. Additionally, this project tends to discover the strengths and weaknesses of MASS when it is used to implement data applications. Specifically, this project tested four applications of Top K, Markov Chain, Connected Components, and Matrix Multiplication in MASS, MapReduce, and Spark and the key criteria are programmability and execution performance.

2. Related Work

This section reviewed two conventional data-streaming approaches of Hadoop and Spark, and further summarized and discussed the previous endeavor with the MASS library.

2.1 Data-streaming Tools in Data Analysis

Hadoop and Spark are two of the most popular software for big data analysis. This section provides the background information on these two software tools, and further interprets why these data-streaming models are so popular, and what problems exist.

2.1.1 Hadoop

Hadoop is a platform for distributed storing and analyzing very large data sets. It provides both infrastructure and programming models: Hadoop distributed file system (HDFS) and MapReduce. The way these modules are woven together is what makes Hadoop so successful.

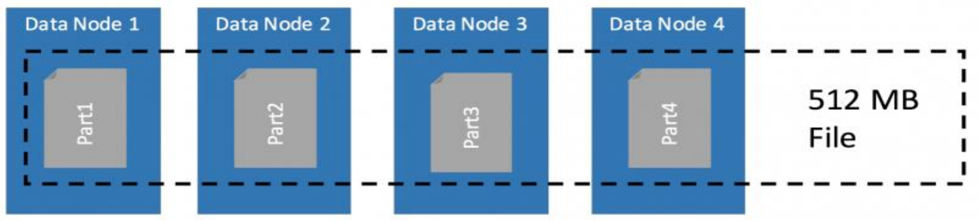


Figure 1. Data Splitting on Hadoop Distributed File System [3]

HDFS has superior fault tolerance and no limit on how big the files can be [4]. As shown in Figure 1, HDFS splits the file into blocks, and then it automatically replicates and distributes these blocks on the Hadoop cluster. Splitting files into blocks makes it easy to analyze the data in a distributed fashion.

MapReduce is a framework for distributed data analysis. There are two important tasks in the MapReduce: Map and Reduce. The Map task takes a set of data and converts it into another set of data, where each element is broken down into tuples (key/value pairs). The Reduce task takes the output of the Map as input and combines these data tuples (key/value pairs) into a smaller set of tuples. The reduce task is always executed after the map job [5].

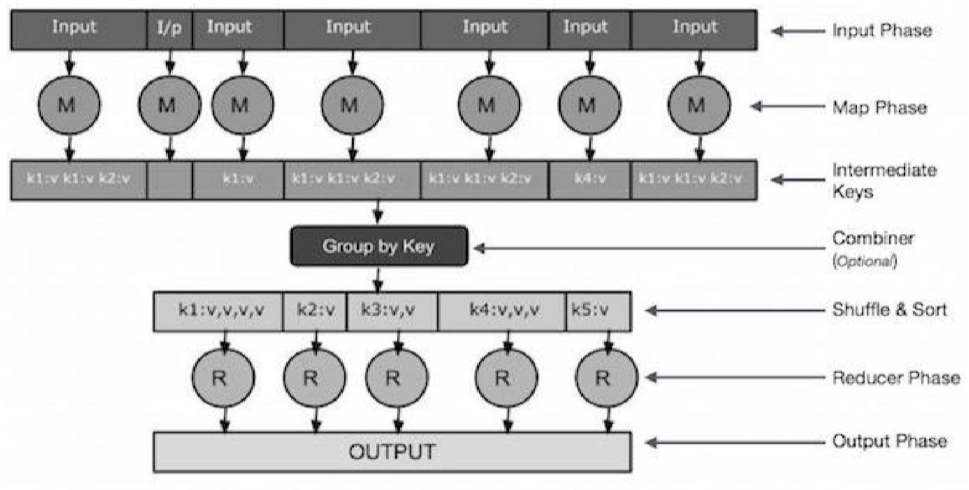


Figure 2. Execution Overview of MapReduce [6]

Figure 2 shows the flows of a MapReduce operation and its several phases. In conjunction with HDFS, MapReduce is able to analyze the blocks of a file in parallel.

The MapReduce framework provides several benefits: Programming model together with infrastructure; The ability to write programs that run on lots of machines; automatic parallelization and distribution; fault tolerance; and program/job scheduling, status checking, and monitoring [7].

2.1.2 Spark

Spark started in 2009 at the University of California, Berkeley and is considered to be the successor to MapReduce for data processing. Spark introduces an abstraction named Resilient Distributed dataset (RDD). An RDD is a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost [8]. Spark and its RDDs were developed in response to limitations in the MapReduce cluster computing paradigm [9]. Because MapReduce was not efficient enough for iterative and interactive jobs, Spark strives to be fast for interactive queries and iterative algorithms and further support in-memory storage and efficient fault recovery [10].

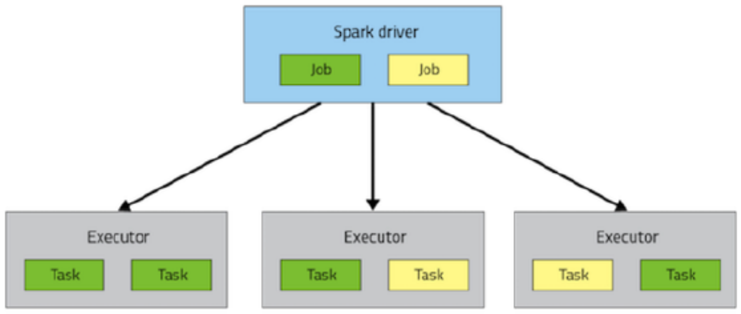


Figure 3. Execution Overview of Spark [11]

As shown in Figure 3, the execution of Spark involves runtime concepts such as driver, executor, task, job, and stage [12]. At runtime, Spark application is mapped to a single *driver* program execution process and a set of distributed processes on the hosts in the cluster. The driver process manages the job stream and schedules the task and is available for the entire time the application is running.

The *executors* are responsible for performing work in the form of tasks and storing any cached data. The execution life of the program depends on whether dynamic allocation is enabled. The executor has a number of slots for running tasks and will run many slots simultaneously during its lifetime. Invoking an action in a Spark application triggers a startup *job* to implement it. Spark checks the data set that the operation depends on and develops an execution plan. The execution plan assembles the data set into phases. A *stage* is a collection of tasks that run the same code, each code on a different subset of data [13].

Spark mainly executes in-memory computations in a distributed environment. It offers incredible processing speed, making it desirable for everyone interested in big data analytics [14].

2.1.3 Limitations

There are several reasons made conventional approaches to data streaming approaches very popular. Firstly, they are endorsed and maintained by Apache—an influential open-source community. The community improves the core software and contributes practical add-on packages [15]. Because of all those side projects, Hadoop and Spark have turned more into an ecosystem for storing and processing big data.

In addition, these tools are particularly suitable for flat texts [16], e.g. statistical analysis of data streamed from social, business, and IoT environments. Services of partitioning, flattening and streaming multidimensional data extend the practicability of analyzing structured datasets. In general, these approaches provide straightforward programming frameworks for parallel computing and interpretive execution environments.

However, these frameworks still have problems with programmability and execution performance of data discovery, especially for structured datasets [17]. For instance, Hadoop and Spark usually decompose the entire data operation into multiple small tasks. Each task must deal with the entire file or stream, complete the previous task and then start the next task. Usually, these data stream models do not maintain the entire data structure in distributed memory on the cluster. However, MASS can maintain the data structure in-memory over the distributed system, which we will discuss later. Therefore, if there is a large data set that repetitive MapReduce invocations and Spark transformations need to process, the processes of transferring back and forth or swapping in and out data between disk and memory could slow down the execution speed.

Moreover, these data stream models each provide their unique program paradigm (e.g. map/reduce functions in MapReduce, lambda expressions in Spark). Following these program paradigms will increase a certain amount of boilerplate code. Meanwhile, from the perspective of programmability, developers need to disassemble the original algorithm and transform it into these models. There is a certain learning curve to overcome. ABM expresses the algorithm more intuitively and its programming is closer to basic java programming and requires less conversion for the users.

2.2 Agent-based Modelling in Data Analysis: MASS

2.2.1 MASS Library

As mentioned in Section 1.2, MASS is a parallel-computing library to develop and execute multi-agent simulations with multiple computer [18]. It provides the efficiency of agent-based parallelization while abstracting parallel environments for ABM. It provides an intermediate layer that only requires the user to provide code related to their project without having to manage the details of the parallelization [19].

Two key concepts within MASS library are: *places* and agents. Each *Place* element is automatically mapped to one of the compute nodes, has a logical array index, and is capable of calling a given function in parallel and exchanging data with other elements. On the other hand, agents are a set of execution instances that can reside somewhere, migrate to any other *place* using a matrix index (thus copying itself), and interact with other agents through their local location [20].

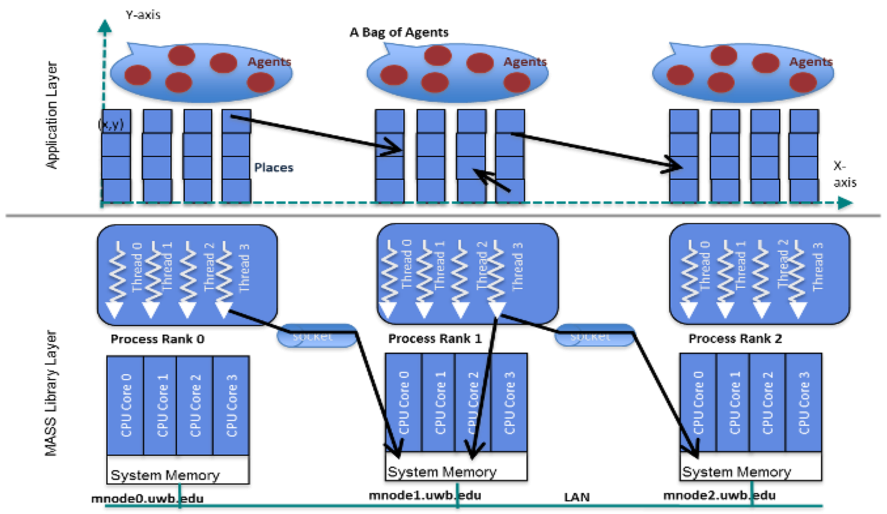


Figure 4. Execution Model of MASS Library [21]

As illustrated in Figure 4, *Places* maps to threads, and agents map to processes. Unless the programmer indicates the places-partitioning algorithm, the MASS library divides the *place* into smaller strips in the vertical or X coordinate direction, and then assigns each stripe to and executes by a different thread. In contrast to the *place*, the agents are grouped into bundles, each of which is assigned to a different process, in which multiple threads continue to check in and check out the agents one by one from the bag while they are ready to execute the new agent [22].

As a tool of agent-based data analysis, MASS has some features and advantages that are exclusive to conventional data-streaming tools. The main difference is the computing units. MASS dispatches agent(s) as computing units to the dataset. There is a parallel loading of a structured dataset and the structure is held in memory. It further facilitates repetitive but various analysis [23].

In addition, the programming in MASS is also intuitive that algorithm developers can code behaviors of agents as if they drive agents out to a dataset [24]. Moreover, as agent-based modelling becomes more popular, it has been applied to computational optimization with many biological-inspired algorithms such as ant colonial optimization. This study will use these features to explore the potential of MASS for some specific big data analysis issues.

2.2.2 Previous Work on MASS

There has been increasing research on the MASS and exploration of its function, and its programmability and performance using different algorithms. As the founder of MASS library, Fukuda and his DSL lab team [25] discusses the advantages and challenges of agent-based data discovery, introduced several agent-based analyzing algorithms, and further strategized the implementation. In the continuing studies [26][27], Fukuda, Gordon and other team members explored algorithms including K Means Clustering, K Nearest Neighbor Classification, Triangle Counting in Graphs, and the Traveling Salesman Problem. It is found that MASS, as a tool of ABM, is quite intuitive with respect to coding programs for distributed data analysis while being efficient. Woodring et. al [28] examined the practicability of MASS via climate change research with web-interfaced climate analysis. In this research, MASS demonstrated practical advantages and performance improvements. Sell [29] explored the enhancement of programmability by designing, implementing, and evaluating these two agents descriptively enhancements of event-driven agent behavioral execution and direct inter-agent broadcast.   With previous studies on

MASS, DSL lab still need to extend more benchmarks to identify the application domains that agents can take advantage of. Therefore, this study borrowed the idea of Gordon's research and extended his research with more algorithms, which might contribute to the future work to create some data science library and graph library. Moreover, Gordon's project only compared the performance of multiple node on the Spark and MASS. Because of no sufficient memory to the research account used in his project, Gordon only study the performance results for a single node on MapReduce. This project tried to fill the gap by implementing four applications on all three frameworks and comparing the performance of both single and multiple nodes.

2.3 Summary

Extending Gordon's research, this study selected more data analysis algorithms to further verify his conclusions. When selecting data analysis algorithms, the rationales are that Top K and Markov Chain are data-streaming tools and Connected Components and Matrix multiplication are more conducive to ABM. The following sections will explain these algorithms in detail, the implementation on the three frameworks, and results and observations from the perspective of performance and programmability.

Based on previous research and conclusions, this project also intends to examine the strengths and weaknesses of MASS when it is used to implement data analysis applications. This study will also discuss and suggest possible improvements of the MASS library.

3. Selection of Applications

Since Hadoop, Spark, and MASS, as clustering programming, often solve the problems with large dataset, we selected four algorithms that tend to solve the common problem of big data. The four algorithms are: Top K, Markov Chain, Connected Components and Matrix Multiplication. This project implemented these algorithms in Hadoop, Spark, and MASS, benchmarked their performances, and made some further discussions.

In addition, based on previous research, Hadoop and Spark usually are suitable for solving the data-streaming-based problems. Therefore, two algorithms are selected, Top K and Markov Chain, that usually involve more data-streaming operations when processing these problems. On the other hand, another set of two algorithms were chosen: Connected Components and Matrix Multiplication, that are suitable for problem solving via modeling into ABM.

This chapter will explain each algorithm and the type of problems it usually solves. The next chapter introduces the implementation details of each algorithm on the three frameworks of Hadoop, Spark, and MASS.

3.1 Data-streaming-based Applications

3.1.1 Top K

The notion of a "Top K" is ubiquitous in the field of information retrieval. A top 10 list is usually associated with the "first-page" of results from a search engine" [30].

A common way to implement the Top K algorithm in Java is to use a priority queue or treemap. The top of the priority queue or the first entry of the treemap is the smallest element among k elements. Moreover, we maintain the largest k elements within the priority queue or treemap. We keep comparing each element to the priority queue or treemap. If the element is larger than the top, pop the current top and add a new element to the priority queue or treemap, otherwise, ignore this element.

3.1.2 Markov-Chain

Markov chain is a machine learning algorithm whose model is described by an undirected graph. A Markov Chain model makes the prediction based on the probability from one state to another state. Let S = {S1, S2, S3, ...} be a set of finite states. We want to collect the following probabilities: P (Sn|Sn–1,Sn–2, ..., S1).

Table 1. City Weather Pattern (Markov Probability Transition Table) [31]

	Tomorrow's Weather			
Today's weather	**sunny**	**rainy**	**cloudy**	**foggy**
sunny	0.6	0.1	0.2	0.1
rainy	0.5	0.2	0.2	0.1
cloudy	0.1	0.7	0.1	0.1
foggy	0.0	0.3	0.4	0.3

The Markov's first-order assumption is the following: P (Sn|Sn–1,Sn–2, ..., S1) ≈ P (Sn|Sn–1). This approximation states the Markov property: the state of the system at time t + 1 depends only on the state of the system at time t. The Markov second-order assumption is the following: P (Sn|Sn–1,Sn–2, ..., S1) ≈ P (Sn|Sn–1,Sn–2). With Markov assumption, we can predict the possibility from one state to another by expressing the joint probability. After we calculate the possibilities from one state to another based on a large amount of data, we can build a Markov probability transition table, which is the machine learning model we use to perform prediction. Table 1 is shown as an example using the city Weather Pattern [32], the sum of each row is 1.00. Typically, a machine learning–based solution consists of two distinct phases: (1) Phase 1: build a model by using historical training data; (2) Phase 2: make a prediction for the next new data using the model built in Phase 1.

This project mainly focuses on Phase 1. This phase involves processing a large amount of data to get the statistics from one state to another and builds a Markov probability transition table. Once we have created this table, getting the possibilities from one state to another is a simple computation, so it is not the scope of this project.

3.1.3 Summary

The first two applications in this study, Top K and Markov Chain, mainly processed plain data. During the processing, the data is distributed to different execution units, and operations on each execution unit are relatively independent. In other words, the data processing on each unit involves little information exchange with each other. This is what this research meant by data-streaming applications and also why we implemented them first in this research. In Section 5, the research will further discuss the execution performance of these two applications on MapReduce, Spark, and MASS.

3.2 Analysis of Structured Data

3.2.1 Connected Components

The third application of this study parallelized is connected components (CC) in an undirected graph. A Connected Components represents "a maximal set of vertices such that there is a connection between every pair of vertices. The components are separate 'pieces' of the graph such that there is no connection between the pieces." [33]. As shown in Figure 5, Connected Components means partitioning a graph into chunks in such a way that there are paths to connect all pairs of vertices within the chunk.

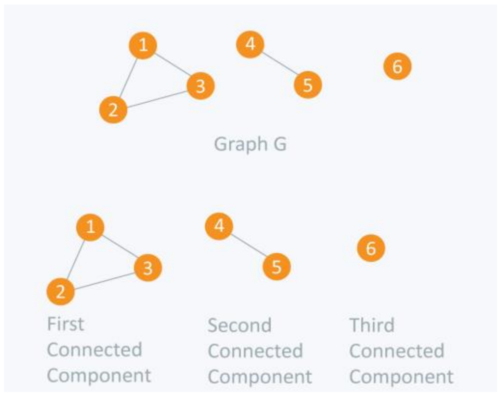

1
2
3
4
5
6
Graph G
First Connected Component
Second Connected Component
Third Connected Component

Figure 5. The Connected Components of a Graph [34]

We could use different graph search algorithms, such as breadth-first search and depth-first search, to solve the Connected Components problem. Taking breadth-first search as an example, we could start from the very first vertex, and then everything we find during this search must be part of the same connected component. We further repeat the search to identify the next component from any undiscovered vertex (if one exists), and so on until all vertices are found [35].

3.2.2 Matrix Multiplication

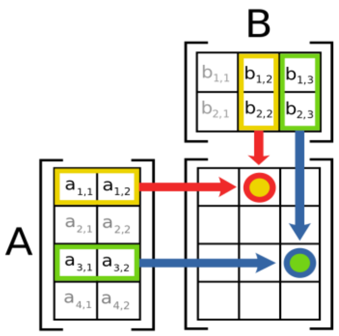


Figure 6. Matrix Multiplication [36]

Matrix Multiplication, in mathematics, is a binary operation that produces a matrix from two matrices. For matrix multiplication, the number of columns in the first matrix must be equal to the number of rows in the second matrix (See Figure 6).

A simple algorithm can be constructed which loops over the indices *i* from 1 through *n* and *j* from 1 through *p*, computing the above using a nested loop: [37]

Input: matrices *A* and *B*

Let *C* be a new matrix of the appropriate size

```
for i from 1 to n:
    for j from 1 to p:
        Let sum = 0
        for k from 1 to m:
            set sum ← sum + Aik × Bkj
            set Cij ← sum
return C
```

3.2.3 Summary

The two algorithms, Connected Components and Matrix Multiplication are selected in this research as two typical algorithms of processing structured dataset. As mentioned in Section 3.1, the first two algorithms (Top K and Markov Chain) processed plain data, and the computation of each unit is relatively independent. When designing and implementing the MASS-paralleled version of Top K and Markov chain, we found there is no need to use agents since using *Places* alone could achieve MASS-version parallelization. Therefore, it is not necessary to use agents to carry parts of the information and then walk, disseminate, or swarm on data structure to observe emergent group behavior.

In order to be able to benefit from “data discovery”, Section 3.2 further chose two algorithms that involve structured data (graph for Connected Components and 2D-array for Matrix Multiplication). In this way, when using MASS to parallelize these two algorithms, the projects need to maintain structured dataset in memory, and then use the agents to achieve “data discovery”. In Section 5, we will further discuss their performance and the observation.

4. Application Development with MapReduce, Spark, and MASS

This section covers the implementations of Tok K, Markov Chain, Connected Component in Graphs, and Matrix Multiplication using MapReduce, Spark, and MASS. Section 5 will discuss the quantitative and qualitative analysis of four algorithms. This section built on Parsian’s work on data algorithms including Top K and Markov Chain for Spark and MapReduce. In addition, this project developed on and modified Nihar Suryawanshi’s MapReduce implementation of Connected Components project [38] and Archana Masilamani’s MapReduce implementations of Matrix Multiplication [39].

4.1 Top K: Implementation and Issues

4.1.1 Top K Hadoop Implementation

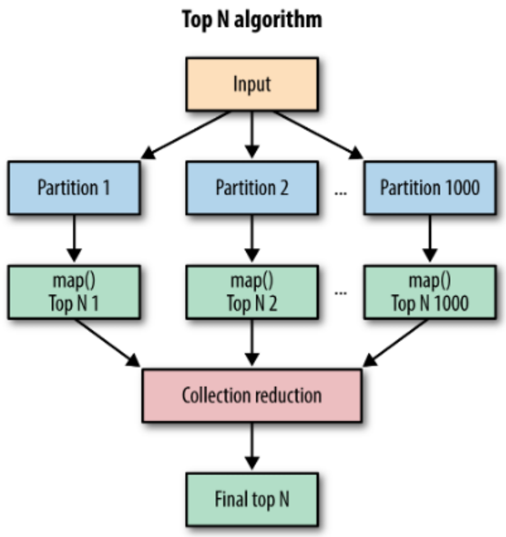


Figure 7. Top K Hadoop Implementation [40]

The MapReduce solution for Top K algorithm is presented in Figure 7. The HDFS partitioned the input into smaller chunks, and each chunk is sent to a mapper, which creates a local top k and then emits the local top k to the reducer. In emitting the mappers' output, we use a single reducer key so that all the mappers' output will be consumed by a single reducer. Finally, the single reducer will find the final top k from all the local top k passed from the mappers.

In general, in most of the MapReduce algorithms, using a single reducer is problematic and will cause a performance bottleneck (because one reducer in one server receives all the data, and all the other cluster nodes do nothing, so all of the pressure and load is on that single node).

However, a single reducer in this project works appropriately, because the size of each Top k passed from the mapper is k. Assuming we have 1,000 mappers, the reducer will get 1,000 * k records. It is not enough data to cause a performance bottleneck [41].

4.1.2 Top K Spark Implementation

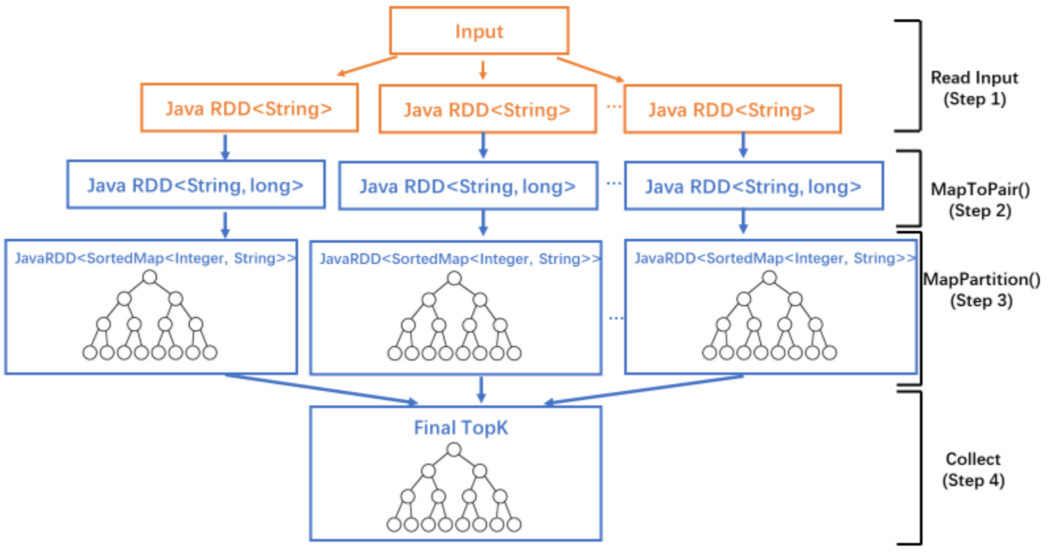


Figure 8. Top K Spark Implementation

The parallel implementation of Top K Spark follows the same logic that of MapReduce - finding the local top k in each partition and then get the final top k from all the local top k. The main difference is that Spark provides a rich set of functional programming APIs. There are four main steps (shown in Figure 8) of Top k Spark implementations:

Step 1: Read the input file from the local filesystem and create an RDD (*JavaRDD<String>*).

Step 2: Use *mapToPair()* to create a new RDD (*JavaPairRDD<String, Long>*) from an existing RDD (*JavaRDD<String>*). The output of this step is *Tuple2<K, V>*, in which Key corresponds to ID and Value corresponds to frequency.

Step 3: Create a local top k for each input partition. Use *mapPartitions()* to create a new RDD (*JavaRDD<SortedMap<Integer, String>>*) from an existing RDD (*JavaPairRDD<String, Integer>*). Create and initialize a *TreeMap<Integer, String>*, and maintain a local top k using this treeMap from each partition. Finally, return the result of a local top 10 list.

Step 4: Create the final top k using *collect()*. Use the *collect()* method to get all local top k. Iterates over all local top k created per partition and creates a single final top k.

4.1.3 Top K MASS Implementation

The solution of MASS is inspired by that of MapReduce as well. This project implemented a *TopK* class that extends *Place* class. The MASS program will create number of clusters * number of cores *places*. In an ideal situation, each *place* runs on a core with no *places* waiting, and we could use the computation resources on the cluster most efficiently. Each *place* handles two functions: *fileread()*, and *getTopK()*.

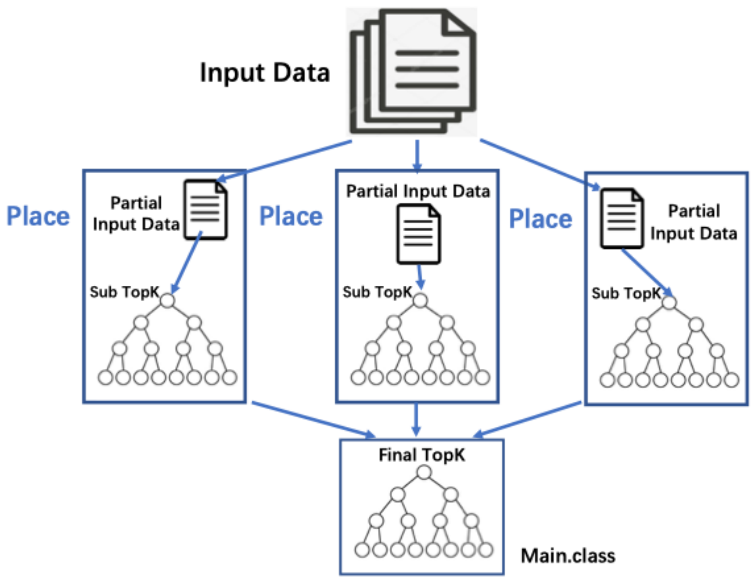


Figure 9. Top K MASS Implementation

- As shown in Figure 9, *fileread()* (line 5) uses MASS parallel IO to read equal portions of data into memory.
- Then *getTopK()* (line 6) reads its portion of data, maintains a local treeMap of size k to find a local sub-top k, Then, returns the local top k to the main class.
- Finally, the main method collects local top k (line 7) from all *places* and gets the final top k.

```
1  main(){
2          int numberOfPlace = MASSBase.getSystemSize() * NUMBER_OF_CORE;
3          Places places = new Places(1, TopK.class.getName(),
4                                          (Object) topKValue, numberOfPlace);
5          places.callAll(TopK.fileread_, (Object) filenames);
6          Object[] subTopKs = places.callAll(TopK.topK_, null);
7          Integer[][] finalTopK = finalTopK(subTopKs, kValue);
8  }
```

The MASS implementation does not require discovering specific data attributes when finding the top k. Therefore, it does not use agent, but only the *Place* to parallelly data processing.

4.2 Markov: Implementation and Issues

4.2.1 Markov Chain Hadoop Implementation

The Markov Chain algorithm involve many steps of data transformation. As shown in Figure 10, The MapReduce solution for Markov chain involves two MapReduce jobs.

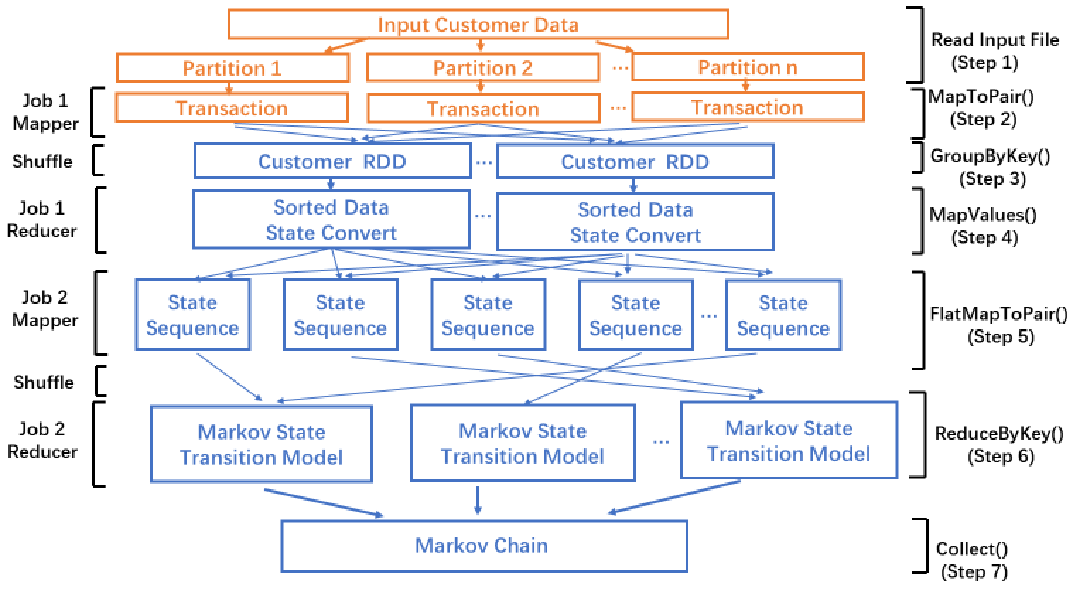


Figure 10. Markov Chain MapReduce and Spark Implementation

For Job 1, the goal is to generate Time-Ordered Markov state for each customer.

- Mapper: Accept the customer transaction data and map each input record *to <customerID, (Date1, Amount1)>* pairs (line 3 - 6).

```
1   public void map(LongWritable key, Text value, Context context){
2       String[] tokens = StringUtils.split(value.toString(), ",");
3       reducerKey.set(tokens[0]); //customer-id
4           //(purchase-date, amount)
5       reducerValue.set(tokens[2], tokens[3]);
6       context.write(reducerKey, reducerValue);
7   }
```

- Reducer: Reduce all transactions (Date, Amount) from the same customerID to a list and sort the transactions by purchase date in ascending order (lines 2 - 5); Then, convert transactions into a two-letter symbol that stands for a Markov chain state (lines 7 - 16, Table 3). The final output generated from this reducer is as follows: *(<customerID>, <State_1>, <State_2>, ..., <State_n>)*

```
1   public void reduce(Text key, Iterable<PairOfLongInt> values,              Context context){
2      List<PairOfLongInt> list = new ArrayList<PairOfLongInt>();
3      for (PairOfLongInt pair : values) { list.add(pair.clone());}
4      // sort by purchase-date: (Date1, Amount), (Date2, Amount)...
5      Collections.sort(list, (o1, o2) -> { o1.compareTo(o2);});
6
7      for (int i = 1; i < list.size(); i++) {
8         long daysDiff = curDate - prevDate;
9         String dateState = (daysDiff < 30) ? "S" :
10                                     (daysDiff < 60)? "M" : "L";
11          String amountState = (prevAmount < 0.9 * curAmount)? "L" :
12                                     (prevAmount < 1.1 * curAmount)? "E": "G";
13          output.append("," + dateState + amountState);
14       }
15       // output: <State1><,><State2><,>...<,><StateN>
16       context.write(null, new Text(output.toString()));
17   }
```

For job 2, the goal is to combine all customer's Markov state to get state sequence and count occurrence of each state sequence to build a Markov State Transition Model. (line 3 - 6)

```
1 protected void reduce(PairOfStrings key, Iterable<IntWritable>
2                                     values, Context context) {
3      int finalCount = 0;
4      for (IntWritable value : values) { finalCount += value.get();}
5      context.write(new Text(fromState + "," + toState),
6                              new IntWritable(finalCount));
7 }
```

4.2.2 Markov Chain Spark Implementation

Follows the same logic of the MapReduce implementations, the Spark solution is implemented in seven basic steps as shown in Figure 10:

Step1: Convert the input into a *JavaRDD<String>*, in which each line is an input record (as defined in the preceding section) and is formatted as:

<customerID   transactionD purchaseDate   amount>

Step 2: Convert *JavaRDD<String>* into *JavaPairRDD<K,V>* and the return format is as *<customerID, (purchaseDate, Amount) >.*

Step 3: Group transactions by *customerID*. Applying *groupByKey()* to the output of step 2, the result is a *JavaPairRDD<K2,V2>*, in which K2 is customerID and V2: is *Iterable<Tuple2<purchaseDate, Amount>*

Step 4: Generate each customer's list of Markov state. Sort each customer's translation (Date, Amount) by purchase date in ascending order. Then convert (Date, Amount) into a Markov chain state. The final output is each customer's list of Markov state.

Step 5: Create a Markov state sequence. Using *PairFlatMap()* function, to map each customers' state into state sequence of State_1, State_2, ..., State_n, as follow format: <(State_1, State_2) , 1>, <(State_2, State_3), 1>..., <(State_n-1, State_n), 1>

Step 6: Calculate the frequency of each state sequence. Use *recudeByKey()* function to get the occurrence of each state sequence. *<(State_i, State_j) , 12>, <(State_m, State_n), 2345>..., <(State_n, State_k), 2>*

Step 7: Emit the final output. Use *map()* function to convert *JavaPairRDD* into *JavaRDD* and return the final output.

4.2.3 Markov Chain MASS Implementation

For Markov MASS implementation, this project created a class *Markov* that implements *Place* class. *Place* implements two methods: *fileread()*, and *getState()*.

fileread(): Because we need to group the same customer's transaction together to generate Markov state, I did not use MASS parallel IO to make each *place* to process portions of data. If we do so, the transaction data of each customer would be fragmented to *places*. When merging the transactions from the same customer, it would involve lots of data shuffling.

Therefore, rather than using MASS parallel IO, I let each *place* read the entire file. While reading, I used the hash value of the customerID mod total number of *places* to get a hash number (line 4-8). In this way, only the transactions whose customer hash number has the same value as the current *place* index are placed to this *place*. When the hash number is different from the current *place* index value, ignore this transaction record.

```
 1  while (scanner.hasNextLine()) {
 2      String[] tokens = scanner.nextLine().trim().split(",");
 3      String customerId = tokens[0].trim();
 4      if(customerId.hashCode() % nNodes == nodeId){
 5          if(transactions.get(customerId) == null){
 6              transactions.put(customerId, new TreeMap<Long, Long>());
 7          }
 8          transactions.get(customerId).put(date, amount);
 9      }
10  }
```

I initially expected this design would allow each *place* to process specific customer data and reduce data shuffling. However, the performance is different from what I expected. We will discuss this in detail in Section 5.

getState() is equivalent to integrating the steps 4, 5, 6, and 7 of Spark implementation into the same method. First, sort its portion of customer's transactions (Date, Amount) by purchase date in ascending order and convert (Date, Amount) into Markov state; Then, create a Markov state sequence and return it to the driver; Finally, the primary method collects all partial Markov state sequences in each *place* and gets the final Markov state sequence.

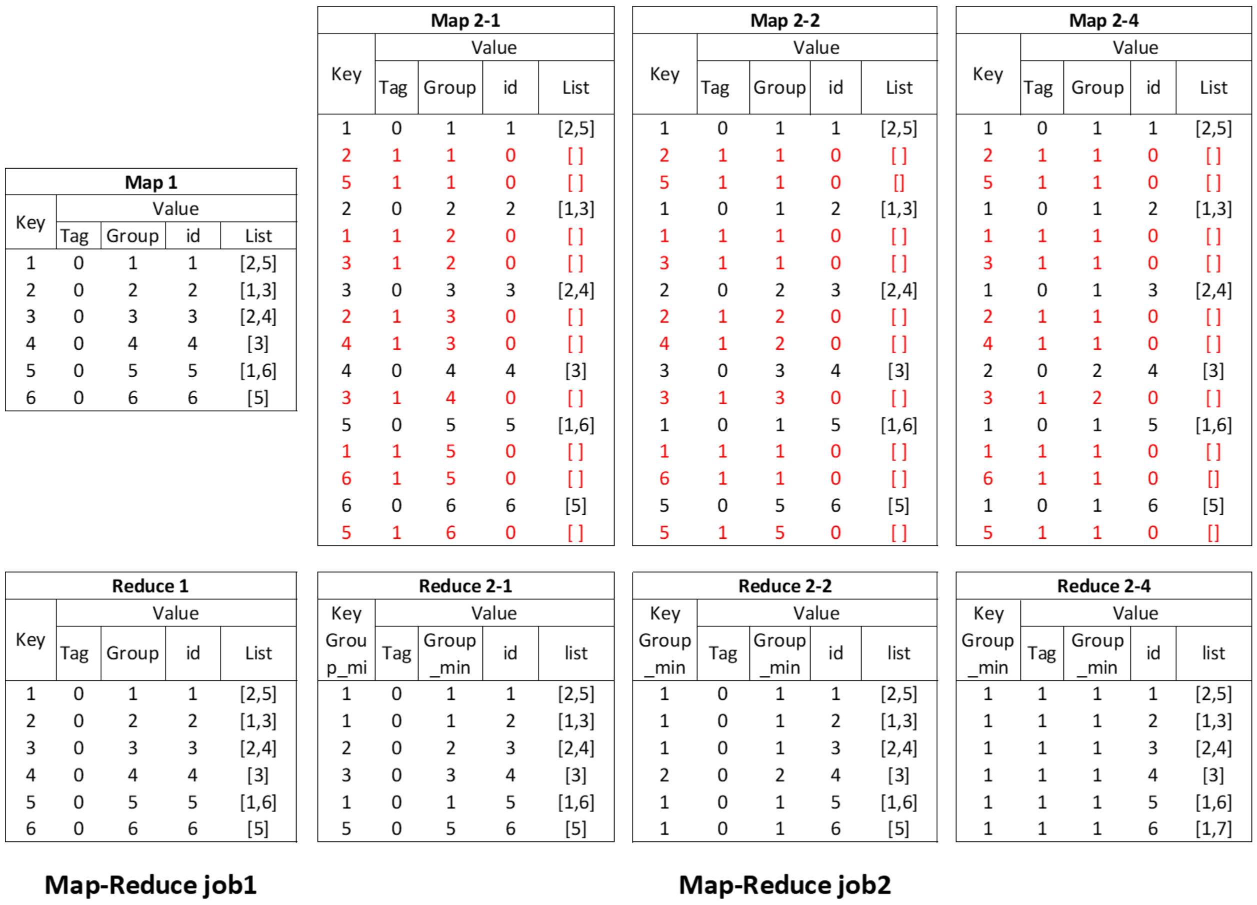
4.3 Connected Components Implementation and Issues

4.3.1 Connected Components Hadoop Implementation

The implementation of Connected Components on Hadoop builds on the MapReduce design of Connected Components in Friso van Vollenhoven's blog [42]. It Applies two map-reduce jobs to solve the problem:

For the first Map-Reduce job, it transforms the edge lists to a representation of the adjacency list. For the second Map-Reduce job, the Mapper maps the adjacency list of the source node and label them with partition id numbers; the Reducer finds the smallest partition ID for each node ID it belongs to, and set the partition ID of each record to the smallest partition ID. The program keeps repeating the second Map-Reduce job until nothing changes, and it mean all connected node have been grouped. Figure 11 indicates the details of how each node spreads the lower component ID to one more level of neighbors at each Map-Reduce step.

The code of implementation on Hadoop builds on and modifies Nihar Suryawanshi's project on GitHub [43].


Map-Reduce job1
Map-Reduce job2

Map 1

Key	Value			
	Tag	Group	id	List
1	0	1	1	[2,5]
2	0	2	2	[1,3]
3	0	3	3	[2,4]
4	0	4	4	[3]
5	0	5	5	[1,6]
6	0	6	6	[5]

Map 2-1

Key	Value			
	Tag	Group	id	List
1	0	1	1	[2,5]
2	1	1	0	[ ]
5	1	1	0	[ ]
2	0	2	2	[1,3]
1	1	2	0	[ ]
3	1	2	0	[ ]
3	0	3	3	[2,4]
2	1	3	0	[ ]
4	1	3	0	[ ]
4	0	4	4	[3]
3	1	4	0	[ ]
5	0	5	5	[1,6]
1	1	5	0	[ ]
6	1	5	0	[ ]
6	0	6	6	[5]
5	1	6	0	[ ]

Map 2-2

Key	Value			
	Tag	Group	id	List
1	0	1	1	[2,5]
2	1	1	0	[ ]
5	1	1	0	[]
1	0	1	2	[1,3]
1	1	1	0	[ ]
3	1	1	0	[ ]
2	0	2	3	[2,4]
2	1	2	0	[ ]
4	1	2	0	[ ]
3	0	3	4	[3]
3	1	3	0	[ ]
1	0	1	5	[1,6]
1	1	1	0	[ ]
6	1	1	0	[ ]
5	0	5	6	[5]
5	1	5	0	[ ]

Map 2-4

Key	Value			
	Tag	Group	id	List
1	0	1	1	[2,5]
2	1	1	0	[ ]
5	1	1	0	[ ]
1	0	1	2	[1,3]
1	1	1	0	[ ]
3	1	1	0	[ ]
1	0	1	3	[2,4]
2	1	1	0	[ ]
4	1	1	0	[ ]
2	0	2	4	[3]
3	1	2	0	[ ]
1	0	1	5	[1,6]
1	1	1	0	[ ]
6	1	1	0	[]
1	0	1	6	[5]
5	1	1	0	[]

Reduce 1

Key	Value			
	Tag	Group	id	List
1	0	1	1	[2,5]
2	0	2	2	[1,3]
3	0	3	3	[2,4]
4	0	4	4	[3]
5	0	5	5	[1,6]
6	0	6	6	[5]

Reduce 2-1

Key	Value			
Grou p_mi	Tag	Group _min	id	list
1	0	1	1	[2,5]
1	0	1	2	[1,3]
2	0	2	3	[2,4]
3	0	3	4	[3]
1	0	1	5	[1,6]
5	0	5	6	[5]

Reduce 2-2

Key	Value			
Group _min	Tag	Group _min	id	list
1	0	1	1	[2,5]
1	0	1	2	[1,3]
1	0	1	3	[2,4]
2	0	2	4	[3]
1	0	1	5	[1,6]
1	0	1	6	[5]

Reduce 2-4

Key	Value			
Group _min	Tag	Group _min	id	list
1	1	1	1	[2,5]
1	1	1	2	[1,3]
1	1	1	3	[2,4]
1	1	1	4	[3]
1	1	1	5	[1,6]
1	1	1	6	[1,7]

Figure 11. MapReduce of Connected Components Spreads the Component ID to Neighbors

4.3.2 Connected Components Spark Implementation

The implementation of Spark Connected Component follows the same logic of implementation on MapReduce. Like Markov Chain (in Section 4.2.2) and Top K (in Section 4.1.1) on Spark, Spark implementation uses its API and involves more data transformation steps. The data transformation steps are as shown in Figure 11. Since the implementation of Connected Components are similar to that of MapReduce, further interpretation is waived here.

4.3.3 Connected Components MASS Implementation

For the MASS version of Connected Components, this project develops a *Node* class that implements *Place* class and it represents a graph node in the graph. Each node maintains the current node index and component ID. The component ID indicates which component the current node belongs to, and it is initialized with the node index ID.

In addition, this project created a *Spreader* class that implements *Agent* class. Initially, one agent is populated over each *place*. This agent takes two information: the node index (component ID) it was populated and a list of node indexes that this agent has visited. Hereafter, let agents roll down to the neighbor nodes with higher index.

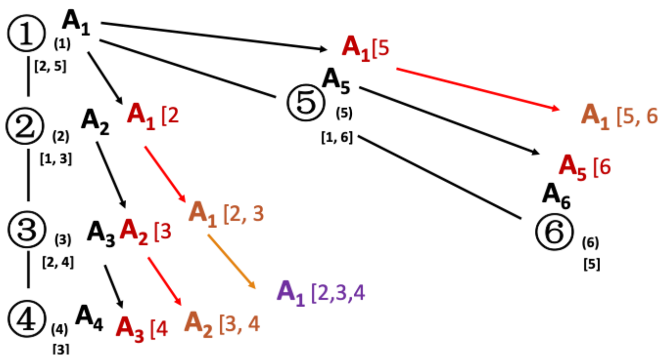


Figure 12. Connected Components MASS Implementation

Figure 12 gives a view of how agents update the components information. The agents spread the current lowest component ID to all neighbors who have a higher index than the current graph node via *onArrival()* and *onDeparture()*.

- *onArrival()*: If this node has a higher component ID than the one current agent carrying, updates the component ID of this node. Then, give the index of the next neighbor node to the current agent to let it spread. However, if the neighbor node has a lower component ID, terminate the current agents.

- *onDeparture()*: If the next neighbor has not been visited, migrate the agent toward the next neighbor.

4.4 Matrix Multiplication: Implementation and Issues

4.4.1 Matrix Multiplication Hadoop Implementation

For Matrix Multiplication, there are two popular MapReduce designs: CPMM and RMM. The RMM implements a replication-based strategy in one single MapReduce job, while CMPP implements a cross product strategy requiring two jobs. In a study [44], Ghoting et. al. analyzed the performance differences between RMM and CPMM, and found CPMM is better because CPMM has higher degrees of parallelism, performs stably, and outperforms RMM. This is the reason why we choose CPMM as our implementation plan.

The implementation of Matrix Multiplication on Hadoop builds on Archana Masilamani's implementation [45]. This project further reviewed and implemented the code based on her work.

This algorithm requires two rounds of MapReduce jobs (See Figure 13). From the first job, input matrix sub-blocks $A_{i,k}$ and $B_{k,j}$ are aggregated together according to the same key k, and then the reducer phase obtains the intermediate result matrix $P^{k}_{i,j}= A_{i,k}\ B_{k,j}$. Then, the second job accumulate the previous $P^{k}_{i,j}$ and obtain the final result submatrix $C_{i,j}=kP^{k}_{i,j}$.

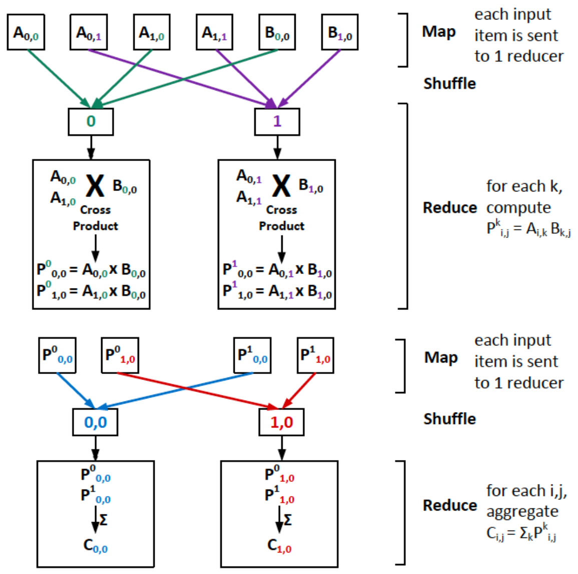

Map
each input item is sent to 1 reducer
Shuffle
Reduce
for each k, compute $P^k_{i,j} = A_{i,k} B_{k,j}$
Cross Product
Map
each input item is sent to 1 reducer
Shuffle
Reduce
for each i,j, aggregate $C_{i,j} = \Sigma_k P^k_{i,j}$

Figure 13. Matrix Multiplication Hadoop Implementation [46]

4.4.2 Matrix Multiplication Spark Implementation

Similar to MapReduce implementation, there are two shuffle phases for the Spark implementation. As shown in Figure 14, Spark also implemented CPMM. Since the algorithm design follows the same logic of MapReduce, further interpretation is saved here on the process of Spark implementation.

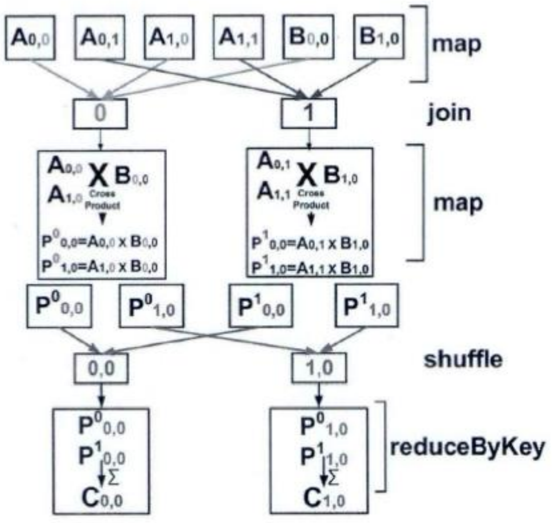


Figure 14. Matrix Multiplication Spark Implementation [47]

4.4.3 Matrix Multiplication MASS Implementation

The MASS project is implemented using Cannon's algorithm. Cannon's algorithm is a distributed solution for matrix multiplication. It is especially suitable for computers laid out in an $n * n$ mesh [48]. Considering distributed two $n * n$ matrices A and B into p blocks. Therefore, each process gets a block of matrices A and B which is size $(n/\sqrt{p} \times (n/\sqrt{p})$. Process P(i , j) stores matrix A(i , j) and matrix B(i , j) and computes the block of the result matrix C(i , j).

The algorithm can be majorly described in two steps: matrices alignment and matrices shifting. Figure 15 indicates how matrices A and B are initialized and aligned, how data is shifted, and how the data unit is performing the multiplication.

- Step 1 matrices alignment: The initial step of the algorithm regards the alignment of the matrices. As shown in Figure 15, 15.a and 15.b show the initial metrices A and B that distributed on a 4 * 4 clusters. Before performing the matrix multiplication, we first want to shift all submatrices A(i , j) to the left (with wraparound) by i steps and all submatrices B(i , j) up (with wraparound) by j steps. Figure 15.c and 15.d shows a view of matrices distribution after alignment.
- Step 2 matrices shifting: At each iteration step, each process would perform a multiplication at local block and accumulate the current block of multiplication result to the local matrix C. When local multiplication is done, each block of A moves one step left and each block of B moves one step up (again with wraparound). The matrices shifting is shown in Figure 15.e and 15.f. Thereafter, each process could perform next block multiplication, add to partial result, repeat until all blocks have been multiplied.

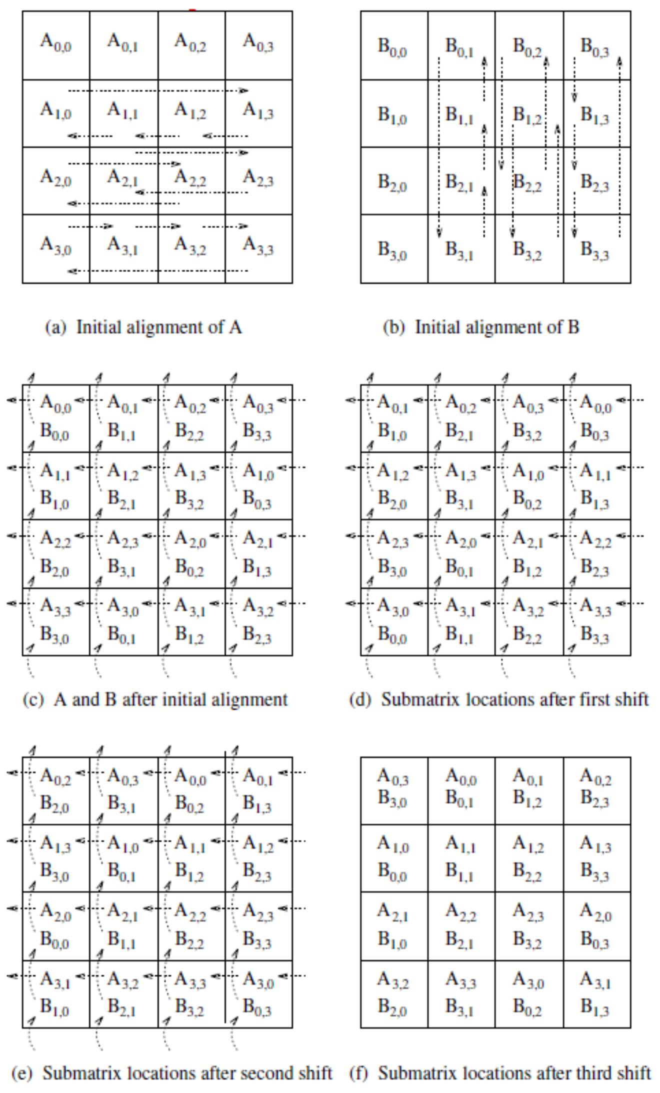

(a) Initial alignment of A
(b) Initial alignment of B
(c) A and B after initial alignment
(d) Submatrix locations after first shift
(e) Submatrix locations after second shift
(f) Submatrix locations after third shift

Figure 15. Cannon's Matrix Multiplication algorithm [49]

The Matrix Multiplication for MASS simulates the implementation of Cannon's Algorithm. I created a *SubMatrix* class that implements *Place* class. As mentioned previously, for two matrices A and B of size n * n, initialize p *SubMatrix* to parallelize the matrix multiplication, each *SubMatrix* holds a block of matrices A and b of size $(n/\sqrt{p})\times(n/\sqrt{p})$. Each *SubMatrix* also holds a matrix C of size $(n/\sqrt{p})\times(n/\sqrt{p})$ that keep accumulating the partial matrix multiplication results.

I also created a *Carrier* class that implements the *Agent* class. Two agents are initialized at each *place*. The agent with even index carries matrix A, and agent with odd matrix carries matrix B. The two agents carry each matrix to perform matrix alignment and shifting according to the Cannon' algorithm.

5. Comparison of MapReduce, Spark, and MASS

To answer the research question of how the application fits with agent-based parallelization, this project applies two metrics of programmability and performance.

5.1 Comparison of Programmability

To analyze the difficulties of programming, this section will compare three frameworks based on programming intuitivity (including algorithmic modelling, data representation, number of methods), and boilerplate code ratio.

5.1.1 Intuitive Programming

As mentioned in Section 2, due to the limitation of the programming paradigm, implementing the parallel version of the algorithm by MapReduce and Spark is quite different from that of basic java programming for sequential version.

Taking MapReduce as an example, all operations of MapReduce must be converted into two operations: Map and Reduce, which is not easy to adapt to all situations. To fit with Map-Reduce program framework, the programmer often could not use familiar programming patterns and data structures (such as Map and Queue) provided by Java library. Therefore, MapReduce is generally considered difficult to program. Meanwhile, it is a similar issue on Spark.

In comparison, when the algorithm is suitable for the agent-based model, MASS can describe the problem more intuitively. MASS programming needs to implement *Place* and *Agent* as paradigm, but their methods are all user-defined. Compared to MapReduce and Spark, the learning curve for MASS is lower.

Particularly when the algorithm needs to deal with complex data structures, parallel programming on MASS framework can simulate the problem-solving process using agents. It allows frequent agent migration and fine-grained data discovery. The intuitive programming of MASS will be further analyzed and evidenced by algorithmic modelling, data representation, and the number of methods.

5.1.1.1 Algorithmic Modelling

Taking the implementation of Connection Component on MASS as an example, the program of MASS could be intuitively described as: Each agent carries the component ID and walks around the graph to group components. When the agent arrives the current node whose component ID is smaller than the one the agent carries, this agent stops spreading and terminates; Otherwise, it updates the current node’s component ID with the smaller one brought by this agent, and spawn children agents to all neighbor nodes with greater index than the current node to expand the current connected component. From this example， we find that MASS parallelizes the algorithm by using multiple agents to simulate the problem-solving process.

On the contrary, the implementations of MapReduce and Spark programs are abstract. As described in 4.3.1, and 4.3.2， the MapReduce and Spark solutions are implemented according to Frigo Vanllenhoven’s Blog [50]. Taking MapReduce as an example. Generally, every MapReduce step serves the purpose of spreading the lower component ID to one more level of neighbors. But each step of map and reduce data transformation is not a straightforward directive to group more nodes to the current component. The author uses a long blog post to explain this MapReduce design and every data transformation step. That is saying that MapReduce is not as direct as MASS to describe the problem.

Taking Matrix Multiplication as a further example. As described in 4.4.3, the MASS implementation simulates the algorithm of Cannon's algorithm very well. Two agents are generated at each *place*, and an agent with even index carries matrix A, and agent with odd matrix carries matrix B. The two agents carry each matrix to perform matrix shifting (A matrix moves one step left and B matrix moves one step up), Then at each iteration each matrix block accumulates partial results, and repeats until all blocks of the matrix have been multiplied.

Now we compare the implementation of MASS with MapReduce and Spark, both of them use the CPMM algorithm [51]. Because Spark implementation is derivative from MapReduce, here we still take MapReduce as an example. If we look back to Section 4.4.1, as indicated by Figure 13, in the first Map-Reduce step, it aggregates $A_{i,k}$ and $B_{k,j}$ to get $P^{k}_{i,j}= A_{i,k} B_{k,j}$ according to the same K. In the second Map-Reduce step, it accumulates $P^{k}_{i,j}$ to obtain the final result submatrix $C_{i,j}=kP^{k}_{i,j}$. The design of the Map-Reduce algorithm is more abstract than MASS. What to select as the key in each MapReduce step is difficult to think of. Therefore, if engineers are not very skilled in MapReduce and Spark, the parallel algorithm design is hard.

In summary, observing the programmatic modeling MASS parallelizes the algorithm by using multiple agents to simulate the problem-solving process. This makes the MASS program easier to design and easier to understand. Meanwhile, we have to acknowledge that observations above are based on algorithms involving structured data. If these algorithms are processing plain data, or the algorithms are quite suitable for MapReduce framework such as Top K, the difficulty of implementation on three frameworks are quite similar.

5.1.1.2 Data Representation

Another evidence of intuitive programming is data representation, which refers to how the paradigm internally represents the input data used in each algorithm. As indicated in Table 2, this study found that MASS is able to represent both plain and structured data. This result corresponded to Gordon's finding on the flexibility of MASS [52]. The flexibility of MASS allows better representing input data and therefore has more advantages in algorithm design.

Table 2. Data Representation

Paradigm	Top K	Markov Chain	Connected Components	Matrix Multiplication
MASS	Plain	Plain	structured	structured
Spark	Plain	Plain	Plain	Plain
MapReduce	Plain	Plain	Plain	Plain

5.1.1.3 Number of Methods

Besides the difficulties of algorithmic modelling and data representation, a further evidence for intuitive programming is the number of methods used for programming. The number of methods represents the number of modules used to get the final data transformation. It refers to the complexity and easiness of each step of data transformation. We can see from Table 3 that MASS can generally solve the problem with fewer methods. The other two frameworks use more methods because they are limited by their program paradigm, whose data transformation must be operated using the inherent API.

Table 3. Number of Methods

Parallel Framework	Top K	Markov Chain	Connected Components	Matrix Multiplication
MapReduce	5	10	6	6
Spark	4	12	7	8
MASS	5	8	5	5

Taking Markov Chain as an example, Markov chain processes plain data, but Spark uses 12 methods, MapReduce uses 10 methods, and MASS uses only 8 methods. This is because this algorithm involves many steps of data transformation. For Spark, transforming the data from one-to-many, many-to-many and many-to-one must use certain fixed APIs. On the other hand, since MASS is similar to basic java programming, users can use the common data structure to combine several relative data transformation operations in one method. With proficiency in Spark program paradigm, programming can become smooth, but if engineers are not very proficient and the data transformation of the algorithm is complex, modelling problems to Spark can be more difficult than to MASS.

In summary, the Section 5.1.1 interprets the intuitivity of MASS programming from algorithmic modelling, data representation, and number of methods. Comparing with the other two frameworks, MASS parallel framework is not significantly different from common java programming. When converting complex algorithms (such as graph problems) and closely related data structures (such as Matrix Multiplication), MASS is easier to design and understand than

Hadoop and Spark. For general engineers, once they understand the design concept of MASS, programming is more intuitive in MASS.

5.1.2 Boilerplate Code Ratio

In a previous project on MASS, Gordon found that both MASS and Spark use less boilerplate code than MapReduce [53]. In his project, Gordon explained that MapReduce needs extra boilerplate code to know what files to set up and where to find them. It requires understanding of the underlying HDFS. By comparison, MASS and Spark save this complexity.

Table 4. Boilerplate Code

Parallel Framework	Top K	Markov Chain	Connected Components	Matrix Multiplication	Average
MapReduce	10.7%	8.5%	13.9%	16%	12.6%
Spark	4.5%	2.4%	6.2%	6.2%	4.5%
MASS	5.2%	6.6%	9.6%	12.6%	8.6%

This study adopts the same metrics of boilerplate ratio. Table 7 shows the boilerplate ratios for each algorithm as well as the average ratio among all frameworks. As indicated from Table 4, this study generally proves the advantages of MASS and Spark，MapReduce requires the most boilerplate code than the other two frameworks. This corresponds to Gordon’s finding on boilerplate ratio yet the difference is that this study applies a definition of boilerplate code that is slightly different from Gordon’s.

In Gordon’s definition, boilerplate refers to all configuration lines in MapReduce and Spark are counted, and all lines that use MASS.* are counted in MASS. The lines that parse command line arguments in all three frameworks are also counted. Meanwhile, the *place* and constructor lines are left out.

However, this study defined boilerplate as "sections of code that have to be included in many *places* with minor alteration and accomplish only minor functionality" [54]. Therefore, in my statistics, codes that serve the requirement of the framework are also included. Besides configuration lines and parse command line, I also include three types of lines: (1) *the repeated header* in MapReduce, which must be included each time when using Map class and Reduce class; (2) *switch method* in MASS that invokes internal methods of *place* and agent through *callAll()*; (3) *constructor lines* in MASS, since these are also part of the MASS framework that have to be included.

In summary, Section 5.1 compares the programmability on MapReduce, Spark, and MASS. It is found that MASS has more intuitive programming, and it also has less boilerplate ratio, and less difficulties in configuration compared to MapReduce and Spark.

5.2 Performance

This section compares the performance (running time in millisecond) of applications of Top K, Markov Chain, Connected Components, and Matrix Multiplication on three frameworks, and further discusses the performance and shortcomings of three frameworks.

The performance is assessed by the running time with the unit of millisecond. Performance results of this project were executed on the cssmpi cluster at the University of Washington, Bothell. The cluster includes eight Dell Optiplex 710 desktops, each with an Intel i7-3770 Quad-Core CPU at 3.40 GHz and 16 GB RAM. MapReduce, Spark, and MASS were executed on these machines using 1 to 8 nodes. The result of each performance was from the average outcome of 5 to 10 times stable running.

5.2.1 Top K Application and Performance

5.2.1.1 Top K Application

The Top K algorithm can be used to solve problems like "top 50 hottest topics" in real life. The application of this study simulates this problem by finding k most frequent IDs in a given document. A python script was written and generates an input file with 10 million ID-frequency pairs (See Appendix 1 for a screenshot of the input file generated via the Python script and Appendix 2 for a screenshot of output top 50 most frequent IDs get after running the Top K algorithm).

5.2.1.2 Top K Performance

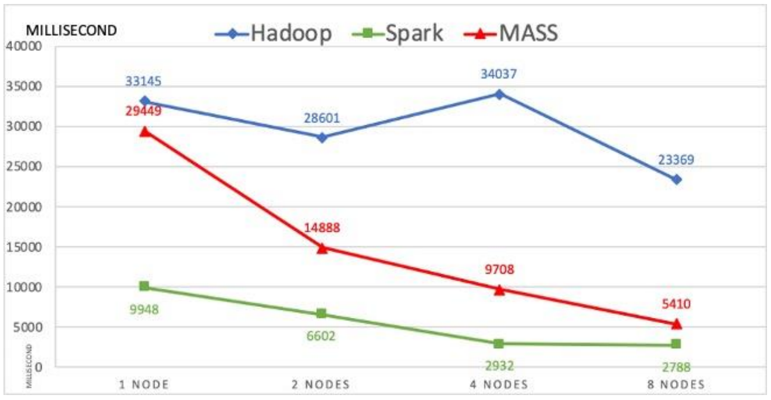


Figure 16. Top K's Performance Comparison of Hadoop, Spark, and MASS

Figure 16 indicates the performance of Top k programs implemented on Hadoop, Spark, and MASS on one, two, four, and eight nodes on cssmpi cluster. The data size is 10 million, and the unit of running time is millisecond.

We could tell from Figure 16 that Spark always uses the least time from single to 8 nodes, and MASS ranked second. The reason is that Spark and MASS processes data in random access memory (RAM). It helps with both frameworks with better performance, but there are memory issues for both Spark and MASS. We will discuss more in detail in Section 5.1.4. Hadoop uses the longest time because MapReduce persists data back to the disk after a map or reduce action.

Another observation is that from 4 nodes to 8 nodes, the performance of MASS keeps improving significantly, but the performance of Spark does not improve much.

Last observation is as indicated in Table 5, reading files costs the longest time during the MASS operation. MapReduce, on the other hand, kills its processes as soon as a job is done, so it can easily run alongside other services with minor performance differences.

Table 5. Top K MASS Running Time on Each Step (1 - 8 nodes)

Time on Each Step	1 node	2 nodes	4 nodes	8 nodes
Time of *Places* creation	91	139	1156	196
Time of Read input	29249	14605	8450	5066
Time of GetSubTree	104	143	100	146
Time of GetFinalTop K	2	1	2	2
Total time	29449	14888	9708	5410

5.2.2 Markov Chain Application and Performance

5.2.2.1 Markov Chain Application

The data used in this project is a simulated customer transaction history data generated by Pranab Ghosh's Ruby script [55] (See Appendix 3 for a screenshot of the input file generated by the Ruby script). In total, there are 10 million transaction records in the simulated customer transaction history. Each input record has the following format: <customerID, transactionID, purchaseDate, amount>. (See Appendix 5 for a screenshot of the output of this program. Each line represents the number of times that a Markov state sequence has occurred. This is almost a Markov Probability Transition Table (MRTT). This MRTT can be used as a machine learning model to predict the next effective date to send a marketing email to that customer.)

5.2.2.2 Markov Chain Performance

Figure 17 indicates the performance of Markov Chain implementation on Hadoop, Spark, and MASS running with one node, two nodes, four nodes, and eight nodes.

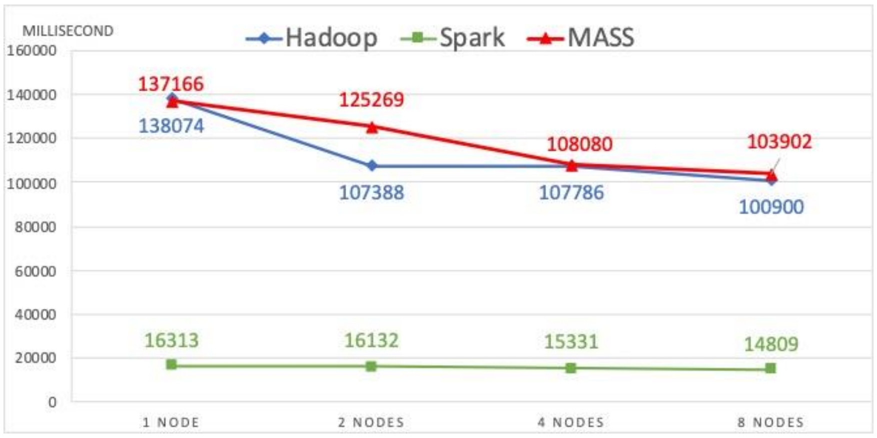


Figure 17. Markov Chain's Performance Comparison on Hadoop, Spark, and MASS

We could tell from Figure 17 that the performance of Spark outperforms the other two frameworks on every cluster setup. Hadoop is slightly better than the MASS, and MASS has the worst performance.

The reason why MASS performs not well is that the expense of IO is significantly dominant than computation for this algorithm. We can observe the IO expenses from Table 6 that the MASS program spends 100 to 140 seconds in total to build the Markov Chain Model, in which 100 seconds is spent on IO. Although the time spent on the computing part to build the model decreases along with more nodes, it does not contribute much to the overall performance. Section 4.2.3 will explain the algorithm design of Markov Chain on MASS at Section 4.2.3. The performance shows that the current parallel strategy might not be the best one for Markov Chain Algorithm. In the future, with more time, it is possible to redesign other parallel strategies. One possible design is to make better use of MASS parallel IO to make each *place* only read partial data to improve the performance.

Table 6. Markov Chain MASS Running Time on Each Step (1 - 8 nodes)

Time on Each Time	1 node	2 nodes	4 nodes	8 nodes
Time of Reading input file	116737	112592	105172	98115
Time of getting sub Markov State	2032	1188	708	356
Total Time	137166	125269	108080	103902

5.2.3 Connected Components & Matrix Multiplication Application and Performance

Considering the performance of the Connected Components and Matrix Multiplication shares a lot of similarities. This study will discuss the performance of these two applications together in this section.

5.2.3.1 Connected Components Application

Connected Components can be used to calculate the connectivity of different network configurations [56] when measuring routing performance in multihop wireless networks in real life. Other than that, Connected Components is often used as the first step for many graph algorithms. For example, in social networks, a group of people usually have a close relationship. Many of these groups usually like some common websites or play common games. The CC algorithm is used to find such groups and recommend them to people in the group who have not yet liked these websites or games.

This project modified the original existing class called GraphGren.java in the DSL lab, and used it to generate a graph of size 2000. This graph is by default separated into at least four different components. (Appendix 5 shows the input format of this application. Each line consists of two parts, representing the current node index before the colon and all the neighbors of the current node listed after the colon. The output of this application is formatted as follows: <number of node in this connected component> : < list of nodes within the current connected component>.)

5.2.3.2 Matrix Multiplication Application

This project wrote a simple java class MatrixGen.java that creates two matrices: A (of dimension 2048 * 2048) and B (dimension 2048 * 2048). After parallel performs matrix multiplication, the output is 2048 * 2048 matrix C where C [i] [j] is the dot product of the ith row of A and the jth column of B.

5.2.3.3 Connected Components and Matrix Multiplication Performance

Figure 18 indicates the performance of Connected Components implementation on Hadoop, Spark, and MASS running with one node, two nodes, four nodes, and 7 nodes. Figure 19 indicates the performance of Matrix Multiplication running on one node, and four nodes. In the final phase of this project, node 8 of cssmpi cluster (see Section 5.2 for cluster equipment information) had technical issues of logging. Therefore, both programs could not be tested using eight nodes.

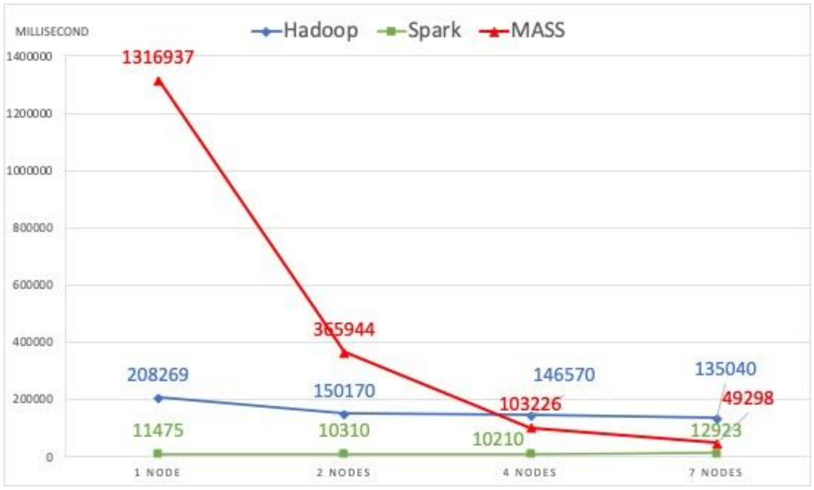


Figure 18. Connected Components' Performance Comparison on Hadoop, Spark, and MASS

Because MASS version of Matrix Multiplication was achieved via Cannon's Algorithm. Cannon's Algorithm is especially suitable for computers with *n** *n* cores. Cssmpi cluster has 8 node and 32 cores in total. If we want to use the number of cores as a square number (*n** *n*), this project would only be able to test single node (4 cores) and 4 node (16 cores) in cluster.

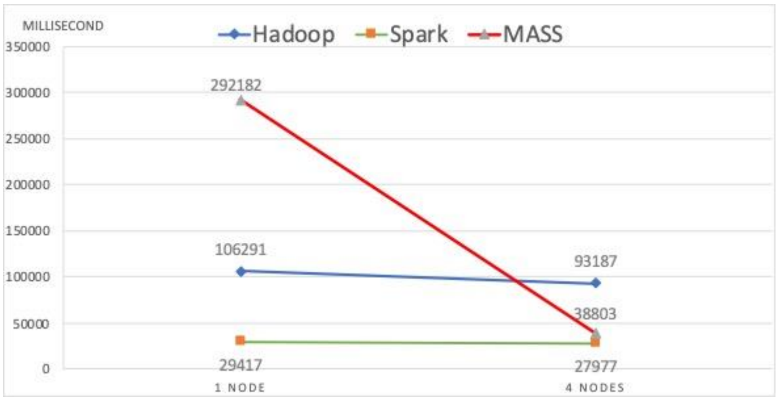


Figure 19. Matrix Multiplication's Performance Comparison on Hadoop, Spark, and MASS

As discussed in Section 3.2.3, algorithms of Connected Components and Matrix Multiplication are selected to be able to benefit from "data discovery". These two algorithms involve structured data (graph for Connected Components and 2D-array for Matrix Multiplication).

As described in Sections 4.3.3 and 4.4.3，both algorithms use multiple agents to parallelize the problem-solving process.

Considering these similarities, the performance of these two algorithms are discussed together. As indicated Figures 18 and 19, when these two algorithms are tested in fewer nodes (1 node, 2 nodes), the execution time of MASS is much higher than the other two frameworks. The reason is that when the total amount of data to be processed remains the same, the fewer nodes are used in the cluster, the more agents populated on each node, and the larger overhead caused by the agent object creation. That is to say, when fewer nodes are used in the cluster, the overhead of MASS cause by agent creation is larger.

However, with increasing nodes in the cluster, the performance of MASS improved dramatically. As indicated in Figures 18 and 19, the velocity of performance improved from both algorithms on MASS is much greater than the other two algorithms. Especially for Matrix Multiplication, when the cluster increases the number of nodes from one to four, the performance of MASS improves 6.5 times. If we can add more nodes to the cluster for testing, the performance of MASS might even exceed Spark.

5.3 Summary

Section 5 discussed the programmability and performance of each algorithm on three frameworks above. In the rest of this session, we will summarize the observation about performance and programmability across all frameworks and applications.

Firstly, when we compare the performance, Spark performs the best, followed by MASS, and MapReduce performs the worst. As we discussed the reasons in Section 5.1.1, Spark and MASS processes data in random access memory (RAM). Hadoop persists data back to the disk after a map or reduce action. Therefore, from both theoretical and practical perspectives, Spark and MASS outperform Hadoop MapReduce. On the other hand, this also makes Spark and MASS limited in memory.

As for Spark, its RDD applies in-memory processing computation when data set can fit into memory. However, as stated in the *Spark Programming Guide*, "When data does not fit in memory, Spark will spill these tables to disk, incurring the additional overhead of disk I/O and increased garbage collection." [57] Although this project did not test data bigger than memory, theoretically speaking, if the data is too big to fit into memory, Spark will suffer major performance degradations.

MASS has a similar issue of memory. Taking Top K as an example, when distributing data, MASS parallel IO first reads the entire input file into memory at once, and then process partition data according to the offset of each *place*. If the input data file is larger than RAM size, MASS is not able to process. As mentioned in Section 5.1.1, the design of MASS parallel IO leads to high cost.

Secondly, there are different overhead of each framework. From the performance of Top K, Connected Components, and Matrix Multiplication, we could find that with increasing nodes in the cluster, the velocity of performance improvement from MapReduce and Spark is not as steep as MASS. This is probably because the design of MapReduce and Spark has more overhead coming from iterative computations that need to pass over the same data many times. For data-streaming-based applications of Top K and Markov Chain, bottle neck is I/O, because data should be read from and send back to the main program. For structured-data-applications of Connected Components and Matrix Multiplication, the communication between different nodes would causes overheads. These applications maintain structured data which has a close relationship and more updates between data. When more node is being used in the cluster, the communication cost increases.

If the number of nodes in the cluster is small, MASS tends to have more overhead. As discussed in Section 5.1.3.3, this is probably caused by the needs to initialize too many *places* and agents on each core.

Considering the performance and programmability comprehensively, this project found that MASS achieve a proper balance as MASS has better performance with increasing nodes in the cluster. In addition, MASS has more intuitive programming, and also has less boilerplate ratio, compared MapReduce and Spark. The balance of programmability and performance of MASS might appeal to engineers dealing with structured data analysis.

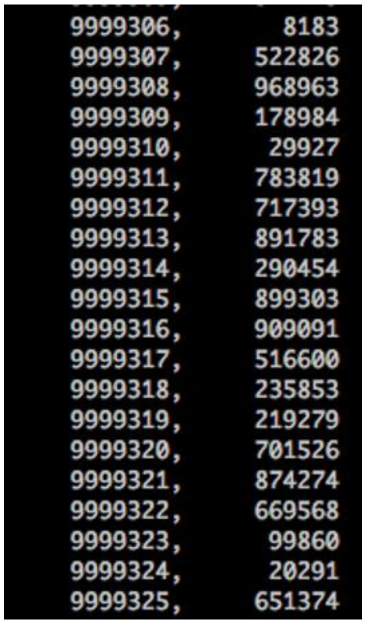
6. Conclusion

This project strives to extend the initial achievement in agent-based data analysis by exploring more data science applications that would be the better fit to agent-based parallelization. This project presents the algorithm designs and implementation process for four applications of Top K, Markov Chain, Connected Components, and Matrix Multiplication on MapReduce, Spark, and MASS. This project further compares the programmability and execution performance of four applications on three frameworks. Comprehensively speaking, MASS demonstrated the good balance between programmability and execution performance for structured data analysis.

Based on the previous work on MASS and this project, future work could be taken in the areas of Computational optimization, Graph algorithm, and Matrix operation library. In addition, we could develop some agent-based modeling package in above for other researcher to extend the study on the MASS.

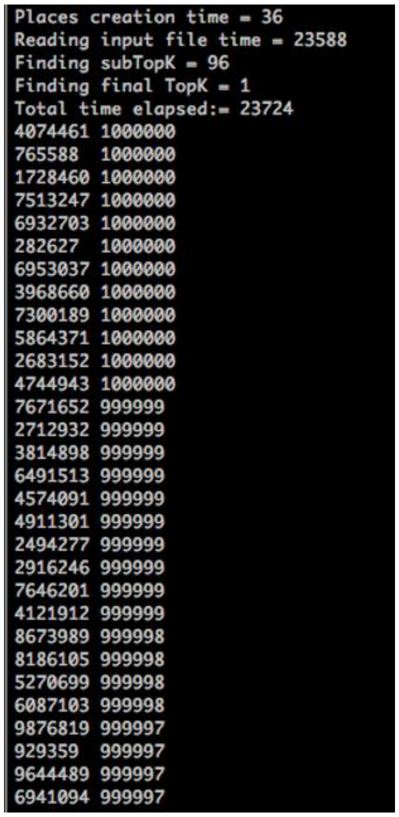
There are also few limitations of this project. This project only chose four data analysis algorithms. Should this project have more algorithms, there would be more and stronger evidences for the findings of this study. In addition, some findings of this study, such as the programmability, could have more quantitative and qualitative evidence. Another limitation is the experience with MapReduce, Spark, and MASS. Since the researcher still have rudimentary understanding of the low-level implementation of these frameworks, the configuration and implementation of algorithms could be optimized and the comparison would be more accurate.

Appendix 1. Screenshot of Input of Top K



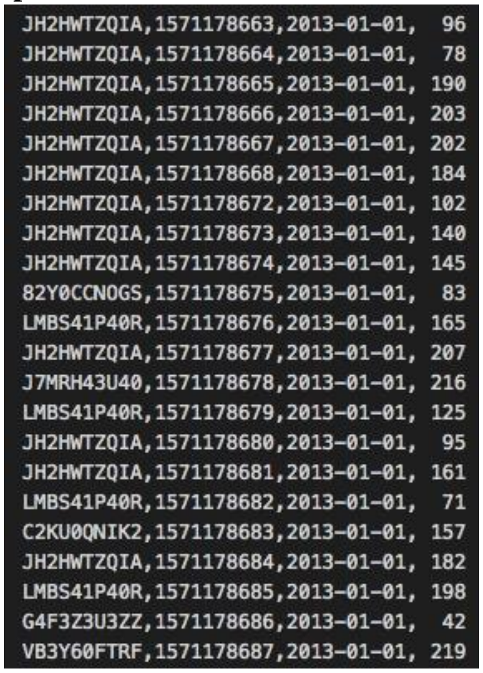
```
9999306,      8183
9999307,    522826
9999308,    968963
9999309,    178984
9999310,     29927
9999311,    783819
9999312,    717393
9999313,    891783
9999314,    290454
9999315,    899303
9999316,    909091
9999317,    516600
9999318,    235853
9999319,    219279
9999320,    701526
9999321,    874274
9999322,    669568
9999323,     99860
9999324,     20291
9999325,    651374
```

Appendix 2. Screenshot of Output of Top K



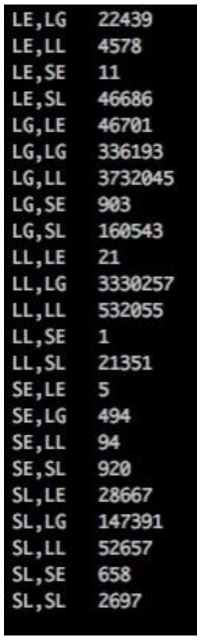
```
Places creation time = 36
Reading input file time = 23588
Finding subTopK = 96
Finding final TopK = 1
Total time elapsed:= 23724
4074461 1000000
765588  1000000
1728460 1000000
7513247 1000000
6932703 1000000
282627  1000000
6953037 1000000
3968660 1000000
7300189 1000000
5864371 1000000
2683152 1000000
4744943 1000000
7671652 999999
2712932 999999
3814898 999999
6491513 999999
4574091 999999
4911301 999999
2494277 999999
2916246 999999
7646201 999999
4121912 999999
8673989 999998
8186105 999998
5270699 999998
6087103 999998
9876819 999997
929359  999997
9644489 999997
6941094 999997
```

Appendix 3. Screenshot of Input of Markov Chain



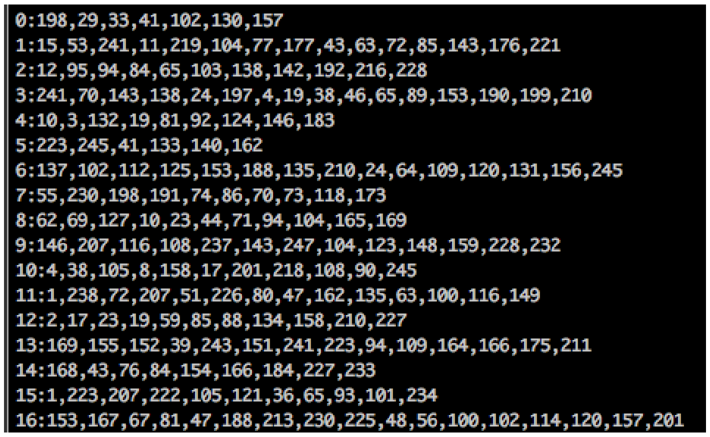
```
JH2HWTZQIA,1571178663,2013-01-01,  96
JH2HWTZQIA,1571178664,2013-01-01,  78
JH2HWTZQIA,1571178665,2013-01-01, 190
JH2HWTZQIA,1571178666,2013-01-01, 203
JH2HWTZQIA,1571178667,2013-01-01, 202
JH2HWTZQIA,1571178668,2013-01-01, 184
JH2HWTZQIA,1571178672,2013-01-01, 102
JH2HWTZQIA,1571178673,2013-01-01, 140
JH2HWTZQIA,1571178674,2013-01-01, 145
82Y0CCNOGS,1571178675,2013-01-01,  83
LMBS41P40R,1571178676,2013-01-01, 165
JH2HWTZQIA,1571178677,2013-01-01, 207
J7MRH43U40,1571178678,2013-01-01, 216
LMBS41P40R,1571178679,2013-01-01, 125
JH2HWTZQIA,1571178680,2013-01-01,  95
JH2HWTZQIA,1571178681,2013-01-01, 161
LMBS41P40R,1571178682,2013-01-01,  71
C2KU0QNIK2,1571178683,2013-01-01, 157
JH2HWTZQIA,1571178684,2013-01-01, 182
LMBS41P40R,1571178685,2013-01-01, 198
G4F3Z3U3ZZ,1571178686,2013-01-01,  42
VB3Y60FTRF,1571178687,2013-01-01, 219
```

Appendix 4. Screenshot of Output of Markov Chain



```
LE,LG   22439
LE,LL   4578
LE,SE   11
LE,SL   46686
LG,LE   46701
LG,LG   336193
LG,LL   3732045
LG,SE   903
LG,SL   160543
LL,LE   21
LL,LG   3330257
LL,LL   532055
LL,SE   1
LL,SL   21351
SE,LE   5
SE,LG   494
SE,LL   94
SE,SL   920
SL,LE   28667
SL,LG   147391
SL,LL   52657
SL,SE   658
SL,SL   2697
```

Appendix 5. Input of Connected Components Application



```
0:198,29,33,41,102,130,157
1:15,53,241,11,219,104,77,177,43,63,72,85,143,176,221
2:12,95,94,84,65,103,138,142,192,216,228
3:241,70,143,138,24,197,4,19,38,46,65,89,153,190,199,210
4:10,3,132,19,81,92,124,146,183
5:223,245,41,133,140,162
6:137,102,112,125,153,188,135,210,24,64,109,120,131,156,245
7:55,230,198,191,74,86,70,73,118,173
8:62,69,127,10,23,44,71,94,104,165,169
9:146,207,116,108,237,143,247,104,123,148,159,228,232
10:4,38,105,8,158,17,201,218,108,90,245
11:1,238,72,207,51,226,80,47,162,135,63,100,116,149
12:2,17,23,19,59,85,88,134,158,210,227
13:169,155,152,39,243,151,241,223,94,109,164,166,175,211
14:168,43,76,84,154,166,184,227,233
15:1,223,207,222,105,121,36,65,93,101,234
16:153,167,67,81,47,188,213,230,225,48,56,100,102,114,120,157,201
```

References

[1] IBM Corporation. "What is batch processing?". *zOS Concepts*. Retrieved Oct 10, 2019 from https://www.ibm.com/support/knowledgecenter/zosbasics/com.ibm.zos.zconcepts/zconc_whatisbatch.htm. Para. 1

[2] Woodring, J., Sell, M., Fukuda, M., Asuncion, H., & Salathe, E. (2017, June). A Multi-agent Parallel Approach to Analyzing Large Climate Data Sets. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)* (pp. 1639-1648). IEEE.

[3] A. Kretz, What is Hadoop and Why is it so Freakishly Popular? https://andreaskretz.com/2016/11/30/what-is-hadoop-and-why-is-it-so-freakishly-popular/, Para.8

[4] A. Kretz, What is Hadoop and Why is it so Freakishly Popular? https://andreaskretz.com/2016/11/30/what-is-hadoop-and-why-is-it-so-freakishly-popular/, Para.17

[5] MapReduce: Parallel & Distributed Programming Model, Tutorials Point, https://www.tutorialspoint.com/map_reduce/map_reduce_tutorial.pdf, p.4.

[6] MapReduce: Parallel & Distributed Programming Model, Tutorials Point, https://www.tutorialspoint.com/map_reduce/map_reduce_tutorial.pdf, p.5.

[7] M. Parsian, *Data Algorithms Recipes for Scaling Up with MapReduce and Spark*. O'Reilly Media, Inc. 2015.

[8] Karau, H., Konwinski, A., Wendell, P., & Zaharia, M. (2015). *Learning spark: lightning-fast big data analysis*. O'Reilly Media, Inc.

[9] Apache Spark. https://en.wikipedia.org/wiki/Apache_Spark, Para. 3.

[10] Karau, H., Konwinski, A., Wendell, P., & Zaharia, M. (2015). *Learning spark: lightning-fast big data analysis*. O'Reilly Media, Inc.

[11] Spark Guide, Cloudera, October 16, 2018 retrieved from https://www.cloudera.com/documentation/enterprise/5-6-x/PDF/cloudera-spark.pdf, p.8.

[12] Spark Guide, Cloudera, October 16, 2018 retrieved from https://www.cloudera.com/documentation/enterprise/5-6-x/PDF/cloudera-spark.pdf, p.8.

[13] Munehiro Fukuda. A Parallel Multi-Agent Spatial Simulation Environment for Cluster Systems (18 pages), the 16th IEEE International Conference on Computational Science and Engineering, Sydney, Australia, December 4, 2013. http://depts.washington.edu/dslab/MASS/docs/ieee_cse2013.ppt

[14] Spark vs Hadoop MapReduce – Comparing Two Big Data Giants https://www.intellectsoft.net/blog/spark-vs-hadoop/, Para. 4

[15] Spark vs Hadoop MapReduce – Comparing Two Big Data Giants https://www.intellectsoft.net/blog/spark-vs-hadoop/, Para. 5.

[16] M. Sell and M. Fukuda. Agent Programmability Enhancement for Rambling over a Scientific Dataset, unpublished paper

[17] Fukuda, M., Gordon, C., Mert, U., Sell, M.: Agent-Based Computational Framework for Distributed Analysis. IEEE Computer (to appear in March 2020). https://doi.org/doi:10.1109/MC.2019.2932964

[18] Munehiro Fukuda. MASS: A Multi-Agent Spatial Simulation Library (a one-Page flier), Office of Research - Undergraduate Research Fair, University of Washington Bothell, October 15, 2013, http://depts.washington.edu/dslab/MASS/docs/Fukuda-UGradResearch.pptx, Para. 4

[19] Fukuda, M. & Kim. W. Toward the Practicalization of Agent-Based Data Discovery. Computing and Software Systems, Unpublished Manuscript.

[20] Ma, Z., & Fukuda, M. (2015, December). A multi-agent spatial simulation library for parallelizing transport simulations. In *2015 Winter Simulation Conference (WSC)* (pp. 115-126). IEEE.

[21] Ma, Z., & Fukuda, M. (2015, December). A multi-agent spatial simulation library for parallelizing transport simulations. In *2015 Winter Simulation Conference (WSC)* (pp. 115-126). IEEE.

[22] Fukuda, M. MASS: Parallel-Computing Library for Multi-Agent Spatial Simulation. Retrieved Feb. 10, 2020. from http://faculty.washington.edu/mfukuda/doc/MassSpec_050710.pdf. Para. 5.

[23] Fukuda, M., Gordon,C. Mert, U., & Sell, M. "An Agent-Based Computational Framework for Distributed Data Analysis," IEEE Computer, Vol.53, No.3, DOI:10.1109/MC.2019.2932964, March 2020

[24] Fukuda, M. & Kim. W. Toward the Practicalization of Agent-Based Data Discovery. Computing and Software Systems, Unpublished Manuscript.

[25] Fukuda, M. & Kim. W. Toward the Practicalization of Agent-Based Data Discovery. Computing and Software Systems, Unpublished Manuscript.

[26] Fukuda, M., Gordon,C. Mert, U., & Sell, M. "An Agent-Based Computational Framework for Distributed Data Analysis," IEEE Computer, Vol.53, No.3, DOI:10.1109/MC.2019.2932964, March 2020

[27] Gordon. C. (2018). Agent-Based Data Science and Machine Learning. unpublished master's thesis. University of Washington, Bothell, WA.

[28] Woodring, J., Sell, M., Fukuda, M., Asuncion, H., & Salathe, E. (2017, June). A Multi-agent Parallel Approach to Analyzing Large Climate Data Sets. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)* (pp. 1639-1648). IEEE.

[29] M. Sell and M. Fukuda. Agent Programmability Enhancement for Rambling over a Scientific Dataset, Unpublished paper

[30] Fagin, R., Kumar, R., & Sivakumar, D. (2003). Comparing top k lists. *SIAM Journal on discrete mathematics, 17*(1), 134-160.

[31] M. Parsian, *Data Algorithms Recipes for Scaling Up with MapReduce and Spark*. O'Reilly Media, Inc., p.259, 2015.

[32] M. Parsian, *Data Algorithms Recipes for Scaling Up with MapReduce and Spark*. O'Reilly Media, Inc., p.258, 2015.

[33] Steven S. Skiena. (2008). The Algorithm Design Manual. Springer, p.166

[34] “Depth First Search” Retrieved Feb. 26, 2020, from https://www.hackerearth.com/practice/algorithms/graphs/depth-first-search/tutorial

[35] Steven S. Skiena. (2008). *The Algorithm Design Manual*. Springer. p. 166

[36] “Numpy Matrix Multiplication.”Retrieved Feb. 26, 2020, from https://hackr.io/blog/numpy-matrix-multiplication. Para 1.

[37] “Matrix multiplication algorithm.” Retrieved Feb. 26, 2020, from ”https://en.wikipedia.org/wiki/Matrix_multiplication_algorithm. Para. 5.

[38] https://github.com/niharsuryawanshi/Graph-Analysis-Projects-Using-Hadoop-Map-Reduce

[39] https://github.com/nm4archana/BigDataAnalysis-Comet

[40] M. Parsian, *Data Algorithms Recipes for Scaling Up with MapReduce and Spark*. O'Reilly Media, Inc., p.44, 2015.

[41] M. Parsian, *Data Algorithms Recipes for Scaling Up with MapReduce and Spark*. O'Reilly Media, Inc., p.44, 2015.

[42] https://blog.godatadriven.com/graph-partitioning-in-mapreduce-with-cascading

[43] https://github.com/niharsuryawanshi/Graph-Analysis-Projects-Using-Hadoop-Map-Reduce

[44] Ghoting, A., Krishnamurthy, R., Pednault, E., Reinwald, B., Sindhwani, V., Tatikonda, S., ... & Vaithyanathan, S. (2011, April). SystemML: Declarative machine learning on MapReduce. In *2011 IEEE 27th International Conference on Data Engineering* (pp. 231-242). IEEE. p.232

[45] https://github.com/nm4archana/BigDataAnalysis-Comet

[46] Ghoting, A., Krishnamurthy, R., Pednault, E., Reinwald, B., Sindhwani, V., Tatikonda, S., ... & Vaithyanathan, S. (2011, April). SystemML: Declarative machine learning on MapReduce. In *2011 IEEE 27th International Conference on Data Engineering* (pp. 231-242). IEEE. p.232

[47] Tang, Y. (2016). Study and implementation on distributed large scale matrix computation algorithms with Spark. Unpublished Master’s Thesis. Nanjing University, China. p.21.

[48] Matrix Multiplication on a Distributed Memory Machine, www.phy.ornl.gov

[49] https://iq.opengenus.org/cannon-algorithm-distributed-matrix-multiplication/

[50] https://blog.godatadriven.com/graph-partitioning-in-mapreduce-with-cascading

[51] Ghoting, A., Krishnamurthy, R., Pednault, E., Reinwald, B., Sindhwani, V., Tatikonda, S., ... & Vaithyanathan, S. (2011, April). SystemML: Declarative machine learning on MapReduce. In *2011 IEEE 27th International Conference on Data Engineering* (pp. 231-242). IEEE. p.232

[52] Gordon. C. (2018). Agent-Based Data Science and Machine Learning. unpublished master's thesis. University of Washington, Bothell, WA. p.26.

[53] Gordon. C. (2018). Agent-Based Data Science and Machine Learning. unpublished master's thesis. University of Washington, Bothell, WA. p.24.

[54] Lämmel, R., & Jones, S. P. (2003). Scrap your boilerplate: a practical design pattern for generic programming. *ACM SIGPLAN Notices*, *38*(3), 26-37.

[55] buy_xaction.rb, https://github.com/pranab/avenir/blob/master/resource/buy_xaction.rb. as cited in M. Parsian, *Data Algorithms Recipes for Scaling Up with MapReduce and Spark*. O'Reilly Media, Inc., p.263, 2015.

[56] "Routing performance in the presence of unidirectional links in multihop wireless networks". Retrieved Feb. 26, 2020, from http://u6.gg/rDnun. Para.1.

[57] "RDD Programming Guide." Retrieved Feb. 26, 2020, from hhttp://spark.apache.org/docs/latest/rdd-programming-guide.html. Para. 35.