UNIVERSITY OF WASHINGTON, BOTHELL
COMPUTER SCIENCE & SOFTWARE ENGINEERING DEPARTMENT

UTKU MERT

Masters Capstone Project Term Report
Autumn ‘16

Master of Science in Computer Science & Software Engineering

University of Washington, Bothell

12/15/2016

Project Committee:

Prof. Munehiro Fukuda, Committee Chair

Asst. Prof. Wooyoung Kim, Committee Member

Dr. Johnny Lin, Committee Member

- **Introduction**

As it is explained in my capstone project proposal, there are 3 technical problems that I intend to solve which are stated as “Developing big data test applications and improving the asynchronous agent migration implementation of the MASS Java library, developing agent population control mechanism and developing agent recycle system”. After recent application executions with the library, it is observed MASS Java has a major memory problem. In order to get more accurate results about MASS execution performance, this issue is prioritized. During October and November, I have conducted research and developed the design that me and Prof. Fukuda believe this design is going to solve this problem. In the first week of December, however, I started developing 2 test applications, “Von Neumann” and “RollDown”, that are going to be mostly done by the end of the 3rd week of December.

OCTOBER

- **Research on overcoming agent population expansion problem**

The main research task that I focused on this month is overcoming agent population expansion which is a major issue in MASS Library. Previous tests with Biological Network Motif and UWCA revealed that an agent object in MASS consumes nearly 1MB space in the memory and cause huge negative impact to computation when agent number goes all the way to million. Therefore, I conducted some research coming up with possible solutions to overcome this issue.

After working on the issue deeply, I came up with the logic that may results in effective memory use. The logic offers two new classes to be implemented which are called AgentSpawnRequestManager and AgentSpawnRequest. AgentSpawnRequestManager is the class who is responsible for evaluating any agent spawn request and deciding whether there is enough memory for new agent allocation. This class contains a queue who holds AgentSpawnRequest objects that have the initial necessary information about new agent. Depending on the physical capabilities of computing nodes, each node will specify the max amount of agents that can run simultaneously within that node. When max number is achieved, new agent spawn requests are created and put into the queue. As soon as there is a room for new agent, the head of queue is fetched and new agent is allocated in the memory.

The proposed implementation logic also brings 2 problems. First of all, how are we going to decide max agent number? Even though specifications of computing nodes are important, user application may want to specify this number according to the application’s logic. Secondly, queue itself is going to consume space in memory. Therefore, queue size should also be taken into account and it should not affect used memory space drastically.

In order to implement this logic 2 techniques are offered. One is complete serialization of the new agent and the other one is creating a custom class that holds only the necessary information about new agent. Former one offers huge advantage on memory

consumption but also brings additional cpu time for the exection. Latter one, however, does not bring any significant cpu time but creates only little improvement on memory usage.

- **Research on serialization techniques**

After the meeting sessions with Prof. Fukuda when the research for agent population control techniques is over, we decided to move on with serialization to handle agent population control problem.

There are different types of serialization techniques that are offered online. Two of these techniques are well-known ones which are XML and JSON. In addition to these, since the major language of development is Java, JDK's default serialization implementation is considered as another option. Finally, an open-sourced, frequently supported serialization project is also taken into consideration called Kryo.

JSON is a powerful syntax that even can handle very complicated and complex data structures using JavaScript engine to parse data. However, even though JSON natively offered by JavaScript, systems do not necessarily require JavaScript engine to make use of JSON since parsers of JSON mostly exist in every platform. Therefore, this feature makes JSON is a strong candidate for serialization.

XML, on the other hand, has similar features like JSON has. XML parsers are also available for each platform. The only downside of XML when it is compared to JSON is that XML uses tags for each field which slightly could increase the output stream size.

Java serialization can only be used if the communicating platforms are running on Java. It requires both sides of the communication channel to have class implementation. The overhead of passing entire object generates problems in terms of memory. However, using RMI as messaging protocol, the development phase could be simplified.

As I stated above, Kryo is an open-sourced serialization technique that is popular among developers in the community. Detailed explanation and the major reasons why Kryo is picked for the serialization technique for our project can be found in the following sections.

- **Kryo**

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.[1] Kryo offers serializers for most of the data structures in Java by default. However, it is easy to register your custom class and keep the serialized object in byte stream and read it back to the memory very easily. Due to its nature of serialization, the overhead costs considerably significant cpu-time while serializing the object but deserialization takes very little time comparing to serialization.

[1] Definition taken from "https://github.com/EsotericSoftware/kryo"

- **Development for Kryo integration & testing**

Kryo integration for MASS Library requires certain steps to be completed. First of all, since our project uses Maven for dependencies, XML dependency string is retrieved from Maven website and added to MASS Library's pom.xml file. However, this is not enough for Kryo integration since Kryo is also dependent to "Objenesis" library. Therefore, Objenesis library Maven dependency string is also added to pom.xml file to finalize integration.

It is important to mention that the all changes were made to "utku_verification" branch because the required & most updated package of MASS Java Library, mass-core-0.9.1-SNAPSHOT.jar, can be found in this branch.

Order with 1 OrderLine

Serializer	Size (bytes)	Serialize (operations/second)	Deserialize (operations/second)	% Difference (from Java serialize)	% Difference (deserialize)
Java Serializable	636	128,634	19,180	0%	0%
Java Externalizable	435	160,549	26,678	24%	39%
EclipseLink MOXy XML	101	348,056	47,334	170%	146%
Kryo	90	359,368	346,984	179%	1709%

Order with 100 OrderLines

Serializer	Size (bytes)	Serialize (operations/second)	Deserialize (operations/second)	% Difference (from Java serialize)	% Difference (deserialize)
Java Serializable	2,715	16,470	10,215	0%	0%
Java Externalizable	2,811	16,206	11,483	-1%	12%
EclipseLink MOXy XML	6,628	7,304	2,731	-55%	-73%
Kryo	1216	22,862	31,499	38%	208%

Figure 1. Comparison of serialization techniques[2]

In order to understand how well Kryo performs for our project, of course testing needs to be done. For this purpose, I picked the RandomWalk implementation. The major reason is that since the agent number is static, the performance could be measured more accurately. RandomWalk.java is modified and new file, RandomWalkCPUTest.java, is created to measure memory improvement and cpu cost. RandomWalk.java is edited in a way that it supports Kryo serialization and calculates how much space is used for an agent and a Kryo-serialized agent. On the other hand, RandomWalkCPUTest.java is

[2] Image taken from "http://java-persistence-performance.blogspot.com/2013/08/optimizing-java-serialization-java-vs.html"

implemented to understand how much additional cpu time is spent for both serialization and deserialization processes.

The test files can be found under /CSSDIV/research/dslab/utku/apps/RandomWalk/ folder and mass-core-0.9.1-SNAPSHOT.jar could be found under /CSSDIV/research/dslab/utku/mass/ folder.

- **Preliminary results**

The tests could be repeated by using the following commands for compilation and run.

In order to check memory improvement:
Compilation: javac -cp mass-core-0.9.1-SNAPSHOT.jar:. RandomWalk.java
Execution: java -cp mass-core-0.9.1-SNAPSHOT.jar:. RandomWalk 100 500 100 5 1

In order to check CPU cost:
Compilation: javac -cp mass-core-0.9.1-SNAPSHOT.jar:. RandomWalkCPUTest.java
Execution: java -cp mass-core-0.9.1-SNAPSHOT.jar:. RandomWalkCPUTest 100 500 100 5 true (to enable serialization, false for disabling it)

Here are the results for memory usage:

```
MASS.init: done
-- AGENTS SET SIZE --
361
-- NORMAL AGENT OBJECT SIZES --
time: 0 the total agents size is: 348273408 bytes
time: 0 the average agents size is: 964746 bytes
-- KYRO SERIALIZED AGENT OBJECT SIZES --
time: 0 the total agents size is: 89827 bytes
time: 0 the average agents size is: 248 bytes
-- REQUEST AGENT OBJECT SIZES --
time: 0 the total agents size is: 348400608 bytes
time: 0 the average agents size is: 965098 bytes
-- AGENTS SET SIZE --
361
-- NORMAL AGENT OBJECT SIZES --
time: 5 the total agents size is: 348362168 bytes
time: 5 the average agents size is: 964992 bytes
-- KYRO SERIALIZED AGENT OBJECT SIZES --
time: 5 the total agents size is: 92517 bytes
time: 5 the average agents size is: 256 bytes
-- REQUEST AGENT OBJECT SIZES --
time: 5 the total agents size is: 348388160 bytes
time: 5 the average agents size is: 965064 bytes
-- AGENTS SET SIZE --
361
-- NORMAL AGENT OBJECT SIZES --
time: 10 the total agents size is: 348320488 bytes
time: 10 the average agents size is: 964876 bytes
-- KYRO SERIALIZED AGENT OBJECT SIZES --
time: 10 the total agents size is: 91463 bytes
time: 10 the average agents size is: 253 bytes
-- REQUEST AGENT OBJECT SIZES --
time: 10 the total agents size is: 348346480 bytes
time: 10 the average agents size is: 964948 bytes
```

As you can see 361 agents are run for the application. An average not serialized agent size is roughly 965,000 bytes. While serialized agent size is around 250 bytes. The request agent object, which is a custom class that holds only the data of the agent instance, is almost same as not serialized agent. When Kryo is used the memory consumption of one agent object goes down to 0.00025%.

Let's look at the cpu cost:

```
MASS.init: done
---------------------------
Serialization IS enabled!
Total execution time(including accessing all places and agents objects): 30607 milliseconds
Total execution time(only for serialization for all cycles): 25371 milliseconds
Execution time(only for one serialization cycle for all agents): 1268 milliseconds
Execution time(average for one serialization): 3.514121 milliseconds
Total execution time(only for deserialization for all cycles): 3892 milliseconds
Execution time(only for one deserialization cycle for all agents): 194 milliseconds
Execution time(average for one deserialization): 0.539179 milliseconds
---------------------------
MASS::finish: done
```

As it is seen above, one serialization cycle costs additional 3.5 ms cpu time whereas deserialization adds only 0.5 ms. Therefore for each serialization/deserialization process total of additional 4 ms is the trade-off of decreasing an average agent memory consumption to 0.025%.

NOVEMBER

- **MASS Java version check-up**

Beginning from this month, I have been making sure the performance of the MASS Library. As it is pointed out multiple times by the other research members, MASS is currently experiencing high memory consumption issue when the number of agent increases drastically. Apart from this problem, it is also suggested that asynchronous migration implementation needs to be revised since it takes days to understand the logic behind the code. It is also pointed out that a new application is required to verify the results that we collect after the capstone development is done. Finally, in case everything works out as we predict, it is discussed that MASS could be released to the public for further improvement.

- **High level software design before development**

After asynchronous migration development is added as to-do task, high level software designs for development objectives are specified.

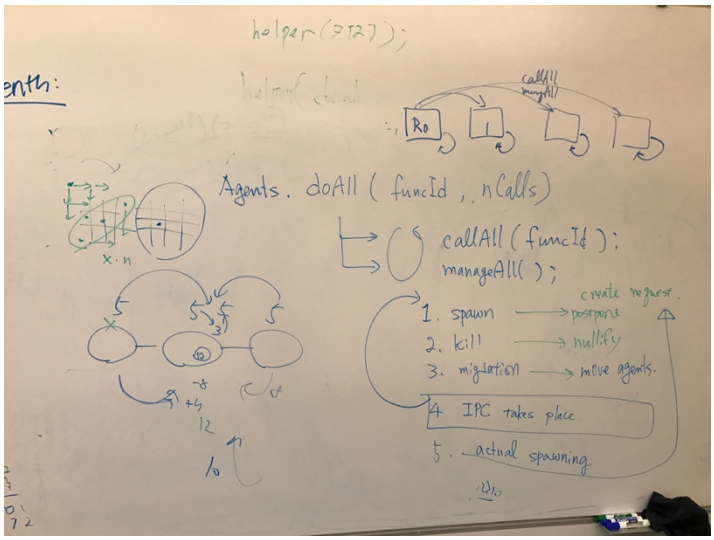
1. **Agent Population Control**

New classes are introduced for agent population control which are called AgentSpawnRequestManager and AgentSpawnRequest. In current version of the MASS, when user application calls “spawn” method, a new agent is spawned and it starts running within the system. Since allocated agent causes high memory consumption (the reasons are explained in detail in previous month section) MASS is not able to utilize its performance. Therefore, instead of letting “spawn” method to create new agent directly, the system should get these calls as requests for further investigation. In to-be-developed version, each computing nodes will have AgentSpawnRequestManager that is going to evaluate these requests and decides whether new agent allocation is allowed or they should be stored as AgentSpawnRequest in the queue. AgentSpawnRequest object contains simply the serialized version of the new agent object. For this purpose, Kryo serialization technique is going to be used (explained in detail in previous report). Maximum number of agent has a default value but user application will be able to override this value.

2. **Asynchronous Migration**

Since the current implementation of asynchronous migration may give some headache to the developers who would like to learn the logic behind the implementation, it is better to revise the code to make it more clear. After regular meetings with Prof. Fukuda, it is acknowledged that current implementation could be taken out and a new simple asynchronous functionality can be developed. Right now, user application needs

to handle correct callAll and manageAll calls during asynchronous execution process. Since MASS needs to be user friendly and should keep the weight of effort to itself, one simple function call is going to be required from the user application which is currently



specified as “doAll” (name is subject to change). The doAll method will be responsible from calling callAll and manageAll functions recursively. In each manageAll call the system now postpone the spawn process. It is going to terminate the agents and complete the migration process as usual. Then Inter process communication will take place. After this step system will collect more spawn requests, apply terminations and perform migrations. When there are no more previous actions left, AgentSpawnRequestManager will come into play and does its work.

3. Agent Recycling

In the current version of MASS, there are no such implementation that enables us to reuse agents. In the previous quarter it has been discussed that having such system will also help our memory consumption problem. However, after the verification tests that I applied last month, we came into conclusion that if agent population control development could be done in success, agent recycling development might not even be needed.

As for the design, agent recycling consists of creating a queue that only contains the pointers to terminated agents. Instead of leaving these agents to garbage collector, they will retain the queue and they are going to be reset with the values of newly spawned agent.

- **Von Neumann application**

In order to verify all development process, it is decided that a new application that would abuse MASS with many number of agents is required. Therefore, von Neumann application is considered to satisfy this need. In this application, an agent will be spawned in the middle of the places array. Then, it is going to select a random neighbor and migrate to that place. It will create spawn requests for other unvisited neighbors. This process will lead us to have many agents in short interval and help us to identify if the new development will affect the execution performance and memory consumption of the MASS significantly.

- **Public access to MASS Java**

It has been also discussed that MASS Java needs to be publicly available to other students so that they can develop applications using MASS or contribute actual library development directly. There are many advantages of making MASS publicly available. First of all, it is effective to learn/teach about the parallel computing concepts with a solid example project. Secondly, developers can partially improve the MASS and make the code more readable or cpu-effective. Furthermore, there are always shallow bugs and edge cases for every project. With more people, these bugs and edge cases could be identified much easily. Lastly, more people means more solutions to a certain problem. For example, in future if MASS encounters another problem like high memory consumption, more number of solutions would definitely be offered by all the developers together.

As of now, MASS Java is not ready to be publicly released yet. However, after the development discussed in earlier sections, the library will be in good shape and become ready to be released shortly after. It needs to be noted that even though everything goes well in terms of technicality, proper commenting and documentation are required since the new developers should be quickly able to understand the logic behind the implementation.

DECEMBER

- **Test application development**

As it is mentioned before, in order to verify MASS execution performance after development tasks, we need test applications. Since the beginning of this month, I have been developing the Von Neumann application. As soon as this development is completed, I will be moving forward to RollDown application.

As of now, since none of the applications are ready for execution, there is no quantitative result that I am able to present. However, it is predicted that these application will be ready before December ends.