Agent-Based Data Science and Machine Learning

Collin Gordon

A whitepaper
submitted in partial fulfillment of the
requirements for the degree of

Master of Science in Computer Science & Software Engineering

University of Washington
2018

Project Committee:
Munehiro Fukuda, Committee Chair
Robert Dimpsey, Committee Member
Dong Si, Committee Member

Program Authorized to Offer Degree:
Computer Science & Software Engineering

Contents

University of Washington

Abstract

Agent-Based Data Science and Machine Learning

Collin Gordon

Chair of the Supervisory Committee:

Professor Munehiro Fukuda

Computing and Software Systems

The Multi-Agent Spatial Simulation (MASS) library is a library that implements an agent-based programming paradigm in Java, C++, and CUDA. This library has been used to great effect in the parallelization of a variety of simulations and data analysis programs. Building on this foundation, the Agent-Based Data Science and Machine Learning project is an exploration into the advantages of using MASS Java to parallelize computationally complex clustering, classification, and graph algorithms.

This project presents the algorithm designs for agent-based versions of K Means Clustering, K Nearest Neighbor Classification, Triangle Counting in Graphs, and the Traveling Salesman Problem. In addition to the designs of the algorithms, we present an analysis of programmability and performance comparing MASS Java to the widely used MapReduce and Spark paradigms. We also explore the contributions of previous graduate researchers and position the project as a launching point for expanding the use-cases for MASS.

1 Introduction

The Agent-Based Data Science and Machine Learning project aims to prove the hypothesis that agent-based parallelism can be used to improve the performance and ease the programmability of some complex data science and machine learning applications.

1.1 Motivation

The motivation for this project lies in the spatial nature of machine learning algorithms. Descriptions of these algorithms often involve operations on objects in a plane or other space. This spatial quality is often lost when the concepts are implemented in code. Additionally, many of the algorithms involved in machine learning are complex and often require parallel programming to run efficiently. The Multi-Agent Spatial Simulation (MASS) library offers an ability to work with spatial representations of data in a parallel format. These qualities make an intriguing case for research into the possible benefits of using MASS for machine learning tasks.

1.2 Goals

Throughout its progression, the Agent-Based Data Science and Machine Learning project has had three main goals:

1. **Design Agent-based Algorithms**: The primary goal at the outset of the project was to design agent-based algorithms for K Means Clustering and K Nearest Neighbor Classification.

2. **Performance Analysis**: The secondary goal was to measure the performance of the K Means Clustering and K Nearest Neighbor algorithms compared to their sequential counterparts. If it was possible, these algorithms would be tested against MapReduce and Spark, two competitor paradigms.

3. **Programmability Analysis**: Examine the programmability of the two algorithms compared to other MapReduce and Spark.

All three of these goals were met and exceeded. The final version of the project saw the design and implementation of four algorithms K Means, KNN, Triangle Counting, and Traveling Salesman. Additionally, performance and programmability was compared between MASS, MapReduce, and Spark. A potential fourth goal would be to test the accuracy of the machine learning algorithms to see if the models created by MASS were more accurate than those from MapReduce and Spark. However, this was not considered a goal for the project since the approaches used to design the algorithms were already deemed accurate in other research [1, 2, 3]. This is covered more in Section 6.1.

MapReduce and Spark were chosen for this project over other available frameworks due to their use in industry and the variety of applications that can be created with each [4, 5, 6]. Another framework we could have used for this project is Apache Storm [7]. However, due to this framework's strict adherence to the data streaming paradigm, it would not have made a good comparison in either algorithm design or performance.

1.3 Audience

This project is aimed towards researchers in data intensive scientific fields who are currently attempting to process a lot of data on a single machine with either a program they wrote or statistical software and who have an intermediate knowledge of computer science. It is the goal of this project to enable researchers to access more computing power without having to go through an entire computer science degree or spending valuable grant money on hiring a software engineer.

Additionally, this project is meant for future researchers at UW Bothell to expand the existing catalog of work on MASS. It has led to a successful publication at the Practical Applications of Agents and Multi-Agent Systems (PAAMS) 2018 Demonstration and will contribute to more publishable works in the near future [8]. It is a goal to be able to use MASS and its features to help students and professors in other areas of the university.

This paper will present an overview of background information in Section 2, implementations of algorithms in Section 3, programmability and performance analysis in Sections 4 and 5, and conclusions in Section 6.

2 Background

This section will provide background information on the three parallelization techniques covered in the paper as well as the concept of biological optimizations.

2.1 MASS

The MASS library, which we have developed at University of Washington Bothell, provides the efficiency of agent-based parallelization while abstracting the parallel environment for the agent-based model [9]. It provides a middle layer that only requires the user to contribute code relevant to their project without having to manage the details of parallelization.

MASS is made up of two different component objects: *Agents* and *Places*. Agents are serializable, migratable objects that are mapped to processes. Places, on the other hand, are objects that are represented by an N-dimensional matrix which are mapped to threads.

A unique feature of MASS is the customizability of the behavior of its components. A frequently referenced concept in this paper is the concept of *static agents* versus *dynamic agents*. Static agents perform computations and store information, but do not create more agents or migrate. In contrast, dynamic agents can migrate from place to place and can spawn children from themselves to increase the agent population. MASS also allows the user to customize the behavior of places by setting which other places a place can communicate to as well as the amount of agents it can have and the type of data it holds. Currently, places can communicate with each other in either Von Neumann or Moore neighborhood format [9]. They can also communicate with any agents currently residing on them. This high level of customization allows for MASS to address problems in a variety of formats.

2.2 MapReduce

MapReduce is a framework for operating on distributed files stored in a cluster running the Hadoop File System (HDFS). Like MASS, it provides a middleware that abstracts the parallelism allowing developers to work on task-specific code [6]. The MapReduce paradigm consists of three components.

1. **Mapper**: The component responsible for parsing and manipulating data from the file.

2. **Reducer** This component takes the output from the Mapper component and combines it into a new format to be written to HDFS.

3. **Job**: The component that executes the parallel code. It sets up file paths, global variables, and file formatting which allow the Mapper and Reducer objects to perform functions on the cluster. Paths and global variables can be referenced by Context objects which are passed to Mappers and Reducers behind the scenes.

Execution of a MapReduce program can be broken down into three stages [6]:

1. **Map** is the stage where a mapper object is created for each line in a file.

2. **Shuffle** is the stage where the output from the Map stage is organized by key before it is passed to the Reducer objects. This stage is hidden from the user.

3. **Reduce** is the stage where a reducer object is created for each unique key from the shuffle stage. This stage will feed its output to a results file known as a *part* file.

While this paradigm has its advantages and does make programming easier for the user, it is very restrictive. In many situations, like the ones in section 3, the desired algorithm requires the chaining of many jobs together or the creation of objects known as *identity mappers* and *identity reducers* that do not perform any manipulation of the data except to pass it on to the reduce step or the next job. There are library options available to help manage job chaining in the form of *counters* and *status indicators* [6]. Examples of both of these concepts can be found in section 3.

2.3 Spark

Spark is an open source project designed to provide a program structure that is compatible with a wide variety of data sources [5]. The Spark Core code is written in Scala which is a hybrid of purely functional and purely object oriented languages. In addition to being available in Scala, Spark and most of its modules are available in Java, Python, and R and it is compatible with HDFS, Amazon S3, and SQL Databases [5]. This project uses Spark Java to maintain comparability with MASS Java and MapReduce.

Spark differs from MapReduce and MASS by its use of immutable data [5]. The building block of Spark is the Resilient Distributed Data structure (RDD). Under the hood, Spark builds a directed acyclic graph (DAG) to represent program flow. Figure 1 shows a graphical representation of the DAG.

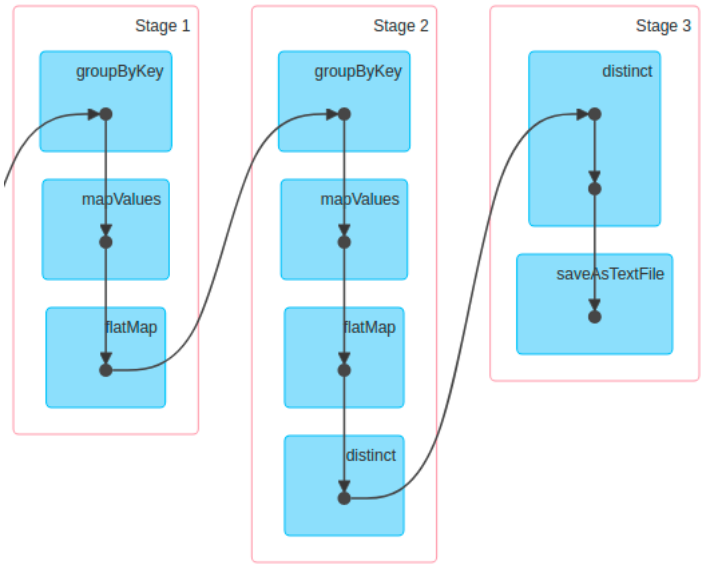

Figure 1: DAG visualization

The DAG is made up of nodes called *stages* that represent an RDD. Each stage outlines the execution of *transformations* and *actions* to alter the data in the RDD [5]. Spark operates on the principle of lazy evaluation meaning that a function does not execute until its data is requested from another function [5]. Following the lazy evaluation principle, the DAG in Figure 1 would be executed in the order of: stage 3, stage 2, stage 1.

When stage 3 is executed, it calls for information that was processed in stage 2 which results in the execution of stage 2. This process is repeated from stage 2 to stage 1 as

well. Spark adds an additional requirement that states that a transformation will only be performed if an action is called upon it. This can be seen in Figure 1 by following the arrow through stage 1.

In stage 1, *groupByKey* is an action function that in turn calls *mapValues* which is a transformation. The *mapValues* transformation needs to know how to map its values which leads to the call of *flatMap*. The function *flatMap* is a higher order function or lambda expression that dictates the transformation of the data.

This order of execution provides a benefit to the Spark architecture that is not replicated in the other frameworks. When Spark is processing data it only needs to store what it needs for each stage to complete which reduces the memory used by its processes. This is referred to as *data streaming* where data is processed as a stream of information rather than a block of memory [5].

Another benefit of the DAG is the improved handling of fault tolerance. In the event that a stage experiences failure, the program retrieves data from the previous stage and restarts its calculation. This is handled internally by Spark itself in contrast with MapReduce and MASS where failures are handled by the user [6, 9, 5].

2.4 Biological Optimizations

Biological optimizations are optimizations that use the behavior of animals in the wild to improve the speed and accuracy of complex problems [10, 11, 12]. These optimizations were a necessary area of exploration for this project due to the ease at which the individuals in the population can be translated to an agent-based format as well as the backlog of previous work simulating different biological principles with MASS [13, 14, 15].

While each optimization is different, they all can be characterized by two steps: *Exploration* and *Exploitation*. These steps are [10]:

1. **Exploration**: The process where agents navigate the data to find a satisfactory target.

2. **Exploitation**: The process where agents use the condition they have found to

attract others to their position.

While there are many biological heuristics that can be used to optimize algorithms, this research explored the Particle Swarm Optimization(PSO) and Ant Colony Optimization (ACO) algorithms [2, 12, 3].

PSO was explored with the hope of using its principles to design a K Means clustering algorithm [2]. When applied to clustering, PSO first generates sets of clusters for a data set equal to the number of members in the swarm. Each member of the swarm determines the most optimal cetroids and communicates the optimal centroids to other members of the swarm. This allows for an optimal solution to be found quicker than computing and analyzing one solution at a time as in traditional K Means [2]. Figure 2 shows a high level depiction of this algorithm.

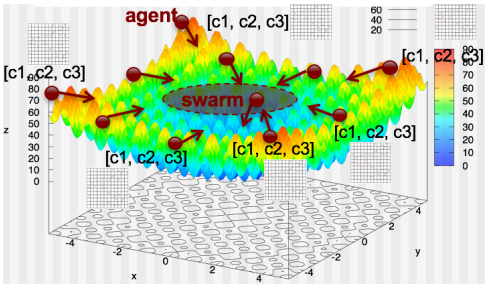

Figure 2: Image representation of the Particle Swarm Optimization algorithm

ACO was explored in part through the parallel implementation of a sequential Traveling Salesman Problem (TSP) optimization [3]. ACO enhances TSP by having each ant visit a node and deposit pheremones. Other ants, in turn, will determine the node that they will visit next based on the amount of pheremones deposited on each node. This algorithm will result in all ants eventually traveling the path with the heaviest deposit of pheremones which in turn is the most optimal path available [3].

3 Algorithms

The following section will cover implementations of K Means Clustering, K Nearest Neighbor Classification, Triangle Counting in Graphs, and the ACO optimized TSP using each parallel method described in Section 2. Each subsection will describe the implementations along with their advantages and disadvantages. Section 4 will discuss quantitative and qualitative analyses of the algorithms in further depth.

The algorithms for K Means, K Nearest Neighbor, and Triangle Counting for Spark and MapReduce have been derived from and checked for correctness against Mahmoud Parisian's work in [4]. Additionally, information about the implementation of counters for the K Means MapReduce program was gleaned from Thomas Jungblut's programming blog [16].

3.1 K Means Clustering

3.1.1 MASS Implementation

The K Means algorithm in MASS uses static agents paired with places to run the K Means algorithm many times over a dataset in parallel. This mimics the exploration phase of the PSO clustering method covered in Section 2.4.

The code for the algorithm is divided into three distinct Java classes:

1. **Storage**: The storage components primarily serve to store and pass information on the data points to the static agents that perform the calculations.

2. **Tool**: Tools are the static agent constructs that perform the calculations involved with the K Means algorithm.

3. **Centroid**: This object is a wrapper consisting of a vector of centroids for the data and a map of data points to each centroid.

Clustering is performed following the traditional K Means algorithm with two methods. These methods are summarized with their parameters below.

1. **kmeans**: This method checks the euclidean distance of each data point with k amount of centroids and assigns the data point to the closest centroid.

2. **collect**: This method presents clusters to the user based on a definition of a desired perfect clustering.

The code in Listing 1 represents the critical section of the main function of the algorithm. The main point of interest is line 7. At this line the clustering is performed using the *kmeans* function. *doAll()* is a wrapper function that repeatedly calls the function *callAll()* for a set number of iterations. This number of iterations is decided by the user and assigned to the *nIter* variable. The benefit of using *doAll()* over *callAll()* is that *callAll()* must be called from the main program, meaning that each iteration using *callAll()* would have to defer back to the main program before executing again. This deferrence requires a barrier synchronization of all the nodes in the cluster which takes additional time.

```
1   MASS.init();
2   Places storage = new Places(100, Storage.class.getName(),
3                               dataPoints, sizeX,sizeY);
4   Agents tools   = new Agents(100, Tool.class.getName(),
5                               null,storage, nAgents);
6   tools.callAll(Tool.init_);
7   tools.doAll(Tool.kmeans_,nIter);
8   Object[] finalRes = (Object[])tools.callAll(Tool.collect_,
9                                 new Object[tools.nAgents()]);
10  MASS.finish();
```

Listing 1: Critical Section of the K Means MASS program

There are both advantages and disadvantages to this implementation. The advantage of implementing K Means in this way is the reduction of time spent on multiple runs of the algorithm. Since how a cluster set is produced is dependent on where the centroids for the clusters are picked, the algorithm is often run more than once [4]. MASS allows for numerous runs of the data to be processed in parallel over multiple nodes which is an

advantage over other implementations. With the addition of *doAll()*, the advantage of this implementation is multiplied since the agents can run their algorithm without barrier synchronization [17].

The disadvantage of this algorithm is that the data being clustered must be small since each place and agent must have a copy of the full dataset. A massive set of data would likely crash this algorithm. The remedy for this is to use dynamic agents allowing data to be partitioned among places as in the other MASS implementations described in Section 3. However, due to limitations in the migration capabilities of MASS, this is not yet possible to do [18].

3.1.2 MapReduce Implementation

The MapReduce implementation of K Means uses several advanced constructs from the library. While there is only one job, the algorithm must run it multiple times. On each iteration, the job is reinitialized, opens the file containing the current data points with their centroids, operates on it, and closes the file. The reducer step is responsible for checking each centroid to see if it has changed between iterations. If there is a change, a counter stored in the Hadoop configuration is incremented. The algorithm ends when the counter is zero. The main execution loop is shown in Listing 2.

```
1        while(update > 0){
2            conf = new Configuration();
3
4            conf.set("centroid.path", centroids.toString());
5            conf.set("cycles", iter + "");
6
7            job = Job.getInstance(conf);
8            job.setJobName("KMeans Clustering " + iter);
9            job.setJarByClass(KMeans.class);
10           job.setMapperClass(KMeansMap.class);
11           job.setReducerClass(KMeansReduce.class);
12
```

```
13              in = new Path("files/clustering/depth_" + (iter - 1) + "/");
14              out = new Path("files/clustering/depth_" + iter + "/");
15
16              FileInputFormat.addInputPath(job, in);
17
18              checkFileExists(fs, out);
19              FileOutputFormat.setOutputPath(job,out);
20              job.setInputFormatClass(SequenceFileInputFormat.class);
21              job.setOutputFormatClass(SequenceFileOutputFormat.class);
22              job.setOutputKeyClass(CenterVector.class);
23              job.setOutputValueClass(PointVector.class);
24              job.waitForCompletion(true);
25              iter++;
26              update = job.getCounters();
27              findCounter(KMeansReduce.Counter.UPDATED).getValue();
28          }
```

Listing 2: Critical Section of K Means MapReduce

This algorithm has an advantage over the previously described MASS implementation. Each point in the dataset is represented by a map object meaning that the majority of data being processed in the algorithm will be partitioned amongst the nodes in the cluster. MapReduce's underlying parallel code ensures that map objects will be efficiently distributed. The centroids of the clusters are kept in a separate file in HDFS, but are small in number. Therefore, this is not as taxing on the map objects as containing the dataset might be on the places in MASS.

There are two major disadvantages to running K Means on MapReduce. The first disadvantage is that a new job must be instantiated for every iteration. This means that files must be read from, written to, and deleted along with a reinitialization of the job itself which as is shown in Listing 2 is not only taxing to the programmer to write, but involves creation of new objects or accesses to underlying class variables. While the programmer may be able to use copy and paste to allieviate the pressure from this, HDFS

must create objects fresh which takes up computation time and slows the program down.

The second disadvantage is that the algorithm only generates one cluster set. As noted in the previous section, K Means is normally executed many times in order to find an optimal clustering. Rather than providing a convenient way to do this programmatically, MapReduce involves using an outside script to coordinate multiple runs of the algorithm.

3.1.3 Spark Implementation

The Spark implementation of K Means involves the use of the built in Spark machine learning library *mllib* [19]. The library implementation provides different options to the user for both the distance measurement and the centroid initialization.

Mllib offers both manhattan and euclidean distance for the measure between data points and centroids and allows users to initialize centroids at random or pick them using a method called *kmeans parallel* which is a parallel version of the *kmeans++* method [20, 19]. Once the distance measure and centroid initialization have been set, the algorithm performs traditional K Means clustering that ends once the centroids have converged. Listing 3 shows the main function of the Spark K Means program.

```
1        public static void main(String[] args){
2            String inputFile = args[0];
3            int k = Integer.parseInt(args[1]);
4            int iterations = Integer.parseInt(args[2]);
5            int runs = args.length >= 4? Integer.parseInt(args[3]) : 1;
6            SparkConf conf = new SparkConf().setAppName("KMeans");
7            JavaSparkContext context = new JavaSparkContext(conf);
8            JavaRDD<String> lines = context.textFile(inputFile);
9            JavaRDD<Vector> points = lines.map(new ParsePoint());
10           KMeansModel model = KMeans.train(points.rdd(), k,
11           iterations, runs, KMeans.RANDOM());
12           context.close();
13       }
```

Listing 3: Main function of Spark K Means

A unique feature of the Spark built-in function is the *run* parameter. This parameter allows the user to run the K Means algorithm more than once [4]. If the user elects to provide a value to *run*, the underlying library repeats the K Means function for the specified value and returns the best set of clusters for the data. This functionality is important because it mimics both the MASS implementation described in Section 3.1.1 and in the implementation of PSO described in Section 2.4 [2]. Unlike MASS, the runs in Spark are not all performed simultaneously. Instead they are performed in groups based on the division of tasks by the underlying task implementation in Spark. It is hard to predict whether the runs are executed in a dynamic or static fashion. However, it seems practical to use dynamic execution since the algorithm is not reliant on steps assigned to other threads to complete its run.

Additionally, *mllib* stores its information about the K Means algorithms in a *KMeans-Model* class which stores the centroids and memberships as well as additional functions that compute the cost and other information pertaining to the algorithm.

3.2 K Nearest Neighbor

3.2.1 MASS Implementation

The agent-based K Nearest Neighbor (KNN) program classifies unlabeled data points based on the k number of labeled neighboring data points to its position. MASS uses dynamic agents combined with the water strider heuristic to perform a parallel version of the KNN algorithm.

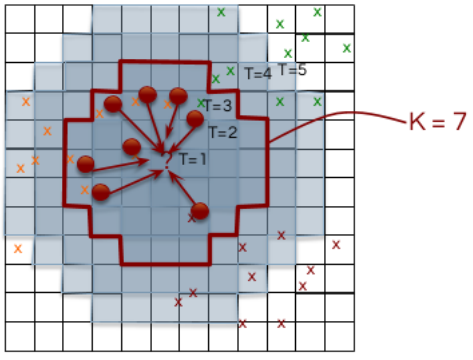

Figure 3: Image representation of the agent-based K Nearest Neighbor algorithm

The KNN algorithm works by placing an agent containing an unlabeled data point at a position in the places grid. Each place contains one or two labeled data points with each data point grouped by label. At intervals of T, the agent spawns children on places near it which then deliver the label attached at their place by migrating back to the origin. As each T increases, the amount of places considered increases. In figure 3 the desired k number of neighbors is seven. T stops increasing once seven neighbors have been captured by the ripple. The agent in question in Figure 3 would be classified as belonging to the orange cluster since its children reported five of seven neighbors to be orange.

In the KNN program, places are referred to as space and the agents as reporters. After the space array is created it reads data from a file and disseminates it to the other places at line 1. Line 2 is the main loop that repeats until all places have been touched by an agent. Inside this loop, the *detect()* and *propogate()* methods disseminate and respond to the ripple. At line 6, the reporter agents spawn children on the space that has been touched by the ripple. Each child is created as a new instance of an agent class creating additional performance overhead. Line 10 exits the loop once enough reporters have been created to satisfy the given k number of neighbors.

```
1    space.callAll( Space.init_ );
2    for ( int i = 0; i < placesize; i++ ) {
3        space.callAll( Space.detect_ );
4        space.callAll( Space.propagate_ );
5        space.exchangeBoundary( );
6        reporters.callAll( Reporter.spawnChild_ );
7        reporters.manageAll( );
8        if ( reporters.nAgents( ) >=
9             k + placesize * placesize )
10           break;
11   }
```

Listing 4: Critical Section of the agent-based K Nearest Neighbor Program

This algorithm works well for small queries of data as each point must be entered by the user however, it is not ideal for large quantities of data. Classifying whole sets of unlabeled data is far more common than a manual request for classification.

Originally, the algorithm attempted to classify large amounts of data. This implementation was impractical due to the overhead incurred when using dynamic agents. When MASS agents create children they create brand new agents and initialize them with new values. Depending on the algorithm, this could mean making a copy of an agent that takes up a lot of room in memory. In the case of KNN, not only are agents spawning new children, they are also determining whether they need to be removed or not due to being in a previous ripple and no longer being needed. Initial performance results for this algorithm against a sequential program yielded a performance reduction. Therefore, until agent overhead is examined in depth, this algorithm will be limited to user input.

3.2.2 MapReduce Implementation

KNN in MapReduce consists of one map task and one reduce task. Firstly, the map task reads in a training file from memory and stores it into a data structure during the setup phase. The map function takes each point from a testing file and compares it to different points in the training set using the euclidean distance formula. Once each test point has been grouped with all close training points, the reducer narrows down the list of training points to the k nearest points and uses the majority vote to give the test data its label.

This algorithm is advantagous because it can classify a truly large set of data as the unlabeled data can be stored on HDFS. However, this algorithm suffers from the same drawback as the MASS K Means algorithm in Section 3.1.1. The training set is read into each map object and stored entirely within it. This means that the training set has to be limited in size which in turn may cause problems with the accuracy of the classification in a situation where the unlabeled data is larger than the training data.

3.2.3 Spark Implementation

KNN in Spark is similar to the MapReduce implementation with one key difference. The difference is the grouping of training and testing data based on the cartesian product. The cartesian product is necessary in Spark since both the training and test data are represented by RDDs. This representation means that we need a way to combine two RDDs into one and also group items from one RDD to the other RDD. The *cartesian()* function takes an RDD as a parameter and returns a key value pair of items from the calling RDD paired with items from the parameter RDD. These pairs are then operated on in a map function that computes the distance between each testing item and its associated training item and stores possible classifications and shortest distances as values along with the training key. The resulting RDD is operated on using a *groupByKey()* reduction followed by a map that acts as a reducer where each test data is classified based on a majority vote of all the training data that is closest to it. The resulting RDD is then saved to a text file.

This implementation is very beneficial when it comes to practical classification problems. Since both unlabeled and labeled data are stored in RDDs the only limit on data size is the number of nodes in the cluster. Additionally, storing both in RDDs ensures that the code that is abstracted away from the user properly partitions both datasets for optimal processing power. The drawback to this implementation is that all the data must be numerical as with many implementations of machine learning algorithms [19]. This means that the user will likely spend more time preprocessing their data to fit the confines of the algorithm.

3.3 Triangle Counting

3.3.1 MASS Implementation

The MASS implementation of Triangle Counting uses places to represent the nodes in the undirected graph and agents as travelers along the graph. Each agent travels along the graph migrating to a node with a lower ID than the one it is currently on. Agents

spawn children when they encounter a node with multiple edges. Agents check if an array of visited nodes on every third move. If the agent has returned to the same node after three moves, it has found a triangle.

The advantage of this implementation is that it removes the need to account for duplicate triangles. However, as in Section 3.2.1, the agent migration creates additional overhead to consider. This is a small price to pay for the elimination of a whole step in the algorithm. However, the drawback is in the building of the graph structure. This is done by generating the graph in memory, by reading a file sequentially into the master node and distributing it among the places, or by using MASS Parallel File I/O. These approaches are slower than their MapReduce and Spark counterparts.

3.3.2 MapReduce Implementation

The implementation in MapReduce is split amongst three jobs [4]. The first job reads a graph in from a text file and outputs each connection in the graph. This output is different from the original input as it outputs a destination to source connection for each source to destination connection in the original file. Based on the mapper output, the reducer outputs two values. The first is a pair of nodes with no connecting node. This occurs in paths between nodes that do not have a node in common. These paths are output with a zero signifying there is no connecting node. The second value output is the nodes that share a path that includes a common node. These values become the possible triangles in the second job.

The second job takes the output file from the first job and finds all potential triangles in the graph. The map object in this class is an identity mapper. Identity mappers are objects that pass their input data into the reduce object without any modification. Listing 5 shows the detail of this job's identity mapper. The reducer takes the information from the mapper and filters out of the zeroes from the previous step's reducer leaving only the nodes that share a connecting node. The connecting nodes are output along with the pair of nodes they connect.

```
1    public class TriangleMap
```

```
2     extends Mapper<EdgePair,IntWritable,EdgePair,IntWritable>{
3     @Override
4     public void map(EdgePair pair, IntWritable connector, Context context)
5     throws IOException,InterruptedException {
6         context.write(pair, connector);
7     }
8}
```

Listing 5: Identity Mapper

The third stage takes the triangles output from the previous reducer and sorts them. Each sorted triangle is output as a key to the reducer step. Since the keys in a reducer are unique the final stage in the algorithm is an identity reducer meaning that the input from the mapper is passed directly through to output with no additional processing.

There are two problems with this implementation. The first is the use of identity mappers and reducers. It must be noted that while the code shown in Listing 5 is visually short, the programmer is overriding only one of several methods that are called by the mapper which in turn have several variables and routines associated with them. This can mean an unecessary consumption of memory and time.

Logically, the algorithm should be able to skip this step seeing as the mapper only funnels information to the reducer in the next step. However, the paradigm does not work this way. A reducer cannot exist without the phases before it being completed. This means that due to the paradigm, additional code has to be written that does very little for the progress of the overall algorithm.

The second problem with the implementation comes from the formatting of the file. The structure of the file for both MapReduce and Spark required every pair of nodes to be on a separate line resulting in the structure shown in Listing 6 [4].This can lead to very large files which can greatly slow performance. This issue is covered more in Section 5.3.

```
1     1 2
2     2 3
3     2 4
```

```
4    2 5
5    3 4
6    4 5
```

Listing 6: File structure

3.3.3 Spark Implementation

The Spark implementation of Triangle Counting mimics the one in 3.3.2 however, instead of chaining jobs together, Spark uses RDDs to represent each step [4].

The key difference from 3.3.2 is the final step of the program. This step, shown in Listing 7, is worth highlighting due to the fact that it determines which of the possible triangles are actually triangles and removes duplicates in on step. The highlight of this code is line 37. The *distinct()* function is a form of filter that is applied to the RDD. As the name implies, it removes duplicates of data found in the RDD. In Section 3.3.2, it was shown that MapReduce required an entire job to perform this function and that job contained an identity reducer. Additionally, the implementation in Section 3.3.1 required that the agent check its itinerary of visited nodes to see if it had already passed the node in question. While the checking of the itinerary is more efficient and easier to manage than an extra job, the Spark implementation of a filter is easier on the cognitive load of the programmer.

```
1   JavaRDD<Tuple3<Long,Long,Long>> uniqueTriangles =
2   possibleGroup.flatMap(
3   new FlatMapFunction<Tuple2<Tuple2<Long, Long>,
4   Iterable<Long>>, Tuple3<Long, Long, Long>>() {
5       @Override
6       public Iterator<Tuple3<Long, Long, Long>>
7       call(Tuple2<Tuple2<Long, Long>, Iterable<Long>> angles)
8       throws Exception {
9           Tuple2<Long,Long> key = angles._1;
10          Iterable<Long> connectors = angles._2;
```

```
11
12
13                  boolean seenZero = false;
14                  List<Long> connector = new ArrayList<>();
15                  for(Long node : connectors){
16                      if(node == 0){
17                          seenZero = true;
18                      } else {
19                          connector.add(node);
20                      }
21                  }
22
23                  List<Tuple3<Long,Long,Long>> result = new ArrayList<>();
24                  if(seenZero){
25                      if(!connector.isEmpty()) {
26                          for (Long node : connector) {
27                              long[] Triangle = {key._1, key._2, node};
28                              Arrays.sort(Triangle);
29                              result.add(new Tuple3<>(
30                              Triangle[0], Triangle[1], Triangle[2]));
31                          }
32                      }
33                  }
34
35                  return result.iterator();
36              }
37          }).distinct();
```

Listing 7: Finding Unique Triangles with Spark

3.4 Traveling Salesman: Ant Colony Optimization

3.4.1 MASS Implementation

The MASS implementation of the Ant Colony Optimized TSP (ATSP) borrows the graph set up used in the implementation of Triangle Counting from Section 3.3.1. The places are used to represent the different cities in the graph and agents represent the ants that travel along different routes. As ants walk along the same route the path of pheremones that they leave becomes stronger eventually resulting in all of the ants moving along the same route.

This algorithm is a prime example of the strengths of MASS. ATSP is reminiscent of one of the example programs for MASS, Sugarscape [14]. In that program, places represent sugar and ants move from sugar source to sugar source until all the sugar has been consumed. One of the difficulties in adapting the sequential version of ATSP to a parallel implementation is that the algorithm is described in a spatial manner, but programmatically, the spatially oriented description must be altered to a flat representation. MASS, being a spatial simulation library, allows for the algorithm to easily be translated from its description to reality without making the sacrifices MapReduce and Spark make to bring the algorithm to fruition.

One of the big advantages of the MASS implementation is that it splits the computational load among places and agents. This allows for objects to take up less space in memory and not be bogged down by having to perform a bulk of calculations on one thread or process. As with other implementations using dynamic agents, there is the factor of agent migration overhead, however this overhead is acceptable as data sets increase in size.

3.4.2 MapReduce Implementation

MapReduce presents two options for the implementation of ATSP. The first is an implementation where each map task is representative of an ant and each reduce task is responsible for manipulating the graph to update trails. The second is a batch algorithm similar to the MASS K Means implementation in Section 3.1.1. In this implementa-

tion, each map task runs its own ATSP problem and a single reduce task picks the most efficient route from among all of the map tasks.

Both of these implementations present slightly different challenges. The first implementation is programmatically far more complex than the second. Each map task needs to keep track of its position in the graph and check to make sure its not processing the same node at the same time as another task. While the difficulty is ramped up for programming, the advantage is that MapReduce used in this way could potentially process a very large graph. The second implementation, while easier, does have one drawback. The drawback is that the graph must be able to be loaded into the memory taken up by the map tasks. This puts a limit on the size of graph that can be processed. [3] came with pregenerated files designed to replicate the results of the paper. Therefore, we used the batch implementation for this project.

Since map tasks are generated per line in a text file, the input for the map object is a line of text that contains a number. This ensures that for every number in the text file, a map object is created. Once the map is instantiated it reads user defined variables such as the number of iterations from the configuration. The graph must be read from a file in the setup phase of each mapper. Then, each map goes through the sequential implementation of the ATSP problem and passes its results to the reducer. The reducer cycles through each map output and outputs the best tour found by the mappers.

3.4.3 Spark Implementation

The Spark implementation mirrors the MapReduce implementation in the previous section. However, there are some key advantages to Spark that MapReduce does not have. Firstly, the ATSP program is wrapped in an ATSPInstance class which is contained within an RDD. Spark has a *parllelize()* function which takes a collection and turns it into an RDD. Therefore, rather than using a text file to generate instances, the user passes a parameter to the program which initializes an ArrayList of ATSPInstance classes. This ArrayList is then passed to parallelize and turned into an RDD.

There are two ways to represent the graph in Spark. The first is through a Broadcast

object and the second is through a global variable. For ATSP we used the global variable. This is due to having the graph size in advance. Broadcast objects help ensure that information is only passed once to the worker nodes in the Spark cluster which improves performance when working with large data. However, the graphs used for the tests in Section 5 were small enough to where passing them would not have signifcantly affected performance time.

4 Programmability Analysis

The following subsections provide an insight into the programmability of each algorithm described in Section 3. Programmability for each paradigm was evaluated based on the boilerplate ratio, number of classes written, and how each algorithm represents its data.

4.1 Boilerplate Ratio

Boilerplate Ratio is calculated by taking the number of lines of code dedicated to setting up the parallel environment and dividing it by the total lines of code in the main function. Boilerplate refers to code involved in preparing the paradigm for execution. This means that in MapReduce and Spark the configuration lines are counted and in MASS all lines that use *MASS.** are counted. For all three frameworks, lines that parse command line arguments are counted as well. To maintain fairness, the place and agent constructor lines are left out since we cannot see similar lines in MapReduce and Spark due to abstraction. Table 1 presents the ratios for each algorithm as well as the average ratio and total source lines of code among all of the main functions for each framework.

Table 1: Boilerplate Ratio

Paradigm	K Means	KNN	Triangles	ATSP	Average	Main SLOC Total
MASS	0.12	0.20	0.16	0.30	0.19	148
Spark	0.10	0.20	0.10	0.37	0.19	130
MapReduce	0.75	0.80	0.81	0.85	0.80	147

The results in Table 1 reveal that Spark and MASS both use less boilerplate code than MapReduce. The reason this is significant is because all three paradigms aim to reduce the interaction between the user and the underlying parallel systems. MapReduce needs extra boilerplate code in order to tell it what files to set up and where to find them. This means that in addition to knowing how the MapReduce paradigm works, the user is also expected to know how the underlying HDFS works.

In contrast, both Spark and MASS work to incorporate as much of core java into their setup as they can. All of the programs mentioned in Section 3 that read from a file in MASS and Spark either use core java methods and classes such as BufferedReader or

invoke provided methods. Files are read into RDDs in the majority of Spark applications. As a result, the RDD class comes with built in methods for text, sequence, and binary files. The user can instruct Spark to read files from its different supported cluster managers and file systems by providing the URL for the file as the path [5]. Therefore, an HDFS text file that contains data for the Triangle Counting program would be read in as follows:

```
1    JavaRDD<String> lines = context.textFile("hdfs://user/tridata.txt");
```

The equivalent MapReduce code would require that the user tells MapReduce what kind of data the file contains as well as what type of file it is. Spark removes this complexity.

MASS has a lower ratio than Spark in every program except for KNN. What makes KNN unique is that it has 10 command line arguments. KNN is also the only program to accept user input which requires additional setup and code. The reason that MASS has a lower ratio than either Spark or MapReduce is due to the way the parallel environment is set up. *MASS.init()*, the method responsible for kicking off the parallel environment, used to take in command line arguments, but now reads all of its setup data from an external *nodes.xml* file. This file is universal to all MASS java applications and can be reused several times for different programs. Therefore, the only additional boilerplate required besides application specific command line arguments is setting the number of threads to use on each computing node.

4.2 Classes

Data considered for Table 2 included mappers, reducers, places, agents, and RDDs. Utility classes such as the Centroid class from Section 3.1.1 are counted as well. The decision to count RDDs as classes came from the observation that in many cases each RDD corresponded to a map and reduce pair.

Table 2: Class Count

Paradigm	K Means	KNN	Triangles	ATSP
MASS	4	5	5	3
Spark	3	6	6	4
MapReduce	5	4	8	3

Given this data, the results in Table 2 are quite interesting. Despite the efficiency in the reduction of boilerplate, the three paradigms stay consistent with the amount of classes that need to be implemented. Judging from Table 2, the most egregious offender in terms of classes is the Triangle Counting program. While the MASS implementation of this algorithm clearly edges out MapReduce in the number of classes it requires, it is important to note that the classes counted for MapReduce do include an identity mapper and an identity reducer. Both of which are not difficult to implement.

At the other end of the spectrum, ATSP has the least additional classes associated with it. This is because it fits perfectly into all three paradigms. MASS can take advantage of its spatial nature using a place class to represent the graph nodes and an agent class to represent the ants. Similarly, MapReduce and Spark only need a way to read the file, process it, and output it. Spark only has four classes due to the ATSPInstance class which is a wrapper for the algorithm so that it can be called as an element in an RDD.

4.3 Data Representation

Data representation refers to how the paradigm internally represents the input data it uses in each algorithm. When data is represented flatly, it is represented as a list of items similar to the input file it came from. Conversely, when data is represented spatially, it is represented as it would be in a visualization or diagram.

Table 3: Data Representation

Paradigm	K Means	KNN	Triangles	ATSP
MASS	Flat	Spatial	Spatial	Spatial
Spark	Flat	Flat	Flat	Flat
MapReduce	Flat	Flat	Flat	Flat

While Table 3 does not show much variation between the representation of different algorithms, it represents an important advantage of MASS over MapReduce and Spark. This advantage is the ability to easily live in both states of data representation.

Unlike Spark or MapReduce, MASS can be both spatial and flat due to the existence of places. Since places themselves are represented as an N-dimensional matrix, they can be used in a variety of contexts. For instance, in both Triangle Counting and ATSP, the

places are used to represent nodes in a graph, while in KNN they are used to represent data points on a plane. Even more still, in K Means they don't represent anything and simply exist to store data between calculations of the algorithm.

This flexibility is achieved through the access the user has to each parallel component's behavior. While MapReduce and Spark do effectively abstract almost the entire parallel environment and can be used to do a lot of parallel tasks efficiently with little knowledge of parallel programming, MASS proves that there can be more benefit than detriment to leaving component behavior up to the user.

As stated in Section 2.1, MASS allows for a high degree of customization of its components allowing for it to encompass both flat and spatial data representations. While proponents of MapReduce and Spark might argue that this level of customization increases the difficulty of working with MASS, Table 3 shows that this difficulty is offset by greater flexibility in algorithm design.

5 Performance Results

The following performance results were evaluated on the University of Washington, Bothell's general purpose linux cluster. The Linux cluster contains 16 Dell Optiplex 710 desktops, each with an Intel i7-3770 Quad-Core CPU at 3.40 GHz and 16 GB RAM. MASS and Spark were evaluated on clusters consisting of 1, 2, 4, and 8 nodes. MapReduce programs were only evaluated on a single node. An explanation of this phenomenon can be found in Section 6.2. In order to be presented in a graph, results had to complete within 10 minutes. Anything that took longer is addressed in writing and not in graphical format.

5.1 KMeans Performance

Performance for K Means was evaluated with 1,000 two dimensional points with a max X and Y coordinate of 30. For MASS and Spark, the frameworks where the algorithm could be run more than once, the number of agents and runs was set at 5,000.

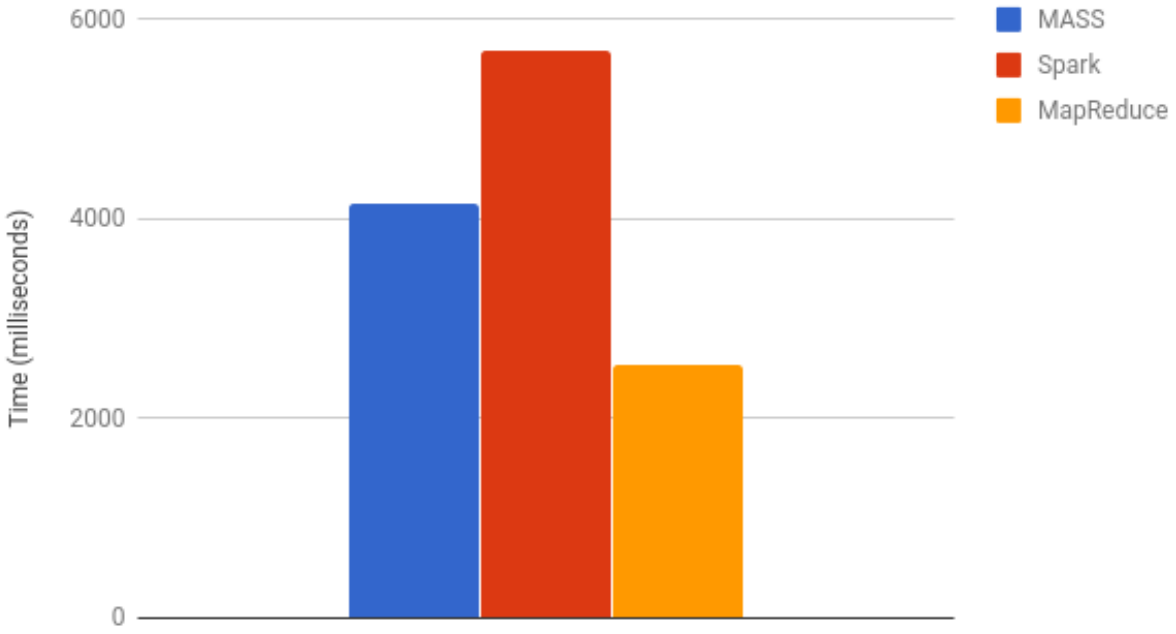

Figure 4: Performance difference between MASS, Spark, and MapReduce

In the Figure 4 we can see the single node performance of the K Means algorithm across the three paradigms. The observation made here is that MapReduce vastly out-

performs MASS and Spark. This is due to the design of the algorithm. Unlike MASS and Spark, MapReduce's algorithm only runs once. Therefore, to be on par with the other frameworks, the performance of 2542 milliseconds should be multiplied by 5,000. This results in 12,710,000 millseconds or 3.5 hours.

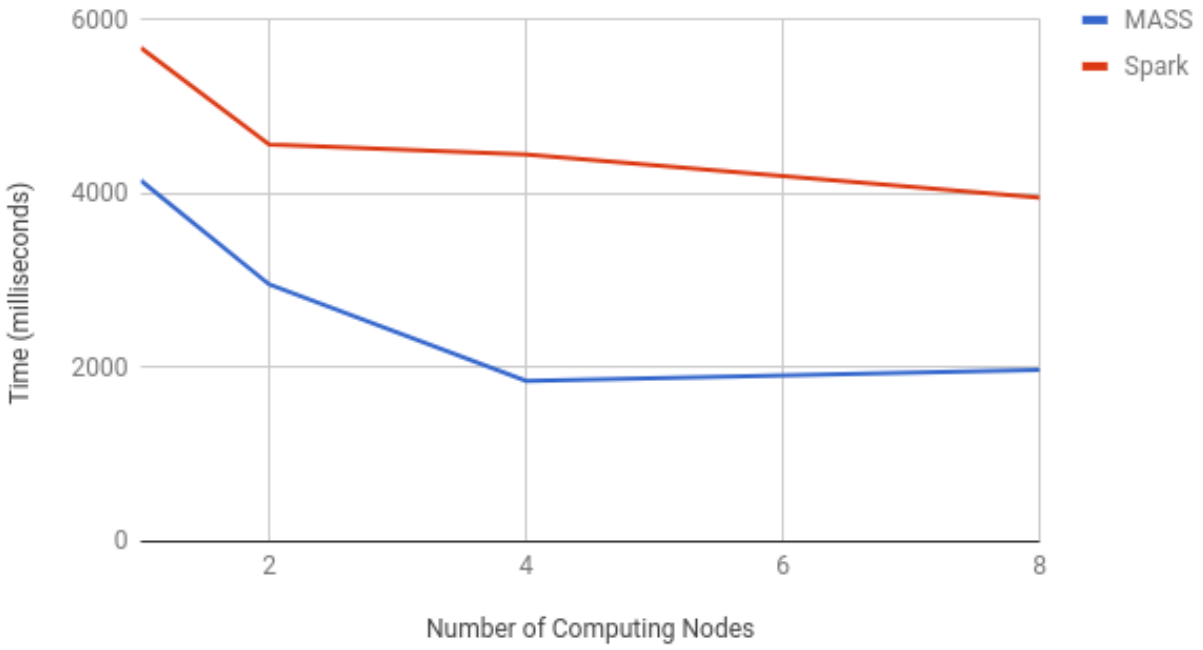

Figure 5: Performance difference between MASS and Spark

As can be seen in Figure 5, MASS outperforms Spark on every cluster setup. This showcases the advantage of being able to run all 5,000 runs without synchronization and without reverting to the driver until completion.

5.2 KNN Performance

KNN performance was judged using a grid of 4,000 x 4,000 data points. This grid was represented by places in MASS and a file in Spark and MapReduce.

MapReduce proved too slow to compare as it surpassed the limit of 10 minutes. However, Figure 6 shows the difference between MASS and Spark.

As can be seen in Figure 6, Spark proved to be much slower than MASS clocking in at a minute of execution time at the slowest and 42 seconds at the fastest. This is likely due to two factors: (1) The training file size and (2) only classifying one point. If Spark and MASS were asked to classify more than 1 data point, the results would likely be reversed

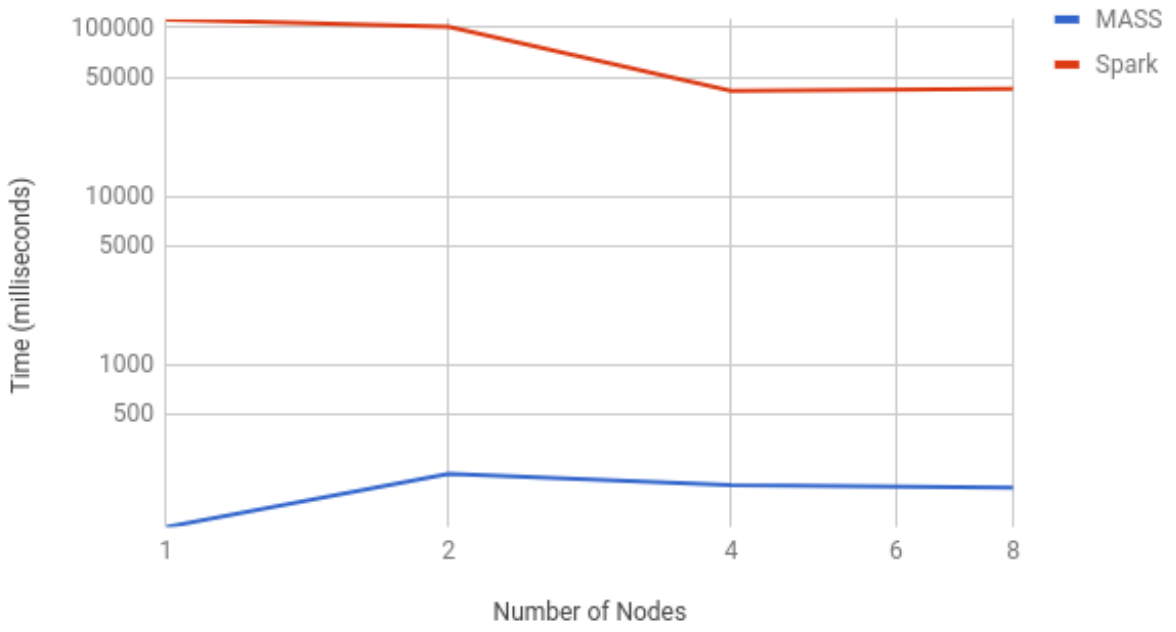

Figure 6: KNN Performance with MASS and Spark

as Spark does not have to account for the agent overhead as MASS does.

The training file size is also likely the case for MapReduce running slowly. The primary difference betweeen the MapReduce and Spark implementations is that in Spark, the training set is represented with an RDD which only stores information in memory that it is currently using. MapReduce, on the otherhand, attempts to store all 16,000,000 data points in a single ArrayList before moving to the classification of the point.

5.3 Triangle Counting Performance

Triangle Counting performance was measured using a graph consisting of 3,000 nodes. With this test case, both the MapReduce and Spark programs took longer than an hour to run on any node configuration. The probable cause is that the algorithms followed were not meant for use beyond proof of concept. The main issue for both algorithms occured in the graph reading stage.

For a graph of 3,000 nodes a file structure like the one in Section 3.3.2 means that for each of the 3,000 nodes, each connection would be on a separate line. Although the user has no direct visibility into the creation of map objects and RDD instances, it is known that map tasks are spawned for each line in a file and Spark instances work

similarly [5, 6]. This means that a large number of map tasks and instances are created that are all demanding resources at once. A better file structure would have the each node and a list of all the connections on one line so that map tasks and instances would be less numerous.

Since the algorithms from MapReduce and Spark did not work out, the performance will instead show the difference between the regular MASS and MASS with *doAll()*.

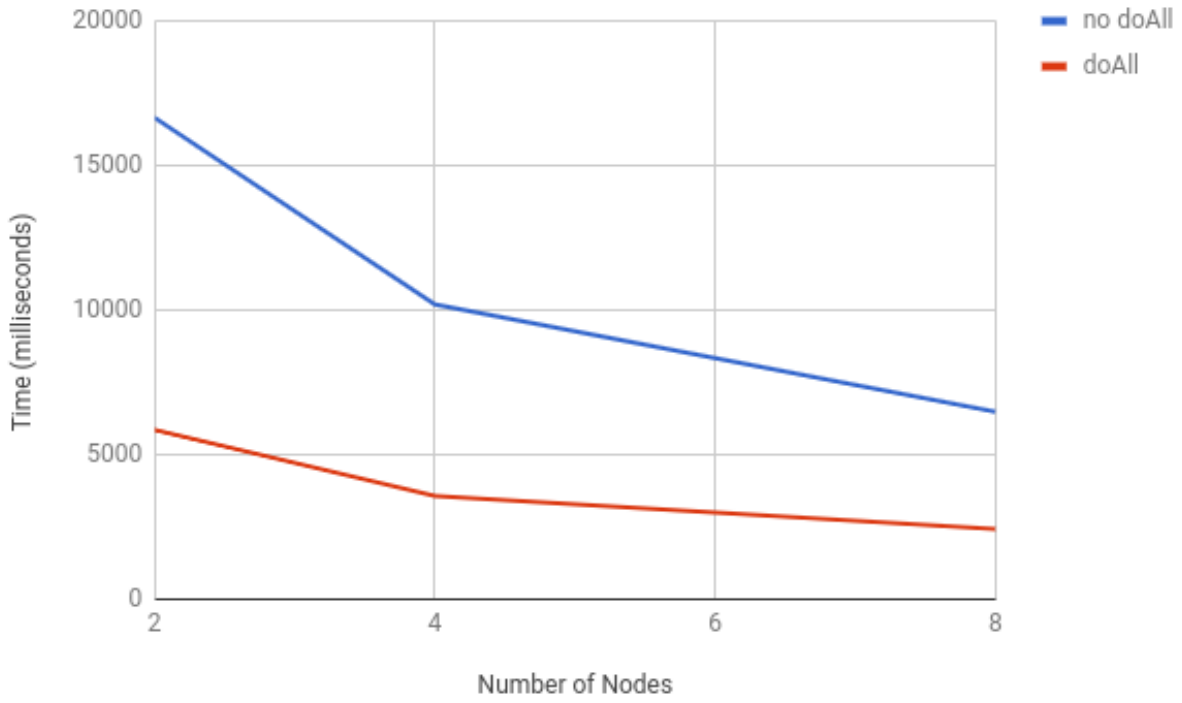

Figure 7: Performance difference using doAll

Figure 7 shows a significant performance improvement between using *doAll()* and *callAll()*. This is due to the way *doAll()* is structured. *doAll()* calls both *callAll()* to perform a function on the data in the agents and *manageAll()*, which is responsible for processing agent migration, spawning, and termination [17]. So, the Triangle Counting program and other dynamic agent alogrithms get twice the benefit from *doAll()* that K Means does. The agents in question migrate and process data completely avoiding the synchronization that occurs when control is passed back to the main program.

5.4 ATSP Performance

ATSP performance was measured over 17 and 48 cities. The number of ants that traversed the graph was equal to 80% of the number of cities meaning each run had 13 and 38 ants respectively [3]. The MASS program took abnormally long in comparision with the previously mentioned programs. Due to the long time, it was determined that one and two node execution were sufficient for comparison as increasing the nodes would have likely violated the 10 minute time limit.

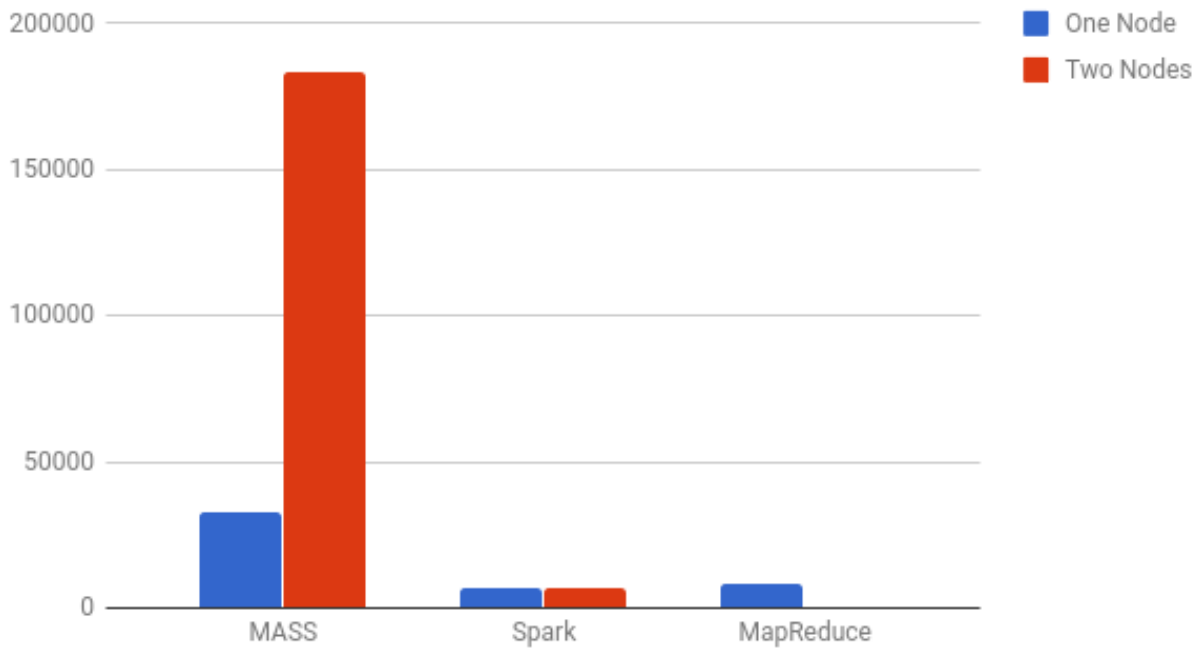

Figure 8: Performance with 17 cities

Figures 8 and 9 show the comparison between MapReduce, Spark, and MASS on one and two nodes. The broad difference in performance between MASS and the other frameworks is attributable the size of the data relative to the overhead incurred by moving agents.

We can see in Figure 9 that although the change appears slight on the graph, the three frameworks move closer together in performance numbers. In addition, combined with the evidence presented in Section 5.3, we can see that when the number of nodes in a graph is large, MASS outperforms both Spark and MapReduce. These results prove the supposition in Section 3.4.1 that MASS's agent overhead becomes acceptable as data

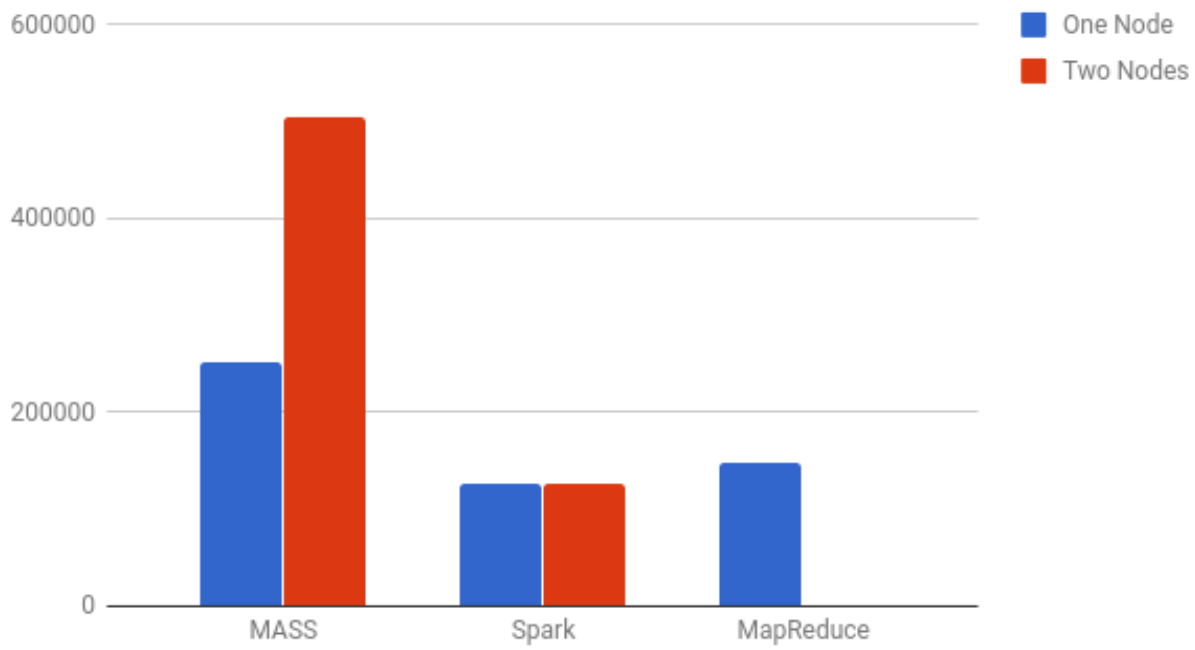

Figure 9: Performance with 48 cities

size increases.

6 Conclusion

The project was successful in proving that MASS is a good paradigm for machine learning and data science. In programmability, MASS was overall more efficient than MapReduce and Spark. Performance also proved that MASS implmentations performed significantly better than their MapReduce and Spark counterparts for K Means, K Nearest Neighbor, and Triangle Counting. While the results are a net positive, the research conducted to reach them has shown that there is still room for impovement to be made. Additionally, this research has suffered from some limitations that warant revisitation.

6.1 Future Work

Future work based on this project can take a variety of forms. As this project has drawn on past works in MASS, so can future projects draw on it in the following areas:

1. **Machine Learning algorithm exploration**: While K Means and KNN are two of the most common machine learning algorithms, there are numerous others to explore. Some examples are Gradient Descent and Naive-Baynes classification. Both of these algorithms have pitfalls that may allow for agents to be beneficial.

2. **Collision Free Migration exploration**: While not explicitly discussed, early versions of K Means attempted to use dynamic agents. The limiting force that led to the abandonment of dynamic agents was the lack of flexible collision free migration patterns in the MASS library. Research in this area may yield more efficient implementations of K Means.

3. **Agent Overhead**: As shown in the performance results of the algorithms that use dynamic agents, overhead is costly on the performance of the library. This is due to the size of agents inside MASS Java. Currently, agents are 1MB of memory each [17]. Future research in size reduction would make dynamic agents more palatable to a wider array of algorithms.

4. **Agent Communication**: One of the early approaches to an implementation of K Means on MASS involved agents traversing places and acting as centroids. This version was impossible to implement due to a lack of agent to agent communication. Research in this area would make such an algorithm possible and could open up possibilities other than the algorithm described in Section 3.1.1.

6.2 Limitations

6.2.1 MapReduce

One of the pitfalls of the information presented is the lack of sufficient MapReduce data. HDFS experiences several issues due to the specific cluster setup that UW Bothell uses.

The primary issue is the blocking of connectivity between nodes in the cluster. HDFS requires the use of YARN to connect and delegate resources on a cluster as of MapReduce 2.x. While YARN is a practical option in an industry or home cluster, the lack of memory available to the research account used in this project restricted the capability of YARN to properly distribute resources to the cluster. Therefore, it is likely that the performance results listed above are exclusively for a single node. In the future, research on MapReduce performance against MASS will best be measured on a cluster where the account being used can properly allocate resources. In addition, this project revealed that MapReduce may not be worth considering for future comparisons of MASS. Despite the configuration issues, this research has sufficiently presented evidence for an argument against MapReduce being used for further comparisions.

A major piece of evidence behind this argument is that in both programmability and performance for three of the four algorithms, MapReduce underperforms both MASS and Spark. The amount of boilerplate and classes alone prove that programmability-wise, MapReduce is more time consuming and difficult to use. Additionally, issues with comparability particularly in Section 5.1 led to having to extrapolate and guess at equivalent performance which does not equate to sound, provable research.

The other piece of evidence is the comparability in model, goal, and target audience between Spark and MASS. Both of these paradigms attempt, in different ways, to

alieviate the overhead of having to work with parallel computing in order to appeal to non-computer scientists [5].

6.2.2 Other Limitations

Another pitfall of this experiment are the algorithms that made up the test programs in MapReduce and Spark. It became apparent particularly in the instance of Triangle Counting that these algorithms may not be designed to be used in production since neither the MapReduce or Spark programs could handle a graph with more than 500 nodes. This also may be the cause of the file generators created to make test files. As MASS can run with most configurations due to its emphasis on spacial simulation rather than data streaming or data parsing, researching a mathematically correct formula for generating a realistic 1000+ node graph was not considered.

The final limitation in with the validity of K Means between MASS and Spark. No comparison was made on the accuracy of the cluster set reached by the algorithms. This was based on an assumption that accurate data would definitely be found in 5,000 runs of an algorithm. Future research can be done on the optimal number of runs to produce an accurate cluster set.

References

[1] S. Lukasik, P. A. Kowalski, M. Charytanowicz, and P. Kulczycki, "Data Clustering with Grasshopper Optimization Algorithm,"

[2] D. W. Van der Merwe and A. P. Engelbrecht, "Data clustering using particle swarm optimization," in *Evolutionary Computation, 2003. CEC'03. The 2003 Congress on*, vol. 1, pp. 215–220, IEEE, 2003.

[3] M. Dorigo, V. Maniezzo, and A. Colorni, "Ant system: optimization by a colony of cooperating agents," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 26, no. 1, pp. 29–41, 1996.

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

[5] H. Krau, A. Konwinski, P. Wendell, and M. Zaharia, *Learning Spark Lightning-Fast Data Analysis*. O'Reilly Media, Inc., 2015.

[6] T. White, *Hadoop The Definitive Guide*. O'Reilly Media, Inc., 2015.

[7] "Apache Storm."

[8] C. Gordon and M. Fukuda, "Analysis of agent-based parallelism for use in clustering and classification applications," *Advances in Practical Applications of Complex Multi-Agent Systems: The PAAMS Collection*, 2018.

[9] M. Fukuda, "MASS: A Parallelizing Library for Multi-Agent Spatial Simulation, http://depts.washington.edu/dslab/MASS."

[10] S. Saremi, S. Mirjalili, and A. Lewis, "Grasshopper Optimisation Algorithm: Theory and application," *Advances in Engineering Software*, vol. 105, pp. 30–47, Mar. 2017.

[11] R. Eberhart and J. Kennedy, "A new optimizer using particle swarm theory," in *Micro Machine and Human Science, 1995. MHS'95., Proceedings of the Sixth International Symposium on*, pp. 39–43, IEEE, 1995.

[12] C. Blum, "Ant colony optimization: Introduction and recent trends," *Physics of Life Reviews*, vol. 2, pp. 353–373, Dec. 2005.

[13] M. Gardner, "The fantastic combinations of john conway's new solitaire game "life"," *Scientific American*, vol. Vol.223, pp. 120–123, October 1970.

[14] J. Epstein and R. Axtell, *Growing artificial societies: social science from the bottom up*, p. 224. Brookings Institution Press, October 1996.

[15] C. Bowzer, B. Phan, K. Cohen, and M. Fukuda, "Collision-free agent migration in spatial simulation," in *In Proc. of 11th Joint Agent-oriented Workshops in Synergy (JAWS'17)*, (Prague, Czech), September 2017.

[16] T. Jungblut, "Thomas Jungblut's Blog: k-Means Clustering with MapReduce," May 2011.

[17] U. Mert, *Multi-Agent Spatial Simulation (MASS) Java Library Performance Improvement for Big Data Analysis*. PhD thesis, University of Washington, 2017.

[18] C. Bowzer, B. Phan, K. Cohen, and M. Fukuda, "Collision-Free Agent Migration in Spatial Simulation," *Proc. of the 11th International Workshop on Multi-Agent Systems and Simulation*, Sept. 2017.

[19] "MLlib | Apache Spark."

[20] B. Bahmani, B. Moseley, A. Vattani, R. Kumar, and S. Vassilvitskii, "Scalable k-means++," *Proceedings of the VLDB Endowment*, vol. 5, no. 7, pp. 622–633, 2012.