# Multi-Agent Spatial Simulation (MASS) Java Library Performance Improvement for Big Data Analysis

Utku Mert

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

Master of Science in Computer Science and Software Engineering

University of Washington

2017

Reading Committee:

Prof. Munehiro Fukuda, Chair

Asst. Prof. Wooyung Kim

Dr. Johnny Lin

Program Authorized to Offer Degree:
Master of Science in Computer Science and Software Engineering

University of Washington

**Abstract**

Multi-Agent Spatial Simulation (MASS) Java Library Performance Improvement for Big Data Analysis

Utku Mert

Chair of the Supervisory Committee:
Prof. Munehiro Fukuda
Chair of Computer Science and Software Engineering Department

MASS is a parallel-computing library for multi-agent and spatial simulation over a cluster of computing nodes. The library uses two important concepts; Agent and Place. Place represents each array index of the given data set while Agent instances execute set of instructions to perform actual computation. Agents communicate with each other to exchange data and they can spawn new child Agent instances, migrate between Places, or get terminated. The library is used for conducting simulation or big data analysis. This project primarily focuses on memory efficiency of the MASS Java library for big data analysis. Many contributors have improved this library since its initial release in 2010 and some of them made significant changes on MASS Java version during their capstone project or thesis process. Our findings show that an Agent instance uses up to 1MB of memory space and some scientific applications, such as UW Climate Analysis or Biological Network Motif, require 3 to 5 millions of Agents during their execution. This shows us that the system requires terabytes of memory which none of the computing nodes can afford. This project introduces an agent population control mechanism that restricts the number of active agents in the system and serializes new incoming agents into byte streams for later use. In addition, the library communicates with user application in each iteration cycle and transmits data. The overhead of this communication causes performance decrease in terms of both excessive memory consumption and high

CPU usage. We also address this issue by implementing a practical way of doing recursive method calls along with executing spawn, kill, and migrate processes without sending data back to user application in each iteration.

# TABLE OF CONTENTS

# ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my supervisor, Prof. Munehiro Fukuda, for providing me an opportunity to contribute his research and to work as his Graduate Research Assistant. My sincere appreciation also goes to Mr. Matthew Sell for sparing his time to help me setting up the experiment environment.

# Chapter 1

# INTRODUCTION

## *1.1 Agent-Based Algorithms & Multi Agent Spatial Simulation Library (MASS)*

Due to rapid rise of cloud technologies and related services, many technology companies including startups, began providing applications that handle big data. A number of software programs have been developed to serve this purpose, such as Spark or MapReduce. These applications scan through the entire data set in parallel to return desired result sets. However, unlike Spark or MapReduce, some scientific-computing related applications may use multi-dimensional arrays to represent this big data. Even though the memory is large enough to hold entire data simultaneously, scanning through all the values to find proper result sets may cause performance issues. Agent-based algorithm is one of the useful solutions to prevent this performance decrease.

Agent-based algorithms have a great advantage in terms of performance. Agents are capable of comparing the data at any index of the array with its neighbors. This helps them migrate to a neighbor index that will most likely to be in the final resulting set. Therefore, unrelated portions of the data set are not scanned which results in performance improvement.

Parallelism is very important for big data analysis. Libraries that are developed using parallel computational algorithms can decrease the execution time of the applications significantly[19]. Taking this fact into consideration, in 2010, Prof. Munehiro Fukuda began his research of developing a parallel-computing library, Multi-Agent Spatial Simulation (MASS) Library, in order to bring a new perspective overcoming performance challenges in big data analysis by making use of parallelism.

As described in the project description[6], MASS is a new parallel-computing library for multi-agent and spatial simulation over a cluster of computing nodes. The library uses two important concepts; Agent and Place. Place represents each array index of the given data set. For simulation purpose, the user application treats Place objects as locations. In addition, if this user application aims to conduct data analysis, then it searches through the values that Place objects have and retrieve unique ones. Places remain during the whole of simulation or data analysis process. Agent instances execute a set of instructions to perform actual computation. They can spawn new child Agent instances, migrate between Places, or get terminated. In each periodic cycle, Agents communicate with each other to exchange data.

More than 30 students have improved this library since its initial release and some of them made significant changes on MASS Java version during their capstone project or thesis process.

### *1.2 Problem Definition*

Agent-based algorithms, such as the MASS library, have some challenges as opposed to their advantages stated in section above. These algorithms require too many agent synchronization; the number of agents in the system can expand excessively; and during the execution agents are spawned or terminated many times.

C++ is faster and more efficient language than Java. The reason is that less execution time is required when a set of instructions are run in C++[13]. Therefore, developers intend to implement their applications with C++ when they are dealing with simulation. However, speed is not the most important factor in terms of application performance for some applications such as UW Climate Analysis[25] or Biological Network Motif[21] when conducting big data analysis. These applications require certain features for desired execution that Java

can easily provide. For example, distributed memory is essential to spread multi-agents over different computing nodes and Java has many dynamic and configurable data grid computing infrastructures available online for the developers[5]. In addition, Java provides a good GUI that users can interact with the application easily. Ease of interaction improves the usability of the program and usability is one of the key factors that help both users and developers to succeed in completing their tasks effectively[1]. As a consequence, the MASS library project is also implemented in Java in addition to C++ version.

This paper focuses on memory efficiency and CPU performance of the MASS Java library for big data analysis. Agent synchronizations, the expansion of the number of agents, and agent spawn and termination processes increase the memory consumption of the MASS Java library drastically. This increase causes applications show very slow performance or even crash when Java Virtual Machine is out of available memory. In addition, the Agent data is transmitted from the library the user application in every single iteration. The communication overhead between Agents and the user application affects the execution time negatively. Therefore, this project implements the solutions, stated in below section, in order to improve memory usage and execution time of the MASS Java library.

### *1.3 Proposed Solutions*

The paper introduces three new versions of the MASS Java library to overcome each problem. These versions are: 1) Clean Synchronous, 2) Agent Population Control, and 3) Asynchronous Migration.

The current asynchronous agent migration implementation of the MASS library is in stable condition. One of the former grad students developed this feature as his capstone project[6]. However, Prof. Fukuda realized that the current implementation needs to be improved since it lacks proper comments and documentation. First version, clean synchronous, gets rid of all previous implementation about asynchronous agent migration. In addition, it also makes

sure that synchronous functionality does not get affected during this improvement process. This version also becomes a base point for the other two versions.

The MASS library allows agents to spawn child agents during the execution. Our findings, discussed in section 3.2.1, show that an agent instance use up to 1 MB of memory. Under some specific circumstances, the number of agents can exceed the threshold value and this causes system to fail due to insufficient memory space. For example, Biological Motif Network application[21] spawns 3 to 5 million agents during its execution and this means the system needs 3 to 5 TB of memory which is not affordable. Agent Population Control version introduces a control mechanism that restricts the agent-spawning process according to the number of agents alive in the system. If the system already hits its maximum capacity, then the new agents are serialized into byte streams and put in a queue. When there is available space, these byte streams are de-serialized into agent instances and let them run in the system.

Moreover, in synchronous version of the MASS library, the system communicates with user application in every iteration cycle. However, the overhead of this communication causes performance decrease. It is because the agents have to wait to start their next method execution until MASS finishes transmitting data with the user application. Therefore, in order to utilize computing nodes, the MASS library needs to implement asynchronous migration feature. The 3rd version, asynchronous migration, implements a practical way of doing recursive method calls along with executing spawn, kill, and migrate processes without sending data back to user application in each iteration. After the nth iteration, where n is the number of iteration specified by the user application, the MASS library transmits retrieved data to user side.

# Chapter 2

# LITERATURE OVERVIEW

## *2.1 Background*

The MASS library, Multi-Agent Spatial Simulation Library, is a parallel-computing library for multi-agent and spatial simulation over a cluster of computing nodes[6]. The library uses a multi-threaded communicating processes that are connected through TCP sockets. The number of threads is dependent on the number of CPU cores within the computing node because we want to change the scalability of the system for better performance evaluation. These threads are responsible for method calls and data exchange between Agents and Places. All user applications have to call MASS.init() before the computation starts and MASS.finish() once the data analysis process is done.

Agent and Place are the primary concepts for the MASS library. Place object represents an array index of the given data set. The entire matrix, which is the data set, is called as Places. Places are dynamically distributed over the cluster of computing nodes. Each Place object contains the actual data and neighbor Places information. Places are also capable of exchange information between each other. There is no restriction that limits the size of Places array and they can be initialized properly by calling a Places constructor.

Agent instances execute a set of instructions to perform actual computation. Each Agent object resides in a Place. Multiple Agents can exist within the same Place object simultaneously. Agents are capable of spawning child Agents, migrating to another Place or getting terminated once they finish their tasks. They are also eligible to interact with each other. When the user application calls Agents.callAll() method, the MASS library executes the

function of all Agent objects associated with a function id. Meanwhile, all Agent spawn, migration and kill processes are handled upon Agents.manageAll() call.

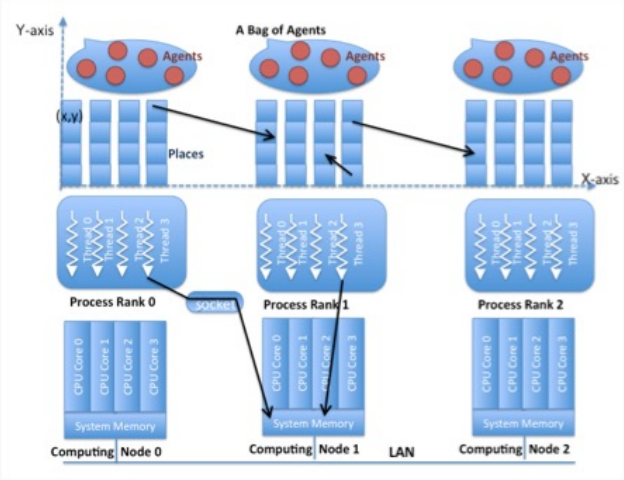


Figure 2.1: The MASS library execution logic.

The other key classes and functions to understand the basic terminology of the MASS library are as follows: [6]

- Places class: Places manages all Place elements within the simulation space. The class instantiates an array of user-defined Place objects shared among multiple nodes.

- Agents class: Agents are a set of execution instances made up of user-defined Agent objects.

- MASS.init(): Each process, running on a different computing node, spawns the same number of threads as that of its local CPU cores, so that all threads can access places and agents.

- MASS.finish(): Each process cleans up all its threads as being detached from the Places and Agents objects.

- Places.exchangeAll(): Calls the user-defined method specified with a function id of all destination cells.

- Places.callAll(): Calls the user-defined method specified with a function id of all array elements as passing an Object argument to the method. Done in parallel among multi-processes/threads.

- Agents.callAll(): Calls the method specified with a function id of all Agents as passing an Object argument to the method. Done in parallel among multi-processes/threads.

- Agents.manageAll(): Updates each agents status, based on each of its latest migrate(), spawn(), and kill() calls. These methods are defined in the Agent base class and may be invoked from other functions through callAll. Done in parallel among multi-processes/threads.

- Agent.spawn(): Spawns new Agent objects, as passing arguments[i] to the i-th new agent upon a next call to Agents.manageAll().

- Agent.kill(): Terminates the calling Agent upon a next call to Agents.manageAll().

- Agent.migrate(): Initiates an agent migration to a new Place upon a next call to Agents.manageAll(). More specifically, migrate() updates the calling agents index[].

The MASS library also provides a simple debugger GUI that visualizes all Place and Agent instances. The user can easily update the number of Place and Agent values before execution. In addition, they can pause the execution, observe the simulation behavior and then continue the execution again.

### *2.2 Related Work*

As it is mentioned in section 1.1, Spark and MapReduce are two important examples of software tools that conduct big data analysis. Apache Spark[23] supports both batch, stream,

and iterative processing. It also lets user applications to combine these processing techniques depending on their algorithm[14], [26]. Spark is one of the leading tools in iterative processing which is time consuming because of multiple read and write operations. This indicates that Spark mainly focuses on conducting high-speed analysis. MapReduce implements two major methods, Map() and Reduce(), that breaks the data into tuples and then integrates these tuples into smaller tuples. However, this framework is batch-oriented. As a consequence, MapReduce mostly focuses on background processing instead of high-speed analysis.

Both Spark and MapReduce frameworks keep data within disks in different types of storage elements such as key-value pair text files, CSV (comma separated values) or SQL schemas. This type of data representation lets these frameworks ignore the size of data set since a disk have way more space than a memory does. However, such design brings some disadvantages, too. First, they cannot easily examine the relation between data portions since the data objects are not allocated within the memory. Second, they do not keep track of specific data items during analysis. Third, application development using Spark and MapReduce is not easy. Even though they are powerful frameworks in terms of high-speed analysis, the developers need to have certain knowledge before starting implementation.

In addition, there are few researches that focused on communication overhead between agents and the overall system including user application. One of the studies[17] searched for an optimal communication topology to redirect messages efficiently to finish communication with least number of iterations. Also, in one of the studies, two framework services are introduced that the number of messages between nodes by implementing object-based middle agent services and dynamic agent distribution[20].

The MASS library is aimed to be developed as a user-friendly agent-based parallel computing library that allocates major data structures, Agents and Places, within memory to conduct big data analysis. Well-known algorithms and frameworks mainly focus high-speed

execution and researches aim to reduce the overhead between agent-to-agent communication. Therefore, as long as I did my investigation there is no such study that directly focuses on memory improvement techniques or agent-to-user process communication overhead issue for big data analysis. In this project, we address these techniques as well as agent-to-user overhead issues by implementing agent population control mechanism and asynchronous migration algorithm.

# Chapter 3

# DEVELOPMENT METHODS

## *3.1 Clean Synchronous Version*

Synchronous and asynchronous versions are the two execution types that the MASS library currently supports in the Java version. The term synchronous version indicates the logic where all Agent instances execute the requested functions and move to another specified Place upon each Agents.callAll() & Agents.manageAll() function calls respectively. The user application calls these functions in this order. Therefore, the execution of Agents.manageAll() function waits for the completion of Agents.callAll() execution. This wait causes a huge overhead. Agents neither start executing the requested functions nor finish these tasks at the same time. The reason is that, some of these Agents finish their execution tasks fast and they become ready to migrate to another place. However, since the process is synchronous, all Agents need to be done with their processes before any of them can move on to migration phase. In order to get rid of this overhead and utilize the CPU performance of the MASS library, one of the former graduate students developed the asynchronous version[6]. This version gets his name from the implemented feature that is asynchronous migration. The user applications do only single Agents.callAllAsync() call instead of multiple Agents.callAll() and Agents.manageAll() calls. The logic simply is letting Agents, which finished their tasks, migrate to another Place without waiting for the completion of the tasks of other Agents. Consequently, this feature keeps the overhead of communication at minimal level and utilizes the CPU time.

Documentation and commenting are essential for continuous software development[27]. Even though the MASS library supports asynchronous migration, the implementation lacks of

proper documentations and comments. Hence, it is hard to understand the exact behavior of the system, or to verify that both the development logic and implementation are consistent.

Documentation should contain essential information about the developed software system[27]. Documentation can simply be considered as a reference book for both the current and future developers. This manual is a powerful tool that provides healthy communication among all software engineers. It includes software requirements, architectural designs, technical information, such as code or APIs, and end-user manuals. Javadoc, Doxygen, ROBODoc and Appledoc are a few example documentation tools that help developers generate meaningful and readable documentations[22]. Commenting is an important practice of coding standards. Proper commenting helps developers understand the key logic of a code block that performs complex computation[3].

In order to address this insufficient documentation and commenting issue, we conducted code reading sessions. In these sessions we went over all the implementation about asynchronous migration and added necessary comments to the code blocks. In addition, we updated some portions of the code to make them more readable. However, in the end of all these sessions we came to the conclusion that the asynchronous migration implementation is too complicated. Hence, this makes impossible for developers to understand all the logic in detail. As a consequence, we decided to take out all asynchronous migration implementation and to improve the synchronous version to make the MASS Java library more readable and understandable.

### *3.1.1 System Design*

The asynchronous migration implementation introduced new classes to the project repository as well as modified the existing ones. We investigated develop branch that supports the asynchronous migration feature, and reverted changes without disturbing the synchronous functionality.

This project develops a new branch, 'utku_clean', which gets rid of the asynchronous functionality. The procedure consists of three phases: 1) removing new implementation files, 2) editing affected existing files, and 3) testing for verification. The first two phases require investigation for every single commit that is done to the branch. The third phase is only about running existing user applications and verifying the process by comparing the original with the new results. This helps us make sure that the synchronous functionality does not get affected by code editing.

| Class Name | Status |
|---|---|
| Agent.java | Modified |
| AgentAsyncComparator.java | Removed |
| AgentList.java | Modified |
| Agents.java | Modified |
| AgentsBase.java | Modified |
| AsyncInputThread.java | Removed |
| AsyncOutputThread.java | Removed |
| MASS.java | Modified |
| MASSBase.java | Modified |
| Message.java | Modified |
| MThread.java | Modified |

Figure 3.1: Modified and Removed Classes.

The asynchronous version introduced new implementation files such as AsyncInput-Thread,java, AsyncOutputThread.java, and AgentAsyncComparator.java. These classes were only used when the MASS library is executed with asynchronous functionality. Therefore, they did not cause any threat to synchronous functionality and they were removed from

the repository. On the other hand, existing files, such as Agent.java, Agents.java, or Agents-Base.java, are used by both execution types. As a consequence, any changes in these classes affect synchronous execution as well. When reverting the changes back on these classes, each line was carefully modified one by one. Figure 3.1 shows which implementation files were removed or edited during the first and second phases.

'RandomWalk', 'QuickStart', and 'SugarScape' are our test applications that are specifically developed for the MASS Java library. After the first two phases are completed, the test applications were run with synchronous execution and their behavior was observed. These test runs lasted for a day. Since the results were as expected, all the changes were committed to 'utku_clean' branch.

### *3.2 Agent Population Control*

When the application calls Agent.spawn() function, the MASS library instantiates a new agent object and lets it run actively in the system in the next simulation cycle. The library lets user applications spawn as many agents as they request. However, each computing node is running on Java Virtual Machine (JVM) and it has a limited available memory. Even though this value depends on the host environment and it can be extremely large, an average desktop computer has a dedicated memory between 2 to 8 GB in todays technology[2].

Previous tests with Biological Network Motif[21] and UW Climate Analysis[25] revealed that an agent instance consumes nearly 1MB of space in the memory and causes huge negative impacts to computation performance when number of agents hits a large value. Unfortunately, these applications require more than a million of agents in order to analyze the data accurately. However, when they request the MASS library to spawn millions of agents, JVM attempts to use TBs of memory. Since this value is way beyond the limit, JVM throws a java.lang.OutOfMemoryError exception which forces the system to crash. This implies that the MASS library is vulnerable to a drastic change in the number of agents, which is referred

as agent population expansion in the remaining parts of this paper, and the library needs to keep its memory consumption value under control.

This paper introduces a new algorithm to restrict the number of active agents in the system. Depending on the physical capabilities of computing nodes, each node will specify the maximum number of agents that can run simultaneously within that node. When this value is hit, new agent spawn requests are collected in a queue. As soon as there is a room for a new agent, the head of queue is fetched and a new agent is allocated within the memory.

The proposed implementation logic also brings two problems. First, specifying the maximum number of agent is not an easy task. Even though the specifications of computing nodes are important, a user application might want to specify this number depending on the applications logic. Second, the queue itself consumes memory space. Therefore, the queue size should also be taken into consideration carefully and this value should not affect the memory consumption extremely.

The implementation logic offers to make use of serialization techniques to achieve this problem. Serialization of an object provides the system to gain huge advantage in terms of memory consumption. On the other hand, this process brings additional CPU time for the execution. When conducting big data analysis, since the MASS library needs to utilize memory efficiency to prevent system failures, this additional CPU usage is a favorable trade-off. Section 3.2.2. discusses the actual test results about agent memory consumption and serialization CPU usage.

### *3.2.1 Serialization*

According to Microsoft Developer Network (MSDN)[9], the method of transforming objects into byte streams is called serialization. These byte streams help developers store the objects in persistent storage or transfer them through the network. The intention of this process is

conserving the state of the object for later uses. When the object is needed, deserialization, reading byte streams and recreating the object, occurs upon a request by the application.

*Serialization Techniques*

Many different types of serialization techniques or formats are available to developers. This project has taken four of them into consideration for implementing Agent population control. Among these four techniques, the best-known and MOST widely used are XML[16] and JSON[12]. In addition to these, since the major language of development is Java, JDKs default serialization implementation is here considered as another option[18]. Finally, Kryo, an open-sourced and frequently supported serialization project, is also considered[4].

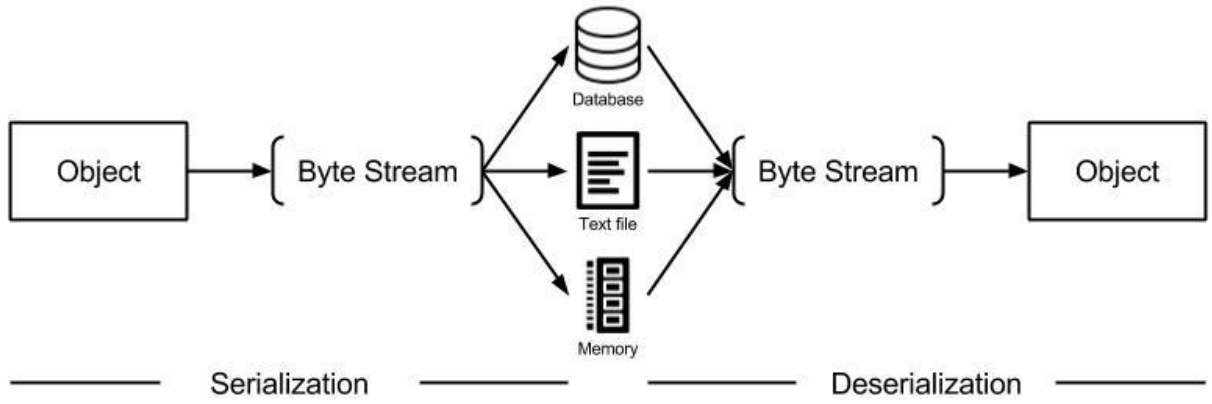


Figure 3.2: Serialization and Deserialization.

JSON is a powerful syntax that can handle very complicated and complex data structures, and it is used by JavaScript engine to parse data[12]; however, even though JSON is natively offered by JavaScript, systems do not necessarily require JavaScript engine to make use of JSON, since parsers of JSON exist in most platforms. Therefore, this feature makes JSON is a strong candidate for serialization.

XML has similar features compared to JSON[16]. XML parsers are also available for both platforms. The only downside of XML, when compared to JSON, is that XML uses tags for each field; consequently, this can slightly increase the output stream size.

Java serialization can be only used if the communicating platforms are running on Java[18]. It requires both sides of the communication channel to have class implementation. The overhead of passing an entire object generates problems in terms of memory. If, however, RMI is used as a messaging protocol, the development phase can be simplified.

Kryo is an open-sourced serialization technique that is popular in Java development community. its performance in terms of space usage is one of the best as compared to others.

Because of their unique advantages, this project implements Kryo and Java serialization techniques into the MASS library in order to create Agent population control mechanism. The following sections discuss these serialization techniques along with their advantages in detail.

*Kryo Serializer*

As it is described in the official Kryo website [4], Kryo is a fast and efficient object graph serialization framework for Java. The goals of the project are speed, efficiency, and an easy to use API. The project is useful any time objects need to be persisted, whether to a file, database, or over the network. Kryo offers serializers for most of the data structures in Java by default. However, it is easy to register a user custom class and keep the serialized object in the byte stream and read it back to the memory. Due to its characteristics of serialization, the overhead costs are significant CPU-time while serializing the object. On the other hand, deserialization takes very little time comparing to serialization.

There are certain steps that MASS needs to follow before starting Kryo serialization pro-

cess. First, all classes that a user application has implemented have to be registered. These classes include not only non-primitive, (a.k.a. object references), but also primitive data types such as int, boolean, and short. When registering these classes, MASS also needs to specify a unique id for each of them. These ids must be positive integers and higher than 9 since ids from 0 to 9 are already reserved for primitive types by default. For example, if a user application implements a class Example.java by extending Agent.java and if this class contains 'int' as a primitive data type, MASS has to execute following expressions:

```
// new Kryo instance
Kryo kryo = new Kryo();
// registering classes
kryo.register(Agent.class, 10);
kryo.register(Example.class, 11);
kryo.register(int.class, 12);
```

Furthermore, Kryo offers different types of serializers for each implementation architecture. Depending on user defined structures, an accurate serializer should be set for a correct serialization process. A desired serializer could be set as follows:

```
// set serializer
kryo.setDefaultSerializer(FieldSerializer.class);
```

Lastly, Kryo makes use of the Objenesis library to instantiate new objects during application execution[4]. The user application might want to customize the re instantiation process of their serialized agent objects. In this case, an instantiator strategy variable of Kryo instance needs to be updated. Note that there are few important points, such as not providing no-argument constructor, that might affect the choice of this strategy. Following expression shows how this variable could be assigned:

```
// sets instantiator strategy
kryo.setInstantiatorStrategy(new Kryo.DefaultInstantiatorStrategy(new
```

```
   StdInstantiatorStrategy()));
```

*Java Serializer*

Java object serialization is introduced in JDK (Java Development Kit) version 1.1 almost two decades ago. By default, Java has its own algorithm to serialize the objects. However, a developer can implement a Java class that could override this algorithm and change the way to serialize objects[8]. As it is mentioned in Oracle website, When an object is serialized, information that identifies its class is recorded in the serialized stream. However, the class's definition ("class file") itself is not recorded. It is the responsibility of the system that is deserializing the object to determine how to locate and load the necessary class files[8]. Therefore, if default algorithm is overridden, these class files must be introduced to Java properly.

Since the primary development language and runtime environment is Java, no external library needs to be imported. All necessary components exist in Serializable class that is provided by JDK. Simply creating an instance executing basic expressions are enough to perform serialization process.

*Challenges in Serialization*

MASS was not ready to use both serialization techniques due to three issues. One of them is Kryo-specific but the other two are common for both Kryo and Java. These issues are class registration, lack of class serialization and having one non-serializable class.

MASS initially had one no argument constructor and another one that takes command-line style expression, number of nodes and number of threads as arguments. However, as it is discussed in Kryo section, the MASS library must register all classes that are going to be serialized in the beginning. Therefore, this project introduced a new constructor along with

necessary getter and setter functions for these classes. If the user application wants MASS to use Kryo serialization, then the application needs to call this new constructor and pass all classes that Agent class makes use of. Even though this sounds a very simple process, the classes that seem very trivial might be overlooked. For example, even if a very common data type such as nxn integer array is used somewhere in the implementation, then MASS needs to register Integer[][].class class type to Kryo instance.

Second issue was a trivial one. When an object serialization process begins, all classes that are referenced within this object class must be serializable. In MASS, Agent class contains a Place variable that holds the pointer to the Place object that the Agent currently resides in. However, Place class was not serializable. Therefore, this project also implemented necessary code changes to make Place class serializable.

Moreover, in Java there are few classes which are not serializable due to their architectural design[7]. One of these classes is called SynchronizedSet which is a private class that is implemented within java.util.Collections class. MASS makes use of a synchronized set instance in Place class to store all active Agent objects. In addition, the value of this set is subject to continuous change even when Agent is serialized. Therefore, this set variable is no longer a part of serialization process. As soon as Agent is de-serialized the Place information of the object is updated so that it can have a reference to the synchronized set again.

### *3.2.2 Preliminary Work*

In order to understand how well Kryo performs for our project, we conducted memory and CPU tests. For this purpose, we picked our RandomWalk application. RandomWalk.java is edited in a way that it supports Kryo serialization and calculates how much space is used by an agent instance. Then, another implementation file is created, RandomWalkCPUTest.java, that serializes each agent object and then de-serializes them. During the execution the application computes how much additional CPU time is spent for each serialization cycle in

average.

| Test run # | Total agents size (bytes) | Agent size in average (bytes) | Total execution time for all serialization & deserialization cycles (ms) | Execution time for one serialization cycle in average (ms) | Execution time for one deserialization cycle in average (ms) |
|---|---|---|---|---|---|
| 1 | 348,273,408 | 964,746 | 29,263 | 3.514121 | 0.539179 |
| 2 | 348,362,168 | 964,992 | 29,322 | 3.519224 | 0.541002 |
| 3 | 348,328,488 | 964,876 | 29,314 | 3.518776 | 0.540857 |
| 4 | 347,998,870 | 964,324 | 29,128 | 3.513573 | 0.530221 |
| 5 | 348,331,268 | 964,904 | 29,308 | 3.518702 | 0.540664 |

Figure 3.3: Memory and CPU Test Results.

Figure 3.3 shows the results about the memory consumption and the CPU usage during test executions. As it is seen in the table, 361 agent instances are created within the application. According to results, an average value of an agent object size is roughly 965,000 bytes. While serialized agent size is around 300 bytes. As a consequence, when Kryo is used as the serializer for the library, the memory consumption of one agent object goes down to 0.025

Figure 3.3 also shows that one serialization cycle costs additional 3.5 ms CPU time whereas deserialization adds only 0.5 ms. Therefore, for a complete serialization/deserialization process, the execution time is increased by 4 ms. This implies that the MASS library can decrease its memory consumption significantly by adding an acceptable additional time to total execution time for each serialization cycle.

In order to verify Agent population control development, this project also introduces a new application called VonNeumann. This application is designed to abuse the number of agents that actively run in the system. In this application, an agent is spawned at the first index (0,0) of the Places array. In each iteration, agent looks available neighbors of the Place that it currently resides in. Each Place object has 4 neighbors: west, north, east, and south. These available neighbors are added into an ArrayList and then the list is shuffled. The agent migrates to the first available neighbor and spawns new Agent instances to other available neighbors. This process leads the MASS library to run with excessive agents due to quick increase in the number of active agents. Consequently, such application helps us to identify if Agent population control affects memory consumption significantly. The project compares the memory consumption values with and without serialization processes.

Figure 3.4: Sample run with arguments, array size: 100x100, total iterations: 400, interval: 40 at times '120', '200', and '280' respectively.

Figure 3.4 shows the screenshots from VonNeumann application execution. The application is run for 400 iterations and the Place matrix size is 100 by 100. At every 40th iteration, Agent information is collected and displayed on user interface. The Agent distribution over Places is shown in the figure at times 120, 200, and 280 respectively.

### *3.2.3 System Design*

The user application does not have any kind of restriction or limitation that prevent them calling Agent.spawn() function. The MASS Java library is obliged to initiate Agent spawn procedures upon every call. However, test runs and system fails show us that either user application or the library needs to consider the number of agents that are running simultaneously during the execution. This paper introduces a new algorithm that restrict this value for effective memory use.

The agent size limit needs to be set before the MASS library executes first iteration cycle. Either the user application passes this value as an argument while it is initializing the library or MASS assigns the default value to the variable, MAX_AGENTS_PER_NODE. There is a dedicated MASS constructor that serves this purpose for user applications. If default constructor is called, then this indicates the user application agrees to start execution using default value.

```
// default constructor
MASS.init();

// dedicated constructor that specifies max agent value per node as 100
MASS.init(100);
```

Each AgentsBase instance used to keep MAX_AGENTS_PER_NODE value to assign ids to Agent instances. New algorithm lets system make use of this value to understand if the limit is reached so that Agent population control mechanism steps in. When Agents.manageAll() function is called, the library checks Agent spawn, kill, and migrate requests in this order. No development changes made for migrate process since its functionality does not depend on the MAX_AGENTS_PER_NODE variable. It is simply responsible for moving Agents from one Place to another by removing from or adding to Agent bag list of the Place and updating Agents indexes. During spawn and kill check, actions are dependent to Agent population

control logic since these actions affect the number of active Agents running in the system.

This project implements three new classes for Agent population control: 1) AgentSpawnRequestManager, 2) AgentSpawnRequest, and 3) AgentSerializer. When there is any spawn or kill request in the list, these classes take necessary actions to keep the number of active Agents in the system as close as possible to value of MAX_AGENTS_PER_NODE variable but prevents this number to exceed the value. All changes are pushed to the branch, utku_agentPopulationControl.

*AgentSpawnRequestManager*

The system is enhanced with a decision-maker which is responsible for evaluating the runtime environment variables such as number of active Agents, number of threads etc. when there is any Agent spawn or kill request. This decision-maker, AgentSpawnRequestManager, checks whether there is enough space for new Agent allocation or a serialized Agent that waits to become active in the system.

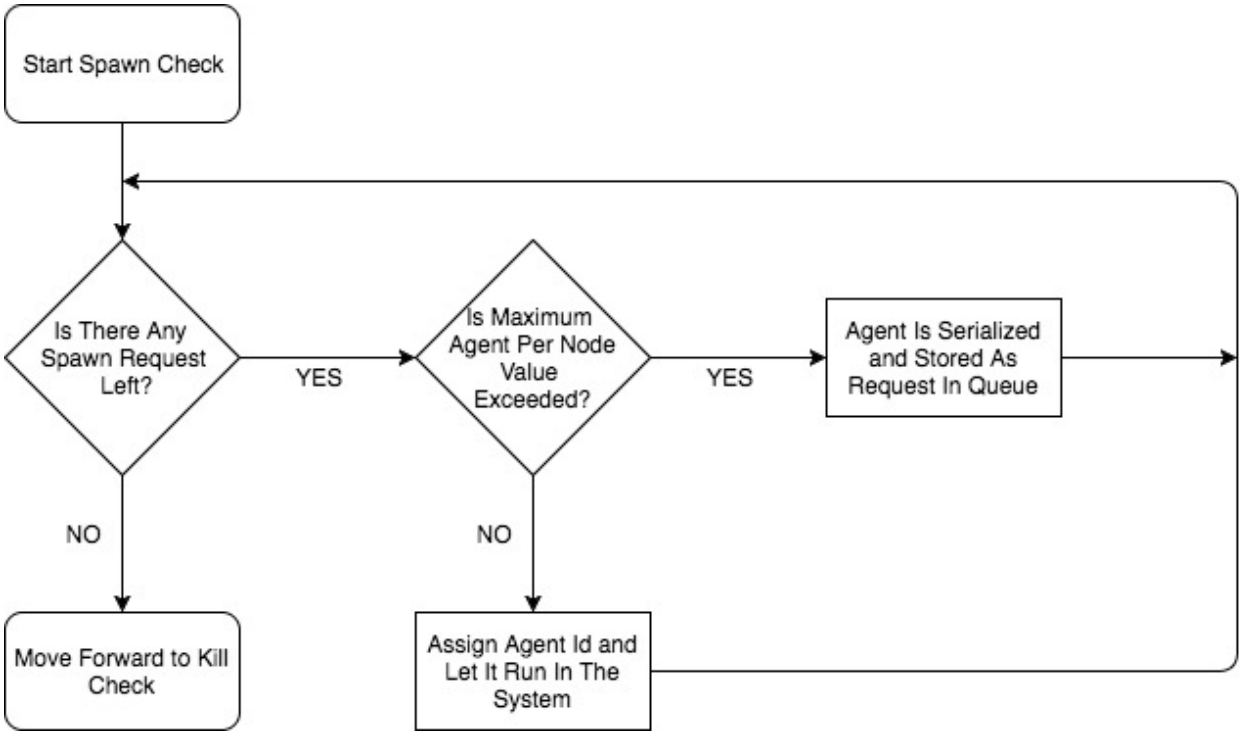


Figure 3.5: Spawn Flow Diagram.

In each spawn cycle, the new Agent object is allocated regardless of the current number of active Agents. The library passes necessary information, such as index array and Place instance, to this new Agent object along with spawn arguments. At this moment, AgentSpawnRequestManager.shouldAgentRunInTheSystem (Agent agent, int currentActiveAgentSize) method checks if there is enough space for the Agent in the system. The method returns true and the library lets Agent become active if the limit, MAX_AGENTS_PER_NODE, is not yet reached. Otherwise, the manager informs AgentSerializer class to serialize this Agent and retrieves the serialized Agent identifier. This identifier value must be unique for each serialized Agent object to prevent conflicts. Therefore, System.currentTimeMillis() followed by serialization extension, is used as this unique identifier. Then, an AgentSpawnRequest instance is allocated for recently serialized Agent object. The manager has to keep track of all serialized Agents in First In First Out (FIFO) order. Hence, the manager contains a

queue variable, initialized as a LinkedList, that holds these AgentSpawnRequest instances in desired order. After the manager finishes serialization process and stores the request in the queue, it returns false to AgentsBase class to inform that the Agent is serialized and the manager does not let it run in the system at the moment.

The kill cycle results in a decrease in the number of active agents in the system. Consequently, some space becomes available for serialized Agent objects. Before AgentSpawnRequestManager steps in, Agent termination process goes as usual. The Agent is removed from the Places Agent bag and the Agent list of AgentsBase instance. As soon as the Agent removal processes are done, AgentsBase calls agentSpawnRequestManager.getNextAgentSpawnRequest() method. This method checks if the manager has any available AgentSpawnRequest object in the queue. If there is not, then the method returns false and no more actions are taken. Otherwise, the manager fetches the head object of the queue and passes the unique id to AgentSerializer instance. AgentSerializer de-serialize the file with this unique id, builds the Agent object, and sends it back to the AgentSpawnRequestManager. The manager informs AgentsBase de-serialized Agent object is ready to run in the system. AgentsBase takes care of the final steps; it sets the Place variable of the Agent, adds the Agent object to the Places Agent bag, and adds the Agent to the Agent list of AgentsBase instance. After these final steps, Agent population control mechanism finishes its work for this iteration cycle and the system moves onto migration process.

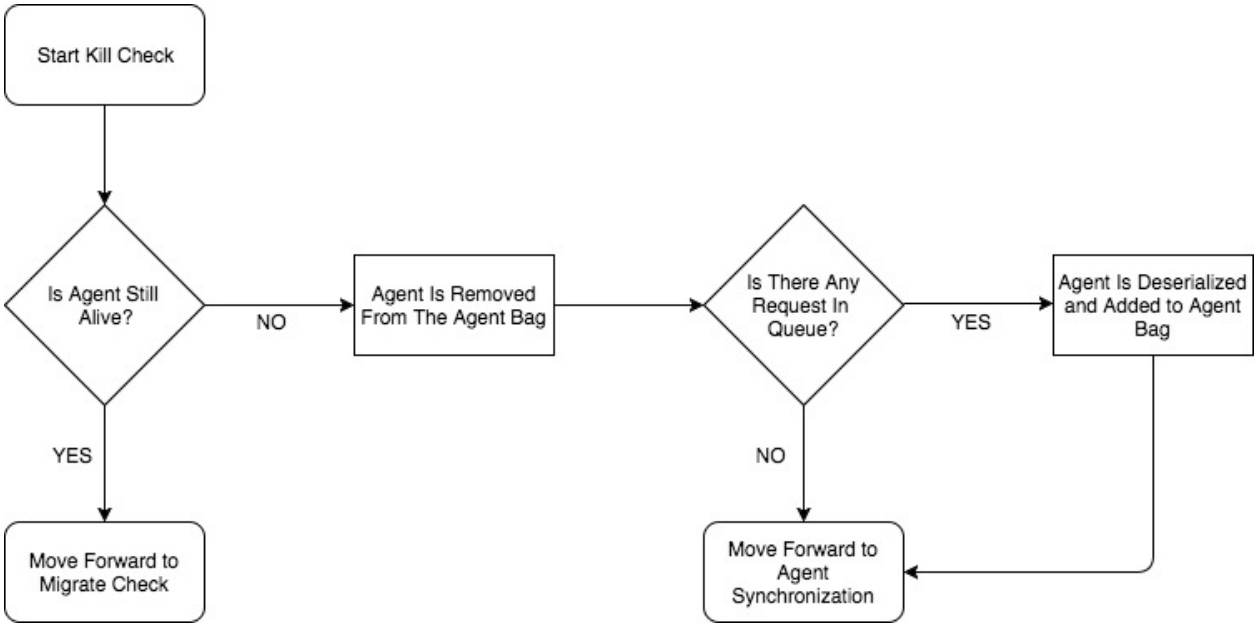


Figure 3.6: Kill Flow Diagram.

*AgentSpawnRequest*

AgentSpawnRequest is the custom class that simply contains the identifier of recently serialized Agent object alongside the index where the Agent should be injected into after deserialization. Depending on the user application needs, this class can easily be extended with new variables. However, it is important to mention that declaring more variables within AgentSpawnRequest object results an increase in size. Since AgentSpawnRequestManager holds a queue that consists of AgentSpawnRequest objects, if an average AgentSpawnRequest object size grows, this also affects the memory consumption of the manager instance. Therefore, in order to get best results in terms of memory improvement, the request object size should be kept as small as possible.

*AgentSerializer*

Previous development tasks caused disorganization in the MASS library implementation. The problem that is described in section 3.1, is one of the significant examples about this

issue. In order to prevent repeating same mistakes that we experienced with previous the MASS library development phases, this paper adopts single responsibility principle for all implementation of classes. Single responsibility principle is one of the five fundamental principles (SOLID) of object oriented programming in software engineering[11]. According to this principle, each implemented class is responsible for single functionality. AgentSerializer class file is the best example of single responsibility principle use among all the implementation files that this project introduces.

AgentSerializer is responsible for performing all the serialization and deserialization tasks. The class implements both Kryo and Java serialization techniques. It also contains the default value for maximum number of agents per node. When dedicated MASS constructor for Agent population control is called, this value is updated. AgentsBase class accesses the value during MASS initialization process.

It is important to mention that AgentSerializer is a singleton class. This helps library to get rid of each AgentSerializer instance initialization and deallocation overhead. However, since the MASS library runs on multiple computing nodes, same AgentSerializer instance might be accessed by more than one computing node simultaneously. As a consequence, all method executions must be thread-safe including singleton instance creation. All variables have local scope within serialization or deserialization method in which they are declared. In addition, AgentSerializer implements an inner class, LazyHolder, for an effective thread-safe singleton cache.

### *3.3 Asynchronous Migration*

The changes, that are discussed in section 3.1, took asynchronous feature out from the library. Hence, the MASS Java library begins supporting only synchronous functionality. Even though these changes gets rid of insufficiently documented and commented implementation, library still needs a simplified version of asynchronous functionality.

In synchronous version of the MASS library, the user application and the library communicates each other after every Agents.callAll() or Agents.manageAll() function calls. This communication results in data transmission from the library to the user application. The most significant advantage of this behavior is that the user application is notified in every state of the simulation or data analysis process. Therefore, the accuracy of final result is optimal. However, the communication overhead is too large. It causes huge negative impact on CPU performance. In addition, the transmitted data consumes memory space as well.

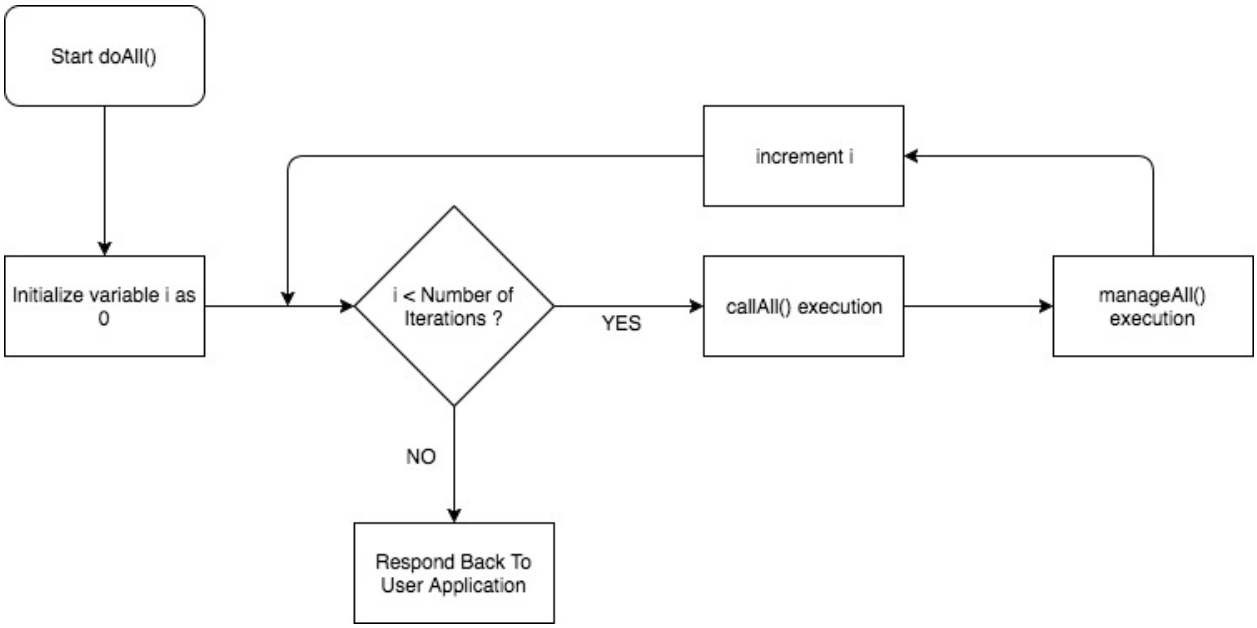


Figure 3.7: Asynchronous Migration Flow Diagram.

As a third task, this project introduces a new asynchronous migration algorithm that improves CPU usage of the system. In new logic, instead of transmitting Agents data to the user application in each iteration, the library returns the collected information in every nth iteration. The iteration number, n, is specified by the user application by passing the value during function call.

### *3.3.1 System Design*

When running an application with the MASS library, the user application needs to follow certain steps. The application calls MASS.init() function and then initializes the Places array and Agents objects. Once these processes are successfully done, the application goes into cyclic simulation. In cyclic simulation, Agents.callAll() and Agents.manageAll() functions are called in every iteration. Both functions respond back to the application when their executions are complete.

New algorithm implements a separate function, doAll(), that performs recursive method calls along with executing spawn, kill, and migrate processes without sending data back to user application in each iteration. Three new constructors are implemented to support method overloading for callAll() function.

```
// doAll method that gets function id and number of iterations as parameters
public void doAll(int functionId, int numberOfIterations)

// doAll method that gets function id, argument for child agents and number of
    iterations as parameters
public void doAll(int functionId, Object argument, int numberOfIterations)

// doAll method that gets function id, arguments list for child agents and
    number of iterations as parameters
public Object doAll(int functionId, Object[] argument, int numberOfIterations)
```

The doAll(int, int) function lets user application to call a specific function for n number of iterations specified by the second parameter. Next, doAll(int, Object, int) function is called by the application when it needs to share some type of object with child agents. Finally, doAll(int, Object[], int) function is also used to share an object but it is mostly used for collecting information from Agents during data analysis process.

The most significant advantage of this design is that the great improvement in CPU time. Therefore, the process ends way faster than synchronous one. However, the user application is not able to learn about Agents behavior within one doAll() execution cycle. For example, we assume that the iteration value is specified as 3. Once the application calls doAll() method, it does not have any information about at which Place that Agents reside in during first and second consecutive Agents.callAll() and Agents.manageAll() calls. Therefore, the accuracy of final result can be affected negatively.

# Chapter 4

# EXPERIMENTS & RESULTS

## *4.1 Environment Setup*

In this project, the experiments are conducted with two applications which are RandomWalk & VonNeumann. RandomWalk is a simple random process application where Agents migrate from one Place to another available neighbor Place in each iteration cycle. As discussed in 3.2.2 in VonNeumann, an Agent is allocated in the first indexes of the Places array and then this Agent migrates to another Place while it spawns child Agents to neighbor Places.

It is also important to mention that all experiments were conducted using the 16 machines in Linux Lab provided by University of Washington and located at UW1-320. These machines are named from UW1-320-00 to UW1-320-15. Each machine has 4 cores and 16GB of RAM.

## *4.2 Clean Synchronous Version*

We conducted the experiments for this version just to verify synchronous functionality did not get affected with the changes. For this purpose, we implemented a new version of RandomWalk application that use the synchronous method calls of the MASS Java library. In addition, we also ran VonNeumann application to check the behavior of recently developed user application that makes use of the library.

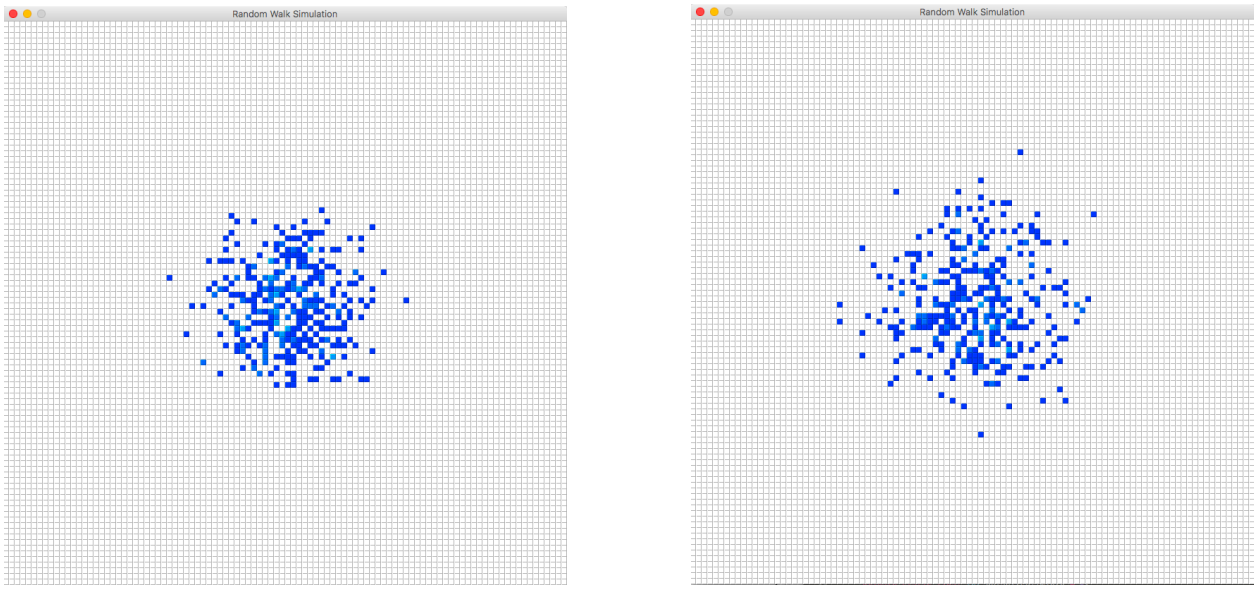


Figure 4.1: RandomWalk test run, array size: 100x100, total iterations: 100, interval: 5 at times '40' and '80' respectively.

Figure 4.1 shows that the execution of RandomWalk having 100x100 Places matrix, 361 agents, 100 iteration cycles at time 40 and 80 respectively.

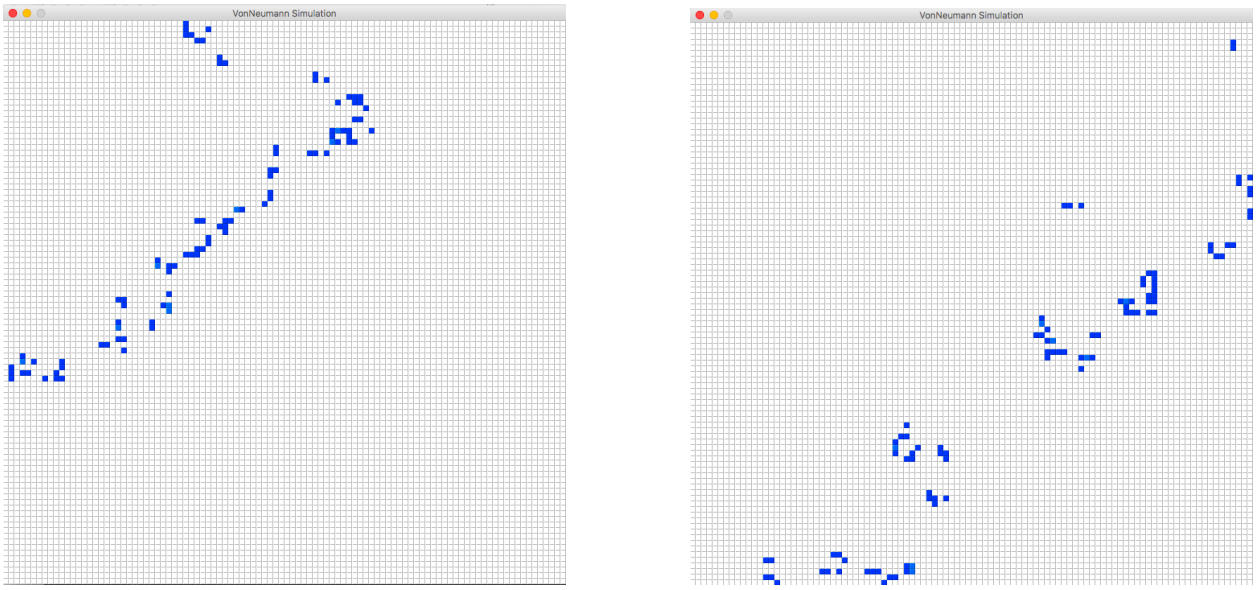


Figure 4.2: VonNeumann test run, array size: 100x100, total iterations: 400, interval: 10 at times '150' and '250' respectively.

Figure 4.2 shows that the execution of VonNeumann having 100x100 Places matrix, single Agent, 400 iteration cycles at time 150 and 250 respectively.

The test results helped us to be sure that the synchronous functionality of the MASS Java library is working properly without having any problems. Since this version is the base point for the other two versions, it is good to know that other version implementations are built on a working and stable condition.

### *4.3 Agent Population Control*

VonNeumann is a good example application in order to understand if Agent population control mechanism is working properly. Since the number of Agent increases quickly, we can easily observe the change of JVMs memory usage data and see if our algorithm improved the memory consumption of the library.

VonNeumann is a good example application in order to understand if Agent population control mechanism is working properly. Since the number of Agent increases quickly, we can easily observe the change of JVMs memory usage data and see if our algorithm improved the memory consumption of the library. In order to get results quickly, we modified the VonNeumann application. In experiments, the application spawns Agents in each side of the array and Agents are being killed randomly during the simulation. The code that is used to trace memory consumption as follows:

```
// getting run time instance
Runtime runtime = Runtime.getRuntime();

// used memory is calculated by subtracting free memory from total memory
long usedMB = (runtime.totalMemory() - runtime.freeMemory()) / 1024 / 1024;
```

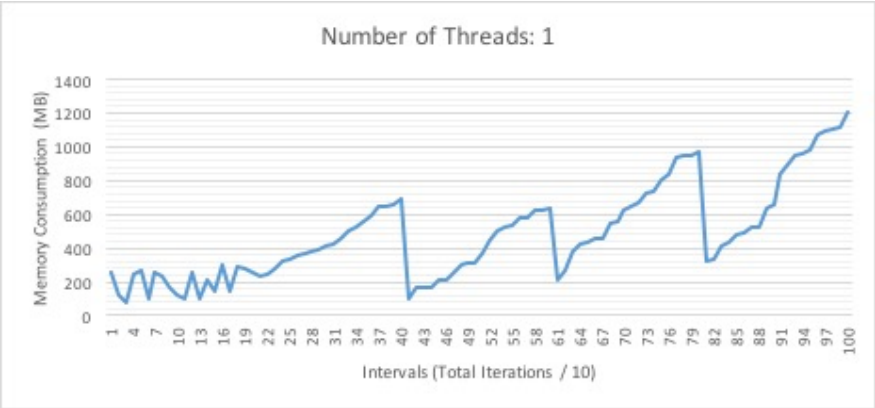


Figure 4.3: VonNeumann application memory consumption without serialization and with one thread.

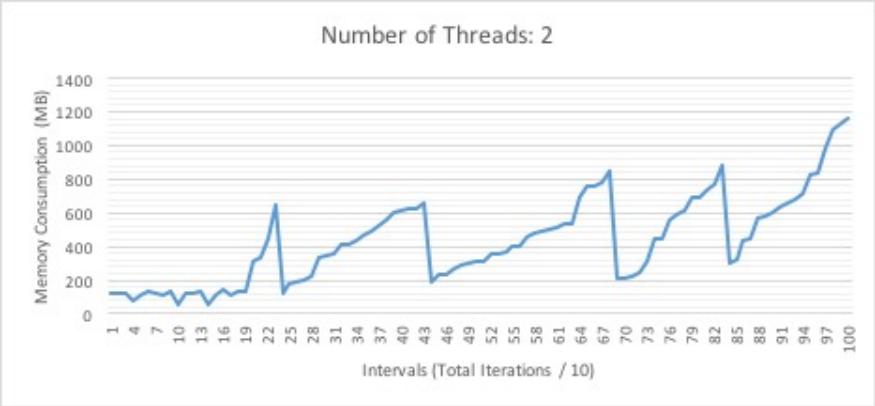


Figure 4.4: VonNeumann application memory consumption without serialization and with two threads.

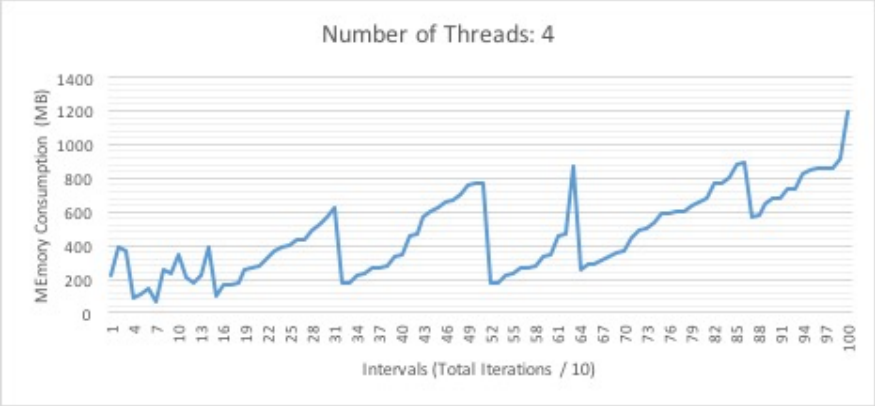


Figure 4.5: VonNeumann application memory consumption without serialization and with four threads.

We conducted the first group of experiments with VonNeumann application without the serialization. The application does not specify any number of maximum agents. The array size is 400x400 and the simulation ran for 10,000 iterations. Figure 4.3 shows that when there is one thread running, the memory consumption fluctuation happens constantly without any drastic change. Similar behaviour is observed when thread number is increased to 2 or 4. It is important to note that the expand of memory consumption does not stop in any experiment.

In the last few iterations, it is clear that the system already started using more than 1.2 GB of memory. The figures conclude that if simulation time increases, the Agent number also goes up, the memory consumption increases, and the system is likely to fail since the memory consumption value is not getting fixed.

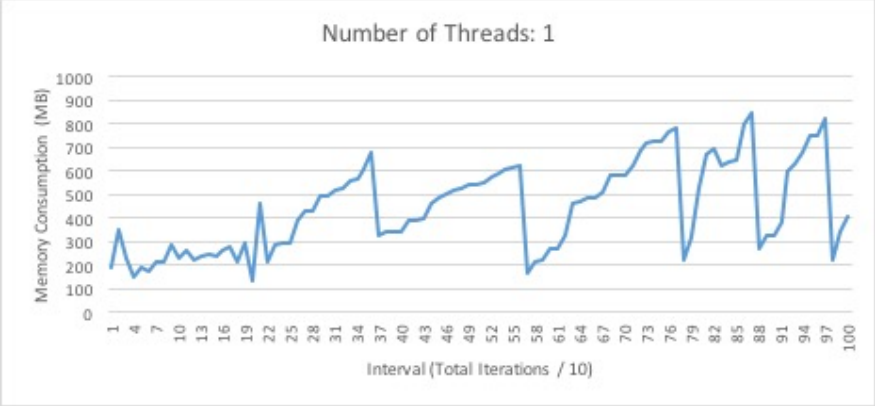


Figure 4.6: VonNeumann application memory consumption with serialization and one thread.

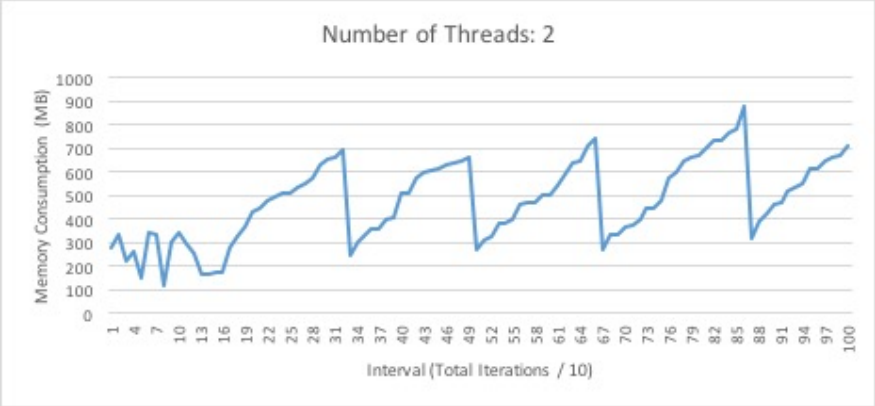


Figure 4.7: VonNeumann application memory consumption with serialization and two threads.

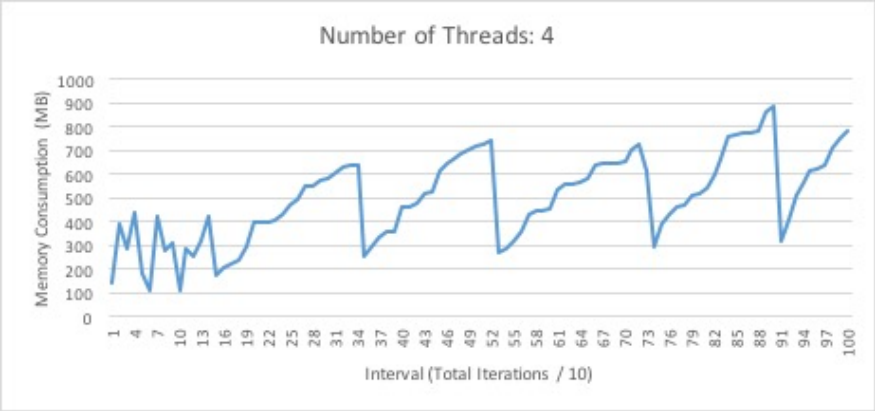


Figure 4.8: VonNeumann application memory consumption with serialization and four threads.

The second group experiments used Java serialization technique. In these group experiments the array size is 400x400 and they were run for 1,000 iterations as well. We specified the maximum number of Agents per node value as 100. Figure 4.6, Figure 4.7 and Figure 4.8 indicate that maximum memory consumption was around 850 MB. The important fact is that once this value is reached, the system uses memory does not get beyond this value in the rest of simulation.

When we compare the results we can conclude that a user application, VonNeumann, uses around 850 MB of RAM as maximum when serialization takes place. On the other hand, not-serialized version hits 1.2 GB of memory space and the memory consumption growth does not get fixed around any value. With these experiments parameters, it is seen that we gain around 300 MB memory space in the last stages of the simulation while we expected more. However, according to Oracle[10], By default, the virtual machine grows or shrinks the heap at each collection to try to keep the proportion of free space to live objects at each collection within a specific range. Therefore, the free memory value is subject to change depending on the JVMs heap size.

In conclusion, it is certain that serialized version helps us to restrict memory usage of the system and guarantee that it does not have any failure due to memory consumption. However, the gain in used memory space over time is dependent on JVMs behavior towards total heap size.

### *4.4 Asynchronous Migration*

Asynchronous migration experiments are conducted with RandomWalk and SugarScape applications. As described in section 3.3, in synchronous version, the user application uses Agents.callAll() and Agents.manageAll() functions while in asynchronous migration version it only calls Agents.doAll() function. Accordingly, in our experiments, we labeled the former as callAll and the latter as doAll. It is also important to note that the number of iterations value for doAll is set to five and log level of MASS is set to DEBUG.

In order to hit true parallelism, the applications need to make large scale computations that produce high CPU usage. When the computation intensity decrease, the overhead of communication between computing nodes becomes significant relative to the total execution time. However, in order to understand if asynchronous migration development improves the CPU usage independent of the intensity of computation, we conducted both small and large scale computation experiments with RandomWalk and SugarScape applications.

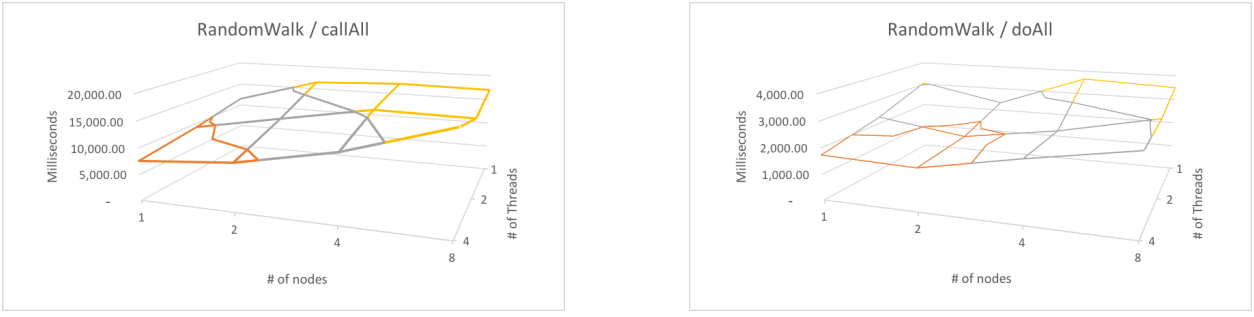


Figure 4.9: Small scale computation experiment results of RandomWalk application

RandomWalk application is very useful to find out the change in execution time of the system since the number of Agents is static. Therefore, we use this application to test our asynchronous migration implementation and learn how well it performs. In this RandomWalk experiment the matrix size is 100x100, there are 361 Agents and the simulation is run for 100 iterations.

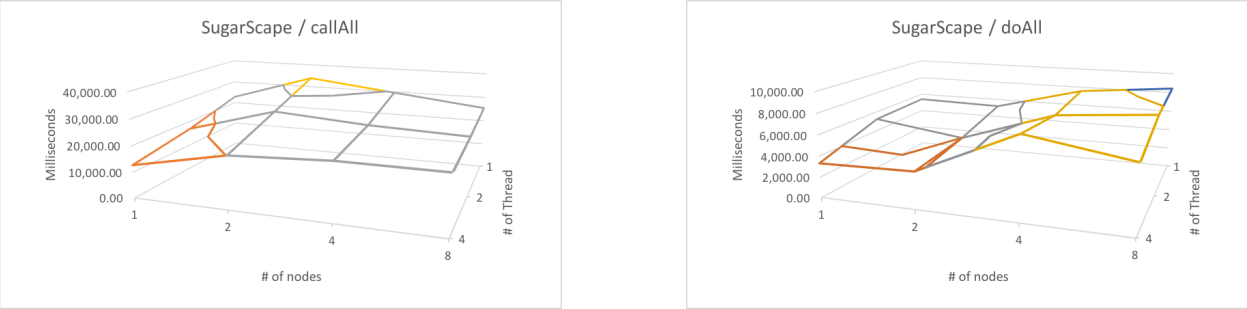


Figure 4.10: Small scale computation experiment results of SugarScape application

SugarScape application is another practical application to find out the change in execution time of the system. Since the Agents and Places that contain sugars are distributed uniformly, it is a very favorable application for parallel computing. In this SugarScape experiment the matrix size is 100x100, the number of Agents is initially 30 and the simulation is run for 100 iterations.

Although the CPU performance goes down as the number of nodes and the number of threads increase, small scale computation experiment results show that doAll execution provided significant decrease in CPU-usage.

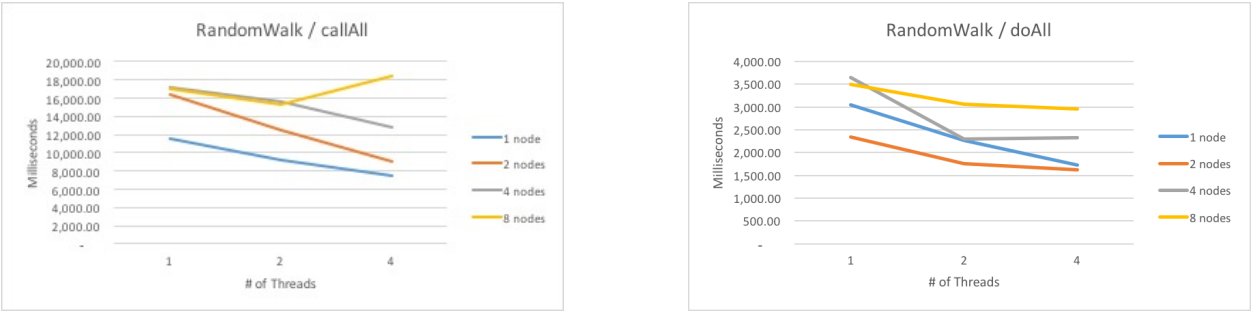


Figure 4.11: The effect of number of threads on callAll and doAll executions in small scale RandomWalk application.

The Figure 4.11 shows that when number of threads increase the execution time decreases but the change is little. However, as it is described above, this behavior is expected since the experiment is small scale. Regardless of this fact, doAll execution completes between 1,500 and 4,000 milliseconds whereas callAll execution takes between 7,000 and 20,000 milliseconds. The CPU improvement is around 79% in average.

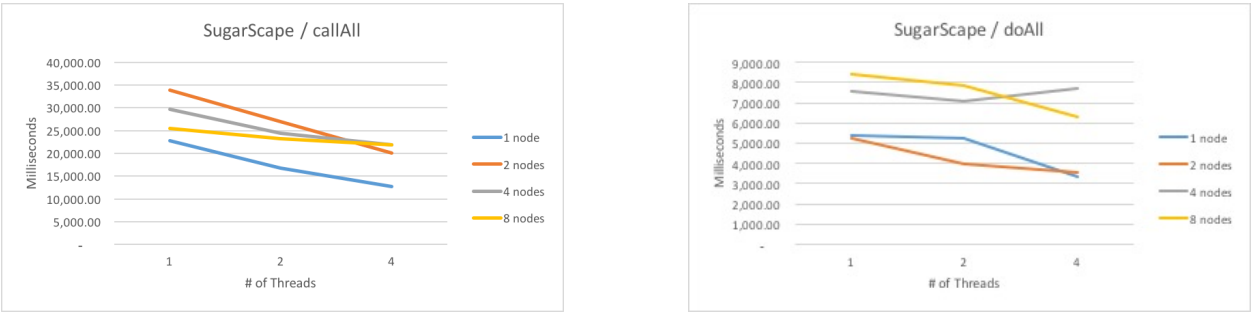


Figure 4.12: The effect of number of threads on callAll and doAll executions in small scale SugarScape application.

The Figure 4.12 shows that the SugarScape experiment results are pretty similar with RandomWalk. The decrease in execution time is little. In this experiment it is seen that doAll execution takes between 3,000 and 9,000 milliseconds while callAll execution takes between 10,000 and 35,000 milliseconds. The CPU improvement is around 72% in average.

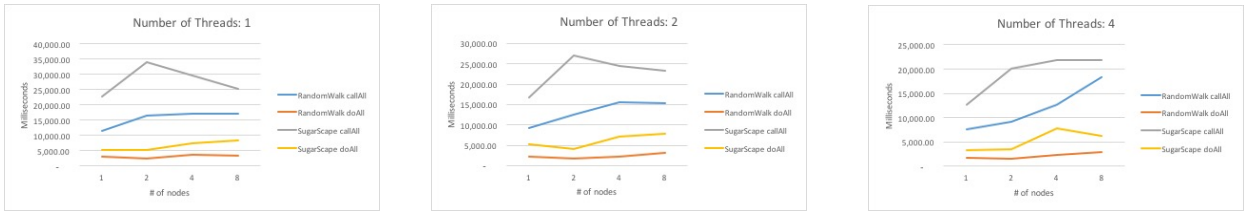


Figure 4.13: The effect of computing nodes on callAll and doAll executions in small scale on callAll and doAll executions in small scale SugarScape application.

The Figure 4.13 shows that execution in one computing node is better than two, four or eight computing nodes since the application computations are small scale. The overhead of communication between master and slave nodes is too high so that parallelism increases the CPU time. However, as It is mentioned above, regardless of this fact doAll has a huge impact on CPU usage.

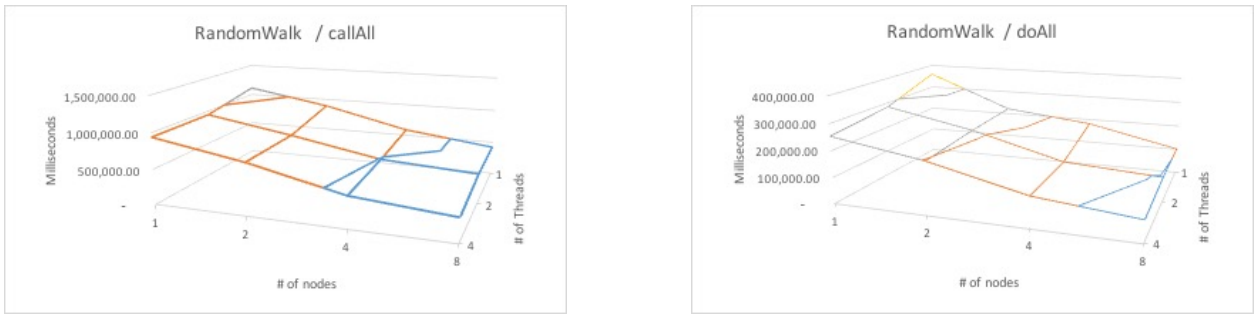


Figure 4.14: Large scale computation experiment results of RandomWalk application

Large scale experiment results are fairly different than small scale ones. This time, the RandomWalk application is run for 1000 iterations with 1521 Agents having matrix size of 200x200.

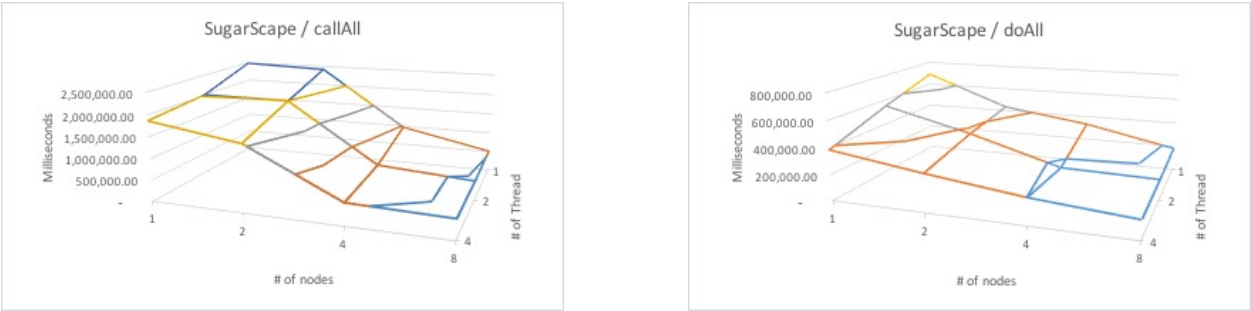


Figure 4.15: Large scale computation experiment results of SugarScape application

SugarScape application is also run for 1000 iterations with 480 Agents having matrix size of 200x200.

In this experiment, it is clearly seen that the CPU performance goes up as the number of nodes and the number of threads increase. In addition, the experiment results show that doAll execution provided significant decrease in CPU-usage.

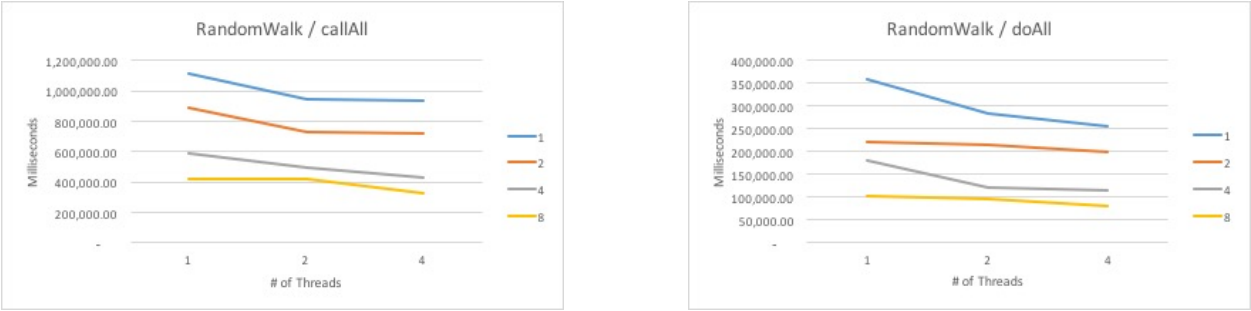


Figure 4.16: The effect of number of threads on callAll and doAll executions in large scale RandomWalk application.

The Figure 4.16 shows that when number of threads increase the execution time decreases and the change is significant. When the experiment is large scale computation we see better performance when the application is run in parallel. It is shown that doAll execution completes between 70,000 and 360,000 milliseconds whereas callAll execution takes between

380,000 and 1,100,000 milliseconds. The CPU improvement is around 74% in average.

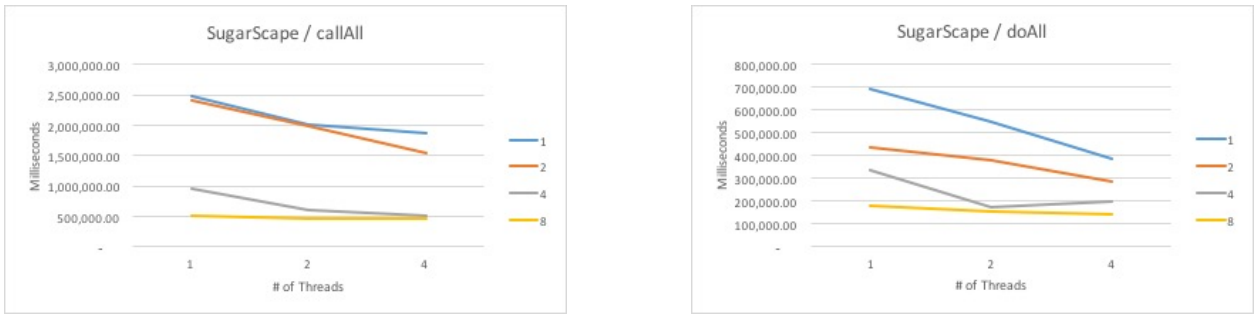


Figure 4.17: The effect of number of threads on callAll and doAll executions in large scale SugarScape application.

The Figure 4.17 shows that in this SugarScape experiment the change in execution time significant as well. It can be seen that doAll execution takes between 150,000 and 700,000 milliseconds while callAll execution takes between 500,000 and 2,500,000 milliseconds. The CPU improvement is around 69% in average.

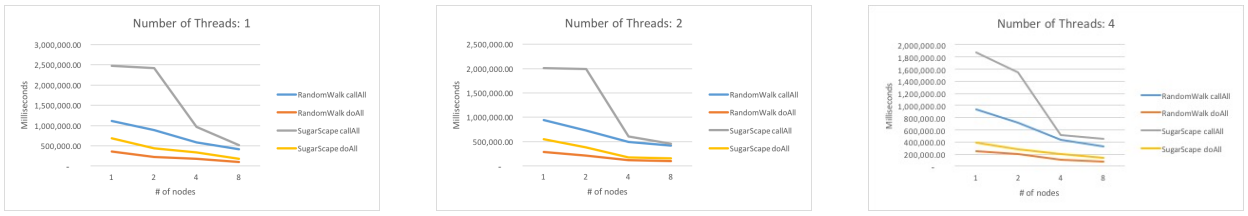


Figure 4.18: The effect of computing nodes on callAll and doAll executions in large scale on callAll and doAll executions in small scale SugarScape application.

The Figure 4.18 shows that execution in eight nodes are better than the others since the application is large scale. The overhead of communication between master and slave nodes is too low relative to the total execution time. As it is seen in any experiment result, doAll has a huge impact on CPU usage.

In conclusion, asynchronous migration improves the CPU usage between 69% and 79% when

it is compared to synchronous functionality. This also implies that there is huge communication overhead between the user application and the MASS library.

# Chapter 5

# CONCLUSION & FUTURE WORK

This project focused on memory efficiency and CPU usage of the MASS Java library for big data analysis and mitigation of communication overhead. We introduced three new versions for the MASS Java library that implements clean synchronous functionality, agent population control mechanism, and asynchronous migration. Our experiments with the applications, VonNeumann, RandomWalk, and QuickStart, have shown that clean synchronous version got rid of asynchronous functionality without breaking synchronous functionality; meanwhile agent population control significantly improved memory consumption and asynchronous migration substantially enhanced CPU usage and slightly improved memory consumption as well. We believe these versions will become good base points for future the MASS Java library development tasks to improve its performance or to prevent it from failing during execution.

This project certainly helped me gain more experience about parallel computing principles and performance improvement practices. I also learned about different serialization techniques such as Kryo project and built-in Java serializer. Developing user applications for the MASS Java library was another worthwhile task since I had the opportunity to observe the system behavior from user-side point of view. Working as the graduate research assistant for MASS project made me take initiatives, responsibilities and meet the deadlines. I am glad that I was able to complete my development tasks successfully and contribute on MASS performance improvement significantly.

As for the future work, our ultimate goal is to run large scale scientific applications, such

as Biological Network Motif or UW Climate Analysis, with MASS Java Library within a desired execution time. It has been also discussed that the MASS Java library needs to be publicly available to other developers so that they can develop applications using MASS or contribute to library implementation directly. There are many advantages of making MASS publicly available. First, such project might help students learn or professors to teach about the parallel computing concepts better. Second, developers can improve the MASS library implementation and improve the CPU usage, memory consumption, or code readability. Furthermore, they can identify shallow bugs and edge cases of this project. Lastly, if there is a significant issue about the library at any time, more developers can come up with more solutions than only a small group of developers can. Therefore, the possibility of finding more effective and faster solutions increases.

# BIBLIOGRAPHY

[1] An approach to improvement the usability in software products (PDF download available).

[2] Choosing the type and amount of RAM in a desktop PC.

[3] Coding techniques and programming practices - MSDN.

[4] GitHub - EsotericSoftware/kryo: Java serialization and cloning: fast, efficient, automatic.

[5] In-memory data grids - DZone big data.

[6] MASS: A parallelizing library for multi-agent spatial simulation.

[7] Serializable (java platform SE 7 ) - oracle.

[8] Serializable objects (the java tutorials > java naming and directory interface > java objects in the directory) - oracle.

[9] Serialization (c# ) - MSDN.

[10] Sizing the generations.

[11] Windows 8 OOPS principles (SOLID principles) sample in c# for visual studio 2010 - MSDN.

[12] Kereshmeh Afsari, Charles M. Eastman, and Daniel Castro-Lacouture. JavaScript object notation (JSON) data serialization for IFC schema in web-based BIM data exchange. 77:24–51.

[13] Asad Mahmoud Alnaser, Omar AlHeyasat, Ashraf Abdel-Karim Abu-Ein, Hazem Hatamleh, and Ahmed A. M. Sharadqeh. Time comparing between java and c++ software. 05(8):630.

[14] Ahsan Javed Awan, Mats Brorsson, Vladimir Vlassov, and Eduard Ayguade. Architectural impact on performance of in-memory data analytics: Apache spark case study.

[15] Jeffrey Dean and Sanjay Ghemawat. MapReduce: a flexible data processing tool. 53(1):72–77.

[16] J.-M. Le Goff, H. Stockinger, I. Willers, R. Mcclatchey, Z. Kovacs, P. Martin, N. Bhatti, and W. Hassan. Object serialization and deserialization using XML.

[17] A. Gonzlez-Pardo, P. Varona, D. Camacho, and F. De Borja Rodriguez Ortiz. Optimal message interchange in a self-organizing multi-agent system. In *Studies in Computational Intelligence*, volume 315, pages 131–141.

[18] Todd Greanier. Flatten your objects; discover the secrets of the java serialization API. page 1.

[19] Andreas Gustavsson, Jan Gustafsson, and Bjrn Lisper. Timing analysis of parallel software using abstract execution. In Kenneth L. McMillan and Xavier Rival, editors, *Verification, Model Checking, and Abstract Interpretation*, Lecture Notes in Computer Science, pages 59–77. Springer Berlin Heidelberg. DOI: 10.1007/978-3-642-54013-4_4.

[20] Myeong-Wuk Jang and Gul Agha. Agent framework services to reduce agent communication overhead in large-scale agent-based simulations. 14(6):679–694.

[21] M. Kipps, W. Kim, and M. Fukuda. Agent and spatial based parallelization of biological network motif search. In *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, pages 786–791.

[22] Bailey says. 18 software documentation tools that do the hard work for you.

[23] Bharathi Shyam R., Sachin Ganesh H. B., Prabaharan Kumar S., Prabaharan Poornachandran, and Prabaharan Soman K. P. Apache spark a big data analytics platform for smart grid. 21:171–178.

[24] Michael Stonebraker, Daniel Abadi, David Dewitt, Sam Madden, Erik Paulson, Andrew Pavlo, and Alexander Rasin. MapReduce and parallel DBMSs: friends or foes? 53(1):64–71.

[25] Brett Yasutake, Niko Simonson, Jason Woodring, Nathan Duncan, William Pfeffer, Hazeline U. Asuncion, Munehiro Fukuda, and Eric Salathe. Supporting provenance in climate science research.

[26] Matei Zaharia, Reynold Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache spark: a unified engine for big data processing. 59(11):56–65.

[27] Junji Zhi, Vahid Garousi-Yusifolu, Bo Sun, Golara Garousi, Shawn Shahnewaz, and Guenther Ruhe. Cost, benefits and quality of software development documentation: A systematic mapping. 99:175–198.