Chris Lee
CSS 600 - Independent Study
Winter 2020

Performance and Programmability of Native code in the MASS Libraries

I. Introduction

Multi-agent simulations are computerized systems composed of one or more interacting intelligent agents used to model mega-scale social or biological agents and their behavior.[1] Given the large number of agents and scale of these simulations an important key to success is their ability to run within an acceptable amount of time which is generally facilitated by parallel computing.[2] An example which has been developed by the Distributed Systems Laboratory (DSL) at the University of Washington Bothell is the MASS (Multi-Agent Spatial Simulation) Library for C++, Java and Cuda.

The different implementations of the MASS Library make use of the features and benefits of each language used in its development. However, maintaining multiple versions of the library can be inefficient, e.g. keeping the features in sync across the three implementations. Additionally, previous performance assessments have shown the MASS Java library to be slower than the C++ version.

In addition to developing an understanding of agent based computing and the applications of the MASS Library the purpose and goal of my research during the Winter 2020 quarter was to assess the performance profile and limitations in the current implementation of MASS Java and to evaluate options to improve performance using native code. Specifically, this research considers the impact on code maintenance and execution performance of two architectural designs for the MASS library as illustrated in Figure 1. The first is to directly translate the MASS Java library to native C++ code. The second is to delegate from the MASS Java and Mass C++ library to a core C++ library which can handle functions best suited to the language. Due to the scale of large simulations an improvement in execution performance would lead to decreased simulation times while improved maintainability would facilitate faster development time and reduce the development time needed for new features.

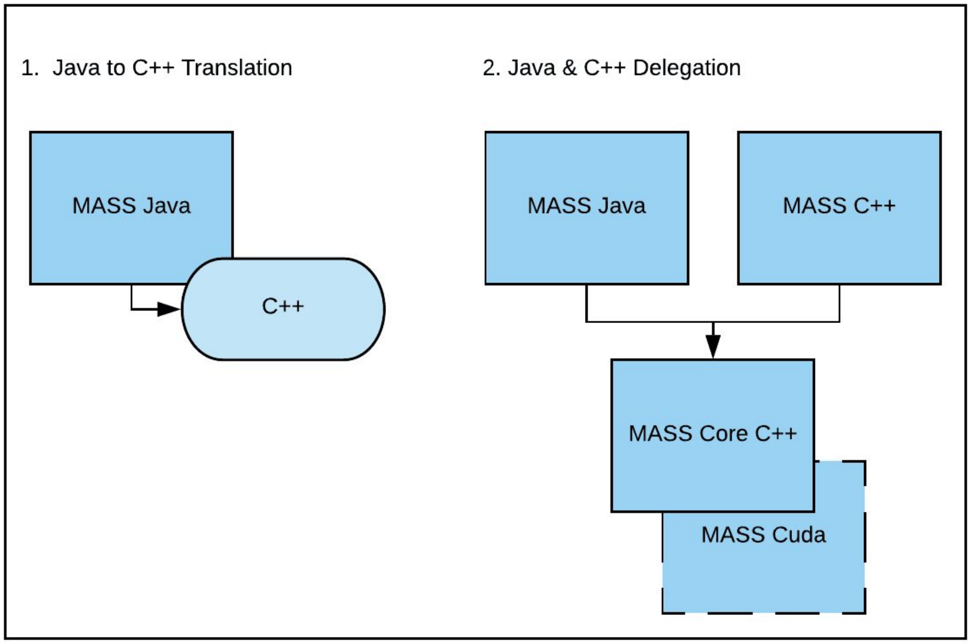


Figure 1. Two possible architectural designs for the integration of native code into the MASS Library.

The remainder of the paper is organized as follows: Section II discusses the MASS programming paradigm and the Sugarscape application which is used for testing; Sections III, IV and V discuss the testing methodologies, results and analysis of MASS Java; Sections VI and VII discuss Java technologies for the use of native code while limitations and further steps follow in Section VIII.

II. MASS Programming and the Sugarscape Application

A MASS application is composed of one or both user implemented Place and Agent classes. MASS instanties two distributed arrays, Places or Agents, which contain these classes and handles the details of parallelization and execution across multiple computing nodes in a MASS cluster. Additional detail can be found in A Parallel Multi-Agent Spatial Simulation Environment for Cluster Systems. [2]

Applications developed for testing the MASS library include SugarscapeCallAll, the application we chose for performance testing. Because this application makes use of both the Place and Agent classes and the associated exchangeAll and ManageAll methods of the Places and Agents classes this application facilitated testing the largest amount of the library. Sugarscape has also been implemented in both the Java and C++ version of the library enabling performance comparisons between the two implementations.

Figure 2 shows the core of the SugarscapeCallAll application. Lines 1 - 15 illustrate the creation of the Places and Agents arrays and the setting of each Place objects neighbors. At line 22 the program enters the simulation portion of the application (where exchangeAll and manageAll are both called multiple times). Due to the large amount of time spent in this portion of the application it is where we focus most of our analysis.

```
1   Places land = new Places( 1, Land.class.getName(), null, size, size );
2   land.callAll( Land.init_ );
3
4   // Populate Agents (unit) on the Land array
5   Agents unit = new Agents( 2, Unit.class.getName(), null, land, nAgents )
6   // Define the neighbors of each cell
7   Vector<int[]> neighbors = new Vector<int[]>( );
8   for( int x = 0 - vDist; x <= vDist; x++ ) {
9       for( int y = 0 - vDist; y <= vDist; y++ ) {
10          if( !(x == 0 && y == 0) )
11               neighbors.add( new int[]{ x, y } );
12          }
13  }
14
15  land.setAllPlacesNeighbors( neighbors );
16
17  Object[] agentsCallAllObjects = new Object[size*size];
18
19  long startNano = System.nanoTime
20
21  // Start simulation time
22  for ( int time = 0; time < maxTime; time++ ) {
23      // Exchange #agents with neighbors
24      land.exchangeAll( 1, Land.exchange_ );
25
26      land.callAll( Land.update_ );
27
28      // Move agents to a neighbor with the least population
29      Object[] callAllResults = (Object[]) unit.callAll(
30  Unit.decideNewPosition_, agentsCallAllObjects );
31
32      unit.manageAll( );
33  }
34
35  long endNano = System.nanoTime();
```

Figure 2. The core of the MASS Java implementation of SugarscapeCallAll

III. Measuring MASS Runtime

Total execution time in a MASS Java test application is measured from the starting and ending system time as illustrated in figure 3. MASS Java testing was performed on the CSSMPI cluster at UW Bothell with the following hardware specifications: Intel(R) Xeon(R) CPU E5-2698 v3 @ 2.30GHz processors with 3 CPU cores and total online memory of 16G.

```
long startNano = System.nanoTime();

// MASS application implementation

long endNano = System.nanoTime();
System.out.println("Total execution time: " + (endNano - startNano) + " nanoseconds");
```

Figure 3. Measuring runtime in MASS Java.

Mass C++ provides a similar method of measurement implemented in the Timer class (Timer.h and Timer.cpp) as illustrated in Figure 4. MASS C++ testing was performed on the Hermes cluster at UW Bothell with the following hardware specifications: Intel(R) Xeon(R) CPU 5150 @ 2.66GHz with 4 CPUs and total online memory of 16G.

```
timer.start();

// MASS application implementation

long elapsed Time =  timer.lap();
printf( "\nEnd of simulation. Elapsed time using MASS framework with %i processes and %i thread :: %ld \n", nProc,numThreads, elapsedTime1);
```

Figure 4. Measuring runtime in MASS C++

The goal was to understand performance bottlenecks in MASS Java, therefore we tested different approaches of measuring the performance time of methods in the MASS API. We first tried tracking the runtime of individual methods similarly to the code in Figure 3 at different execution points but wrote these as log entries. However, we found a significant impact on performance as we scaled up the number of execution threads.

We therefore settled on using a Java profiling tool, VisualVM [3]. VisualVM provides the ability to perform CPU and Memory tracing with an intuitive output with minimal to no performance impact. Our results are discussed in the next section.

IV. Runtime Results

A. Runtime of MASS Java

The first step was to confirm possible performance impact by running the Sugarscape application with and without VisualVM collecting a CPU trace. As illustrated in Table 1 there is not a noticeable performance difference when running MASS Java while a CPU trace is collected.

Host	Threads	Matrix Size	Agents	Iterations	Time	VMVisual
CSSMPI1	3	1000	200000	100	253,185 ms	No
CSSMPI1	3	1000	200000	100	252,800 ms	Yes

Table 1. Runtime of MASS Java with and without running a VisualVM CPU trace

From here, the results of CPU traces with different cluster configurations are summarized in table 2. We found a significant time is spent within ExchangeALL, of that time 50% is self time, 50% is in a call to the Place.CallMethod(). Further discussion of these results will be discussed in section V.

Test	**Host**	**Threads**	**Matrix**	**Agents**	**Iterations**	**Overall Time**
1	CSSMPI1	1	100	4	1000	1,206,395 ms
		Method			**Time**	**% of Total**
		exchangeAll (self time)			599,265 ms	49.7%
		exchangeAll (callMethod)			593,346 ms	49.2%
2	CSSMPI1	2	100	4	1000	663, 490 ms
		Method			**Time**	**% of Total**
		exchangeAll (self time)			316,390 ms	47.7%

		exchangeAll (callMethod)			320,386 ms	48.3%
	CSSMPI1	3	100	4	1000	492,291 ms
		Method			**Time**	**% of Total**
		exchangeAll (self time)			477,923 ms	97 %
		exchangeAll (callMethod)			105 ms	0 %
	CSSMPI1 * 2	3	100	4	1000	264,849 ms
		Method			**Time**	**% of Total**
		exchangeAll (self time)			249,849 ms	94.5 %
		exchangeAll (callMethod)			211 ms	0 %

Table 2. MASS Java Runtimes

B. Runtime of MASS C++

Table 3 is provided as a comparison for the runtime of MASS C++.

Test	Host	Threads	Matrix	Agents	Iterations	Overall Time
1	CSSMPI1	3	1000	200000	100	2,380,868 ms
1	Hermes01	3	1000	200000	100	355,332 ms

Table 3. Runtime of MASS C++ Sugarscape

Testing was performed with different amounts of agents in order to determine possible performance impact, however the increase in time spent in Agents.manageAll was minimal compared to the time spent in exchangeAll. Additionally, we ran the test applications with logging disabled in order to ensure that logging output was not impacting the overall runtime or runtime of specific methods.

V. Analysis and Discussion of Possible Performance Improvements

Line 22 of Figure 2 identified that a simulation starts with O(n) complexity. The exchangeAll method is $O(n)^2$ which makes the entire simulation $O(n)^3$complexity when the exchangeAll method is called. For these reasons it is not surprising that the majority of time in a MASS simulation is spent in the exchangeAll method.

Within the exchangeAll method we were able to identify two primary areas where time is spent. First, the Place array is divided into slices depending on how many hosts and threads are used in the MASS cluster. Individual threads are responsible for iterating over their slice of the distributed array and the neighbors of each Place as illustrated in Figure 5, line 6, . The second is line 31 of Figure 5, where the algorithm makes use of each Place objects' callMethod in order to dynamically call methods implemented in the Place object itself.

Due to the slow performance of Java reflection, a design decision was made to implement a switch statement in the Place object which calls the requested method referenced with an integer.[2] Because it is possible that improvements in the JVM since the original implementation were possible we performed additional testing of the direct calling of a method, the current switch statement implementation and Java Reflection. Results, shown in Table 4, indicate that Java Reflection is still considerably slower than calling a method directly, however there is some inconsistency in the direct method call vs a switch statement. This could perhaps be explained by a JVM optimization and could warrant further exploration.

```java
if (range[0] >= 0 && range[1] >= 0) {
     for (int i = range[0]; i <= range[1]; i++) {
             Place srcPlace = places[i];
             srcPlace.setInMessages(new Object[srcPlace.getNeighbours().size()]);

             for (int j = 0; j < srcPlace.getNeighbours().size(); j++) {

                int[] offset = srcPlace.getNeighbours().get(j);
                int[] neighborCoord = new int[dstPlaces.size.length];

                getGlobalNeighborArrayIndex(srcPlace.getIndex(), offset, dstPlaces.size,
neighborCoord);

                ...

                if (neighborCoord[0] != -1) {

                    int globalLinearIndex =
MatrixUtilities.getLinearIndex(dstPlaces.size, neighborCoord);

                        ...

                        if (globalLinearIndex >= dstPlaces.lowerBoundary
                           && globalLinearIndex <= dstPlaces.upperBoundary) {

                                int destinationLocalLinearIndex = globalLinearIndex -
dstPlaces.lowerBoundary;
                                Place dstPlace =
dstPlaces.places[destinationLocalLinearIndex];

                                ...

                                Object inMessage = dstPlace.callMethod(functionId,
srcPlace.getOutMessage());

                                srcPlace.getInMessages()[j] = inMessage;

                        } else {

                    ...

                        }
                }
        }
}
```

Figure 5. A portion of the exchangeAll Implementation

Test	Iterations	Average Time
Direct Method Call	1000	1488
Switch Statement	1000	152 ms
Reflection	1000	53860 ms

Table 3. Runtime of MASS C++ Sugarscape

Overall, these testing results indicate that we should focus our attention on the exchangeAll method in order to increase the performance of MASS Java simulations. Given the performance differences between MASS Java and Mass C++ the use of native code for the implementation of exchangeAll could lead to performance improvements.

Two architectural options for doing this were described in the introduction, the first, wrapping the C++ library with a Java API or to delegate from the MASS Java to Mass C++ library with JNI or JNA, both discussed in the next section. The use of either approach illustrated in figure 1 will impact the complexity and flexibility of library development. Wrapping the C++ library with Java would, perhaps, be the most straightforward but doing this could limit the use of Java language features. Further research is required to determine the most optimal method of incorporating native code in the MASS Java library.

Another option for improving performance would be to figure out a way to reduce, eliminate or improve the efficiency of the calls to the neighbor places. By doing this we could improve the efficiency of the exchangeAll algorithm to $O(n)^2$.

VI. JNI or JNA

Two options for calling native code from Java are the Java Native Interface (JNI) or Java Native Access (JNA). JNI has been available since JTSE 1.3 released in 2003 [4]. The steps to use JNI are:

1. Write a Java Class which loads a native library, declares the available methods.
2. Compile the Java Class and Generate C or C++ Header files.
    a. Pre Java 10 this uses javac and javah.
3. Implement the C Program.
4. Compile the C program.
5. Run the Java Program

The Java Native Access library is a community-developed library with the goal of providing “easy access to native shared libraries without writing anything but Java code” which was first released in 2007.[5] In contrast to JNI, JNA does not require boilerplate or glue code to be implemented in the native code, making it simpler and more straightforward.

The steps to implement native code with JNA are to:

1. Download the JNA Jar or add the dependency in Maven.

```
<dependency>
        <groupId>net.java.dev.jna</groupId>
        <artifactId>jna</artifactId>
        <version>5.5.0</version>
</dependency>
```

2. Create a Java Interface Class which loads the native library and defines the available native functions in Java.

3. Call the native functions from the Java code.

```java
/** Simple example of JNA interface mapping and usage. */
public class HelloWorld {

    // This is the standard, stable way of mapping, which supports
extensive
    // customization and mapping of Java to native types.

    public interface CLibrary extends Library {
        CLibrary INSTANCE = (CLibrary)
            Native.load((Platform.isWindows() ? "msvcrt" : "c"),
                                CLibrary.class);

        void printf(String format, Object... args);
    }

    public static void main(String[] args) {
        CLibrary.INSTANCE.printf("Hello, World\n");
        for (int i=0;i < args.length;i++) {
            CLibrary.INSTANCE.printf("Argument %d: %s\n", i,
args[i]);
        }
    }
```

```
}
```

Figure 6. An example of the use of a native shared library.

Further exploration of either JNI or JNA would be required to make a determination of how to best incorporate the use of native code.

VII. Limitations and Further Steps

Over the past 10 weeks I was able to achieve my goal of becoming more familiar with the MASS libraries and spatial simulations in general and to develop an understanding of the current performance limitations of the library, however there is additional work to be done in order to understand the opportunities for improving performance in the MASS Java library. Next steps are to perform a literature and application review to determine best practices and other scenarios where native code is used to improve the performance of Java applications. Included, or separate, would be further research in optimizing Java and the JVM for high performance, distributed applications like the MASS library. There is research which indicates that, under certain scenarios, Java can perform as effectively as native code.[5] If this truly is possible it would provide the least complex architecture and the flexibility of continued use of Java's current and future features.

References

[1] "Multi-agent system," *Wikipedia*. 27-Feb-2020.
[2] T. Chuang and M. Fukuda, "A Parallel Multi-Agent Spatial Simulation Environment for Cluster Systems," in *2013 IEEE 16th International Conference on Computational Science and Engineering*, Sydney, Australia, 2013, pp. 143–150, doi: 10.1109/CSE.2013.32.
[3] "VisualVM: Documentation." [Online]. Available: https://visualvm.github.io/documentation.html. [Accessed: 09-Mar-2020].
[4] "JNI APIs and Developer Guides." [Online]. Available: https://docs.oracle.com/javase/8/docs/technotes/guides/jni/index.html. [Accessed: 09-Mar-2020].
[5] J. Nazario Irizarry, "Mixing Java and C for High Performance Computing," The MITRE technical report, 2013, MTR130458.

Appendix A - Measuring Performance using VisualVM

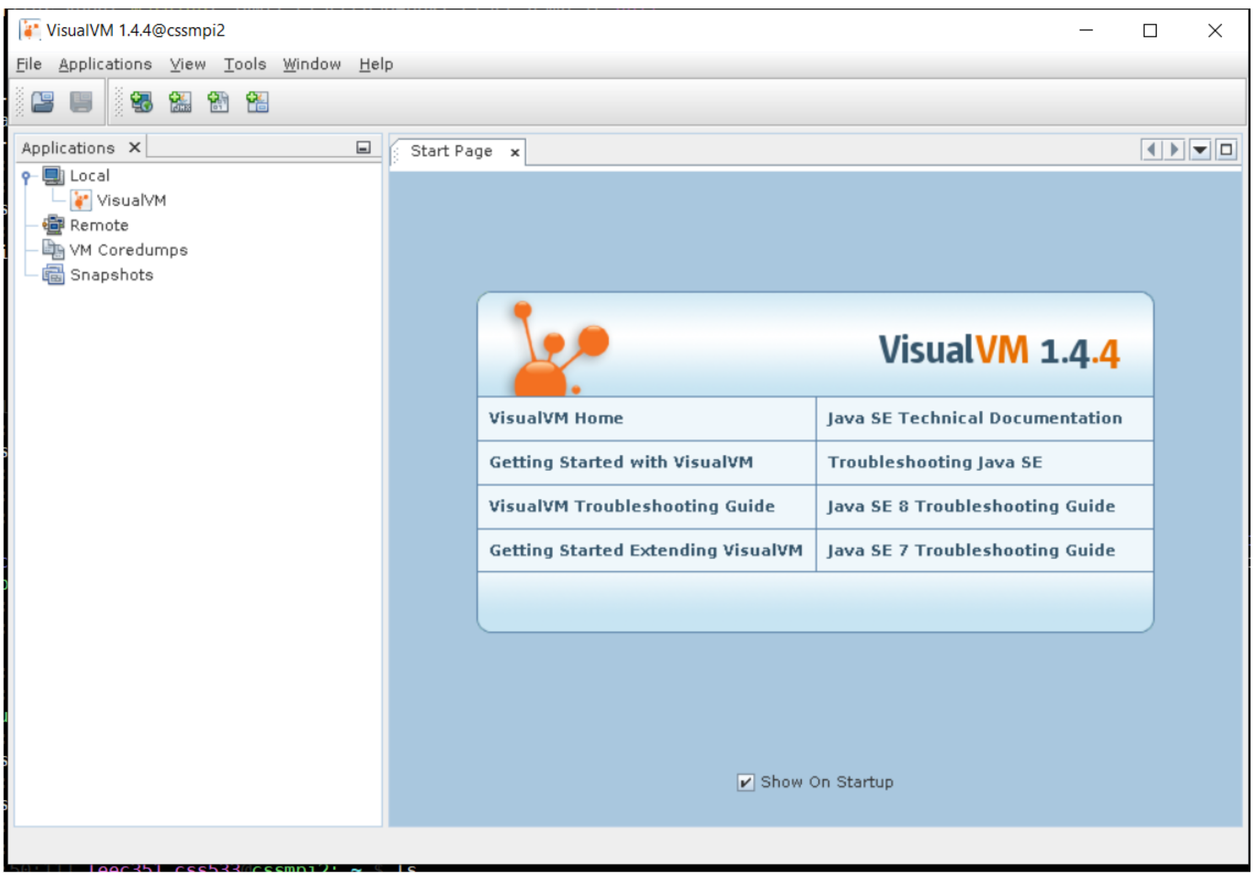
There are three ways to use VisualVM to analyze code running in a Java Virtual Machine. Documentation and download is available from [VisualVM.]

1. Running VisualVM Locally (or with a terminal and X server)

Run VisualVM in the background. Requires an X server (Handled automatically by MobaXterm, documentation for using Putty with an X server can be found [here]).

a. Locally: Run ./visualvm_144/bin visualvm

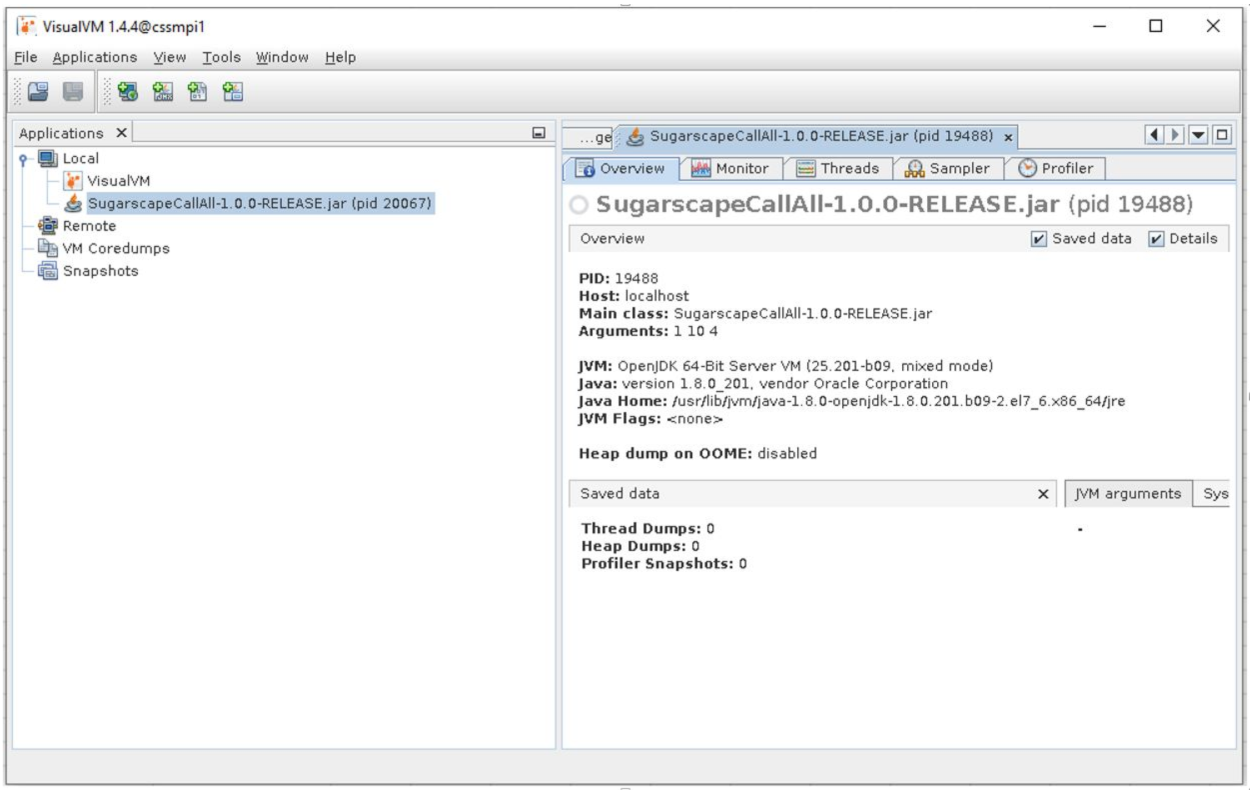
b. To run through SSH use ./visualvm_144/bin/visualvm & (This will cause VisualVM to run as a background process and open on the client machine).



c. Run your MASS application:

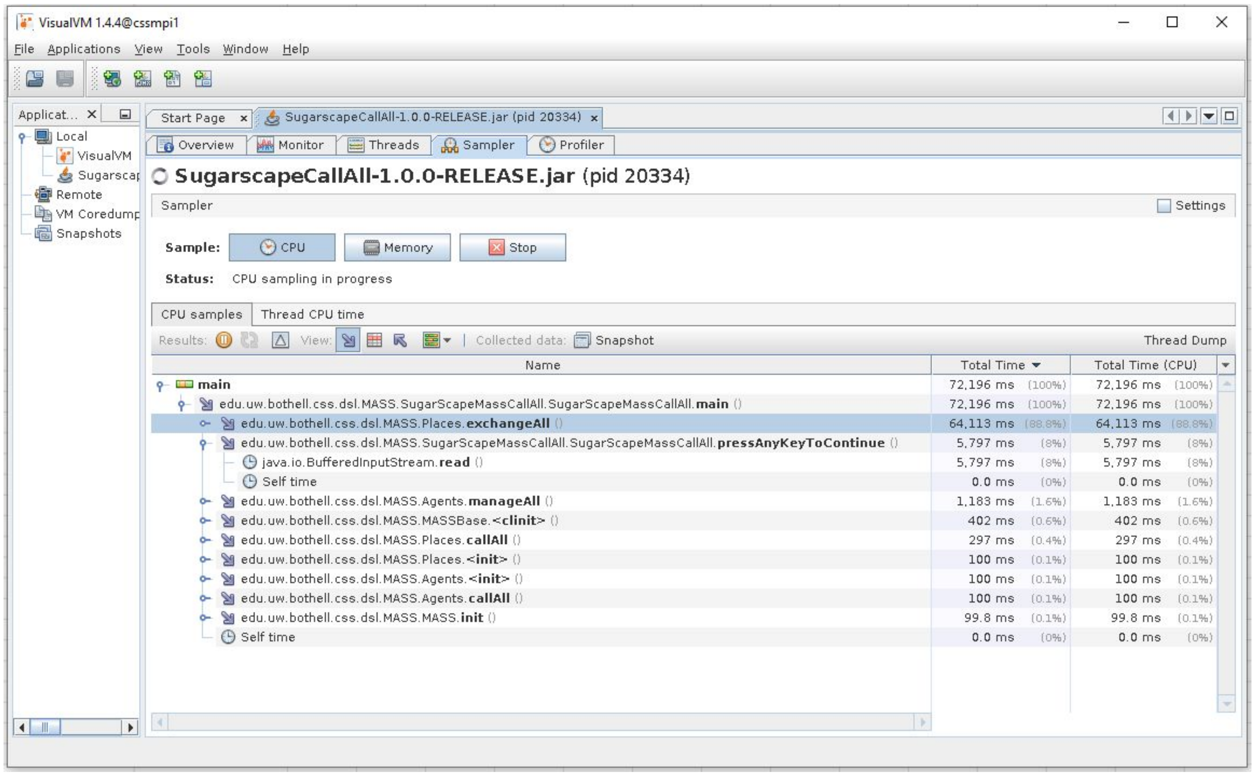
MASS_Application $ java -jar SugarscapeCallAll-1.0.0-RELEASE.jar

The program will appear in the applications list under Local. (When running "locally" the option to use the Sampler or the Profiler will both be available.)



d. On either the "Sampler" or "Profiler" tab select the "CPU" or "Memory" button to start a capture.

Differences between Sampler and Profiler can be found here:
https://blog.idrsolutions.com/2014/04/profiling-vs-sampling-java-visualvm/



e. Use the "Threads" tab for a visual of the status of the running threads.

f. To Stop VisualVM running:

jobs (lists the background processes)

fg %job_id (brings the job_id to the foreground then ctrl-z (verify) or ctrl-c will kill it)

Or:

kill %job_id (terminates)

Note: If you want to capture the full run of your application it can be helpful to have your application pause while VisualVM connects and refreshes the available views.

```
        private static void pressAnyKeyToContinue()
        {
                System.out.println("Press Enter key to continue...");
                try
                {
                System.in.read();
                }
                catch(Exception e)
                {}
        }
```

2. Remotely using JSTATD

The limitation of this method is that you cannot do sampling or profiling.

The following policy file will allow the jstatd server to run without security exceptions. This policy is less liberal than granting all permissions to all codebases, but is more liberal than a policy that grants the minimal permissions to run the jstatd server.

```
grant codebase "file:${java.home}/../lib/tools.jar" {
    permission java.security.AllPermission;
};
```

To use this policy, copy the above text into a file called jstatd.all.policy and run the jstatd server as follows:

```
jstatd -J-Djava.security.policy=jstatd.all.policy
```

Additional information can be found here:
https://docs.oracle.com/javase/7/docs/technotes/tools/share/jstatd.html

3. Using JMX

1. Start your application with the following arguments:

java -Dcom.sun.management.jmxremote.port=3333
-Dcom.sun.management.jmxremote.ssl=false
-Dcom.sun.management.jmxremote.authenticate=false -jar
SugarscapeCallAll-1.0.0-RELEASE.jar

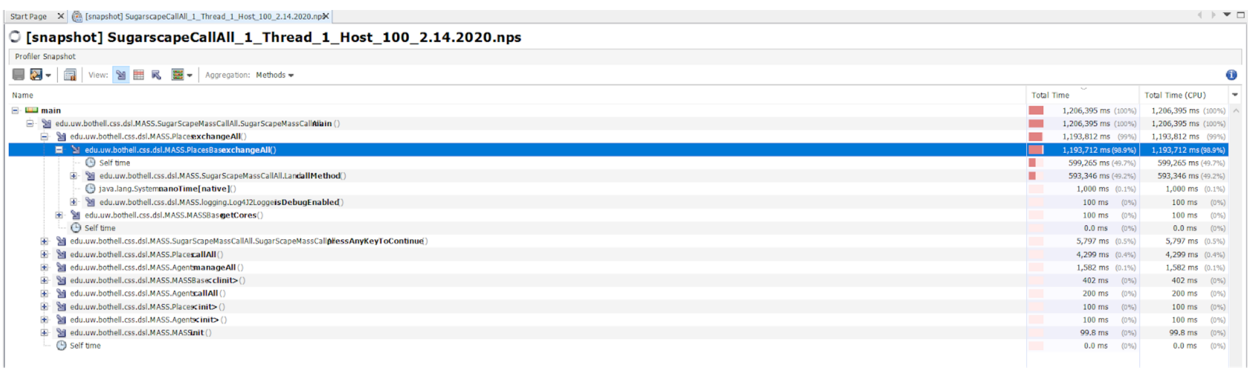
2. Right click “remote” and connect to the remote server and port specified in the above command.

Add this to MASS.init. This will launch the remote process with JMX enabled and all you to connect to the remote nodes.

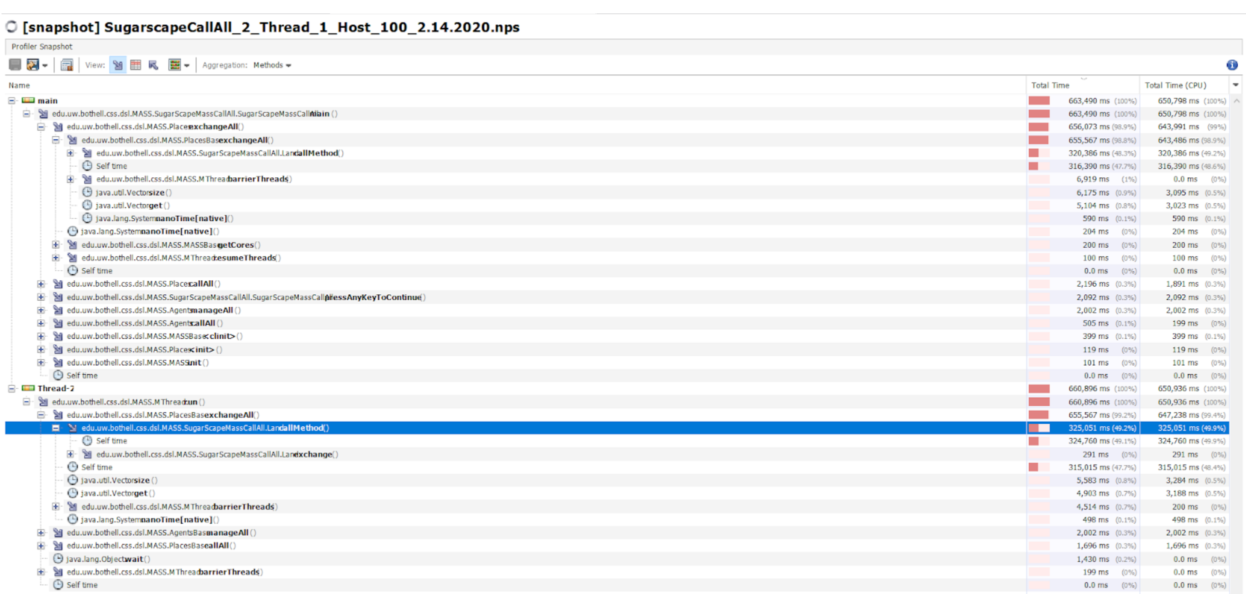
commandBuilder.append("-Xmx9g -Dcom.sun.management.jmxremote.port=3333
-Dcom.sun.management.jmxremote.ssl=false
-Dcom.sun.management.jmxremote.authenticate=false ");

Appendix B - Test Results

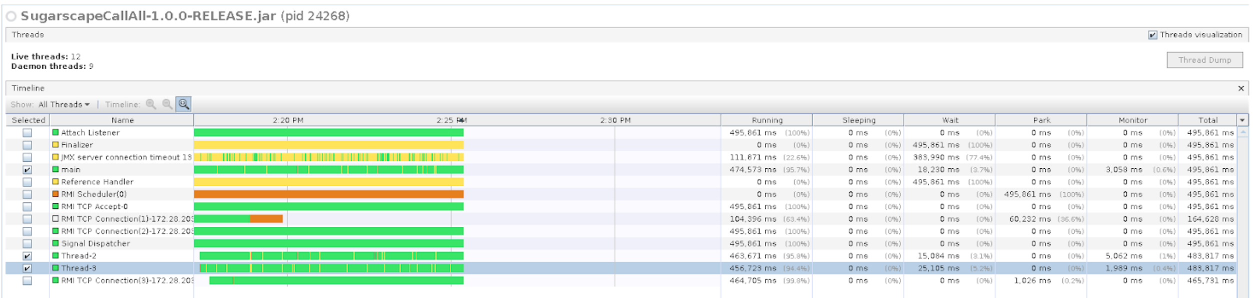
Test 1:



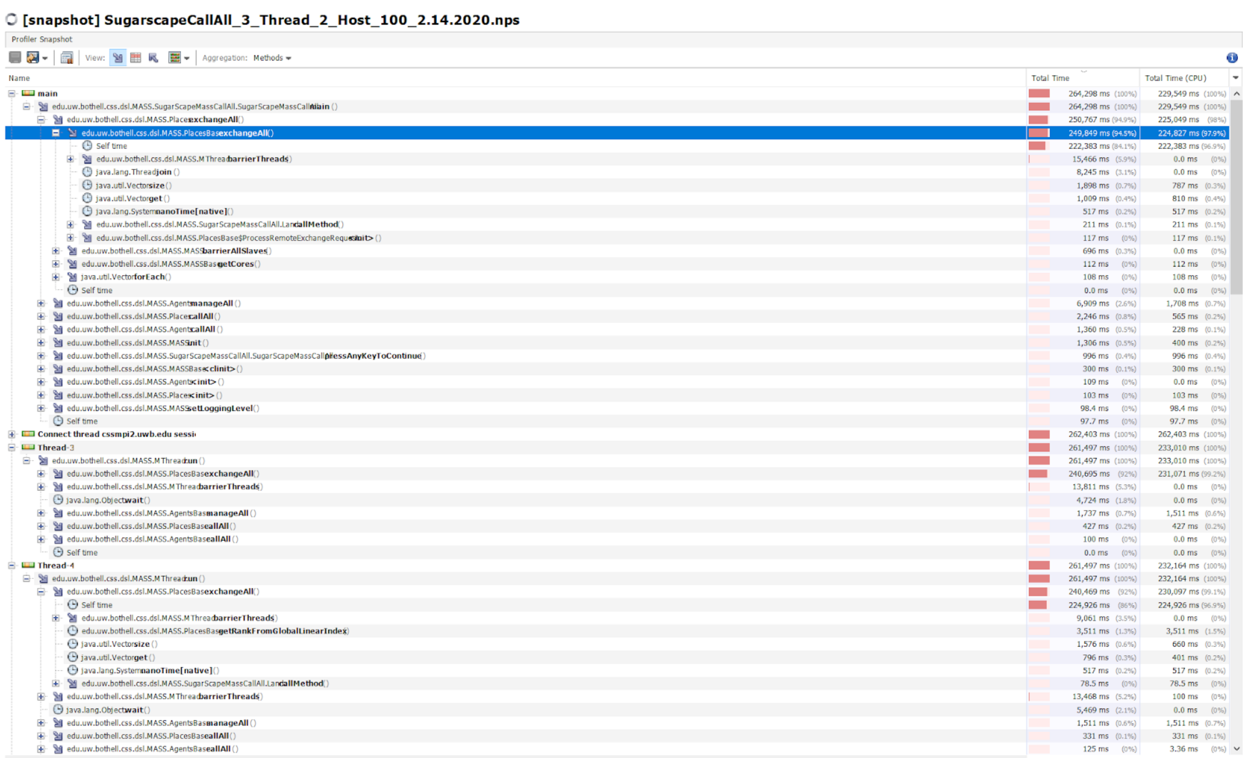
Test 2:



Test 3:


SugarscapeCallAll-1.0.0-RELEASE.jar (pid 24268)

Test 4:


[snapshot] SugarscapeCallAll_3_Thread_2_Host_100_2.14.2020.nps