Nirali Gundecha

Enabling Lambda Expressions and Reductive Communication in Agent-based Data Analysis

Nirali Gundecha

A thesis submitted in partial fulfillment of the

requirements for the degree of

Master of Science in Computer Science & Software Engineering

University of Washington | Bothell

May 2022

Committee:

Prof. Munehiro Fukuda, Chair

Prof. Hazeline Asuncion, Committee member

Prof. Min Chen, Committee member

Prof. Erika Parson, Committee member

University of Washington | Bothell

Abstract

Enabling Lambda Expressions and Reductive Communication in Agent-based Data Analysis

Nirali Gundecha

Chair of the Supervisory Committee:

Prof. Munehiro Fukuda

Computing & Software Systems

The MASS is a parallelizing library that provides multi-agent and spatial simulation over a cluster of computing nodes. Agent-based modeling (ABM) views a simulation with an emergent collective group behavior of many agents. Primarily ABM is applied to big-data computing, focusing on the analysis of a structured dataset such as computational geometric problems and graph algorithms. While the Java version of the MASS library (MASS Java) runs with JShell, thus allowing users interactive computation, the users still must define their agent classes before dispatching them to their dataset. Since MASS retrieves all agent status and dataset values at once, users are responsible to reduce the final data to their desired value.

This paper introduces two new features, *Lambda* and *Reduction* methods, and implementation to achieve the goals. This feature is not implemented and provided in any agent-based library to date. Hence making this sole contribution to the agent-based library. This paper validates the *lambda* and *reduce* method and uses the MASS library to do so.

This project addresses these two problems by allowing a user to pass on-the-fly lambda expression to an agent when dispatching them to a dataset and by automatically reducing the final results into a user-desired value. The former enhances dynamic and interactive analysis with agents, whereas the latter reduces data-gathering overheads and programming complexity. The data collection method is described as a *lambda* method and using *reduce* method, the user can perform tasks of reduction in a single line of code. This improves code reliability and clean code. These features remove the hassle of writing blocks of code and getting involved in agent behavior over a cluster of nodes.

The goal of this capstone is to provide ease to the end-user, reduce the computational, and communication overhead for data and make the user experience effortless. Hence improving the efficiency of MASS. *Lambda and reduce* method implementation is a unique contribution to the agent-based library and its users.

TABLE OF CONTENTS

List of Figures

List of Tables

Chapter 1. Introduction

1.1 Background

The MASS (Multi-Agent Spatial Simulation) is a parallel agent-based model (ABMs) simulator and is applied widely for data discovery in a variety of scientific fields, such as bioinformatics, climate science, space cognition, and computational geometry. Compared to other parallel data analysis tools, such as MapReduce, Spark, and Storm, the MASS maintains the structure of data (e.g., array, tree, and graph) and has great advantages in analyzing scientific datasets with complex structures [1].

The MASS library performs parallel execution of Places and Agents, two classes in the library, using multi-thread communicating processes forked over cluster nodes through JSCH and connected via TCP sockets [1] [2]. The Places are simulation spaces with a multi-dimensional array that is distributed over a cluster system. Agents are entities with a set of mobile objects that can migrate over Places. Agents are the entities responsible to perform operations.

There are two versions of the MASS library: 1) MASS and 2) InMASS (Interactive computing feature for MASS library). The InMASS is running on JShell interactively, users can define and inject agents at run time. This interactive environment works in a traditional way of receiving and then re-iterating over results, considering data is huge, to get insights from that data.

1.2 InMass

Implemented Interactive computing feature for MASS library (InMASS). It is a wrapper around JShell Tool which allows MASS library APIs to be executed interactively. It benefits from both the MASS communication model and JShell Tool functionalities. First, it injects MASS initialization code into JShell. Next, a JShell prompt is returned where the user can interact with MASS components: MASS, Places, Agents, Place, and Agent classes. Finally, cleanup is automatically called on an exit event. It uses MASS communication functionalities to disseminate any class that is defined by the user at runtime.

1.3 Motivation

The MASS library provides an intuitive programming model. However, its rigidity and scope of adding automated features that are available in other tools such as Spark, and MapReduce can draw users away from using the library. Currently, to use the library effectively, users need to know how the existing version of the library works and its dependencies. Due to this overhead, the user gets distracted from their use case to gain an understanding of the library. This overhead is time-consuming and must be unnecessary. Also, whenever a user decides to use any third-party library, it should reduce the amount of unnecessary work that saves development time and cost.

Introducing *lambda* and *reduction* methods, helps users focus on their use cases. MASS is primarily used by data scientists who prefer to code less and analyze results more requiring on-the-fly usage of methods. Having *lambda* and *reduce* method implementation for InMASS leads to faster, easy implementation to check new functions at run time and makes a MASS compelling choice for users.

1.4 GOAL

This paper's goal is to ramp up the library by introducing MASS with supporting features as follows :

1. Implement the Lambda expression in MASS to allow interactive and dynamic agent execution
2. Implement the Reduction method in MASS to simplify collective communication from agents to a user
3. Verification of *lambda* and *reduce* method by implementing triangle counting application in InMASS

These features should be supported by JShell, so that novice users can use these features hassle-free. This paper's work is primarily focused on the implementation of a *lambda* expression that will lead to agents carrying a *lambda* expression and executing it at each place it visits. Using *lambda* expressions on places to perform operations will result in a reduced amount of LoC, clean coding, and better understandability for readers. On top of that, for every single code change, users do not need to follow the process of build and deployment which is not only time-consuming but also repeated. Using *lambda* method support, *reduce* method should distribute work amongst all the places and agents will be responsible for execution. Also, *reduction* method implementation will increase the programmability and reduce the data overhead while filtering data for particular use cases.

This paper verifies these features in the "Triangle counting application" by creating #agents and #places to calculate distance and finding the shortest path of a triangle. This application can test any agent parameter, or any data user wants to test their application

on. As a final goal, these new features will help a user focus more on their use cases. Also, these features are exclusive to an agent-based library and tested on MASS.

The rest of the paper is as follows. Chapter 2, describes Related work done for identifying a need and comparing existing *lambda* and *reduction* methods to replicate similar features in InMASS. In chapter 3, the Architecture diagram and Implementation details for *lambda* and *reduce* method are explained. Chapter 4, Results, covers the demonstration for running *lambda* and *reduction* method in InMASS with the single and multi-node environment. Also, the triangle counting application showcases how it can be used with other applications to remove the hassle of writing lengthy code and how it provides an ease to a user while using these new features.

Chapter 2. RELATED WORK

Interactive MASS (InMASS) was developed during Nasser Alghamdi's graduate work which introduced line-by-line execution of a MASS simulation through the integration of Java's JShell Tool [3]. This work enabled a command-line interface for working with MASS and has fundamentally changed the way users can interact with the MASS Java library because it also allows the user to checkpoint a running application, test new code, and rollback to the checkpointed state without the need to halt execution and then recompile, distribute, and reinitialize the program on the cluster [2]. After that, another student, Daniel Blashaw reengineered InMASS for the practical use – forward and backward computation. These additions significantly reduce the overhead that comes along with working on a distributed system and encourage novice programmers to get into the code, identify mistakes easily, and rapidly adjust. However, there is no lambda implementation in MASS.

The following figure 1 describes the use case before *lambda*, for writing a simple piece of code that novice user needs to go through and for every single and small change in code to test and identify the correctness of the use case.

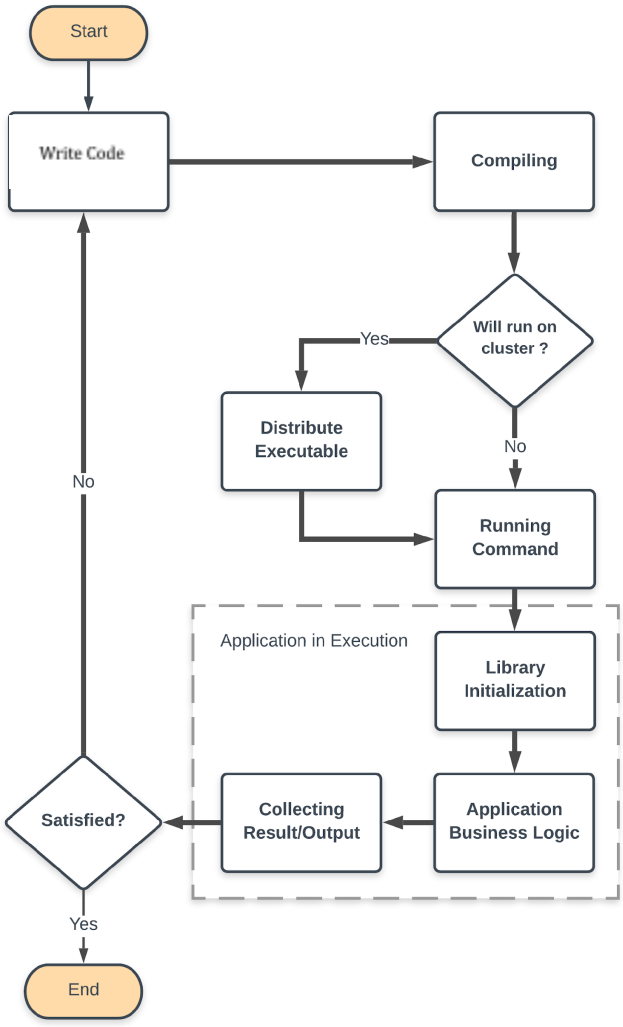

Figure 1. Code flow without Lambda [3]

This flow is not only tedious to follow for every little change but also burdens the user with having to understand the existing code base for MASS. Looking at the users of InMASS, Data scientists, or analysts, this requirement becomes a barrier to using MASS freely because without spending much time to gain an understanding of the library user can't apply their use cases.

This paper describes how easy it becomes with the use of *lambda* and *reduction* method support in the following flowchart figure 2. It dictates user flow for checking any new feature implementation over Agent base library, in this case, MASS.

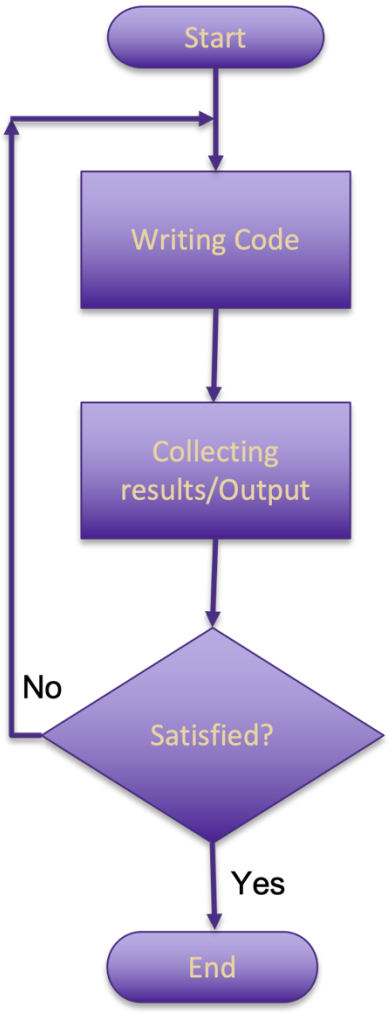

Figure 2. Code flow with Lambda

2.1 Need of Lambda and reduce methods

Lambda expressions are a new and important feature included in Java SE 8 [4]. They provide a clear and concise way to represent one method interface using an expression. Lambda expressions also improve the Collection libraries making it easier to iterate

through, filter and extract data from a Collection. In addition, new concurrency features improve performance in multicore environments [4]. Currently, there is no such feature included in any multi-agent simulation library and also missing in InMASS. The inclusion of the l*ambda* method will be a sole feature that is a must to update the library and bring to industry-level expectations of users.

Reduction is the process of performing a given operation on data items to obtain a single output. Data can be in the form of an array, list, or file provided to *the lambda* method which states the operation user wants to perform on that data. This operation could be finding min, max or taking addition, multiplication of data and reducing it to a single output. This feature is also missing from the MASS library. The inclusion of this feature will bring the library to a competitive level with Spark and Map-Reduce which already provide reduction features for sequential and parallel execution.

2.2 LAMBDA

The *lambda* expressions address the bulkiness of anonymous inner classes by converting multiple lines of code into a single statement. This simple horizontal solution solves the "vertical problem" presented by inner classes. A *lambda* expression is composed of three parts as shown in figure 3.

Figure 3. Lambda Expression Syntax

The body can be either a single expression or a statement block. In the expression form, the body is simply evaluated and returned. In the block form, the body is evaluated like a method body and a return statement returns control to the caller of the anonymous method.

```
//Without using Lambda Expression


int multiply(int a, int b){
   int ans = a * b;
   return ans;
}


int add(int a, int b){
   int ans = a + b;
   return ans;
}
```

```
//With Lambda Expression


interface FuncInter1 {
      int operation(int a, int b);
}


FunInter1 multiply = (int x, int y) -> x*y;


FunctInter1 addition = (int x, int y) -> x+y;
```

Listing 1. Computation without lambda and with lambda

As we compared code from Listing 1. Computation without lambda and with *lambda*, it is easier to write single line expression from JShell in InMASS, instead of writing blocks of code into the main library i.e., MASS_JAVA_CORE or MASS_JAVA_APPLICATION.

Benefits of adding *lambda* into the MASS core:

1. Fewer lines of Code

2. Sequential as well as Parallel execution of code support by passing behavior as arguments to the method
3. Higher efficiency
4. Readability.
5. Less boilerplate code
6. Encouragement of functional programming

This feature is important to users like data scientists and analysts who prefer to work on computational results more than on code. *Lambda* will make changing parameters and check results instantaneously without worrying about the mass core library implementation.

2.3 REDUCE

The reduction is the process of producing result as a single value by performing specific operation on provided data set. The data provided as an input and data after operation is performed will be of same type as we need to perform same operation on resultant data continuously.

RDDreduceExample.java

```
import java.util.Arrays;

import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;

public class RDDreduceExample {

    public static void main(String[] args) {
        // configure spark
        SparkConf sparkConf = new SparkConf().setAppName("Read Text to RDD")
                                        .setMaster("local[2]").set("spark.executor.memory",
        // start a spark context
        JavaSparkContext sc = new JavaSparkContext(sparkConf);

        // read text file to RDD
        JavaRDD<Integer> numbers = sc.parallelize(Arrays.asList(14,21,88,99,455));

        // aggregate numbers using addition operator
        int sum = numbers.reduce((a,b)->a+b);

        System.out.println("Sum of numbers is : "+sum);
    }

}
```

Listing 2.Spark Reduce example [5]

Run the above Spark Java application, and you would get the following output in console.

```
17/11/29 11:26:42 INFO DAGScheduler: ResultStage 0 (reduce at RDDreduceExample.java:20) fin
17/11/29 11:26:43 INFO DAGScheduler: Job 0 finished: reduce at RDDreduceExample.java:20, to
Sum of numbers is : 677
17/11/29 11:26:43 INFO SparkContext: Invoking stop() from shutdown hook
```

Listing 3. Spark Reduce Output [5]

2.4 SPARK REDUCE

In Spark, at a high level, every application consists of a driver program that runs the user's main function and executes various parallel operations on a cluster. The main abstraction Spark provides is a resilient distributed dataset (RDD), which is a collection of

elements partitioned across the nodes of the cluster that can be operated on in parallel. Spark provides the *reduce*() which is responsible for aggregating the elements of the dataset using a function (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel. Also, the important thing to keep in mind is, a reduce operation will return the same type of data it took input. If input data to *reduce*() is of type int, it will return the reduced data also if type int.

The MASS library uses JAVA as the core language. In Java, reducing is a terminal operation that aggregates a stream into a type or a primitive type. Java 8 provides Stream API contains set of predefined reduction operations such as average(), sum(), min(), max(), and count(). These operations return a value by combining the elements of a stream. In Java, *reduce*() is a method of the Stream interface. It is the method of combining all elements. For each element presented in the stream, the *reduce*() method applies the binary operator. In that stream, the first argument to the operator must return the value of the previous application and the second argument must return the current stream element [6]. It allows the user to produce a single result from a sequence of elements by repeatedly applying a combining operation to the elements in the sequence called reducing.

2.5 MapReduce

MapReduce is a processing technique and a program model for distributed computing based on java. The MapReduce algorithm contains two important tasks, Map and Reduce. The map takes a set of data and converts it into tuples (key/value pairs).

Secondly, reduce task, which takes the output from a map as an input and combines those data tuples into a smaller set of tuples until a single result is obtained.

The primary MapReduce abstraction is parallelizable, easy to understand, and hides the details of distributed computing, thus allowing Hadoop to guarantee correctness. However, to achieve coordination and fault tolerance, the MapReduce model uses a pull execution model that requires intermediate writes of data back to HDFS. Unfortunately, the input/output (I/O) of moving data from where it's stored to where it needs to be computed is the largest time cost in any computing system. As a result, while MapReduce is safe and resilient, it is also necessarily slow on a per-task execution. This results in huge amounts of intermediate data written to HDFS that is not required by the user, creating additional costs of disk usage.

Also, Hadoop became more widely adopted, more specializations were required for a wider variety of new use cases, and it became clear that the batch processing model of MapReduce was not well suited to common workflows on-demand computations.

```
public class takeSum {
 public static class TokenizerMapper extends Mapper<Object, Text, Text, IntWritable>{
        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();
        public void map(Object key, Text value, Context context) throws IOException,
InterruptedException {
        StringTokenizer itr = new StringTokenizer(value.toString());
        while (itr.hasMoreTokens()) {
```

```
        word.set(itr.nextToken());
        context.write(word, num);
         }
        }
 }
```

Listing 4. Mapper code

```
public static class IntSumReducer extends Reducer<Text,IntWritable,Text,IntWritable> {
        private IntWritable result = new IntWritable();
        public void reduce(Text key, Iterable<IntWritable> values, Context context) throws
IOException, InterruptedException {
        int sum = 0;
        for (IntWritable val : values) {
        sum += val.get();
        }
        result.set(sum);
        context.write(key, result);
        }
 }
```

Listing 5. Reducer code

```
public static void main(String[] args) throws Exception {
```

```
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "Performing Sum");
        job.setJarByClass(takeSum.class);
        job.setMapperClass(TokenizerMapper.class);
        job.setCombinerClass(IntSumReducer.class);
        job.setReducerClass(IntSumReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        System.exit(job.waitForCompletion(true) ? 0 : 1);
 }
}
```

Listing 6. Main method to call Mapper and reducer

```
Function<Agent, Object> method = (a) -> ((LambdaAgent) a).getAgentId();
BinaryOperator<Object> reduce = (a, b) -> (Integer) a + (Integer) b;
agents.reduceAgents(method, reduce);
```

Listing 7. MASS with lambda and reduce for performing Sum operation

To understand the shift, consider the limitations of MapReduce concerning iterative algorithms. These types of algorithms apply the same operation many times to blocks of data until they reach the desired result. For example, finding a sum of agent ids. To implement this using MASS *lambda* and *reduce* method as shown in listing 7, is simpler than in MapReduce shown in Listings 4-6. In MASS we keep track of a sum amongst the

agents working on a particular node until we find a single value as a result. MapReduce works on key-value pair. We need to provide data by collecting all the agent ids into one file and provide that file as input to the mapper and reducer from main(). First the mapper class maps the data and in reduce operation we reduce data to single output. MapReduce distributes data to multiple nodes by creating smaller pieces of datasets. Also, as we can see in the Listings 4-6 it is less intuitive to write without first understanding the syntax and how it works. Implementing this in MapReduce results in disadvantages as we need to create new job for every iteration until we receive the result. This 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 is not only taxing to the programmer to write but involves the creation of new objects or accesses to underlying class variables. So, this brings an anomaly to compare MASS with lambda and reduce to be compared with MapReduce.

2.6 SUMMARY

Each of the technologies and approaches discussed in this section provides an overview of the goals set out in this research paper. Former students tried to provide the *lambda* feature, but their implementation was too intractable, difficult to understand and use, for novice users. Also, *lambda* methods are too popular and provide ease to a programmer. Lambda methods provide a clear and concise way to represent method interface using an expression. It is useful in the collection library. It helps to iterate, filter, and extract data from a collection.

Also, unlike MapReduce, Spark keeps the dataset in memory as much as possible throughout the course of the application, preventing the reloading of data between iterations. Spark programmers, therefore, do not simply specify map and reduce steps, but rather an entire series of data flow transformations to be applied to the input data before performing some action that requires coordination like a reduction. Because data flows can be described using directed acyclic graphs (DAGs), Spark's execution engine knows ahead of time how to distribute the computation across the cluster and manages the details of the computation. This leads to MASS being similar to Spark and differentiates MapReduce.

As a comparative study, we can compare Spark and MASS. Spark and MASS has common features providing similar comparison parameters as shown in following table 1.

	Spark	MASS
Languages Support	Python, Scala and Java	Java
Interactive shell	PySpark	JShell
Computation across the cluster of nodes	Provides	Provides
In-memory computation	Provides	Provides
Mode of execution	Single	Single and Parallel

Table 1. Similarities Study

To address concerns, this paper shows the successful implementation of *lambda* method. This paper also shows how to use *lambda* with different kinds of examples and use cases and how it provides use to the user. Performance improvement was not primary goal

of this paper, but my implementation shows that performance is also improved for big data sets and reduced the computational and data overhead.

PySpark works well for single execution. However, to achieve parallelism the user requires modifying the codebase which involves importing multiprocessing modules from python and adapting the existing execution as a function to be used in parallel. On contrary, MASS achieves parallelism by adopting configuration as code policy wherein the user simply configures the nodes.xml mentioning nodes with metadata about it with no change in the existing codebase. Thus, the MASS eases the parallel execution by reducing extra coding allowing the user to easily test for multiple nodes thus simplifying the overall user experience for data analysis.

Furthermore, in section 2.3, this paper introduces, importance and feature provided by matured libraries. To enhance MASS and bring to competing level, this paper shows successful implementation of reduction method. In Chapter 4, paper shows the usability and proficiency provided by using reduction method. This paper aims for providing flexibility and ease of use to novice codes and users to concentrate on their work.

Chapter 3. System architecture & implementation details

In this section of the paper, first take look at MASS library methods and internal implementation details. After, implementation details of the *lambda* and *reduction* method is observed in expound. Every new implemented class, fields, program flow and strategy of implementation is narrated in the detail and conclude with its benefits.

3.1 MASS Library

In this research project MASS library is used. Following Figure 4. Provides an overview of the MASS Library Model. In this model, Agent Based Modeling (ABMs) can be represented using the two available modeling objects: Places and Agents. The Places represent the simulation space, while the Agents are the actors within the simulation.

- As shown in the application layer of Figure 4, Places are distributed among the computing nodes on the cluster, and each is mapped using a globally known set of coordinates which are used to reference or locate the Place. Also, Places is a multidimensional distributed array. Each element is called a place, is pointed to by a network-independent array index and is capable of exchanging information with any other places.
- Agents are a set of execution instances that can migrate from one place to another and interact with other agents via variables of the place they currently reside.

MASS performs parallel execution of Places and Agents using multithreaded communicating processes forked over computing nodes through JSCH and connected via TCP sockets. Multi-threads take charge of parallel method invocation and information

exchange among places and agents. Figure 4 illustrates a two-dimensional mapping of Places with x and y coordinates. Regardless of mapping structure, all Places are mapped to processes and may exchange data amongst themselves or with visiting Agents. Agents are then stored in bags on each process and may traverse the cluster to visit Places throughout program execution. When traversing, Agent data is serialized and passed between cluster nodes via TCP communication [7].

The main() function behaves as a simulation scenario that instantiates Places, populates Agents, and describes parallel algorithms using their built-in methods. Places include: callAll() to concurrently invoke each place's function specified with function id; Agents has callAll() to invoke each agent's function in parallel; and manageAll() to commit agent spawning, terminating, and moving operations. Looking at each place and agent object, a place holds Its logical index, sets a list of neighbors to communicate with through exchangeAll(), and implements user-defined functions. On the other hand, an agent with a unique ID is equipped with migrate(int []) to migrate to a place specified with int[], spawn() to spawn children, and kill() to terminate itself, all committed by the Agents.manageAll() call.

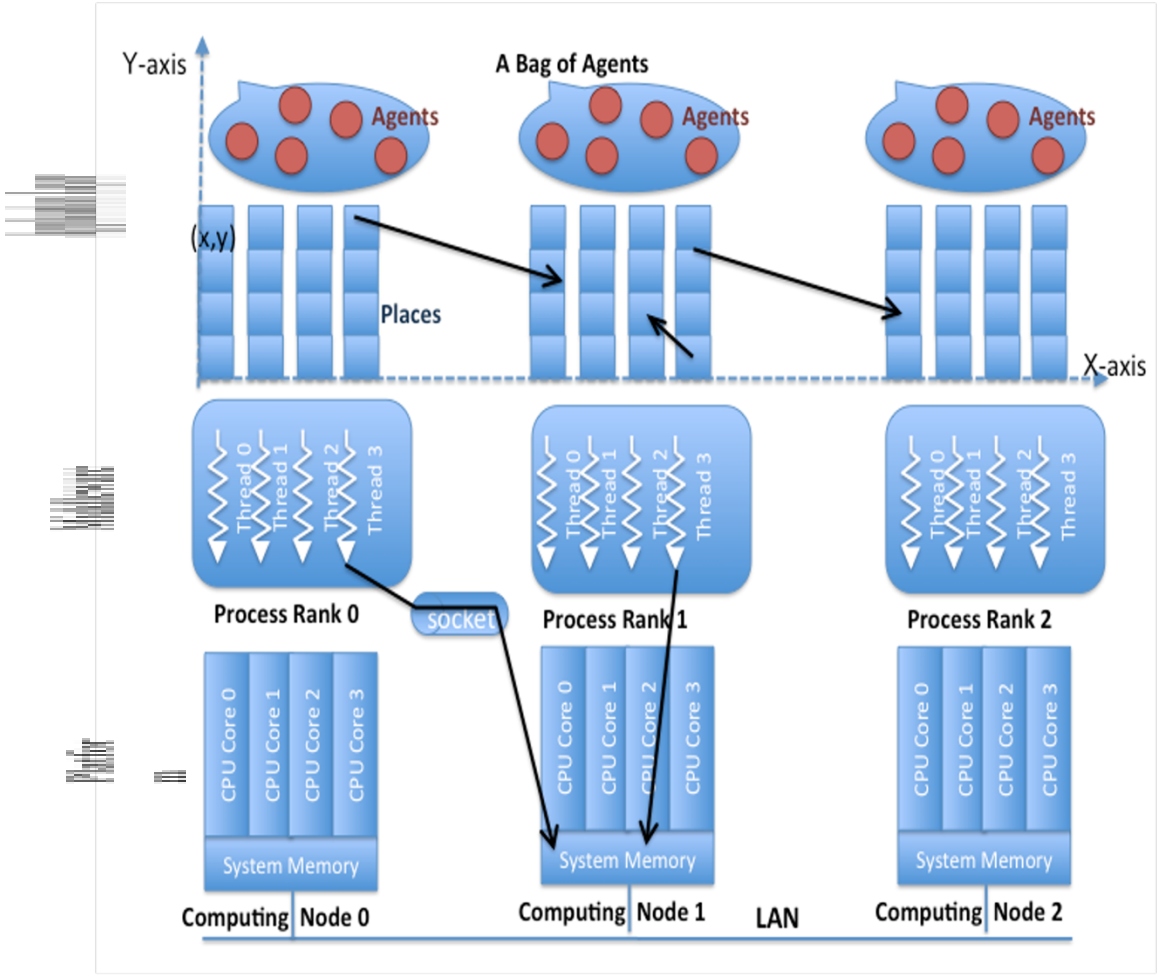

Figure 4. MASS Library Model [2]

3.2 Architecture Design for Lambda

The focus of *lambda* implementation is to make coding easier for novice users of the MASS library. Using *lambda*, users can add, check, and evaluate their features without altering the MASS core or application code. Due to this feature, the hazel of code build and execution of MASS can be avoided. Also, for every single implementation change, users need to follow the same process repeatedly which is not only time consuming but also burdensome.

The very first step to using *lambda* methods will be to start the InMASS and Jshell environment. Jshell should be running on the primary node i.e. Node 0. To execute *lambda* method, agents and places are required. So, first step first, we need to create agents and places. These agents are responsible to execute *lambda* operation at different places by each agent. We will require a cluster of nodes for multi-node execution.

Consider the execution of the *lambda* method on a single node for calculating the sum of two integer numbers. Primary node will create a *lambda* method as an object that is responsible to perform sum operations on integers provided as parameters. This *lambda* will return the object[] with a sum of integers to the primary node on Jshell.

For multi-node execution, the primary node will create agents and places. Agents are distributed to all other nodes randomly. From the primary node lambda method is broadcasted to all the other nodes and agents on each place are responsible to run *lambda* and return results on respective nodes as explained in Figure 5, Lambda Architecture for MASS. Using this feature, it becomes coherent and easy to review execution instantaneously.

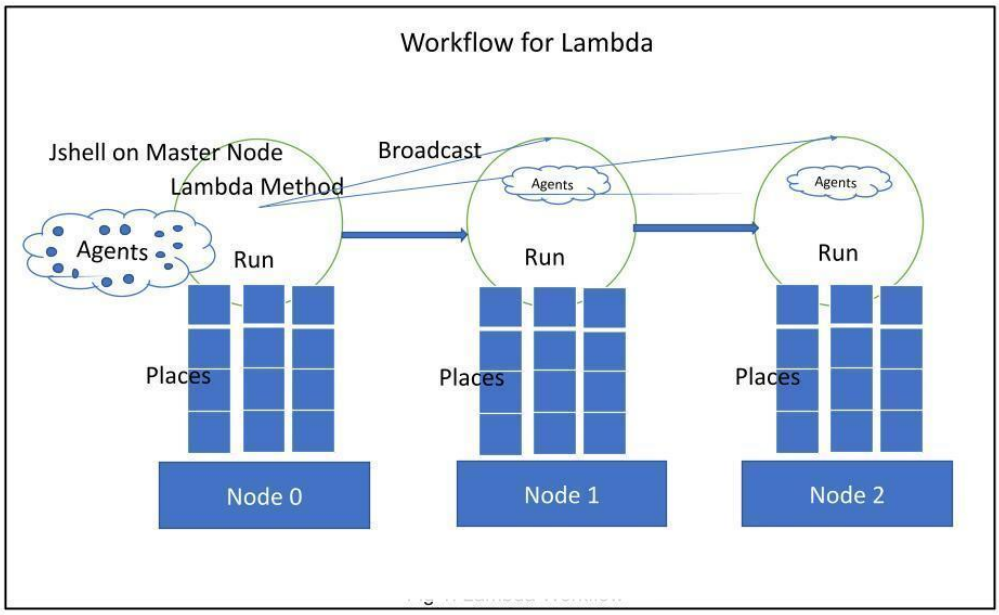

Figure 5. Lambda Model for MASS

3.3 IMPLEMENTATION DETAILS FOR LAMBDA METHOD

After invoking InMASS and starting Jshell platform, user is ready to start their work. Following block diagram, figure 6, denotes the flow when *lambda* is called from Jshell. In order to use *lambda*, first user creates agents and places. When agents are created, they are distributed among the other places randomly, meaning it is not necessary for places to receive same number of agents. After distributing agents, user provided *lambda* method which describes the operation to perform on each data item is broadcasted by Agents. Each agent receives this *lambda* method. The callMethod(functionId, argument[]) containing two primary node. This provided *lambda* method is serialized and provided to each computing node. The receiving computing node is responsible to deserialize it.

Consider a situation where user want to return the value form the map. To obtain desired result, user can send lambda method describing to get value from map. As we know,

it is easier to add function, call function to library and build, compile it to perform action. However, using lambdas, the problem is by default they are not serializable.

Serialization is a process for writing the state of an object into a byte stream so that we can transfer it over the network. We can serialize a *lambda* expression if its target type and its captured arguments have serialized. As the *lambda* function is not serialized by default we simply have to cast our *lambda* to java.io.Serializable. But the submit method requires a Runnable or Callable as the parameter, not a Serializable. So, we have to cast the *lambda* to two interfaces at the same time Runnable and Serializable.Java Serialization is a good generalized, backwardly compatible serialization library. These complexities were successfully implemented to provide feature of lambda to the MASS library from interactive JShell platform.

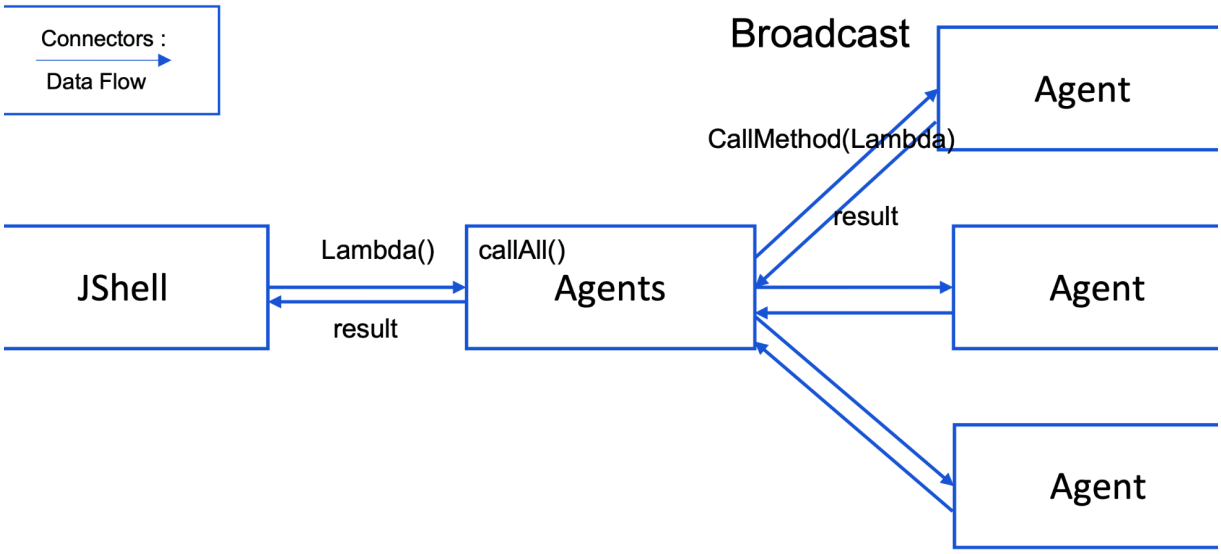

Figure 6. Block Diagram for Lambda

The implementation of *lambda* support functionality in MASS has resulted in the addition of new class and interface to the Java MASS library: LambdaAgent.java and LambdaInterface. Implemented *lambda* class extends Agents in MASS. Figure 7 shows a class diagram of these new classes as well as all associated new or updated methods and variables on existing classes. The arrows define the date flow with agents.

User creates a *lambda* method form Jshell and prepares input data, argument[], containing 1) *lambda* method and 2) data items to perform *lambda* operation on. With this data, newly created, callAll() is called with function id '-1' and argument[] from agents.java. This function id lets program understand to call *lambda* method. Depending on the size of the argument[], appropriate implementation of *lambda* method is called from *LambdaInterface.*

To access the LambdaInterface, the interface must be "implemented" by another class. This means that, implementation of method can be described differently for different object types. The body of the interface method is provided by the "implement" class such as apply() method described by user as shown in listing 1 and listing 2 "method". This is dynamic implementation of method as per user requirement. In the code shown in listing 1 describes the addition operation on agent ids. As shown in listing 4, the argument[] size is 1, when it only contains the *lambda* method. So this *lambda* method can be anything about agents such as getAgentId(), getParentId() or getPlace(). So this gives liberty to write single line of code to access any information about agents. For different applications this use feature is extremely useful (see Chapter 4). Now, as shown in listing 5, if the argument[] is of size 2, it contains *lambda* method and data items to perform *lambda* operation on. Each agent receive the data and *lambda* method, performs the operation and returns the result to

main() at primary node. This flexibility of adding user defined data gives user ability to run program on different kinds of data items.

```
LambdaInterface method = (a) -> ((LambdaAgent) a).getAgentId();
callArgs[0] = (Object) new Object[]{method};
```

Listing 4. One Arguments pass

```
LambdaInterface method = (a) -> {int[] input = (int[]) a; return input[0] + input[1];};
callArgs[0] = (Object) new Object[]{method, new int[]{5, 8}};
```

Listing 5. Two Arguments pass

Following is the description of the components used to implement *lambda* method:

- MASS base: Contains all the shared properties that are used by other classes. It also keeps track of the socket communication among the nodes.
- MASS is a subclass of the MASS base with other properties that are used by the master node only.
- Agents base: Encapsulates variables and logic related to different agent collections such as callAll(), manageAll(). CallAll() calls the user-defined overloaded method where depending on the parameters passed to the function it works in parallel among multi-processes/threads.ManageAll() is responsible to update each agents status
- Agent is a subclass of AgentsBase on other properties and methods that are only executed by the master node.

Mthread : Mthread is responsible to manage all the agents in parallel environment.

Message : Keeps track of all the message types defined at run time, created new or already in use. User can create new message with specified ACTION_TYPE.

Table 2. Introduction of MASS classes

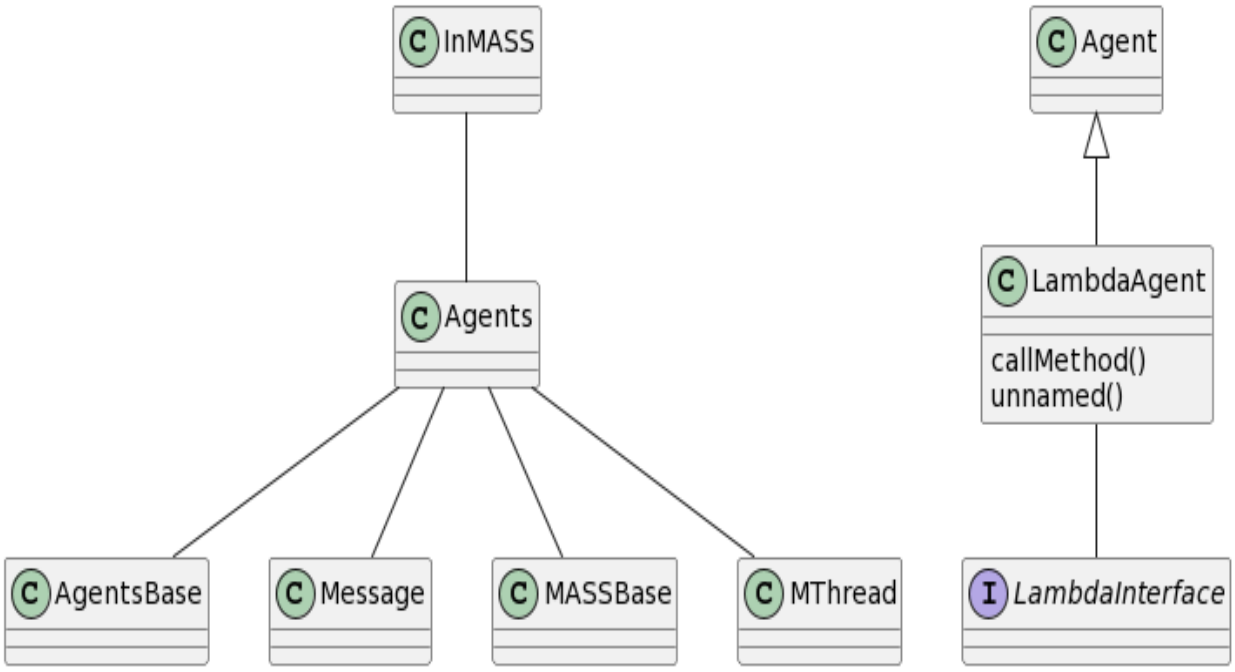

Figure 7. Lambda Class Diagram

Due to the complexity of the MASS library, figure 7 only shows only the classes affected by the addition of *lambda* support and new introduced methods.

In order to complete the execution of the *lambda* method, Agents talk to AgentsBase.java, Messge.java, MASSBase.java and Mthread.java. As described in Table 1, each class is responsible to complete their tasks to run *lambda* successfully.

3.4 Architecture Design for Reduce

On top of *lambda*, this research introduces the new feature to the MASS library: *Reduce* method Implementation. Unlike in Spark, Hadoop, user can provide their data and perform *reduction* on them using *lambda* method of their choice such as sum, max or min. As this feature was missing from MASS, User needs to write block of code and understand existing code base.

Introduced *reduce* method is designed to work similarly to Spark's reduce method. From Jshell, pool of agents is created and distributed to places randomly. As shown in figure 8, the user will provide two main parameters: 1) the *lambda* method describing the data it wants to work on and 2) The *reduce* method to perform desired operation and produce single output. For example, the user can say, work on agentIds and perform 'find min' operation on them. As a result, implemented *reduction* feature will provide the final reduced answer to the user on JShell.

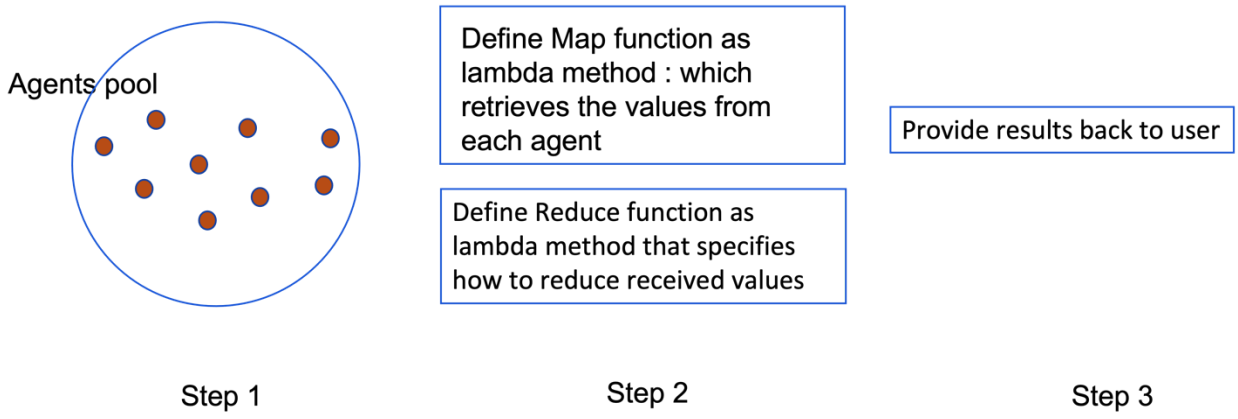

Figure 8. Reduction Method Design

3.5 Implementation details for reduction method

The *reduce* is an operation that aggregates values two by two, by using as many steps as necessary to process all the elements of the collection. This is what allows the frameworks to perform aggregations in parallel, on multiple nodes when needed. The framework will choose two elements and pass them to a defined reduce function. The function must return the new element that will replace the two first. This means that the output type must be identical to the input type, and values must be homogeneous so that the operation can be repeated until all elements are processed.

The implementation of Agent Tracking functionality in MASS has resulted in the addition of a new method to the MASS Java library in AgentsBase.java. Figure 9 shows a class diagram of these new classes as well as all associated new or updated methods and variables on existing classes.

To perform a reduction operation, a user first creates a bag of agents and places. These agents are distributed randomly over the places. From the Jshell platform user will create two *lambda* methods describing 1) mapping method: What data to operate on and 2) reducing method: how user wants to reduce data. Each node will compute the results and send them back to a primary node for final computation. This implementation offloads computational overhead from the primary node.

As shown in figure 9, reduce class diagram, Agents are created from the Jshell platform. AgentsBase.java contains a list of all the agents and data related to agents such as agents id, parent id, and place id. User-defined *lambda* methods to map and reduce data are simultaneously provided to pair of agents. The first *lambda* method, the mapping method,

is an implementation of functional programming introduced in java8 [8]. The function interface is part of the java.util.function package. It represents a function which takes in one argument and produces a result. Hence this functional interface takes in 2 generics namely as follows:

- **T**: denotes the type of the input argument
- **R**: denotes the return type of the function [8]

The function interface consist of apply() method. For reduction method implementation this apply() is *lambda* method describing the data items. This apply() take in only one parameter ***t*** which is the function argument and return the function result of type ***r***. The *lambda* expression assigned to an object of Function type is used to define its apply() which eventually applies the given function on the argument [8].

The second *lambda* method, *reduction* method, is BinaryOperator. This BinaryOperator is also a part of java.util.function package. It represents a binary operator which takes two operands and operates on them to produce a result. However, what distinguishes it from a normal BiFunciton is that both of its arguments and its return type are same [9].

This functional interface which takes in one generic namely:

- **T**: denotes the type of the input arguments and the return value of the operation

The BinaryOperator<T> extends the BiFunction<T, T, T> type. So it inherits the apply() methods from the BiFunction Interface. The lambda expression assigned to an object of BiaryOperator type is used to define its apply() which eventually applies the given operation on its arguments [9].

Also, AgentsBase extends IRef<AgentsInternal>. The IRef is a generic abstract class for making any class self-referenced. The purpose is that if an object needs to be updated without changing already existing references, then this class can be used as a superclass to achieve such requirement.

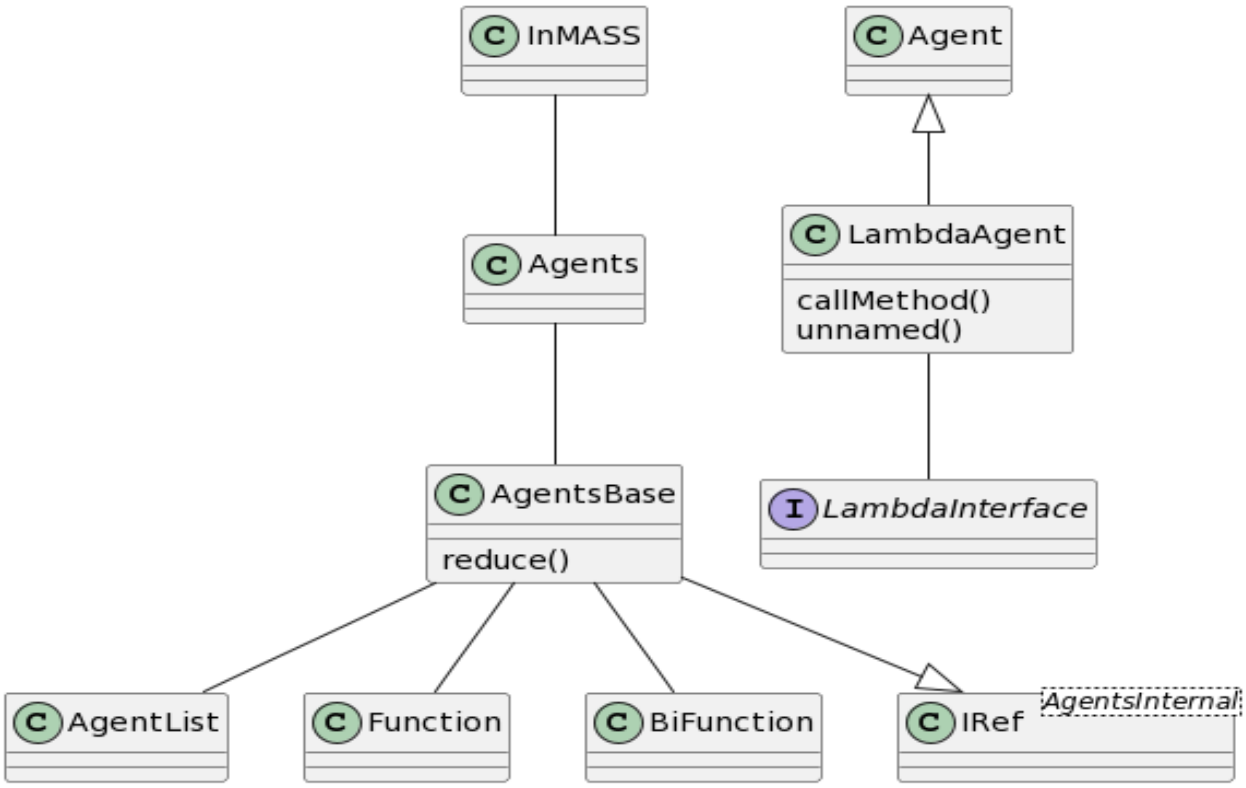

Figure 9. Reduce Method Class Diagram

As shown in figure 10, 8 Agents are created from the JShell platform. AgentsBase.java contains a list of all the agents and data related to agents such as agents id, parent id, and place id. A user defines two *lambda* methods. First to map, type Function, to collect agent ids, and second to reduce, type BiFunction, to find a minimum of id. These methods are simultaneously provided to pair of agents as shown in figure 10. These pairs will first gather the agent ids described in the map function and perform a reduction

operation of finding minimum on them. It results in temporary resultant values as shown in figure 11. These temporary values are of the same type as the originally provided data. This process continues till each place finally has only single resultant data. This single resultant value is sent back to the primary node by each remaining node as shown in figure 11. As a result, the same process of reduction is applied to creating pairs and creating resultant values until the single output is calculated. If an even number of agents remain, it is straightforward to create pairs of agents. However, if an odd number of agents are remained or created at the start, the proposed implementation creates all possible pairs of agents and the remaining single-agent receives the single lambda map method to collect the data and return it as it is. This data is considered for the next round of computation. So main() will contain at max data of the node size. Finally, as a result, the single output is returned to JShell for user visibility. This programming style makes it easier for a user to code and evaluate results at run time.

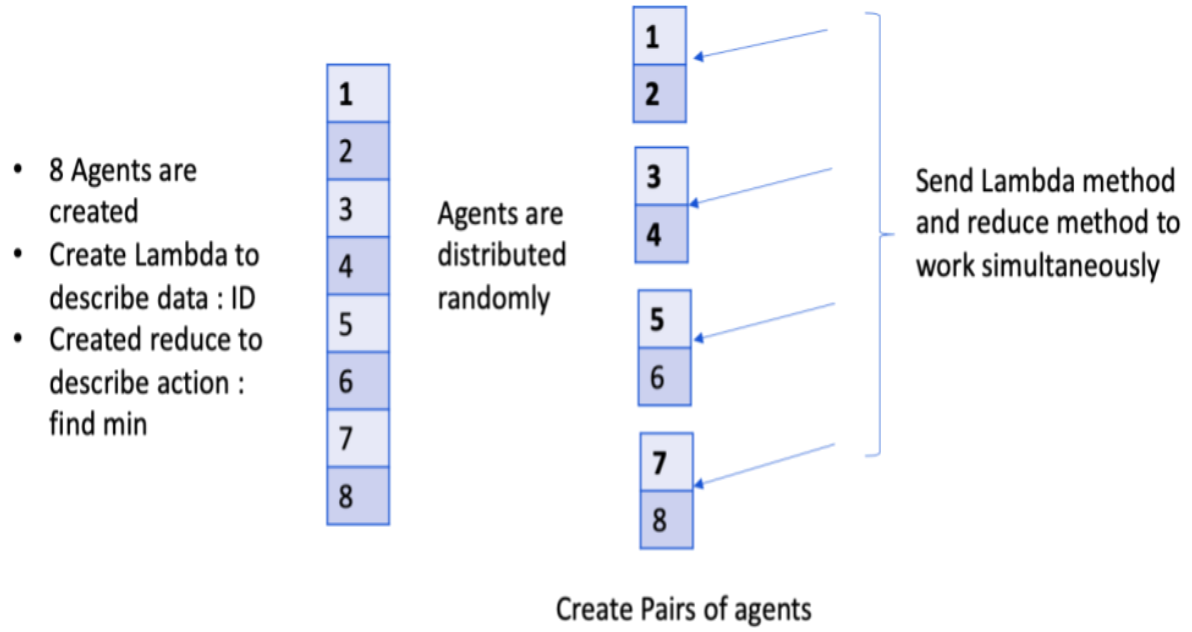

Figure 10. Reduce Strategy

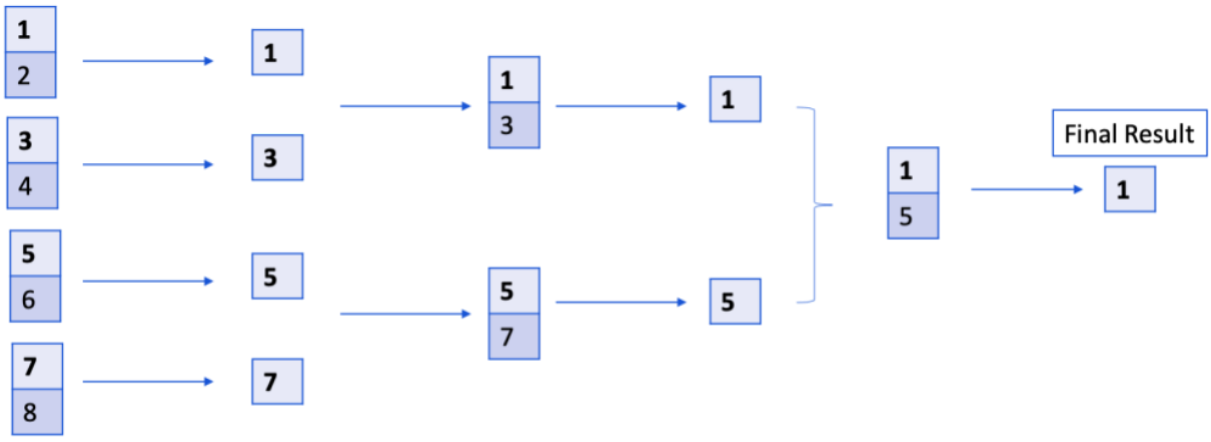

Figure 11. Reduce Strategy Continued

The benefit of using the *lambda* and *reduce* functionality is each node works on data residing or provided to them and return only a single result to the main(). This removes the

data overhead across nodes. Also, due to each node is responsible to perform its computation, it removes the computational overhead from the primary node. Now, a primary node only receives the data of the size of nodes which is very small. On top, the message size distributed and collected across nodes matters to calculate data overhead. Larger the message size, the greater the data overhead. This reduction method implementation overcomes these pitfalls.

Chapter 4. VERIFICATION

This chapter reviews, the performance, and programmability of *lambda* and *the reduction* method. As proof of the working implementation of *the lambda* and *reduction* method, this paper verifies the implementation of the Triangle Counting application. Our implementation of *lambda* and *reduction* is verified below:

4.1 RESULTS

The primary concern behind this research is to achieve usability, making it easier for a user to use and perform various operations worry-free. It is now easier than the traditional way of using the library. Now the user does not need to worry about the implementation details of the library and identifying the area of work for their use cases as shown in Chapters 3 and 4. Also, MASS is a very large library where every year multiple students work to enhance it. This leads to a large number of classes and related helper functions added by every student. This adds complexity to making the earlier version work properly on the user's machine and then the user can start thinking about their use cases and strategy to implement. However, now, because of *lambda* and *reduction* method implementation, a user is free from going into the compilation and building the earlier version of the library. Users can directly start writing lambda methods to perform desired actions and analyzing the results. This is the most important and achieved goal of this paper.

4.1.1 *Lambda*

Following figure 12 and 13 shows the single node execution on, cssmpi1h, UWB machine. *Lambda* method is created for addition of integers. Data provided is object array containing integers ”4,5” as shown in figure 13. Also, user creates 2 places and 4 agents. These 4 agents are responsible for execution program. Expected output is “9” and after execution output is also integer value “9”. This proves the correctness of the implementation. Figure 13 denotes the result received after performing the addition operation on given integer data items successfully.

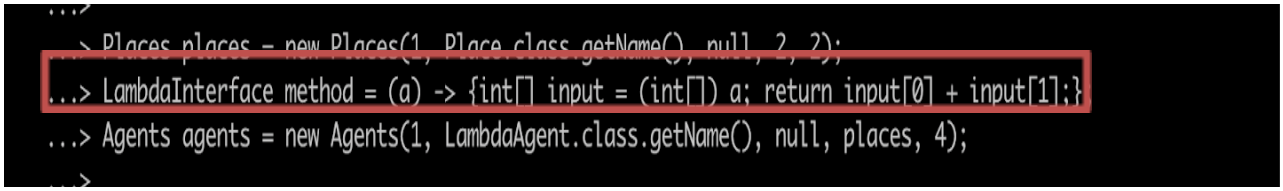

Figure 12. Agents and Places creation

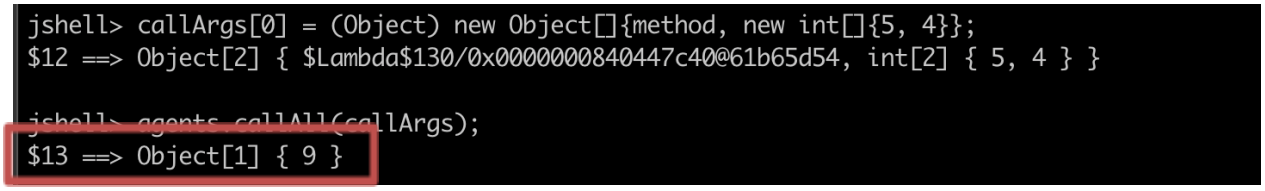

Figure 13. Resultant Output for Lambda

Multi-node execution:

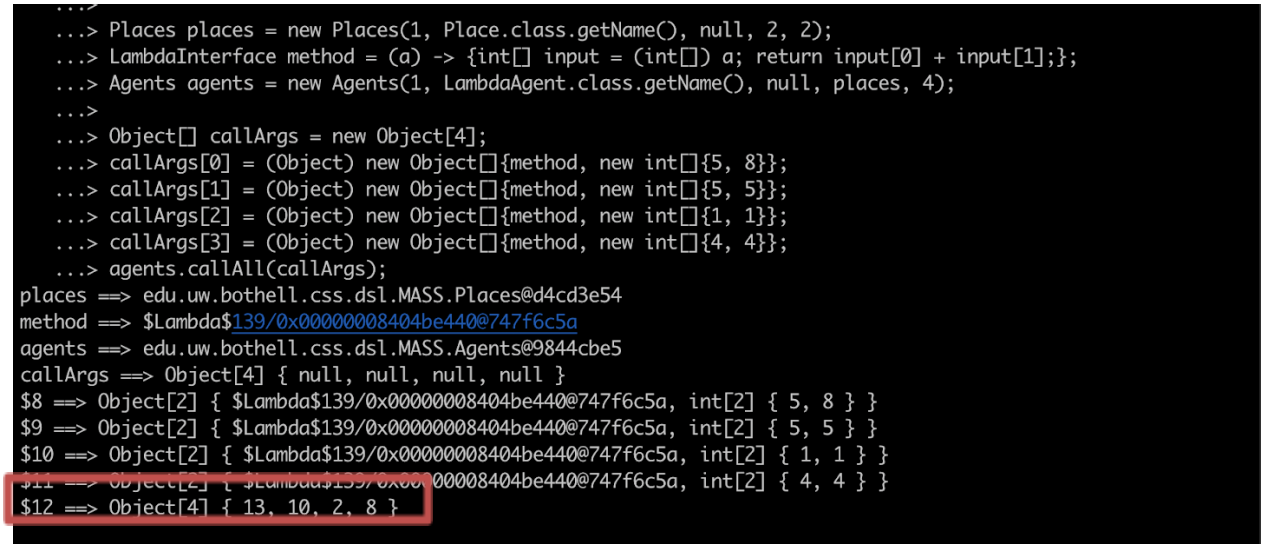

Figure 14. Primary Node in Multi-node execution of Lambda

In figure 14, 4 places and 4 agents created. So callArgs[4] represents every agent respectively. Here *lambda* performing sum operation takes parameters as different integers and returns results as objects[]. As Shown in figure 14, $12 shows the resultant object[] which contains all the results.

Following figure 15 shows logs on cssmpi2h. We can see that results for callArgs[2] and callArgs[3] have been calculated and returned back to the master node. The following image shows logged results on cssmp2h.

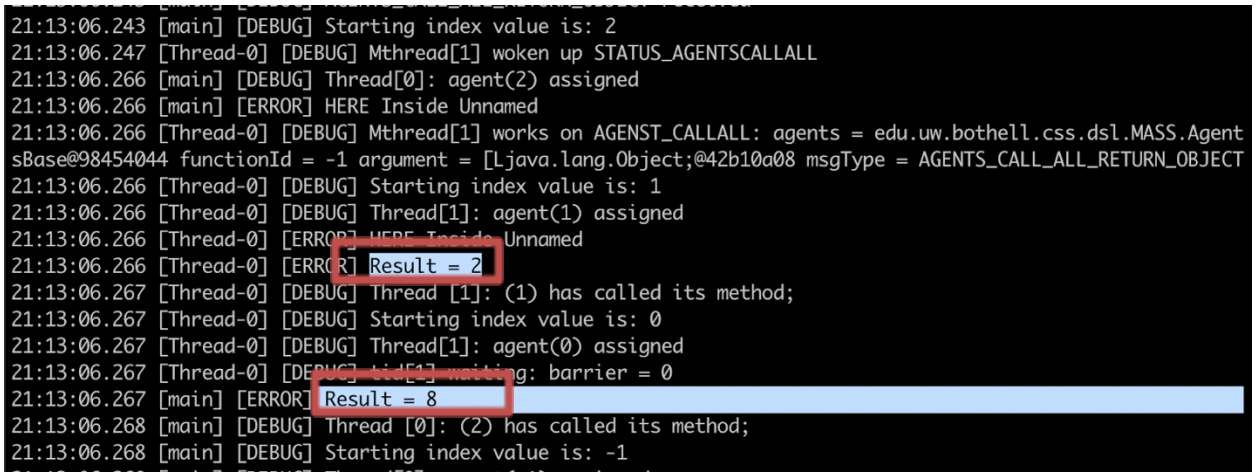

Figure 15. Logs from Secondary Node in Multi-node execution of Lambda

Updating/getting agents information in multimode environment:

In figure 16, user is getting the agents information about places they are residing.

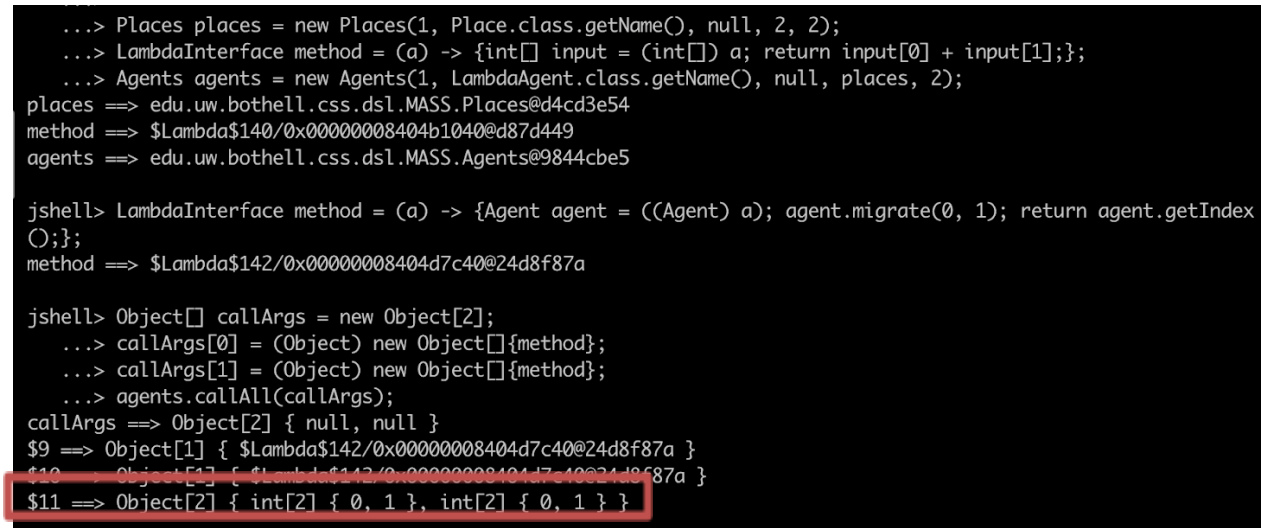

Figure 16. Getting agents information in multimode environment

In figure 17, user updates the agents by migrating them to places. We migrated all agents to [0,0] place.

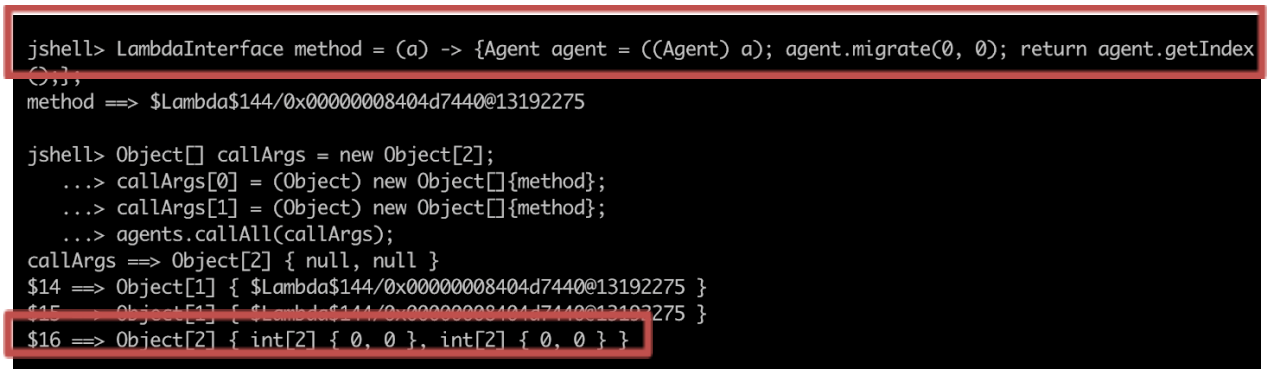

Figure 17. Updating agents information

Using callAll(), user can receive the output instantaneously. Figure 18 is an example of printing out the each data item using *for* loop from JShell.

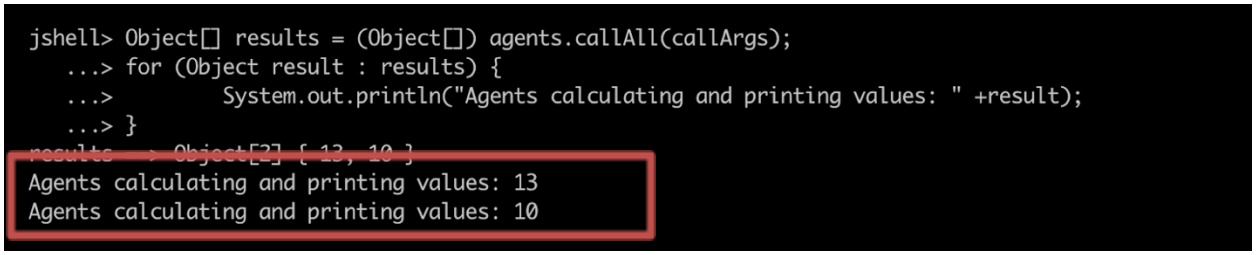

Figure 18. Printing results

4.2 LAMBDA ON PRIVATE METHODS

In Java private methods are the methods having private access modifier and are restricted to be access in the defining class only and are not visible in their child class due to which are not eligible for overridden [12]. So unlike shown in earlier section 4.1.1, we cannot directly access the private methods from JShell. To access the private methods of agents, we need to understand additional programming details in order to get the results successfully. In this section, we show how to use lambda methods on private methods in MASS library. We can replicate this behavior to any ABM library's private methods.

Following figure 19, shows the code written to access the private method from agents such as migrate. In this method we describe that all agents broadcasted should migrate to place[0,1].

```
jshell> LambdaInterface method = (a) -> {
   ...>         try{
   ...>          Agent agent = ((Agent) a);
   ...>          Method m = Agent.class.getDeclaredMethod("migrate", int[].class);
   ...>          m.setAccessible(true);
   ...>          m.invoke(agent, new int[]{0, 1});
   ...>          return agent.getIndex();
   ...>         } catch (IllegalAccessException | InvocationTargetException | NoSuchMethodException e) {
   ...>                 return null;
   ...>         }
   ...>  };
method ==> $Lambda$130/0x0000000840407440@7889b4b9
```

Figure 19. Accessing Private method using Lambda

The following figure 20 denotes that all the agents are successfully migrated to place [0,1] described by the user. In this manner, now user can access private methods to complete their use cases. Identifying this feature extension was really time consuming and requires lots of effort which, now, user does not need to worry about. User can use this information directly and modify as per requirement.

```
jshell> Places places = new Places(1, Place.class.getName(), null, 2, 2);
   ...> Agents agents = new Agents(1, LambdaAgent.class.getName(), null, places, 2);
places ==> edu.uw.bothell.css.dsl.MASS.Places@d4cd3e54
agents ==> edu.uw.bothell.css.dsl.MASS.Agents@9844cbe5

jshell> Object[] callArgs = new Object[2];
   ...> callArgs[0] = (Object) new Object[]{method};
   ...> callArgs[1] = (Object) new Object[]{method};
   ...> agents.callAll(callArgs);
callArgs ==> Object[2] { null, null }
$10 ==> Object[1] { $Lambda$130/0x0000000840407440@7889b4b9 }
$11 ==> Object[1] { $Lambda$130/0x0000000840407440@7889b4b9 }
$12 ==> Object[2] { int[2] { 0, 1 }, int[2] { 0, 1 } }

jshell>
```

Figure 20. Result on Private method using Lambda

4.3 REDUCE METHOD

Following figure 21 is demonstration of user can use *lambda* and *reduce* to perform addition operation on agent Id.

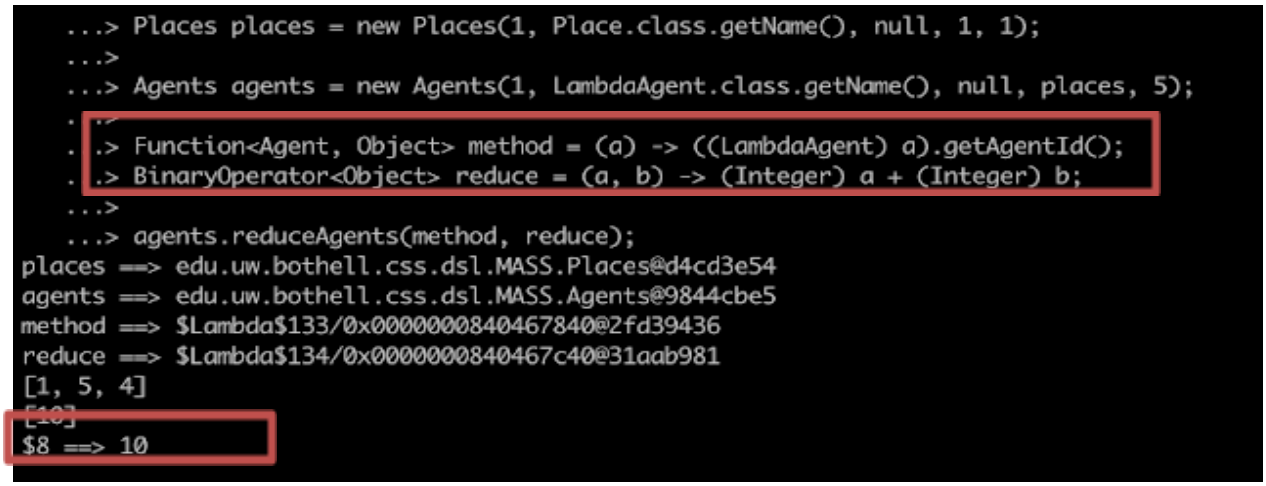

Figure 21. Addition Operation on Agent ids

Figure 22 is the demonstration of user can use *lambda* and *reduce* to perform operation on different data than agent ids successfully.

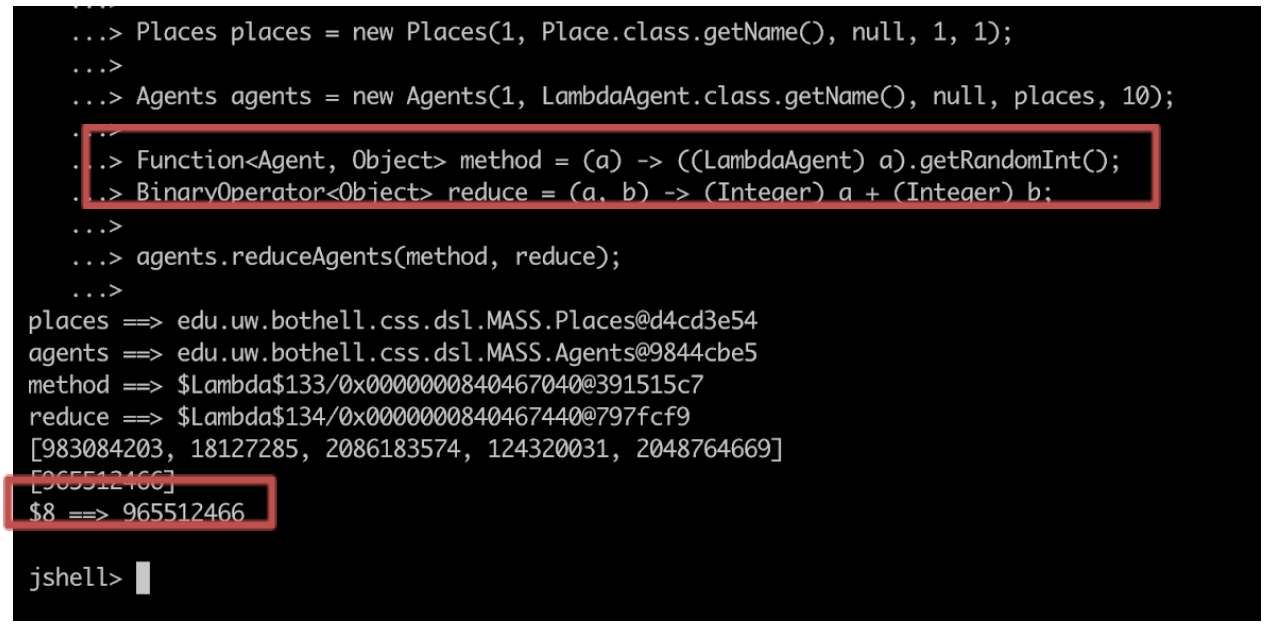

Figure 22. Addition of random generated numbers

4.4 TRIANGLE COUNTING APPLICATION

The triangle counting application is responsible to read the graph and find the potential triangle generated from the given input graph. This application is to prove the successful implementation of the *lambda* and *reduce* method from InMASS.

In the Triangle Counting benchmark, the simulation begins with an Agent on each node. All Agents then traverse the graph structure by first traveling twice to lower-valued Place nodes, spawning child Agents when multiple nodes meet this criterion, and then trying to return to their original node. If the Agent is successful, then a triangle has been found. While a triangle is found, it keeps track of weights associated with that path of forming a triangle. In the end, the program enumerates all the triangles formed and their weights. This is the data set created for the analysis. Earlier, if a user needs to find what is the maximum or minimum distance traveled by an agent to form a triangle, a user needs to write another block of code where it again enumerates all triangles and weights. On this resultant data, now the user writes a block of code to find the shortest, maximum distance, or any desired operation.

```
Function<Agent, Object> method = (a) -> ((CrawlerGraphMASS) a).getDistance();
BinaryOperator<Object> reduce = (a, b) ->  Math.min((Integer)a, (Integer)b);
```

Figure 23. Lambda and Reduce method

Following figure is part of output received after running the triangle counting application from JShell. The input to this application is the graph containing adjacency list

structure with additional parameter of weight associated to each edge connecting two vertices. Figure 23 shows that the two triangles are formed after running the application.

```
jshell> // Enumerate all triangles
   ...>     if (show) {
   ...>         int[][] dummyArgs = new int[nAgents][1];
   ...>         Object[] triangles = (Object[]) crawlers.callAll(CrawlerGraphMASS.getTriangle_, (Object[]) dum
myArgs);
   ...>
   ...>         for (int i = 0; i < nAgents; i++)
   ...>             System.out.println(i + ": " + (String) triangles[i]);
   ...>     }
   ...>
0: agent(4): 4 3 2 4
1: agent(2): 2 1 0 2
```

Figure 24. Triangle itinerary

Distance calculated for these 2 triangles is: 12, 6 as shown in following figures 23. Create Lambda method to calculate the distance of the triangle. Then create *reduce* method which is also the type of *lambda* method to perform reduce operation such as finding minimum/maximum distance travelled to create triangle.

Performing get max operation on distance calculated in triangle counting using reduction method:

```
jshell> Function<Agent, Object> method = (a) -> ((CrawlerGraphMASS) a).getDistance();
   ...>
   ...> BinaryOperator<Object> reduce = (a, b) ->  Math.max((Integer)a, (Integer)b);
method ==> $Lambda$152/0x00000008404adc40@51e7589f
reduce ==> $Lambda$154/0x00000008404afc40@afde064

jshell> crawlers.reduceAgents(method, reduce);
[12]
[12]
$79 ==> 12
```

Figure 25. Get max distance using reduction method

Performing get min operation on distance calculated in triangle counting using reduction method:

```
jshell> Function<Agent, Object> method = (a) -> ((CrawlerGraphMASS) a).getDistance();
   ...>
   ...> BinaryOperator<Object> reduce = (a, b) ->  Math.min((Integer)a, (Integer)b);
method ==> $Lambda$152/0x00000008404adc40@51e7589f
reduce ==> $Lambda$153/0x00000008404af840@2dbfa972

jshell> crawlers.reduceAgents(method, reduce);
[6]
[6]
$76 ==> 6
```

Figure 26. Get min distance using reduction method

Triangle counting with *lambda* and *reduction* allows a user to retrieve each agent's travel distance as well as analyze the data with max, min or any user desired operation.

4.5 PROGRAMMABILITY

In this section, we will compare the effort reduced for a user in writing code. First, we will check how code was written earlier when *lambda* wasn't introduced in MASS. After that, we will check how easy it is now to write code and analyze results.

Earlier, a primary node was responsible to collect results from every other node working and it is the primary node's responsibility to rerun the user-desired operation to receive single output. This adds data overhead and computational overhead to the primary node. The following figure 24 denotes the number of programming lines reduced after using the *lambda* and *reduction* method to calculate the minimum of agent ids. This is an example and can change as the functionality user wants to perform changes. The complete code is listed in the appendix.

Also, to reduce data without using the reduction method implemented users have to write an additional entry into the switch case into the library and implement the

functionality for a user to be able to able to access from JShell. After updating the code into the library, the user still needs to know how to invoke the desired functionality. This is cumbersome and requires additional effort from the programmer to maintain the code. However, now, the user and programmer is free and can directly write *lambda* and *reduction* method to perform the desired functionality.

Now, because of the usage of *lambda* and *reduction* method in MASS, this paper confirms that the user does not need to write another block of code where a user needs to know the internals of the MASS application or core library. Now users can use *the lambda* and *reduction* method and complete this functionality within 3 lines of code. One line to write lambda method identifying the data, second line to write operation user wants to perform to reduce the data and last line to call gents to tell them to perform reduce operation on data items to return a result to the user.

Earlier reduce without boilerplate #Lines of Code: 12

Now reduce without boilerplate #Lines of Code: 3

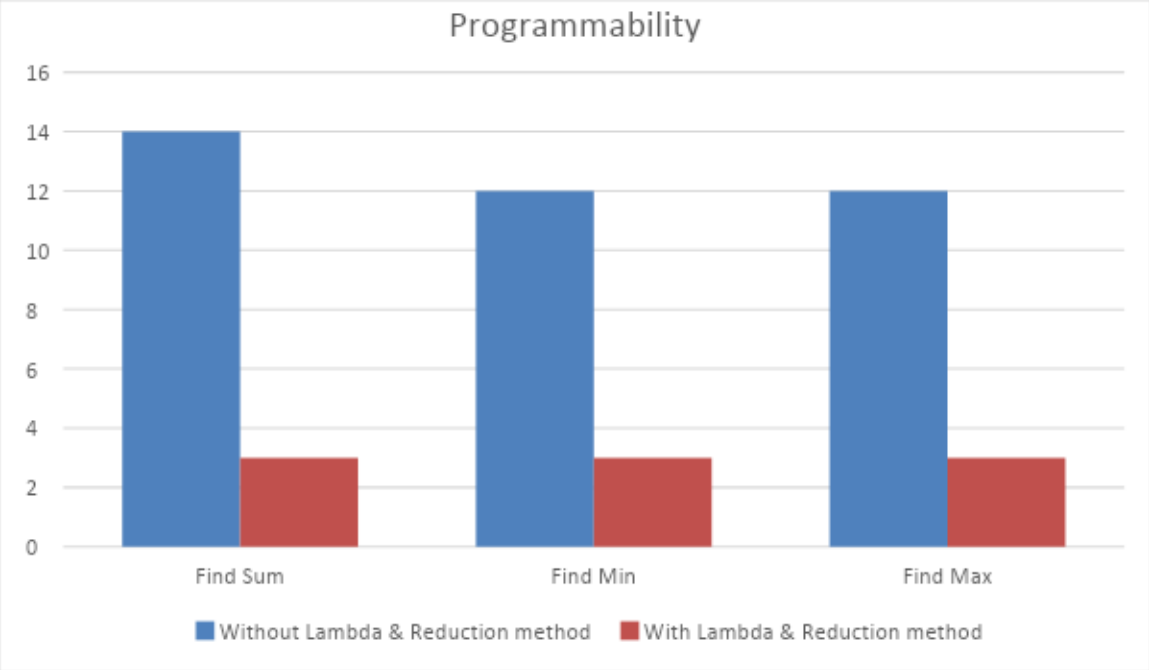

Figure 27. Programmability

This graph denotes the significant difference in programmability aspect. The number of lines written by user to code and test the functionality on the fly is reduced approximately by 75%. As shown in figure 28 user needs to write a block of code to complete the functionality. On the contrary, as shown in figure 29, our implementation shows that user can complete and receive the same results by writing just single line of code. This is huge flexibility and ease of use provided to user.

```
//find shortest distance
if (show) {
Object[] dummyArgs = new Object[nAgents];
    Object[] dist = (Object[]) crawlers.callAll(CrawlerGraphMASS.getDistance_,
    dummyArgs);
    int shortestDist = Integer.MAX_VALUE;
    for (int i = 0; i < nAgents; i++) {
       if (shortestDist > (int) dist[i]){
            shortestDist = (int) dist[i];
       }
    }
System.out.println("Shortest Distance : " +shortestDist);
}
```

Figure 28. Code block without Reduction method

```
Function<Agent, Object> method = (a) -> ((LambdaAgent) a).getAgentId();
BinaryOperator<Object> reduce = (a, b) -> (Integer) a + (Integer) b;
agents.reduceAgents(method, reduce);
```

Figure 29. Simplified Reduction method

As shown in figure 29, few parameters which emphasis on ease provided to user are as follows:

- User can directly state desired operation to perform on data in *reduction* method. Unlike traditional way, where many efforts need to be taken just to reach to a state where user can start focusing on their use case.
- Currently, most of the user using MASS library needs to understand the internals of MASS and how to run applications using InMASS. Now, this complexity and overhead is removed from the user. As shown in figure 28, user needs to know that they need to create agents of type crawlers and then perform operation of collecting distance on them. But now, as shown in figure 26 user do not need to spend time thinking and understanding internals of MASS. Using basic programming knowledge user can complete desired tasks such as getting the minimum of distance as shown in figure 29.
- The main improvement is now user does not need to write boilerplate code including library initialization, classes, and functions to complete user desired use cases. User can directly focus on their work. This also increases the efficiency of the user's work and provides ease.
- Major time is saved by using *lambda* and *reduction* method which otherwise must spend on understanding and identifying how to work with MASS, agents, and places.

4.6 Performance Evaluation

This chapter discusses the result of performance testing of the *lambda* and *reduction* method on the Triangle counting application. The following results shown in figure 30 are run on cssmpi-h UWB machines for running Triangle counting application on 500, 1500, 3000, 6000 nodes graph. The time measured to run the triangle counting application is in a millisecond.

Following results shown in figure 30 are compared with how the same operation, finding the shortest path, will occur in legacy MASS vs implementation with the help of *lambda* and *reduction* method now user can complete this functionality. These trials were conducted using graph sizes between 500, 1,500, 3000, and 6000 vertices with the same number of agents as vertices. As the size of the graph increases, we also observe a corresponding exponential increase in the time taken to complete the operation without using *the lambda* and *reduction* method. However, we see that steady and relatively improved performance when *lambda* and *reduction* methods are in use.

4.6.1 *Single node execution*

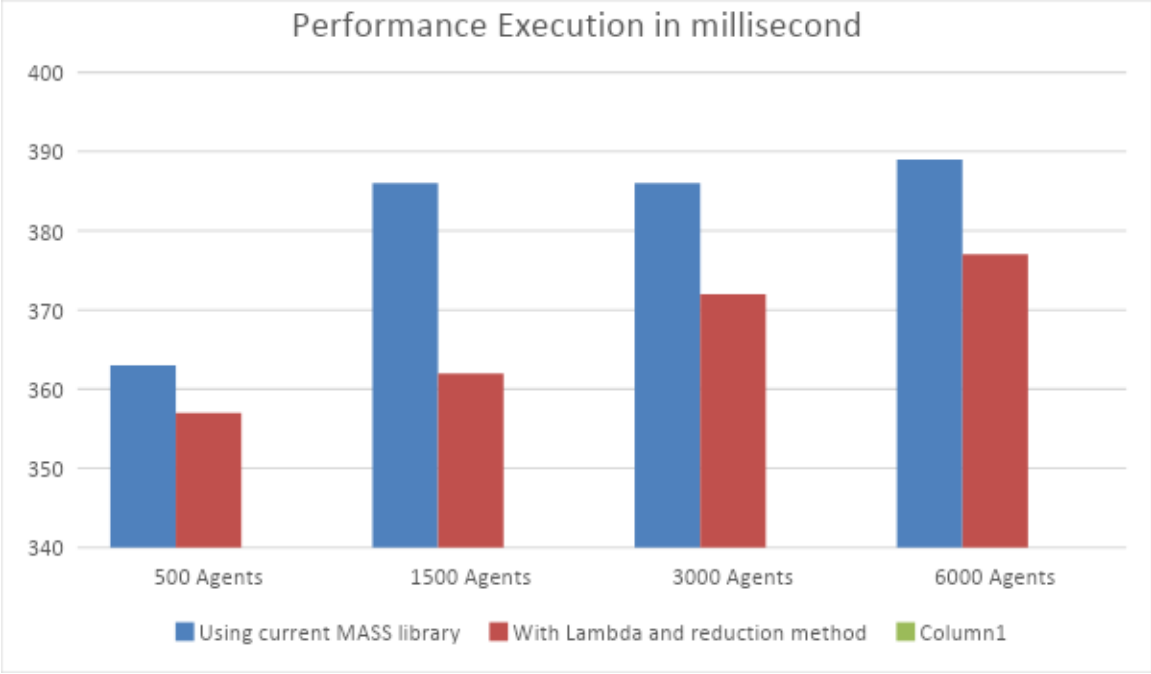

Figure 30. Single Node Execution Performance

From figure 30, we can see that after using the *lambda* and *reduction* method, performance has incremented significantly as we increased the size of the graph. As we go on increasing the size of the graph such as 500, 1500 vertices, 3000, and 6000 vertices, performance using *lambda* and *reduce* method increases as compared to traditional implementation. Figure 30 shows the improvement for 500, 1500, 3000, and 6000 nodes run with *lambda* and *reduction* method vs without these new features.

The primary reasons behind the performance improvement are as follows:

- Message passing overhead is reduced: Current implementation leverages a multinode environment as each node has a list of agents and their data. It calculates the partial results and sends the primary node only intermediate results. This data

size will never be greater than the number of nodes in use. This reduces the data overhead caused by traditional implementation.

- Usage of *lambda* and *reduction* method reduces the computational overhead from the primary node: Each node working on their data and then returning it to the primary node reduces the computational burden from the primary node. Now, a primary node receives the maximum data size of nodes created. This is extremely low as compared to the traditional way.
- Using the *lambda* and *reduction* method, it became easier for a primary node to calculate the desired output on the fly. User is free to focus on desired functionality, modify functions or play with the resultant data to extract meaningful information from it easily.
- Also, we see an exponential difference in legacy MASS performance difference with 500 agents and 1500 agents. This difference is because more agents are running, and data needs to be iterated multiple times to receive the final output, resulting in an increase in performance. Also, initial setup time is similar but after that more number of agents we create, time to create agents increases.

4.6.2 *Multi node execution*

For performance execution, we have run our program on CSSMPI 1h – 12h machines provided by University of Washington, Bothell. As we take measurements for performance, we have created 5000 agents and places same as the number of nodes in use.

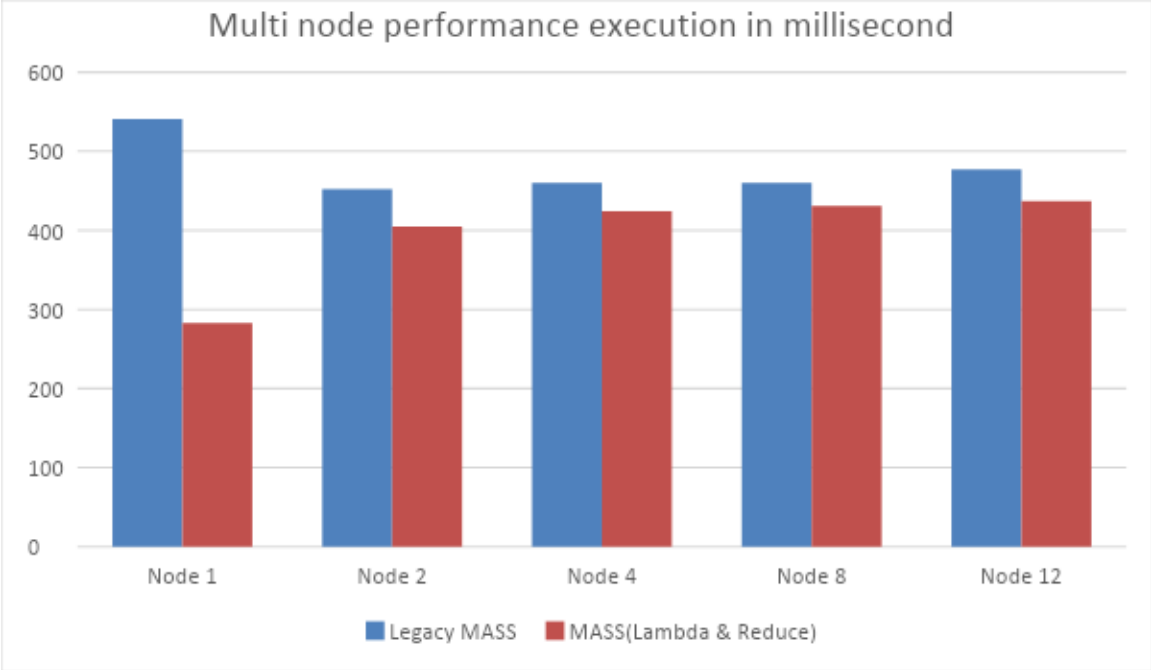

Figure 31. Multi-node performance execution

The difference between legacy MASS and MASS with *lambda* and *reduction* method is because our implementation is working parallelly. Each node is responsibly operating on associated data. It creates the agents and places, provides *lambda*, and *reduces* the method parallelly to pair of agents on each node, reducing overhead from the master node. Resulting in, MASS with lambda and reduction method being faster than legacy MASS.

As we observe in figure 31, when we compare execution on a single node, we see a major difference in legacy MASS vs MASS with *lambda* and *reduced* method in use. This difference is because of the agent residing on the same node the entire operation execution. On top of this, parallelly executing on the same node adds to the performance. On the other hand, legacy MASS is slow because there is no parallel execution.

We have also verified the performance on multimode with 50000 agents and places as same as node size. As shown in figure 32, MASS with lambda and reduce method has

gained exponential performance gain. This performance gain is because of the number of agents in use. More the agents working across nodes, parallel execution of operations is taking place. This performance gain is approximately 58.7% as compared to traditional MASS.

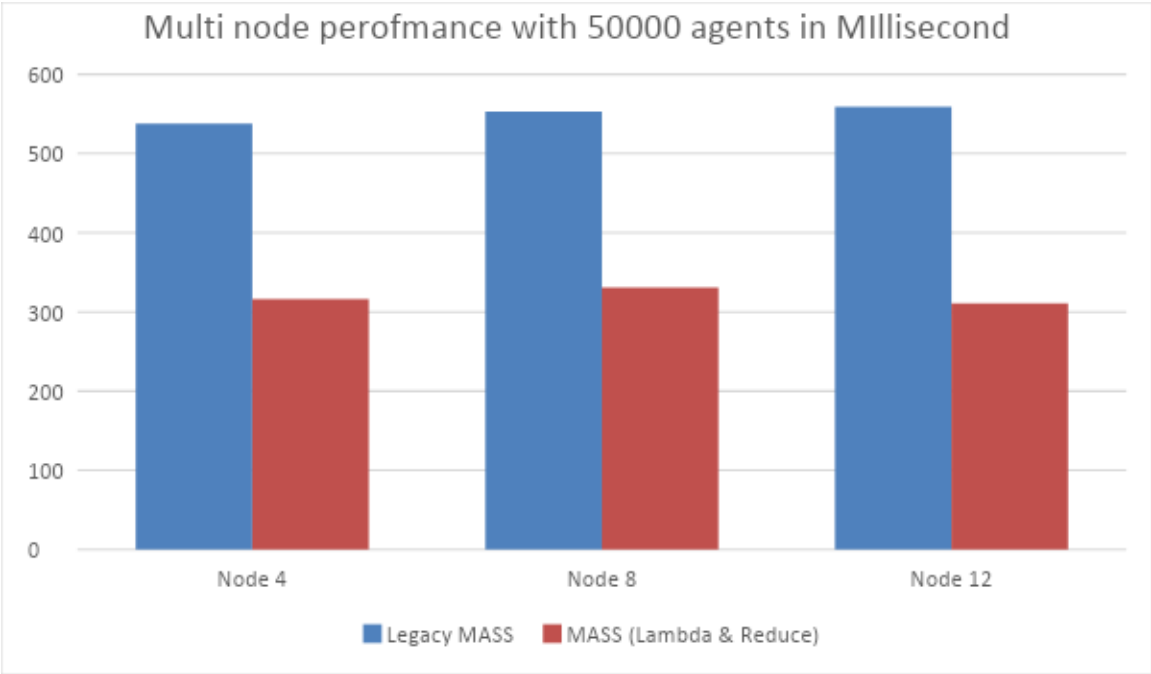

Figure 32. Performance with 50000 agents

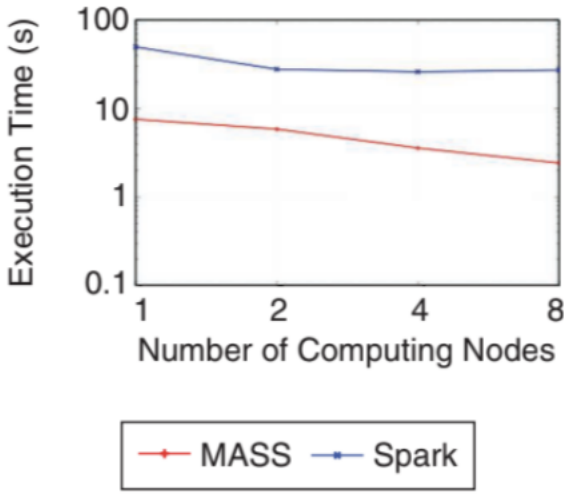

Figure 33. Comparison MASS vs Spark on Triangle Counting [14]

To compare between MASS and Spark, I have referred “An Agent-Base Computational Framework for Distributed Data Analysis” [14]. We can see following criteria analyzed on triangle counting application on MASS and Spark:

(a) Performance: As shown in figure 33, we can see that MASS performs better as compared to MASS from single node to multimode environment.

(b) Readability: To write code using Spark is less intuitive than MASS. User needs to understand the syntax and how it will work in order to implement new features. However, in MASS, User can use their basic Java to write lambda expressions and implement new features.

(c) Data Overhead: MASS reads the places data directly from each computing node’s local disk, whereas Spark creates an RDD in main(), which is then distributed to the other cluster nodes. This causes additional data-transfer overhead.

(d) To write a block of code using Spark, user needs to understand the syntax and basic not intuitive to write code.

(e) There are different types of transformations and actions in Spark User needs to perform these transformations on RDDs in order to perform any actions as RDDs are immutable in nature. These transformations are of two types, Narrow and Wide transformation. To perform any operation, user first need to understand how they work on RDD and manage memory. If we look at Triangle counting application using lambda and reduce method, it becomes intuitive, and user does not need to worry about memory being used.

Now, we compare our implementation with Spark and legacy MASS. We will compare on three technologies working on same setup and are comparable.

Criteria	Spark	Legacy MASS	MASS (lambda and reduce)
Lines of Code	2	12	2
Boilerplate Code Ratio(BPR [14])	0.16	0.1	0.08
Readability	6x	1x	6x
Speed (Triangle Counting) [14]	~85sec	~9sec	~5sec

Table 3. Comparison table

We see improvements due to the implementation of *lambda* and *reduction* method on criteria mentioned in table 3. Spark is used by multiple businesses and it took years of development to bring it to industry standards. Now with legacy MASS, from table 3, we see that it needs more coding and it is burdensome to test and work on new features instantaneously. However, using lambda and reduce methods, MASS becomes more readable and interactivity is increased as user can write and test new features instateneuosly and user does not need to know the internal details of the MASS. Our observation is after the implementation of *lambda* and *reduction*, MASS has become better for processing big data, and obtaining immediate insights with the additional flexibility of writing code for users.

Code readability means writing and presenting your code in such a manner that it can be easily read and understood. Code readability is more in MASS with lambda and reduce as compared to legacy MASS as a user just needs to write the lambda method to perform an operation. Where in legacy MASS, a user needs to know which agents to use and how to handle them while writing code from JShell. We saw earlier code with MapReduce,

Spark, Legacy MASS, and MASS with lambda and reduction method. This we verified from listing 1-7 and figure 22. Also, writing code in MASS with lambda and reduce does not require any additional knowledge or consideration from a user. User can use their basic java knowledge to write lambda methods and perform operations as per requirement. This leads to more intuitive programming using lambdas in MASS instead of MapReduce or Spark. From table3, readability is derived relative with Spark, legacy MASS and MASS with lambda and reduction method. As we can see, Spark and Mass with lambda and reduce become 6 times more readable as lesser LoC is required from the user as compared to legacy MASS.

On top, Spark and MASS with lambda and reduce provides encouragement of functional programming. On contrary to legacy MASS, code becomes modular and reusable and this can be verified from listings 1-7 where we mentioned about Spark, Mapreduce, Legacy MASS and MASS with lambda and reduce. The boilerplate-code ratio refers to the LoC needed to set up the environment against the total number of LoC in the main function[14]. Also, in table 3, speed of execution shows major difference. This is because MASS reads the places data directly from each computing node's local disk, whereas Spark creates an RDD in main(), which is then distributed to the other cluster nodes. This causes additional data-transfer overhead[13].

Interactivity is one of the most caching features when it comes to the user experience of using any technology. Interactivity means allowing the bidirectional flow of information from computer and user. In Spark, interactivity is achieved by using a REPL environment where users can learn and use Spark APIs interactively [15]. In legacy MASS it is achieved by providing support of JShell. On top of that, now, with *lambda* and *reduce*

method included in MASS, interactivity is increased as there is less writing of code, and user can investigate and perform operational analysis and perform rollback opreration and it also supports debugging. Also, as for any single change, a user does not need to go to the mass-core library and change the code. Now because of *lambda* and *reduce* method, users can do that without going into the build, compile, and push code to remote machines resulting to increase in intercativity.

Chapter 5. CONCLUSION

In this research paper, we successfully implemented new features such as *the lambda* method and r*eduction* method for MASS. These features are not only limited to the MASS library but also applicable to ABM libraries. Therefore, we feel that our work has contributed to enabling agent-based on-the-fly data analysis.

These two new features provide a user-desired add-on for rapid agent code development and agent-based data exploration. This research paper explained the technical details of *lambda* and *reduction* method supports for InMASS that runs on top of JShell. Verification of this work is done, using a triangle-counting benchmark, which showed that the new functionality performed as expected, ran faster than the previous implementation, and improved the programmability. Using these features, MASS users can easily implement new features on the fly and therefore user does not need to worry about MASS internals. Also, a user is free from going through the compilation, building, and deployment of code every time the user wants to implement or test a new feature. Indirectly, eases and boosts the working capacity of the user.

5.1 LIMITATIONS

To overcome the shortcomings of MASS, lambda and reduction methods are implemented. Following are the limitations which restrict MASS to become more versatile.

- *Lambda and reduce does not provide static or runtime type safety and does not allow a user to check error at runtime.*

- Performance might degrade when there is not enough memory available to store data in memory.
- To work using MASS, a user needs to write a few lines of code. However, if a user does not want to write any code, we should provide a user interface to allow users to perform some basic operations.

5.2 Future work

To extend this paper's work, future work can be done in a variety of areas. A few of them are listed below:

- The current implementation of the reduction method involves the user writing two lambda methods to define data and action. As an extension to this project, we can merge these methods and the user will need to write a single lambda method defining action and data instead of writing methods as shown in figure 26, user can write "getData.reduce(findMin);". So, here expectation is "getData" is telling which data to consider for performing the find Min operation.
- Currently, the user still needs to write code to perform operations. Instead of writing java code, the goal is to provide a GUI over JShell so that users like data scientists do not need to worry about code writing. This GUI should provide inbuilt functions such as find min, max, and average. Having this GUI might provide ease for the user but restrict a user to perform different operations on data. To provide GUI, MASS is compatible with the Cytoscape plugin. This plugin has already been used to provide a GUI interface to users.
- Also, because Spark is compatible with YARN (Yet Another Resource Manager), it can

run on an existing Hadoop cluster and access any Hadoop data source, including HDFS, S3, HBase, and Cassandra. Providing this compatibility with MASS will provide data for the reduction method to be accessed from these existing clusters. This will result in ease for users to receive and perform an operation on data from the choice of data storage.

Chapter 6. BIBLIOGRAPHY

[1]	C. G. U. M. a. M. S. Munehiro Fukuda, "An agent-based computational framework for distributed data analysis.," Computer (Volume: 53, Issue: 3, March 2020), March 2020. [Online].
[2]	M. F. a. D. S. Lab, "MASS Java Manual," [Online].
[3]	N. Alghamdi, "Supporting Interactive Computing Features for MASS Library: Rollback and Monitoring System," Bothell, 2020.
[4]	"Java SE 8 : Quick Start," Oracle, [Online]. Available: https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/Lambda-QuickStart/index.html.
[5]	V. Nukala, "Java Code Geeks," June 2021. [Online]. Available: https://www.javacodegeeks.com/2021/06/java-spark-rdd-reduce-examples-sum-min-and-max-operations.html.
[6]	"javaTpoint," [Online]. Available: https://www.javatpoint.com/reduce-java.
[7]	D. Blashaw, "An Interactive Environment to Support Agent-based Graph Programming," [Online]. Available: http://faculty.washington.edu/mfukuda/papers/icaart22.pdf.
[8]	h. p. s. @psil123, "Function Interface in Java with Examples," geeksforgeeks, 8 December 2021. [Online]. Available: https://www.geeksforgeeks.org/function-interface-in-java-with-examples/.
[9]	s. @psil123, "BinaryOperator Interface in Java," Geeksforgeek, 30 september 2021. [Online]. Available: ttps://www.geeksforgeeks.org/binaryoperator-interface-in-java/.
[10]	J. Jurinová, "Performance improvement of using lambda expressions with new features of Java 8 vs. other possible variants of iterating over ArrayList in Java," Computer Science , 2018. [Online].
[11]	V. Ram, "tutorialspoint," 03 December 2018. [Online]. Available: https://www.tutorialspoint.com/private-and-final-methods-in-java-programming.
[12]	"tutorialspoint," [Online]. Available: https://www.tutorialspoint.com/what-are-the-advantages-of-lambda-expressions-in-java.
[13]	A. Seigneurin, "ippon," 22 November 2014. [Online]. Available: https://blog.ippon.tech/intro-mapreduce-spark/.
[14]	An Agent-Based Computational Framework for Distributed Data Analysis, Munehiro Fukuda, University of Washington Bothell, Collin Gordon, JBT FoodTech, Utku Mert, Apple Matthew Sell, Fluke Electronics
[15]	R. Savaram, "MindMajix," 01 june 2022. [Online]. Available: https://mindmajix.com/spark/repl-environment-for-apache-spark-shell.

APPENDIX A : DEVELOPER GUIDE

○ A.1 MASS LIBRARY INITIALIZATION

Following installation process is based on MAC OS but should be similar for every other OS.

1. Download ans install MASS_core from Bitbucket : **Link**
2. Build and install MASS_core: From the command line or prefered IDE, prepare package of library. Following figure shows IntelliJ IDE process.

```
with assembly file: /Users/niraligundecha/Documents/Capstone/CSS595_Capstone/Impleme
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  13.143 s
[INFO] Finished at: 2022-05-13T11:41:46-07:00
[INFO] ------------------------------------------------------------------------
```

Figure A.1: Builid MASS_Java_Core

3. Upload the jar, nodes.xml to remote machines i.e cssmpi-h machines or Hermes machines. In nodes.xml we add or remove nodes according to our use case. Also, confirm the correctness of port number in use and active process.
4. Now, user should be able to start InMASS and JShell platform.

Figure A.2: Initializing InMASS and JShell

This is the base of our program. MASS library users are going to use this platform to write their code and complete their use cases to analyze the results.

○ A.2 LAMBDA METHOD

To use lambda methods successfully follow the steps :

1. Download or clone the mass_java_core from bitbucket. Our lambda implementation resides under develop/Nirali branch. This branch is merged with dblashaw_dev2. For cloning this branch use NiraliDevelop branch. After cloning, install the library using the same steps mentioned in A.1.
2. Start the InMASS and JShell platform using mass_java_core jar.

3. Now, this is the time where user can starts working on their use cases. As shown in figure A.3 user creates an agents and places as agents are entities responsible to perform operations over places.

```
jshell> Places places = new Places(1, Place.class.getName(), null, 2, 2);
   ...>
   ...> Agents agents = new Agents(1, LambdaAgent.class.getName(), null, places, 4);
places ==> edu.uw.bothell.css.dsl.MASS.Places@d4cd3e54
agents ==> edu.uw.bothell.css.dsl.MASS.Agents@9844cbe5
```

Figure A.3: Create Agents and Places

4. After initial setup of Agents and places in InMASS, primary operation can be written using lambda expression syntax as shown in A.4. This *lambda* is performing the addition operation.

```
jshell> LambdaInterface method = (a) -> {int[] input = (int[]) a;
   ...> return input[0] + input[1];};
method ==> $Lambda$133/0x0000000840467840@26be9a6
```

Figure A.4: Create Lambda Expression

5. To perform *lambda* operation, user need to provide data on which this operation is supposed to perform. In following figure A.5 data is provided to all the agents with *lambda* method to complete the operation. Also, we see that callall() is called on agents and technical explanation of process is explained in the Chapter 3. Result is printed out on JShell for analyzing and identifying the correctness. We can see that,

agents have performed lambda operation successfully.

```
jshell> Object[] callArgs = new Object[4];
   ...> callArgs[0] = (Object) new Object[]{method, new int[]{5, 8}};
   ...> callArgs[1] = (Object) new Object[]{method, new int[]{5, 5}};
   ...> callArgs[2] = (Object) new Object[]{method, new int[]{1, 1}};
   ...> callArgs[3] = (Object) new Object[]{method, new int[]{4, 4}};
   ...> agents.callAll(callArgs);
callArgs ==> Object[4] { null, null, null, null }
$9 ==> Object[2] { $Lambda$133/0x0000000840467840@26be9a6, int[2] { 5, 8 } }
$10 ==> Object[2] { $Lambda$133/0x0000000840467840@26be9a6, int[2] { 5, 5 } }
$11 ==> Object[2] { $Lambda$133/0x0000000840467840@26be9a6, int[2] { 1, 1 } }
$12 ==> Object[2] { $Lambda$133/0x0000000840467840@26be9a6, int[2] { 4, 4 } }
$13 ==> Object[4] { 13, 10, 2, 8 }
```

A.5: Receive the results

This lambda expression can perform any operation user wants sum, subtraction, addition , multiplication and these operation can be performed on any data related to agent or user created.

- A.3 Reduction Method

To use the reduction method successfully follow the steps:

1. Download or clone the library form Bitbucket. Our *reduction* method implementation resides under develop/Nirali branch. This branch is merged with dblashaw_dev2. For cloning this branch use NiraliDevelop branch. After cloning, install the library using the same steps mentioned in A.1.
2. Create agents and places as per requirement.

```
jshell> Places places = new Places(1, Place.class.getName(), null, 1, 1);
   ...>
   ...> Agents agents = new Agents(1, LambdaAgent.class.getName(), null, places, 10);
places ==> edu.uw.bothell.css.dsl.MASS.Places@d4cd3e54
agents ==> edu.uw.bothell.css.dsl.MASS.Agents@9844cbe5
```

Figure A.6: Create Agents and places for reduction operation

3. Write reduction method to perform addition operation

```
jshell> Function<Agent, Object> method = (a) -> ((LambdaAgent) a).getAgentId();
   ...> BinaryOperator<Object> reduce = (a, b) -> (Integer) a + (Integer) b;
method ==> $Lambda$133/0x0000000840467040@391515c7
reduce ==> $Lambda$134/0x0000000840467440@797fcf9
```

Figure A.7: Describe Reduction operation

4. We provide created reduction method describing operation to perform and lambda expression describing on what data to operation to perform on to reduceAgents. We call this method in agents and internals and technical details are described in chapter 3. In figure A.8 we can see that successfully agents calculate the results.

```
jshell> agents.reduceAgents(method, reduce);
[1, 5, 9, 13, 17]
[45]
$8 ==> 45
```

Figure A.8: Results after performing Reduction operation

Finally, we can see that, the *lambda* and *reduction* method have made very easy for user to concentrate on their work and get results instantaneously. Also, performance gain is addon benefit received from our implementation.