Distributed agent management in a parallel simulation and analysis environment

Cherie Lee Wasous

A thesis

submitted in partial fulfillment of the

requirements for the degree of

Masters of Science in Computer Science and Software Engineering

University of Washington

2014

Committee:

Munehiro Fukuda, Chair

Michael Stiber

Kelvin Sung

Program Authorized to Offer Degree:

Computing & Software Systems

University of Washington

Abstract

Distributed agent management in a parallel simulation and analysis environment

Cherie Lee Wasous

Chair of the Supervisory Committee:

Associate Professor Munehiro Fukuda, Ph.D

Computing & Software Systems

This thesis presents the design, implementation, and evaluation of adding features of automatic distributed injection, diffusion, merger, and termination to reactive agents in a parallel simulation and analysis environment. This provides a powerful simple-to-use agent behavior for information searching or diffusion purposes, improving programmability for the scientist simulating an experiment or analyzing big data. Performance evaluation shows an improvement in execution time over using reactive agents without these features. However, a solution not using agents at all currently shows better performance and an analysis is presented as to possible future improvements.

ACKNOWLEDGEMENT

I am forever filled with deep gratitude to Dr. Munehiro Fukuda, my advisor. The experience of working closely with him during this research has greatly deepened my understanding of parallel and distributed computing. Never enough can be said about his masterful drawings, which convey complex concepts as clearly and simply as possible. His enthusiasm for the topic of parallel programming and distributed systems is contagious. Without his knowledge, patient guidance, and feedback this thesis would not have been possible.

I would also like to thank my thesis committee members: Dr. Michael Stiber and Dr. Kelvin Sung for their encouragement, review and feedback of my work.

I also wish to acknowledge my fellow students in Dr. Fukuda's Distributed Systems Laboratory research group. Their questions and feedback really helped me refine my presentation of concepts for this thesis.

The department staff has always been very helpful and I wish to thank them all. Megan Jewell, as Graduate Academic Advisor, has always been especially timely with any emails, either in response to my endless questions or her announcements reminding me of the next steps in the thesis process. Josh Larios, as Advanced Systems Engineer, has always been very approachable and willing to entertain any questions about symptoms I observed or help I needed with the department's computers and network.

I have thoroughly enjoyed the whole process that went into this thesis. I send thanks to the designers of this master's program for allowing a thesis option.

Finally, I thank my family for their love and encouragement during every day.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1: Introduction

This research aims at investigating pre-determined injection and navigational capabilities that reactive agents could utilize for search or information diffusion purposes in a parallel simulation and data analysis environment. We have designed two algorithms and implemented them in Multi-Agent Spatial Simulation (MASS) [1], which is a parallelization simulation and data analysis library.

This chapter classifies agents and explains our target agents in 1.1, clarifies what is necessary in agent management in 1.2, and states our research goal in 1.3.

1.1 What are Agents

It is more than 20 years since the concept of "software agents" received a highlight in giving intelligence, autonomy, and collaborative capability for execution entities that intend to make predictions, recommendations, and even decisions for their users [2]. This concept has been particularly focused on distributed artificial intelligence, later combined with navigational autonomy in the Internet environment, which resulted in the emergence of mobile agents. The examples are D'Agents [4] and IBM Aglets [5], each respectively focusing on operational support for military field personnel and automatic travel itinerary creation for business users.

On the other hand, there is another type of agents, not necessarily linked to software agents. They are strongly coupled with individual-based models that intend to observe an emergent collective behavior of many simulation entities (i.e., agents). This type of agents, sometimes called multi-agents but distinguished as mega-agents from multi software agents, gained in popularity when the Santa Fe Institute developed Swarm [3] [4] [5].

We distinguish the former and the latter types as cognitive and reactive agents. This research focuses on reactive agents who reside within a simulation or data analysis system.

1.2 What is Agent Management

There are many simulation systems for reactive agents; some classify themselves as a multi-agent system (MAS) and others use the term agent-based model (ABM) [6]. Reactive agents are often provided the following management functions by the system:

- Agent creation and initial placement in the environment
- Ability to interact with their local and nearby environment and agents
- Migration capabilities to move to another location in the environment
- Spawning of children agents
- Self-kill capability

The Distributed Systems Laboratory at the University of Washington Bothell [7] has developed MASS, enabling the above capabilities for reactive agents. The implementation and evaluation phase of this research enhances the agent management capabilities of reactive agents in the MASS platform.

1.3 Research Goal

My research enhances reactive agents' behavior by providing intelligent, distributed injection, diffusion, merger and termination. This allows a user application to easily use these agents for data searching or data diffusion purposes without having to deal with all the details of calculating patterns of migration, allowing the user to focus their efforts on other aspects of the agent (model). My research goal focuses on:

- Developing algorithms for injection and diffusion that will provide good performance
- A straightforward, easy-to-use MASS library API for these enhanced agents.

The rest of this document is structured as follows: chapter 2 differentiates our research goal from the related work; chapter 3 proposes innovative algorithms in agent management, based on the requirements we found in chapter 2; chapter 4 details our implementation of the proposed algorithms; chapter 5 evaluates the programmability and execution performance of the MASS library that takes advantage of the new agent-management algorithms; chapter 6 concludes our discussions.

Chapter 2: Related Work

Based on our research goal, we need automated agent-based information diffusion, distributed data analysis, and result collection. Of importance is efficient execution to complete data analysis as well as simple programmability to automate agent migration over distributed data.

In the following discussions, we examine the past agent-based parallel-computing systems from the viewpoints of execution performance, programmability, and applications which they have focused on.

Table A summarizes the research and simulation platforms that have led up to MASS. The WAVE system used simple instructions that were very cryptic and not easily understandable as they used many symbols rather than similar constructs as used in common programming languages [8]. WAVE had very slow execution times, as it was an interpreted language. The agent movement was implicit along a ring between computing nodes.

UCI Messengers used a language that was very much like C, so it was easier to read and understand. UCI Messengers had a very descriptive move function allowing agents to move directly to other computing nodes across the network. However, it had almost too many details, which could overwhelm the typical application user who just wanted to develop their model (agent) [9].

The next progression was M++, a C++ based implementation that used threads [10]. This platform was targeting mega-agents, but the technology at the time limited the performance of M++ (e.g. too little memory). Also, like UCI Messengers, M++ provided very flexible navigation functions.

Table A: Summary of Simulation Platforms directly leading up to MASS

System	**Execution Performance**	**Programmability**	**Applications**
WAVE [8] ~1994	Interpreted. Slowest.	Very microscopic. Simple instructions, but very cryptic (not easily human readable)	Distributed Knowledge Network. War Game / Military simulation.
UCI Messengers [9] ~1994-1997	Interpretive byte code. Native, for better performance.	C-like, procedural. Navigational methods.	ABM and Scientific Computations (matrix).
M++ [10] ~1998-2002	C++ based. Thread based. Myrinet network & cards.	C++. OOP. Navigational methods.	ABM limited. Really needed mega-agents, but technology at the time (memory sizes) would not allow.
MASS [1] ~2009-2014	Java. C++. Mobile objects. Fine grained. Fastest. TCP/IP network.	Abandoned previous complicated navigational methods. Automatic cloning, diffusion.	ABM, War Game / Military simulation. Big Data analysis.

MASS abandoned the complex navigational methods of M++ and UCI Messengers, and abstracted these details away from the end user, simplifying the API [1]. Support was added to MASS for Parallel NetCDF so that in addition to simulation, MASS can be used as a platform for big data analysis. This thesis research and implementation provides automatic cloning and diffusion behavior to the agents in MASS.

Places and *Agents*, shown in Figure 1, are fundamental concepts in MASS. *Places* is a matrix of *place* elements that MASS distributes over a cluster of multi-core computing nodes. *Agents* are a set of execution instances that can reside on a *place*, migrate to any other *place* using matrix indices, and interact with other *agents* as well as multiple *place* elements.

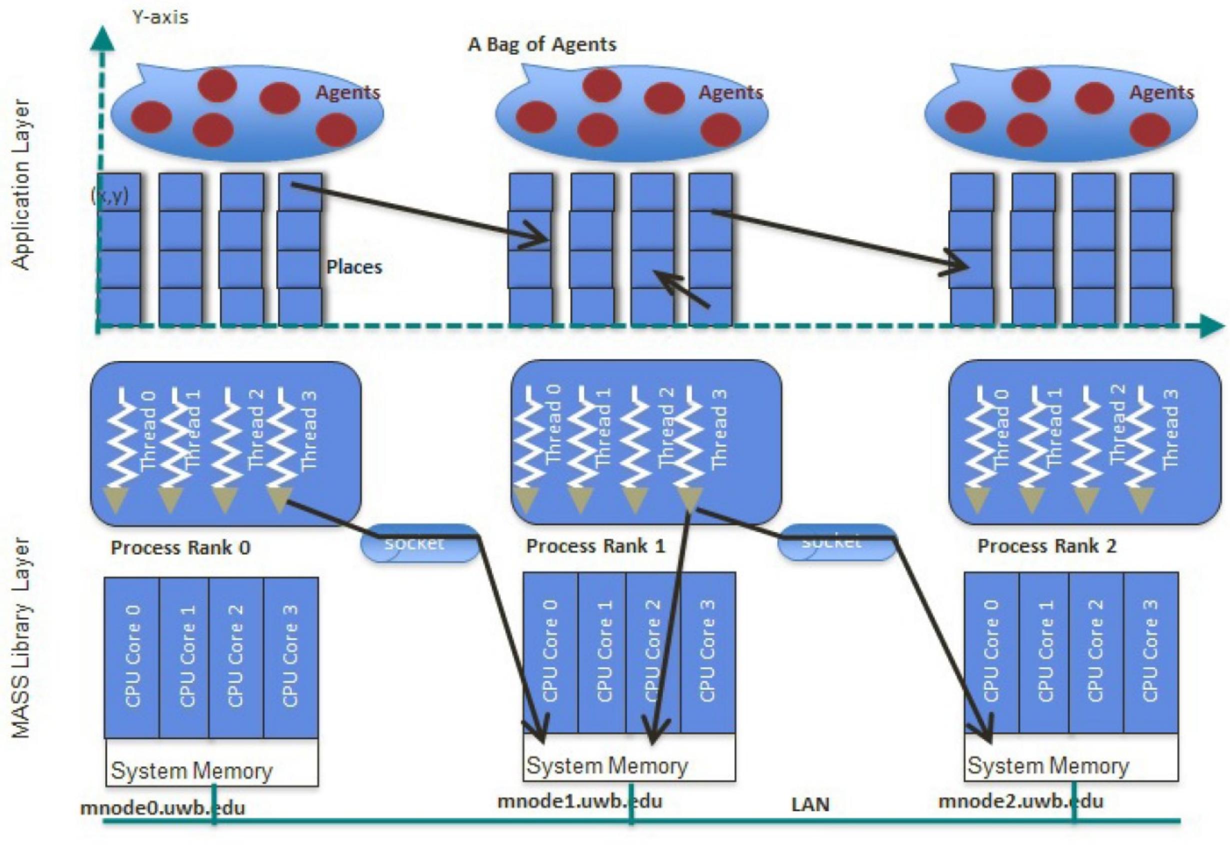


Figure 1: MASS Execution Model [1]

A user designs a behavior of a *place* and an *agent* by extending the Place and Agent base classes respectively. The user application uses the *Agents* methods of callAll( ) and manageAll( ) to have each agent execute a particular user method and to complete any migrate( ), spawn( ), kill( ) commands that were issued, respectively. These calls are performed in parallel among multi-processes/threads.

Two other multi-agent simulation systems that are targeting mega-agent, high performance applications are Repast HPC [11] [12] and D-MASON [13]. From our survey, Repast HPC is a large-scale agent-based modeling system that was implemented on top of MPI and tested on Argonne National Laboratory's Blue Gene/P. The system gives a C++ based agent framework, shared contexts as inter-agent communication media, and ghost spaces visible to adjacent processes, which facilitates basic requirements for agent parallelization. However, it has been designed from the top-down strategy where a user must launch a Repast process at each MPI rank, update agent status with iterative loops, construct logical network spaces with MPI ranks, and use read-only ghost spaces similar to those implemented in distributed arrays [1].

D-MASON is a distributed version of the multi-agent simulation environment MASON, originating from George Mason University over 10 years ago. D-MASON is "designed to harness unused PCs for increased performances" [14]. Their goal is for the user to only do simple modifications to their existing Java based MASON model for D-MASON to run it across multiple computing nodes improving the execution performance. The system consists of a master node, worker nodes, communication server, and logging server. Currently, the system advances a simulation step only at the speed of the slowest worker. A multicast network channel is assigned to each worker, and other workers can subscribe to the updates of agents in neighboring regions. Currently, agents can only migrate between neighboring regions [13].

MASS has the capability of dynamically loading data via parallel NetCDF, enabling data analysis using distributed agents. Our survey did not find any other mega-agent system with such data analysis capability.

Also, we did not find in the literature mega-agent systems that provide built-in capabilities of intelligent, distributed injection, diffusion, merger and termination for their reactive agents.

Chapter 3: Algorithms

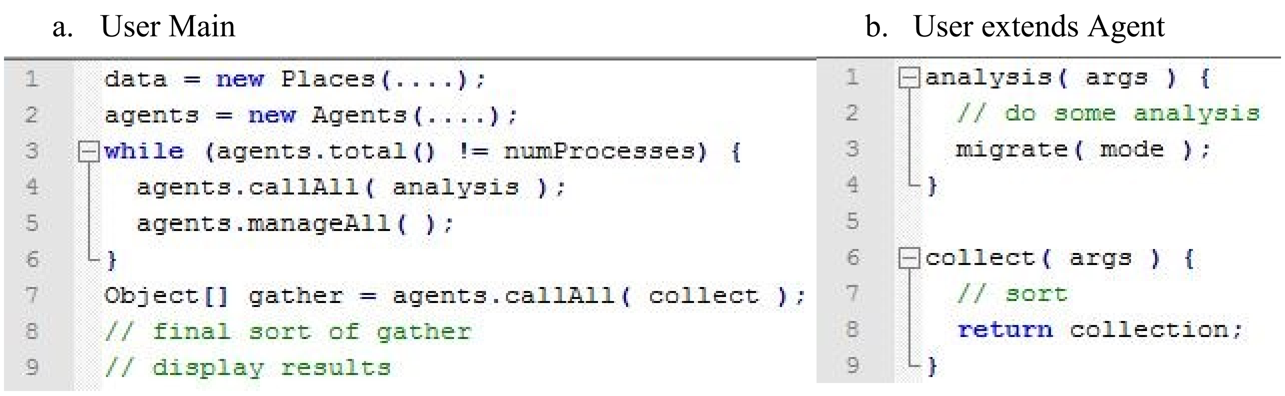
Scientists simulating an experiment or analyzing big data should not be burdened with needing to understand the computer science of parallel programming; their skills are more efficiently utilized in their particular domain of expertise. They need not be troubled with the tedious details of distributed software, especially the challenge of debugging this type of software.

Adding enhanced agents with the features of distributed injection, diffusion, merger, and termination to MASS provides a powerful simple-to-use agent behavior for information searching or diffusion purposes. Climate analysis is an application where these enhanced agents would be useful, automatically injecting and diffusing agents into each place, looking for particular values, such as a maximum moisture flux, or identifying all place elements with a temperature below freezing.

To achieve good performance, the enhanced agents need characteristics of:

- Each place element is visited by an enhanced agent just once.
- Efficient migration patterns that avoid collisions.
- A particular enhanced agent lives its life (injection, migration, eventual death) on one node. This avoids costly inter-node communication for migrating agents across nodes.
- Collection and sorting of data gathered is done on each node, and then the master node gathers the data from the nodes and does a final sort to find the overall results.

Two algorithms were developed that meet the above criteria and are called Central-Axes and Central-Point. While the algorithms are different, the conceptual view of a data-analyzing user program looks similar for either algorithm. Figure 2 shows abstract code that is based on MASS.


a. User Main

```
data = new Places(....);
agents = new Agents(....);
while (agents.total() != numProcesses) {
  agents.callAll( analysis );
  agents.manageAll( );
}
Object[] gather = agents.callAll( collect );
// final sort of gather
// display results
```

b. User extends Agent

```
analysis( args ) {
  // do some analysis
  migrate( mode );
}

collect( args ) {
  // sort
  return collection;
}
```

Figure 2: Data-analyzing User Application

3.1 Central-Axes Algorithm

The Central-Axes algorithm instantiates/injects agents all along the central axes of the distributed Places on each node. Figure 3-a shows the injection for a two dimensional Places, where the agents are labelled with “C” for center/collector, “N” for northbound, “S” for southbound, “E” for eastbound, and “W” for westbound.

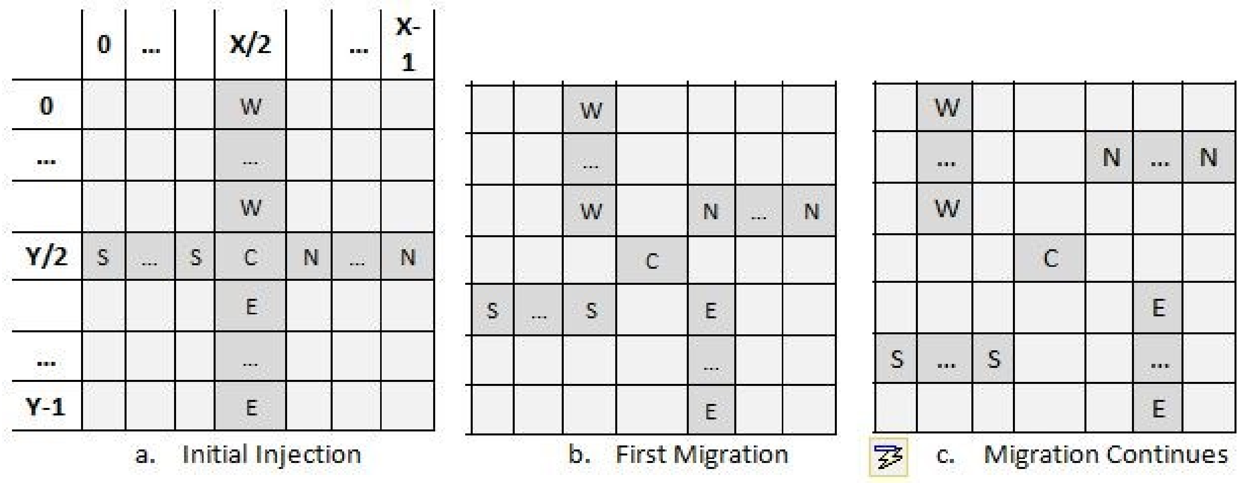


Figure 3 : Central-Axes Algorithm for two dimensional Places

For a node with Places of size X by Y, there are X*Y place elements, and X+Y-1 agents are injected. As each migration step occurs, these Central-Axes agents are migrated in a predetermined pattern, as shown in Figure 3-b and Figure 3-c. The most central agent does not migrate, but stays in its original place element and can provide a static collection data structure for each node, as needed.

As the migrating agents (total of X+Y-2) eventually reach the boundary of the Places on their node, they can deposit their collected value(s) into the central agent on their node and then issue a self-kill.

Figure 4 provides an overview as to the full history of the migration path for these Central-Axes enhanced agents on each node.

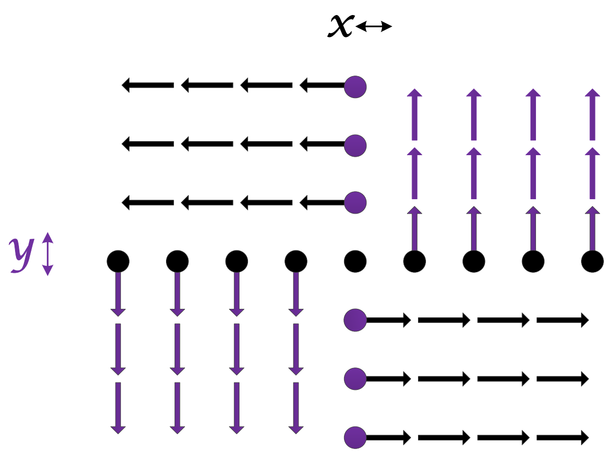


Figure 4: Central-Axes Migration Path Summary for two dimensional Places on one node

The maximum number of migration/kill steps in the diffusion is calculated as:

max [(X/2 + 1) , (Y/2 + 1)]   (integer division)

Once the user application sees that the agents have merged their data and only one agent per node remains, then the user can request the data from the center agents and issue a self-kill.

Figure 5 shows the number of agents present at each node over time. All the agents required are instantiated at time step 1. As the migrating agents eventually migrate to the edge of the node they issue a self-kill after making any deposits into the collector. Then only the one center/collector agent remains, and issues a self-kill after the master node gathers its sorted collected data.

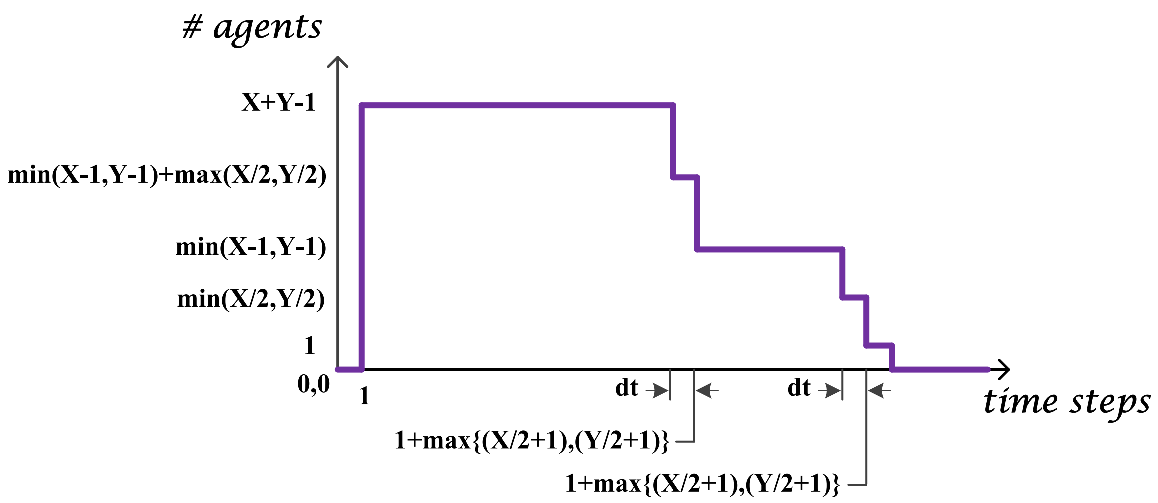


Figure 5: Number of agents present per node for Central-Axes algorithm

The graph with the five steps down as shown in Figure 4 are for the conditions of:

- X not equal to Y,
- X and Y are even numbers (and "dt" as shown on graph has value of 1 time step).

If X and/or Y are odd numbers, then their associated "dt" as shown on graph has value of zero, and the corresponding two-part step down would be just one larger step down, occurring at the time indicated by the formulas below the graph.

If X and Y were equal (X=Y=N), and both were odd numbers, then there would be only one big step down for all the 2N-2 migrating agents, and then the final most central agent would issue a self-kill when it returns the sorted collected values for its node.

3.2 Central-Point Algorithm

The Central-Point algorithm instantiates five agents at the central point area of the Places, as shown in Figure 6-a for a two dimensional Places.

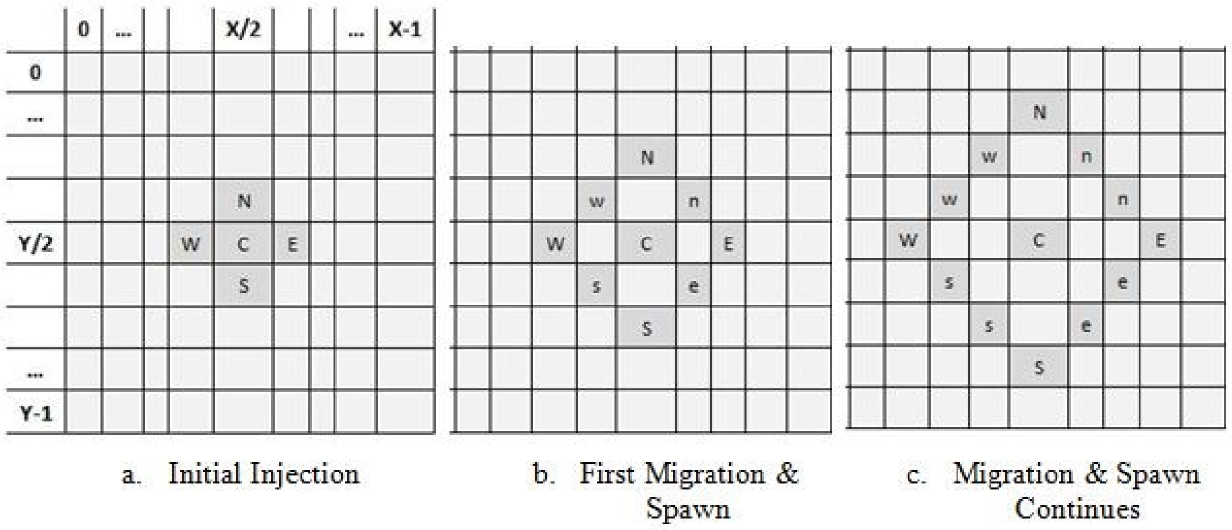
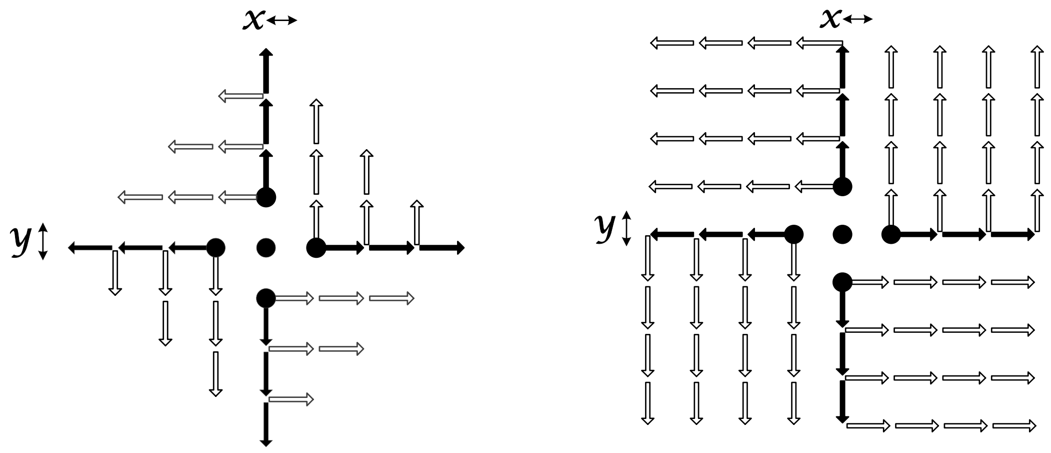


Figure 6: Central-Point Algorithm for two dimensional Places

Like the Central-Axes algorithm, the most central agent does not migrate but remains in the center and can provide a static collector data structure as needed. Figure 6-b and Figure 6-c show the first two migration steps where the four original agents (labeled “N”, “S”, “E”, “W”) move in a predetermined pattern and spawn one agent (labeled “w”, “e”, “n”, “s”) 90 degrees to the left of its direction of travel for each step. The spawned agents (lowercase letters) do not spawn any of their own agents; only the original four agents (uppercase letters) spawn.

Figure 7-a provides an overview of the migration/spawn steps that have happened as the first agents reach the edge of the node. The original N/S/E/W agent steps are shown with solid arrows, and the spawned n/s/e/w agents are shown with hollow arrows.

Figure 7-b shows the total migration/spawn steps that have happened when the final agents reach the edge of the node.



a.) agents first reach edge b.) total migration/spawn path

Figure 7: Central-Point Diffusion for two dimensional Places

As the agents eventually reach the boundary on their node (a total of X+Y+2 agents), they can deposit their collected value(s) into the central agent, and then issue a self-kill. The maximum number of migration/kill steps in the simulation is calculated as:

$$\max [(X - 1), (Y - 1)] \quad \text{(integer division)}$$

Once the user application sees that the agents have merged their data and only one agent per node remains, the user can then request the data from the central agents and issue a self-kill.

Figure 8 shows the number of agents present over time for the condition of X=Y=N and N is odd. Notice that there are a maximum of 2N + 1 agents present at any one time, but there are a total of 2N + 3 agents that are created. This difference occurs because at time step N/2 + 1, the final 4 agents are spawned and 4 previously existing agents make a deposit into collector and are killed. This can be seen by referring to Figure 7-a, as the four original agents have migrated to the edge of the node and each does a final spawn of a new agent as well as a self-kill. For this condition of X=Y=N, and N is odd, once deposits/kills begin to happen they occur at a constant rate of 4 each time step.

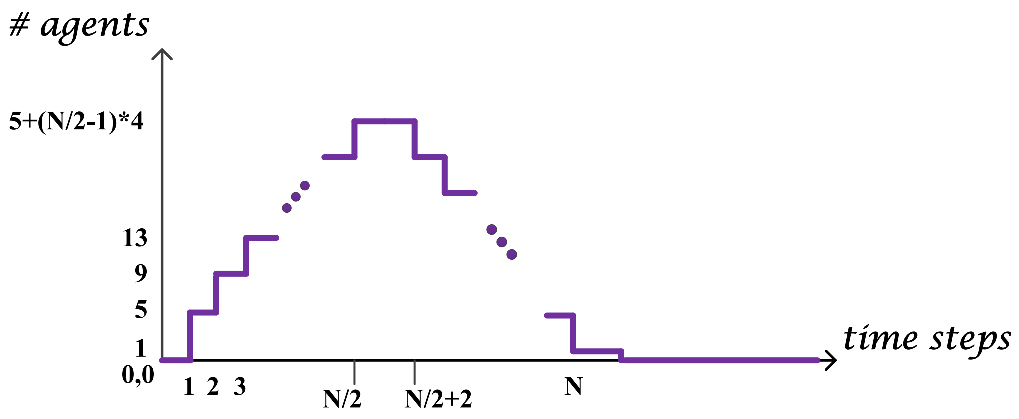


Figure 8: Number of agents present per node for Central-Point (X=Y=N, and N is odd)

The top area of Figure 8 would appear different for the condition of N being an even number, as only 2 agents would be the first to reach the edge of the Places. The next time step the last 2 original agents could reach the edge, as well as 2 spawned agents, and so on.

The top area of Figure 8 would also look different if X and Y were not equal, or X and/or Y were even numbers. The agents migrating in the direction of the smaller dimension would arrive at the edges of Places first and would begin to deposit into the collection earlier, so that the first deposit in time might be for 1 agent (if even dimension) or for 2 agents (if odd dimension). Eventually as the agents migrating in the direction of the larger dimension reach the edge, there could be up to 4 agents making a deposit each time step. Then as the agents in the smaller dimension completed all their deposits, the number of deposits per time step would reduce to just 2 deposits until all the agents in the larger dimension have deposited, with the very last deposit being just 1 agent for the case of the larger dimension being an even number.

3.3 Summary

Table B compares characteristics of the two algorithms, for the condition of X = Y = N.

Table B: Key characteristics of Central-Axes and Central-Point algorithms (X=Y=N)

	Central-Axes	**Central-Point**
Total # of migration/kill steps	N/2 + 1	N
Worst case # of agents depositing into collection in same step	2N - 2	4
Total # of agents created	2N - 1	2N + 3
Worst case # of agents created during initial instantiation	2N - 1	5

Although the Central-Point algorithm requires about double the iterations of migration/kill than for Central-Axes, it has the advantage that the deposits into the central agent are more distributed which may alleviate waits by the agents as they are possibly locked out of the data structure while other agents are depositing.

Currently MASS does the initial instantiation of agents at each node sequentially in that node's process. However, the spawning of agents is done in the parallel threads that are running. So the Central-Point algorithm where worst case only 5 agents are created in the initial instantiation would be faster in this aspect than the 2N-1 agents initially instantiated for the Central-Axes algorithm.

Chapter 4: Implementation

This chapter describes the implementation of the two algorithms from the viewpoint of the user application, as well as the details of how it is implemented inside the MASS library.

The changes to the original MASS library are focused in the two areas of Agents constructor and Agent migrate method. The original version of the agents constructor is specified in Table C [15].

Table C: Original Agents Constructor method specification

public class Agents	**method(arguments)**
public	**Agents(int handle, String className, Object argument, Places places, int initPopulation)** Instantiates a set of agents from the “className” class, passes the “argument” object to their constructor, associates them with a given “Places” matrix, and distributes them over these places, based on the map( ) method that is defined within the Agent class. If a user does not overload it by him/herself, map( ) uniformly distributes an “initPopulation” number of agents. If a user-provided map( ) method is used, it must return the number of agents spawned at each place regardless of the initPopulation parameter. Each set of agents is associated with a user-given handle that must be unique over machines.

The original version of the agent migrate is specified in Table D [15].

Table D: Original Agent Migrate method specification

public abstract class Agent	**method(arguments)**
public boolean	**migrate(int… index)** Initiates an agent migration upon a next call to Agents.manageAll( ). More specifically, migrate( ) updates the calling agent’s index[ ].

These original methods are modified as described in section 4.1.

4.1 MASS API for Enhanced Agents

Two boolean variables (flags) are provided to indicate status of the enhanced agents. They indicate if the enhanced agent is the most center agent and if the agent is currently at an edge location of a node's Places.

In addition, two MASS API methods have been overloaded to provide the enhanced agents to the user; they are the constructor for the Agents class and the migrate method of the Agent abstract class.

4.1.1 Enhanced Agent flags

MASS now provides two new flags for the enhanced agents which ease the implementation of the user application; they are named "iAmCollector" and "atLastLocation" (see Table E).

The "iAmCollector" flag is set TRUE for the one agent that is located in the very center point of the node. This center/collector agent does not move during migrate, but rather sits waiting for the other enhanced agents to provide their gathered data before they self-kill. In the case of an information diffusion application, this center agent does not need to instantiate a central container.

The "atLastLocation" is set FALSE, until the enhanced agent migrates to an edge location of the node. This flag is useful to the user to know when to collect/disperse data at this agent's final place element location, and then to issue a kill( ).

Table E: Enhanced Agent flags specification

public abstract class Agent	**data member**
protected boolean	**iAmCollector** TRUE = this agent is the most center agent. FALSE = this agent is not the most center agent.
protected boolean	**atLastLocation** TRUE = this agent is at the last location (edge) for this node. FALSE = this agent is not located on the edge of this node's Places.

4.1.2 Enhanced Agents Constructor method

The original version of MASS instantiates a user-provided number of agents and does a default distribution of assigning an equal number of agents to each place, with any remaining agents assigned one by one to the place elements starting from the beginning of the Places until all agents have been assigned. This default distribution can be overridden by a user-provided map( ) function, allowing the user to code their own distribution pattern to the initial agents.

For the enhanced agents, the MASS library provides a new overloaded Agents constructor where the injection method is specified by the user. MASS then automatically determines the correct number of initial agents for each node and locates them at the proper place elements (see Table F).

Table F: Enhanced Agents Constructor method specification

public class Agents	**method(arguments)**
public	**Agents(byte inject, byte reserved, int handle, String className, Object argument, Places places)** Instantiates a set of agents from the "className" class according to the algorithm indicated by "inject": 1 = Central-Axes injection algorithm 2 = Central-Point injection algorithm "reserved" should always be zero (to remain compatible with future changes). The other arguments are the same as the original version of MASS, and are specified in the Java MASS Spec[15]. The boolean flag "iAmCollector" is set TRUE only for the most center agent.

4.1.3 Enhanced Agent Migrate method

The original version of MASS provides a migrate(int … index) method which requires input of the coordinates of the Places where the agent should move. This means the user must determine/calculate the new coordinates and provide them to MASS.

The enhanced agents, due to their pre-determined diffusion pattern, now use an overloaded method of migrate to provide a simpler interface to the user (see Table G). The end user just calls migrate(byte migrateMode) and MASS now determines the new index for the enhanced agent (see Figure 2-b, line 3).

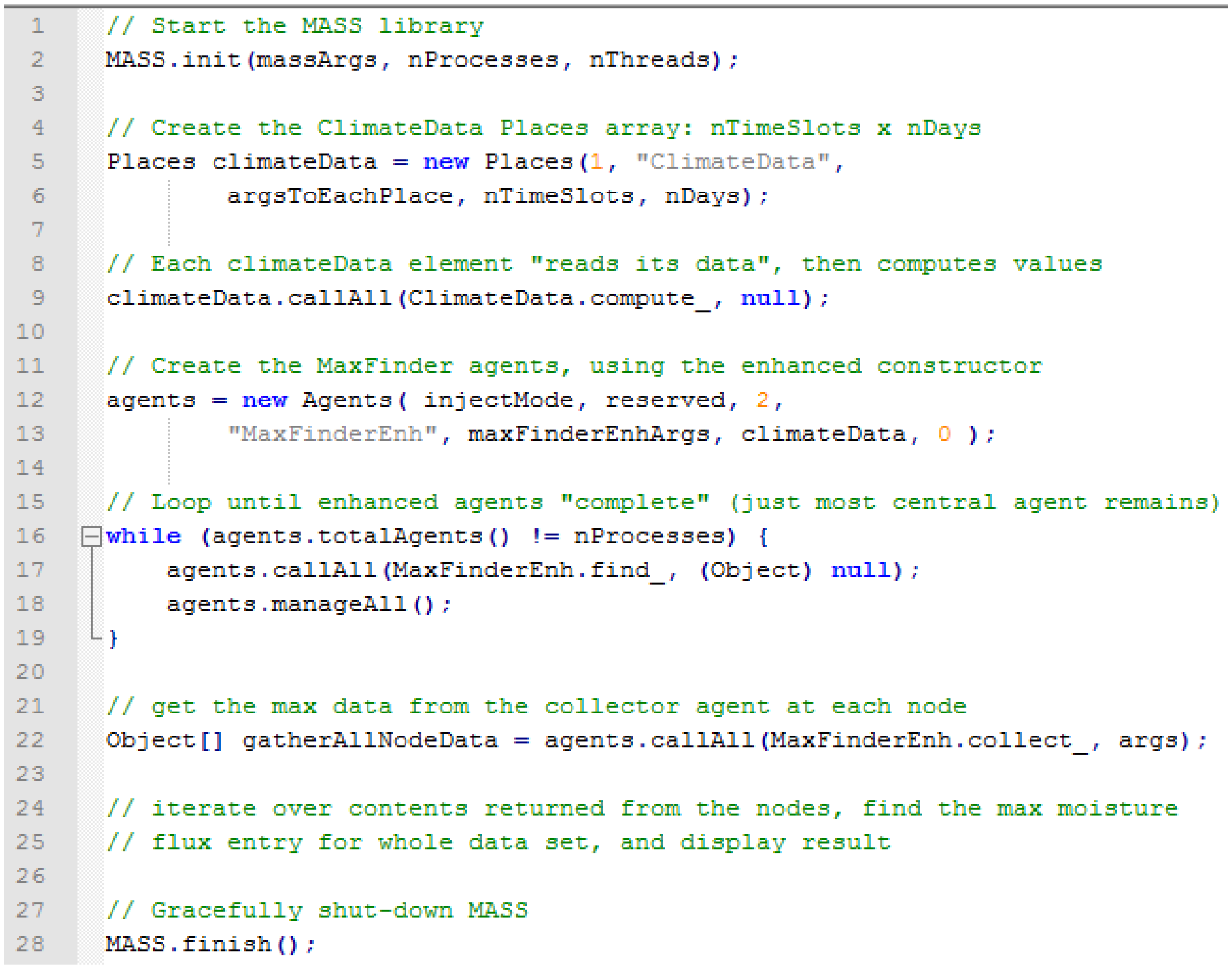
Table G: Enhanced Agent Migrate method specification

public abstract class Agent	**method(arguments)**
public boolean	**migrate(byte diffusion)** Initiates an agent migration upon a next call to Agents.manageAll( ). "diffusion" indicates technique of migration to use: 1 = Central-Axes diffusion algorithm 2 = Central-Point diffusion algorithm (includes auto-spawn as needed) Returns TRUE if migration scheduling was successful, FALSE otherwise. The boolean flag "atLastLocation" is updated if the new location will be at the edge location of the node's place elements.

4.2 User Application Example

A user application, which searches for the maximum moisture flux across data for climate analysis, was written to test, exercise, and evaluate the enhanced agents. This code assumes each node will have only one maximum value, and that the master node will find only one system wide maximum value when it does the final sort of values returned from each node.

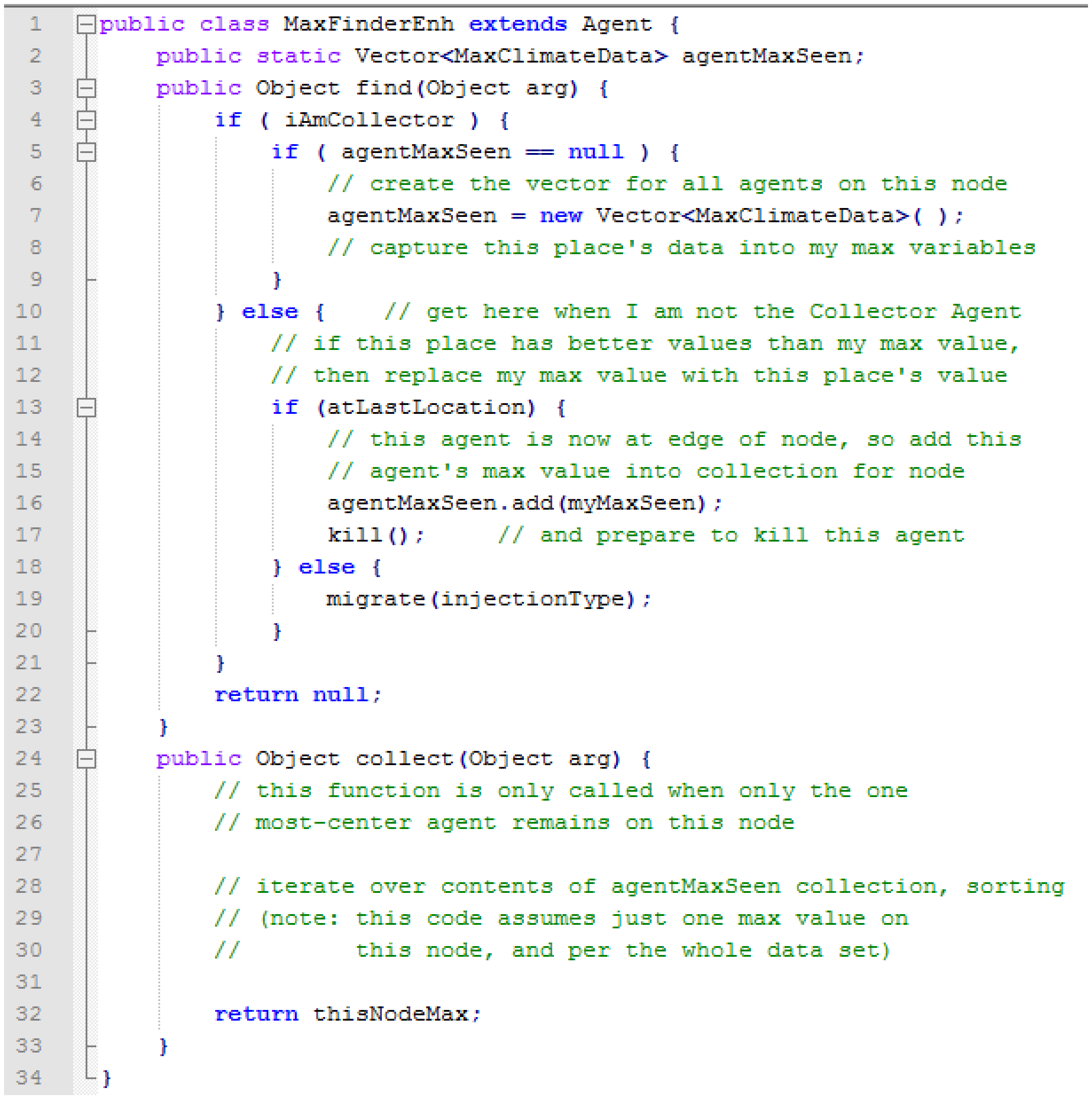
A summary of the user's main program is shown in Figure 9. Initial injection of enhanced agents is done in lines 12-13. The while loop starting at line 16 is where most of the work of the agents occurs. This loop continues until only the collector agent on each node is remaining. Then line 22 calls each collector agent to do a sort of that node's collected values and return that node's maximum value.



```
1   // Start the MASS library
2   MASS.init(massArgs, nProcesses, nThreads);
3
4   // Create the ClimateData Places array: nTimeSlots x nDays
5   Places climateData = new Places(1, "ClimateData",
6           argsToEachPlace, nTimeSlots, nDays);
7
8   // Each climateData element "reads its data", then computes values
9   climateData.callAll(ClimateData.compute_, null);
10
11  // Create the MaxFinder agents, using the enhanced constructor
12  agents = new Agents( injectMode, reserved, 2,
13          "MaxFinderEnh", maxFinderEnhArgs, climateData, 0 );
14
15  // Loop until enhanced agents "complete" (just most central agent remains)
16  while (agents.totalAgents() != nProcesses) {
17      agents.callAll(MaxFinderEnh.find_, (Object) null);
18      agents.manageAll();
19  }
20
21  // get the max data from the collector agent at each node
22  Object[] gatherAllNodeData = agents.callAll(MaxFinderEnh.collect_, args);
23
24  // iterate over contents returned from the nodes, find the max moisture
25  // flux entry for whole data set, and display result
26
27  // Gracefully shut-down MASS
28  MASS.finish();
```

Figure 9: User Application Main

The enhanced agents' user code (which extends Agent) summary is shown in Figure 10. This contains the "find" and "collect" methods that are called from the user main (refer to lines 17 and 22 of Figure 9). The "find" method demonstrates how to use the "iAmCollector" flag (line 4), and the "atLastLocation" flag (line 13).



```java
public class MaxFinderEnh extends Agent {
    public static Vector<MaxClimateData> agentMaxSeen;
    public Object find(Object arg) {
        if ( iAmCollector ) {
            if ( agentMaxSeen == null ) {
                // create the vector for all agents on this node
                agentMaxSeen = new Vector<MaxClimateData>( );
                // capture this place's data into my max variables
            }
        } else {    // get here when I am not the Collector Agent
            // if this place has better values than my max value,
            // then replace my max value with this place's value
            if (atLastLocation) {
                // this agent is now at edge of node, so add this
                // agent's max value into collection for node
                agentMaxSeen.add(myMaxSeen);
                kill();     // and prepare to kill this agent
            } else {
                migrate(injectionType);
            }
        }
        return null;
    }
    public Object collect(Object arg) {
        // this function is only called when only the one
        // most-center agent remains on this node

        // iterate over contents of agentMaxSeen collection, sorting
        // (note: this code assumes just one max value on
        //         this node, and per the whole data set)

        return thisNodeMax;
    }
}
```

Figure 10 : User Application extends Agent

The full user code is provided in the Appendix, starting on page 40.

4.3 MASS Library internal changes

4.3.1 New Action Message for Enhanced Agents Constructor

A new action message was needed, as the Agents constructor has now been overloaded with the "injectionMode" and "reserved" parameters and they must be passed across the SSH link between the Master Node and each Remote Node. This required small changes to Message.java and Constants.java files of MASS.

4.3.2 Constructor for Enhanced Agents

The Enhanced Agents have additional data members as compared to the original MASS Agents. In addition to "iAmCollector" and "atLastLocation" flags mentioned in section 4.1.1, they also have a few variables for use by internals of MASS library.

There is a three element int array called "directionToMove". As agents are created, the "directionToMove" is filled with the value that is added to the enhanced agent's current location when the migrate function is called. In a two dimensional Places, we can think of 4 directions of movement: north, south, east, and west (refer to Figure 3 and Figure 6).

Table H : Enhanced Agents "directionToMove" settings

	directionToMove[0]	directionToMove[1]	directionToMove[2]
north	0	+1	0
south	0	-1	0
east	+1	0	0
west	-1	0	0
center/collector	0	0	0

Also, an int named “agentType” is used to describe the type of enhanced agent, per Table I.

Table I : Enhanced Agents “agentType”

agentType	meaning
0	an original agent
1	a freshly spawned agent
2	an agent who cannot spawn a child

4.3.3 Migrate for Enhanced Agents

When the user application calls migrate for Enhanced Agents, the MASS library calculates the new index for the agent by adding the directionToMove array to the current location of the agent. If the Enhanced Agent is an original agent of the Central-Point type, MASS also calls the spawn method, setting the child’s directionToMove to 90 degrees to the left of the direction of travel of the parent.

4.4 Summary

This implementation was done in a step by step manner, with a testing and verification phase after each small step was implemented. Extensive use was made of MASS’ result logs and error logs for each node to assist in verification and debugging. The user application that was developed has various run modes so that the Central-Axes and Central-Point algorithms can be exercised, as well as run modes for exercising a solution using the original MASS agents (no enhanced agents) and for running a solution that does not use agents at all. In the next chapter we present the evaluation of these various run modes, comparing programmability and execution performance.

Chapter 5: Evaluation

In this chapter we compare the programmability of using the Central-Axes and Central-Point enhanced agents against the code required of the user to arrive at the same the end result without using the enhanced agents. Also, the execution times of the two algorithms are compared, as well as against user code that does not use the enhanced agents.

5.1 Programmability

Figure 9 and Figure 10 (starting on page 21) show the user main and extends agent codes for using either the Central-Axes or the Central-Point enhanced agents (just the injection mode parameters would have a different value). Of particular note is that the user does not need to specify how many or exactly where agents are to be instantiated or spawned, nor give precise index locations for the migrate instruction.

The user does not know how MASS distributes the Places array across the computing nodes. One reason MASS wants this hidden from the user is that if MASS changes the distribution technique in the future, then the user code is not impacted. The user, of course, does know the overall size of Places, so one way a user could write their code using the original MASS agents is to instantiate an agent at each Y=0 location, and then have agents collect data as they migrate along the Y direction, staying in the same X location. Once the agents arrive at the maximum Y location, then the user main could gather the maximum value seen by each agent and then sort to find the overall maximum data. Figure 11 shows this migration pattern.

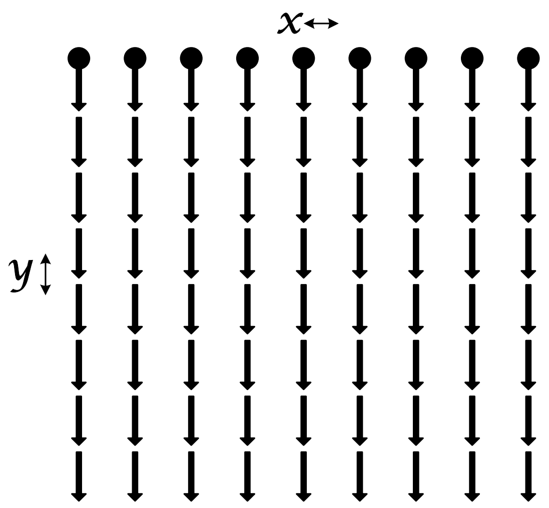
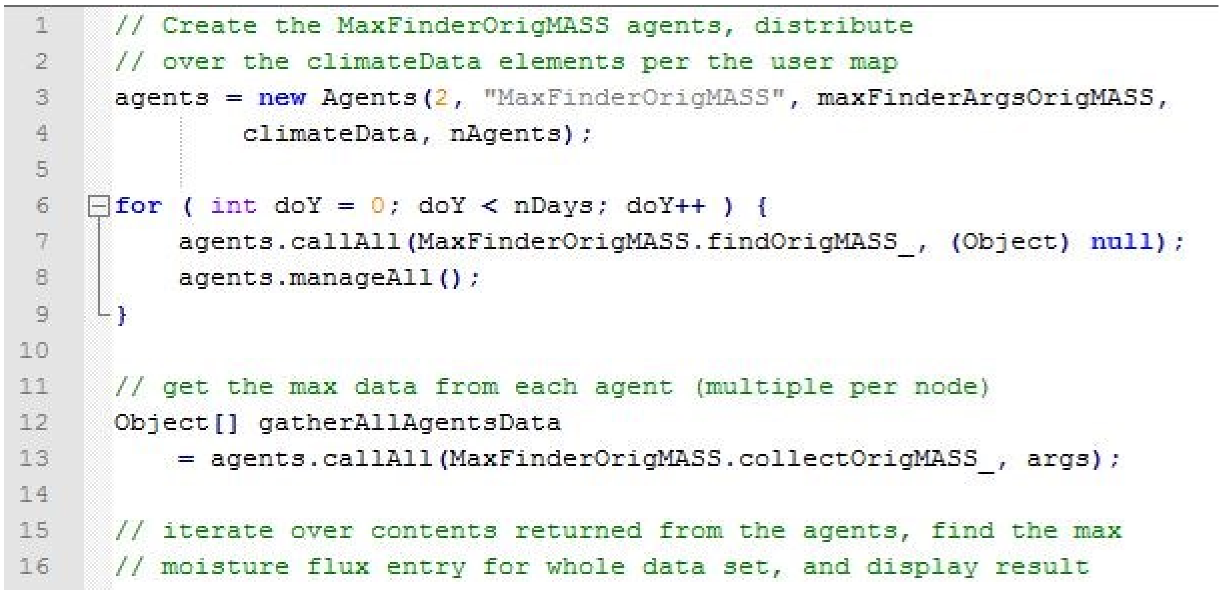


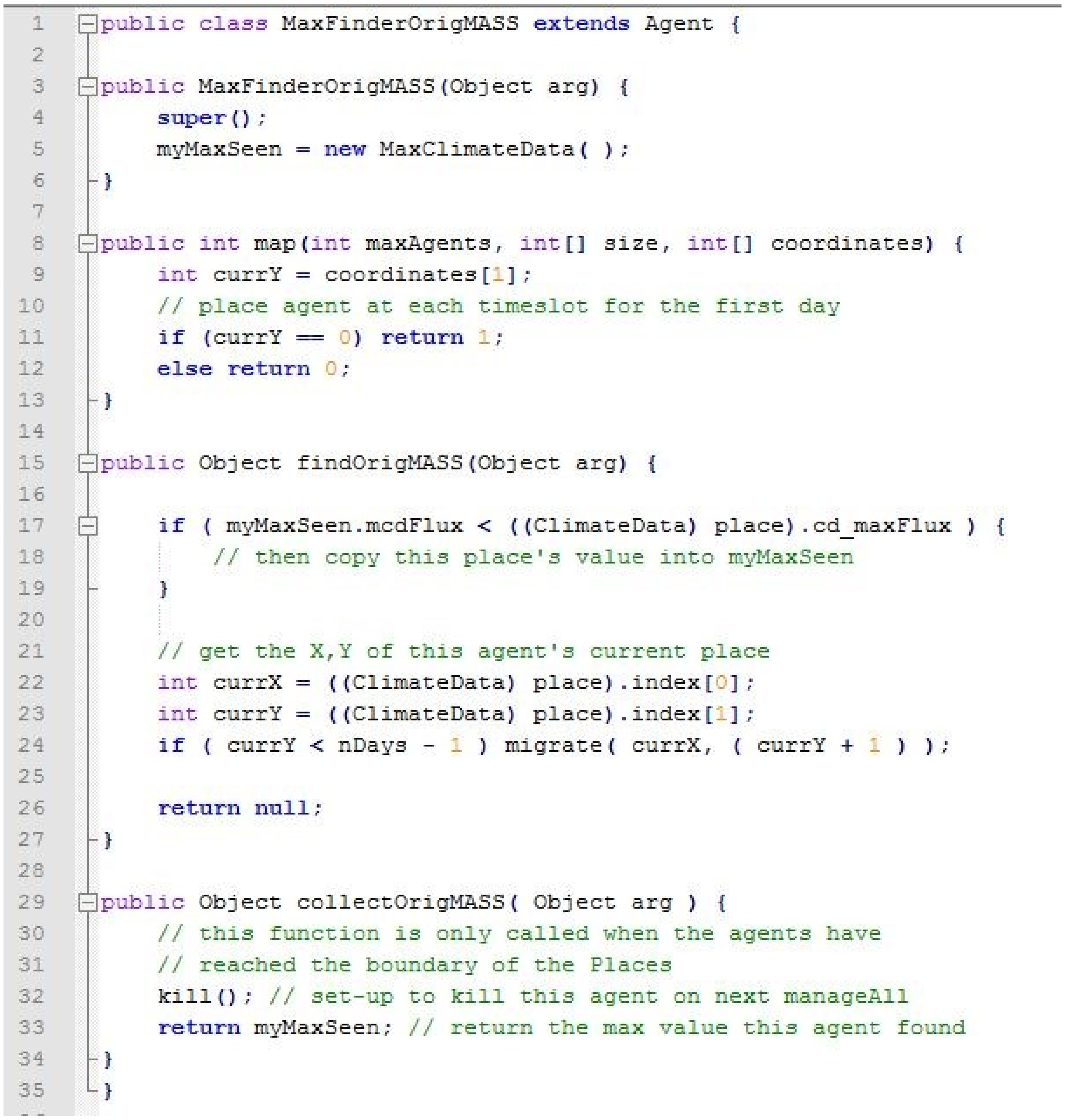
Figure 11: Using Original MASS Agents

Figure 12 and Figure 13, starting on page 26, show the application the original MASS agents to find a maximum climate data value.



```
// Create the MaxFinderOrigMASS agents, distribute
// over the climateData elements per the user map
agents = new Agents(2, "MaxFinderOrigMASS", maxFinderArgsOrigMASS,
        climateData, nAgents);

for ( int doY = 0; doY < nDays; doY++ ) {
    agents.callAll(MaxFinderOrigMASS.findOrigMASS_, (Object) null);
    agents.manageAll();
}

// get the max data from each agent (multiple per node)
Object[] gatherAllAgentsData
    = agents.callAll(MaxFinderOrigMASS.collectOrigMASS_, args);

// iterate over contents returned from the agents, find the max
// moisture flux entry for whole data set, and display result
```

Figure 12: User main code using Original MASS agents



```
public class MaxFinderOrigMASS extends Agent {

public MaxFinderOrigMASS(Object arg) {
    super();
    myMaxSeen = new MaxClimateData( );
}

public int map(int maxAgents, int[] size, int[] coordinates) {
    int currY = coordinates[1];
    // place agent at each timeslot for the first day
    if (currY == 0) return 1;
    else return 0;
}

public Object findOrigMASS(Object arg) {

    if ( myMaxSeen.mcdFlux < ((ClimateData) place).cd_maxFlux ) {
        // then copy this place's value into myMaxSeen
    }

    // get the X,Y of this agent's current place
    int currX = ((ClimateData) place).index[0];
    int currY = ((ClimateData) place).index[1];
    if ( currY < nDays - 1 ) migrate( currX, ( currY + 1 ) );

    return null;
}

public Object collectOrigMASS( Object arg ) {
    // this function is only called when the agents have
    // reached the boundary of the Places
    kill(); // set-up to kill this agent on next manageAll
    return myMaxSeen; // return the max value this agent found
}
}
```

Figure 13: User extends agent using Original MASS agents

The following list of items is what the user code must provide when using the original MASS agents to solve this application:

1. User must specify the initial number of agents to instantiate (line 4 of Figure 12).
2. User must override map function to place these initial agents (line 8-12 of Figure 13).
3. User must specify index for migrate (lines 22-24 of Figure 13).
4. User must specify number of times to loop on callAll/manageAll (lines 6-8 of Figure 12).

The above 4 items are abstracted away from the user when using the enhanced agents, improving programmability.

5.2 Execution Times

Performance was measured for the user application of finding the maximum moisture flux across climate data for these three types of agents:

1. Central-Axes enhanced agents
2. Central-Point enhanced agents
3. Original MASS API agents

Also, performance was measured for the same application without using agents, but rather having each place element return their maximum value to the master node, and the user code then sorts the data to find the maximum value. This solution uses places.callAll method, and all remote nodes send a value from every place element on their node back to the master node. The sorting is then done in sequential user code on the master node. As the number of computing nodes increases, and the amount of memory available to hold larger climate data sets on each node increases, this approach of solving the problem without using distributed agents could become infeasible.

The computing nodes used for these performance evaluations are the machines in the UW1-320 Linux laboratory on the UW Bothell campus. The machines are shared with all other students taking Computing & Software Systems courses, so measurements were done in the early morning hours to minimize impacts from usage by other students. A summary of the machine specifications is given in Table J, and more detailed information can be found in the Appendix starting on page 64.

Table J: Machine Specifications

feature	uw1-320-01.uwb.edu thru uw1-320-15.uwb.edu
RAM	16 GB
CPU	Intel i7-3770 CPU @ 3.40GHz, 4 cores, 8 threads, 8 MB Intel smart cache
Network	1Gbps full-duplex links to 10Gbps backplane LAN switch
OS	Ubuntu 12.10 Quantal Quetzal

The following Table K details the options used when making these performance measurements.

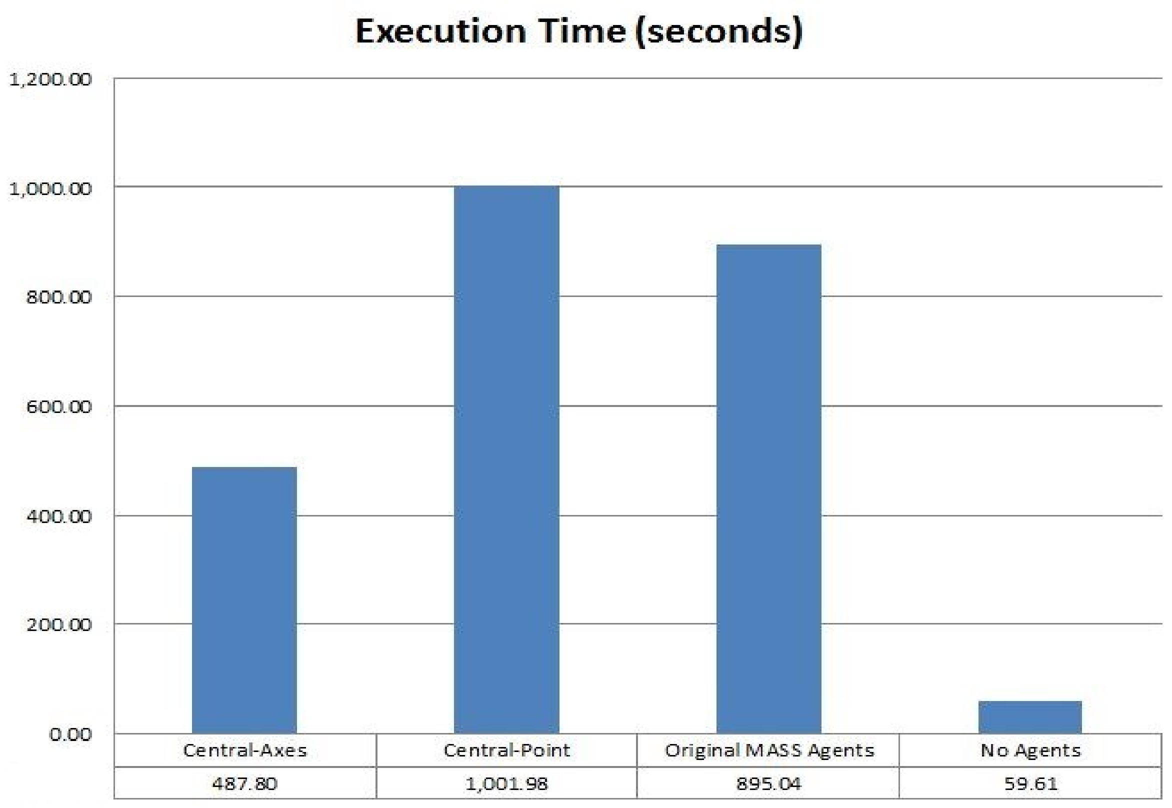
Table K: Options used when running MASS

option	**value**
# nodes	15
# threads/node	6
Places size	20,985 x 1,399 (MASS distributes 1,395 x 1,395 per node)
gridX, gridY	2, 2 (this determines the amount of climate data at each place element)
DLB	disable MASS dynamic thread load balancing (currently only implemented for places, not agents)
JVM	-Xms8g –Xmx12g (heap size of 8 GB min. and 12 GB max.)

Table L summarizes the measurement of execution times for the four different run modes, and Figure 14 shows the average times in a column chart.

Table L: Performance Measurements

	Places of 20,985 x 1,399	
run mode	Measurements taken (seconds)	Average Execution time (seconds)
Central-Axes	482.34, 487.68, 484.67, 496.52	487.80
Central-Point	1015.16, 996.92, 996.89, 998.94	1,001.98
Original MASS agents	898.05, 891.83, 901.76, 888.51	895.04
Not using agents	59.94, 59.21, 59.17, 60.13	59.61



Central-Axes	Central-Point	Original MASS Agents	No Agents
487.80	1,001.98	895.04	59.61

Figure 14: Chart of Execution Times of different solutions

The measurements clearly show that the best performance is achieved by not using agents, at least not in the current MASS implementation. As soon as the early performance numbers became available, it was obvious there was some issue with performance of agents in MASS. Much effort has been spent investigating this very surprising result and is discussed later in this section.

The Central-Point agents take about double the time of the Central-Axes agents. This resonates with the previous analysis of the two algorithms as summarized in Table B on page 15, where it shows that Central-Point requires about two times the number of migrate/kill step iterations.

Using the original MASS agents, the user code does N (dimension size of Places) loops of migrate/kill loop (see line 6 of Figure 12), which is similar to the Central-Point algorithm. The measurements show the Original MASS agents running closer to the time for Central-Point than to Central-Axes times.

5.2.1 Investigation into performance problem of Agents

First, we show a theoretical performance analysis of places-based (or array-based) versus agent-based data analysis. In this study we instantiate a two-dimensional array of scientific data (such as climate data) over multiple computing nodes, and each place element finds its local value of interest (such as maximum temperature). The places-based analysis reads this local value from each place element into the main program and locates the overall value of interest, whereas the agent-based analysis dispatches agents to and marches them over the distributed scientific data.

Our performance estimation focuses on only communication overheads, assuming that data analysis is finding the maximum number and thus is negligibly small. We use the parameters summarized below:

- Each callAll( ) and manageAll( ) spends the same round-trip time as ICMP ping, (i.e., 400usec).
- Each array element provides the local value of interest struct of 20 bytes {long data; double direction; int xGrid, yGrid; int day; int timeslice}.
- The overall two-dimensional array is X * Y.
- The number of computing nodes is Z.

The places-based analysis executes only one callAll( ) that elapses time PerfP:

$$\text{PerfP} = (20 \text{ bytes} * (X / Z) * Y) / 1\text{Gbps} + 400\text{us}) * (Z - 1)$$

$$= (0.16\text{usec} * (X / Z) * Y + 400) * (Z - 1) \text{ usec}$$

The agent-based analysis, particularly central axes, needs (max((X/Z),Y) / 2 + 1) iterations of callAll( ) and manageAll( ) that need time PerfA:

$$\text{PerfA} = (800\text{usec} * (\max((X/Z),Y) / 2 + 1) + (20 \text{ bytes} / 1\text{Gbps})) * (Z - 1)$$

$$= (400 * \max((X/Z),Y) + 800.16) * (Z – 1) \text{ usec}$$

If X= 20985, Y = 1399, and Z = 15, then

$$\text{PerfP} = (0.16\text{usec} * (20985/15) * 1399 + 400\text{usec}) * 14$$

$$= (0.16\text{usec} * 1399 * 1399 + 400\text{usec}) * 14 = 4{,}389{,}730\text{usec} = 4.39\text{sec}$$

$$\text{PerfA} = (400\text{usec} * \max((20985/15),1399) + 800.16\text{usec}) * 14$$

$$= (400\text{usec} * 1399 + 800.16\text{usec}) * 14 = 7{,}845{,}602\text{usec} = 7.85\text{sec}$$

Therefore, we estimate that the places-based analysis is 1.8 times faster than the agent-based approach. However, this is because agents are driven every invocation of callAll( ) and manageAll( ), which takes 400usec per remote computing node. This implies that, if agents migrate asynchronously with each invocation of callAll/migrateAll or at least if agents need a synchronization with the main program only every two or three invocations of callAll/migrateAll, the agents-based analysis would outperform the places-based analysis. This is listed in our future work section on page 35.

However, the measurements shown in Figure 14 show a ratio of about 8:1 for the time for Central-Axes agents compared to the places-based (or not using agents) solution. A closer examination of the MASS Agents and Agent classes revealed two items that are also impacting the performance of the agents:

1. For each callAll and each manageAll, every place element is examined, looking for agents that reside on that place before executing the called agent's function. In the case of enhanced agents, and many user applications using agents, the agents are sparsely populated over the place elements. So in the case of Central-Axes enhanced agents, where only 2N-1 total agents are located on a node, all N*N place elements are searched for every callAll and every manageAll. When an agent is found, it is assigned to a thread and when complete it is returned. A new technique is being investigated to assign equal chunks of agents to the available threads, which will lead to better performance than the current scheme.
2. Vectors are used for storing the agents, both for the overall bag of agents on each node, and for a local collection of agents at each place element. The Java Vector data structure is thread-safe and is easily used for multi-threaded applications. However, Vector is not the best performing data structure. We have started designing a new data structure for holding agents, which is showing significant performance improvements during initial investigations.

We feel confident that with all the above improvements, the agents performance will greatly improve.

Chapter 6: Conclusion

This section summarizes our investigation, describes problems encountered, and lists possible future work.

6.1 Result Summary

Our main focus in this research was to investigate adding enhancements to the reactive agents in a parallel simulation and data analysis system. Our goal was better programmability and performance by providing automatic injection and diffusion, merger and termination, to these enhanced agents.

The goal of better programmability with enhanced agents has been shown by abstracting away four details listed at the end of section 5.1 on page 28.

Performance goals have not yet been fully met, but a multi-part strategy addressing this was outlined in section 5.2.1. If technology continues to increase the number of CPU cores and system memory at a much faster rate than network speed increases, then not using distributed agents to solve problems of information searching or analyzing big data could become infeasible.

6.2 Problems Encountered

Through this research we have identified the following two problems.

6.2.1 Consider allowing Enhanced Agents to run more asynchronously

Since the enhanced agents are on a particular mission with a predetermined pattern that avoids collisions, they do not need to be synchronized after each agents.callAll and agents.manageAll.

The amount of time required for synchronization, which requires network messages between the master and remote nodes, is a costly overhead and if greatly reduced would improve performance. This was previously discussed in detail in section 5.2.1.

6.2.2 Faster Data Structure for Agents, along with improved Agent Servicing Scheme

Currently in MASS, there is one bag of agents per node. During each simulation cycle for agents, the multiple threads check-out an agent, execute, and return that agent one at a time until all agents are executed, as previously discussed in section 5.2.1. Early investigations into creating a new data structure called "agents-array-list", along with a new agent servicing process show signs of improving performance. During each simulation cycle for agents, a range of agents would be assigned to each thread for execution, instead of one agent at a time.

6.3 Future Work

In addition to addressing the suggested improvements to performance, we see two other areas for future work in enhancing the management of agents: include support of the enhanced agents for other than two-dimensional environments, and an automatic random injection of agents.

6.3.1 Enhanced Agents for other than two-dimensional environments

The enhanced agents have only been implemented for a two-dimensional Places. Support for enhanced agents in a one-dimensional and a three-dimensional Places could be implemented. A three-dimensional solution only using the original MASS agents would be especially cumbersome for the user to write.

6.3.2 General agents feature of Auto-injection with Random placement

When using the original MASS agents a default distribution technique is provided, which can be optionally modified by the user code providing its own map( ) function, as discussed in section 4.1.2 on page 18. There are many applications that would like a random placement of their initial population of agents. So another agent management feature we see as being useful is an automatic injection of agents that have a random placement in the environment. A sample application is the Wa-Tor simulation, where there are predators (shark agents) and prey (fish agents) [16]. Parameters could include an optional user supplied random seed and an initial number of agents.

References

[1] T. Chuang and M. Fukuda, “A Parallel Multi-Agent Spatial Simulation Enviornment for Cluster Systems,” in *Proc. of the 16th IEEE International Conference on Computational Science and Engineering*, 2013, pp. 143–150.

[2] H. S. Nwana, “Software agents: an overview,” *Knowl. Eng. Rev.*, vol. 11, no. 03, pp. 205–244, Jul. 1996.

[3] D. Hiebeler, “The swarm simulation system and individual-based modeling,” 1994-11-065, 1994.

[4] N. Minar, R. Burkhart, C. Langton, and M. Askenazi, “The swarm simulation system: A toolkit for building multi-agent simulations,” 1996.

[5] “Swarm Project Homepage.” [Online]. Available: http://www.swarm.org.

[6] M. Niazi and A. Hussain, “Agent-based computing from multi-agent systems to agent-based models: a visual survey,” *Scientometrics*, vol. 89, no. 2, pp. 479–499, Aug. 2011.

[7] “Distributed Systems Laboratory at University of Washington Bothell.” [Online]. Available: http://depts.washington.edu/dslab/.

[8] P. M. Borst, “An Overview of the WAVE Language and System for Distributed Processing in Open Networks,” no. June, 1994.

[9] M. Fukuda, L. F. Bic, M. B. Dillencourt, and F. Merchant, “Messengers: Distributed programming using mobile agents,” *J. Integr. Des. Process Sci.*, vol. 5, no. 4, pp. 95–112, 2001.

[10] M. Fukuda, N. Suzuki, L. M. Campos, and S. Kobayashi, “Programmability and Performance of M++ Self-Migrating Threads,” in *Proc. of the IEEE Int’l Conference on Cluster Computing*, 2001, pp. 331–340.

[11] N. Collier and M. North, “Parallel agent-based simulation with Repast for High Performance Computing,” *Simulation*, vol. 89, no. 10, pp. 1215–1235, Nov. 2012.

[12] “Repast HPC Project Homepage.” [Online]. Available: http://repast.sourceforge.net/repast_hpc.html.

[13] G. Cordasco, R. De Chiara, A. Mancuso, D. Mazzeo, V. Scarano, and C. Spagnuolo, “Bringing together efficiency and effectiveness in distributed simulations: The experience with D-Mason,” *Simulation*, vol. 89, no. 10, pp. 1236–1253, Jun. 2013.

[14] "D-MASON Project Homepage." [Online]. Available: http://www.dmason.org/.

[15] M. Fukuda, "MASS : Parallel-Computing Library for Multi-Agent Spatial Simulation (MASS Java Specification)," Bothell, WA, 2010.

[16] A. Dewdney, "Sharks and fish wage an ecological war on the toroidal planet Wa-Tor," *Sci. Am.*, vol. 251, no. 6, pp. 14–22, 1984.

Appendix A: Source Code – Climate Analysis example MASS application

A.1 User Main

```
// ClimateAnalysisMass.java   -- Cherie Wasous 3.5.2014
//
// This used for exercising and evaluating the enhanced agents of MASS.
// It is a “toy application” based on Climate Analysis, modified to
// represent very large climate data sets in the system memory sizes of
// future generation technology.


import java.util.Date;
import java.util.Iterator;
import java.util.Vector;

import MASS.Agents;
import MASS.Constants;
import MASS.MASS; // Library for Multi-Agent Spatial Simulation
import MASS.Places;

public class ClimateAnalysisMass {

    // runMode
    // = 0: only use places to do the job (so this main finds max)
    // = 201: use central axes enhanced agents
    // = 301: use original MASS agents at every y=0, then march down y

    private static final int USE_NO_AGENTS = 0;
    private static final int USE_ENH_AGENTS_CENTRAL_AXES = 201;
    private static final int USE_ORIG_API_STARTY0_MIGRATEY = 301;

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

        Date startMassInitTime = new Date();
        Date stopMassInitTime, stopInitPlacesTime, stopPlacesComputeTime,
          startAgentsLoopTime, stopPlacesCollectTime, stopUserSearchTime,
          startMassFinishTime, stopMassFinishTime, stopPlacesMyMaxGatherTime;

        // Verify number of arguments
        if (args.length < 11) {
            System.err.println("\nUsage:\n\tjava ClimateAnalysisMass "
                + "login pass port nAgents nProc nThrds nTimeSlots "
                + "nDays runMode weatherGridXrange weatherGridYrange");
            System.exit(-1);
        }
        // Set variables with the user input data
        String login = args[0];
        String pass = args[1];
        String port = args[2];
        int nAgents = Integer.parseInt(args[3]);
        int nProcesses = Integer.parseInt(args[4]);
        int nThreads = Integer.parseInt(args[5]);

        // nTimeSlots -> this is X dimension
        int nTimeSlots = Integer.parseInt(args[6]);
        // nDays -> this is Y dimension
        int nDays = Integer.parseInt(args[7]);

        // runMode (see definitions at top of file)
        int runMode = Integer.parseInt(args[8]);

        // set size of weather grid for each place element
        int gridXrange = Integer.parseInt(args[9]);
        int gridYrange = Integer.parseInt(args[10]);

        // prepare MASS arguments
        String[] massArgs = new String[4];
        massArgs[0] = login; // user login
        massArgs[1] = pass; // user password
        massArgs[2] = "machinefile.txt"; // machine file
        massArgs[3] = port; // port

        // Start the MASS library
        MASS.init(massArgs, nProcesses, nThreads);

        stopMassInitTime = new Date();
        long timeMassInit = (stopMassInitTime.getTime() - startMassInitTime
                .getTime());
        System.out.println("\nt--> Time spent thru MASS init: " + timeMassInit
                + " ms");

        // Create the ClimateData Places array: nTimeSlots x nDays
        //
        // Each place element contains a grid for the Pacific NW, which
        // in a real application is 123 x 162 locations. So at each location
        // in this grid is the climate information that that particular
        // timeSlot and day.  However for this exercising app, we allow
        // for a programmable gridXrange and gridYrange locations.
        int chunk = nTimeSlots / nProcesses;

        Object[] argsToEachPlace = new Object[5];
        argsToEachPlace[0] = chunk;
        argsToEachPlace[1] = gridXrange;
        argsToEachPlace[2] = gridYrange;
        argsToEachPlace[3] = nDays;
        argsToEachPlace[4] = nTimeSlots;
```

```
Places climateData = new Places(1, "ClimateData", argsToEachPlace,
                                nTimeSlots, nDays);

stopInitPlacesTime = new Date();
long timeInitPlaces = (stopInitPlacesTime.getTime() - stopMassInitTime
        .getTime());
System.out
        .println("\nt--> Time spent instantiating Places "
                + "climateData: " + timeInitPlaces + " ms");

// Each climateData element "reads its data", then computes values
climateData.callAll(ClimateData.compute_, null);

stopPlacesComputeTime = new Date();
long timePlacesCompute = (stopPlacesComputeTime.getTime()
                        - stopInitPlacesTime.getTime());
System.out.println("t--> Time spent call all Places 'compute_': "
        + timePlacesCompute + " ms");

// **************************************************************
// runMode Switch statement
//
// **************************************************************
switch (runMode) {

// **************************************************************
// USE_NO_AGENTS
//
// .runMode of using no agents, so just via place callAll gather all
// values and do sequential sort on this Master node
//
// .uses original MASS Agents constructor
//
// **************************************************************
case USE_NO_AGENTS: // do all work without using any agents

    // gather the max values from each place element
    Object[] tempArgs = new Object[nDays * nTimeSlots];
    Object[] temp = climateData.callAll(ClimateData.myMax_,
                                    tempArgs);

    stopPlacesMyMaxGatherTime = new Date();

    long timePlacesMyMaxGather = 0;
    timePlacesMyMaxGather += (stopPlacesMyMaxGatherTime.getTime()
                            - stopPlacesComputeTime.getTime());
    System.out.println("\nt--> Time spent to gather myMax "
                            + "climateData from each Place: "
                            + timePlacesMyMaxGather + " ms");
    // look thru returned values and find the max
    MaxClimateData overallMax = new MaxClimateData();
    overallMax.mcdFlux = 0; // set to very low value

    for (int i = 0; i < temp.length; i++) {
        MaxClimateData nextValue = (MaxClimateData) temp[i];
        if (overallMax.mcdFlux < nextValue.mcdFlux) {
            overallMax = nextValue;
        }
    }

    System.out.println("\nMax value found at day=" + (overallMax.mcdDay +1)
            + " and time=" + (overallMax.mcdTime +1)
            + ", with flux=" + overallMax.mcdFlux
            + ", direction=" + overallMax.mcdDir
            + ", at x=" + (overallMax.mcdX + 1) + ", y="
            + (overallMax.mcdY + 1 ));
    if ( ( ( overallMax.mcdDay + 1 ) == nDays ) &&
        ( ( overallMax.mcdTime + 1 ) == nTimeSlots ) &&
        ( ( overallMax.mcdX + 1 ) == ( gridXrange ) ) &&
        ( ( overallMax.mcdY + 1) == ( gridYrange ) ) ) {
     MASS.printResult("$.$.$.$.$.$.$.$.$.$.$.$.$.$      "
        + "CORRECT !!!  :-)     $.$.$.$.$.$.$.$.$.$.$.$.$.$");
    } else {
     MASS.printResult("~~~~~~~~~~~~~~~~~~~~~~~~~~~~   "
        + "NOT correct...  :-(  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
    }

    stopUserSearchTime = new Date();

    long timeUserSearch = stopUserSearchTime.getTime()
            - stopPlacesMyMaxGatherTime.getTime();
    System.out
            .println("\nt--> Time spent in user MAIN of searching "
                + "for max collected: " + timeUserSearch + " ms");

    break;

// ***************************************************************
// USE_ENH_AGENTS_CENTRAL_AXES
//
// .use Enhanced constructor and inject along central axis
// .only for 2-dim right now
// .then N,S,E,W march
// ***************************************************************
case USE_ENH_AGENTS_CENTRAL_AXES:

    // Create the MaxFinder agents, using the enhanced constructor

    Object[] maxFinderEnhArgs = new Object[8];
    maxFinderEnhArgs[0] = USE_ENH_AGENTS_CENTRAL_AXES;
```

```
maxFinderEnhArgs[1] = nTimeSlots;
maxFinderEnhArgs[2] = nDays;
maxFinderEnhArgs[3] = nProcesses;
maxFinderEnhArgs[4] = nThreads;
maxFinderEnhArgs[5] = 0;   // agentType flag
maxFinderEnhArgs[6] = 0;   // parentsDirectionToMove[0]
maxFinderEnhArgs[7] = 0;   // parentsDirectionToMove[0]

byte injectionMode = Constants.AGENTS_INJECT_CONTROLLED;
byte reserved = (byte) 0;

Agents agents = new Agents(injectionMode, reserved, 2,
        "MaxFinderEnh", maxFinderEnhArgs, climateData, nAgents);

startAgentsLoopTime = new Date();

while (agents.totalAgents() != nProcesses) {
    agents.callAll(MaxFinderEnh.find_, (Object) null);
    agents.manageAll();
}

agents.callAll(MaxFinderEnh.find_, (Object) null);

// get the max data from the collector agent at each node
Object[] tempArgsE = new Object[nProcesses];
Object[] tempE = agents.callAll(MaxFinderEnh.collect_, tempArgsE);

MaxClimateData overallMaxE = new MaxClimateData();
overallMaxE = (MaxClimateData) tempE[0];
// iterate over contents returned from each node and
// find the maximum flux entry for whole simulation

for (int i = 1; i < nProcesses; i++ ) {
    MaxClimateData nextE = (MaxClimateData)tempE[i];
    if (overallMaxE.mcdFlux < nextE.mcdFlux ) {
        overallMaxE = nextE;
    }
}

System.out.println("\n\nOverall Max value found at day="
        + (overallMaxE.mcdDay +1)
        + " and time=" + (overallMaxE.mcdTime +1)
        + ", with flux=" + overallMaxE.mcdFlux
        + ", direction=" + overallMaxE.mcdDir
        + ", at x=" + (overallMaxE.mcdX + 1) + ", y="
        + (overallMaxE.mcdY + 1) );
if ( ( ( overallMaxE.mcdDay + 1 ) == nDays ) &&
    ( ( overallMaxE.mcdTime + 1 ) == nTimeSlots ) &&
    ( ( overallMaxE.mcdX + 1 ) == ( gridXrange ) ) &&
    ( ( overallMaxE.mcdY + 1 ) == ( gridYrange ) ) ) {
 MASS.printResult("$.$.$.$.$.$.$.$.$.$.$.$.$.$      "
            + "CORRECT !!!  :-)     $.$.$.$.$.$.$.$.$.$.$.$.$.$.$.$\n");
        } else {
         MASS.printResult("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~   "
        + "NOT correct...  :-(  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n");
        }

        Date stopAgentsLoopTime = new Date();
        long timeAgentsLoop = (stopAgentsLoopTime.getTime()
                                        - startAgentsLoopTime.getTime());

        System.out
                .println("t--> Time spent in loop of agents searching "
                        + "(until totalAgents=0): " + timeAgentsLoop + " ms");

        break;

    // ****************************************************************
    // USE_ORIG_API_STARTY0_MIGRATEY (301)
    //
    // .use original MASS API, but with user supplied map
    // .user does not know how MASS divides up Places across nodes
    //
    // = USE_ORIG_API_STARTY0_MIGRATEY: put an agent at each timeslot
    // for day=0, then migrate these agents thru each day, staying in
    // same timeslot.
    //
    // ****************************************************************
    case USE_ORIG_API_STARTY0_MIGRATEY:

        // Create the MaxFinderPerAPI agents, which will be distributed
        // over the climateData elements per the user map

        Object[] maxFinderArgsPerAPI = new Object[1];
        maxFinderArgsPerAPI[0] = runMode;

        agents = new Agents(2, "MaxFinderPerAPI", maxFinderArgsPerAPI,
                climateData, nAgents);

        startAgentsLoopTime = new Date();

        if ( runMode == USE_ORIG_API_STARTY0_MIGRATEY ) {
            for ( int doY = 0; doY < nDays; doY++ ) {
                agents.callAll(MaxFinderPerAPI.findPerAPI_, (Object) null);
                agents.manageAll();
            }
        } else {
            for ( int doX = 0; doX < nTimeSlots; doX++ ) {
                agents.callAll(MaxFinderPerAPI.findPerAPI_, (Object) null);
                agents.manageAll();
            }
        }
```

```
stopAgentsLoopTime = new Date();

timeAgentsLoop = (stopAgentsLoopTime.getTime() - startAgentsLoopTime
        .getTime());
System.out
        .println("t--> Time spent in loop of agents searching (until "
    + " totalAgents=0): " + timeAgentsLoop + " ms");

// gather the max values found by each agent (multiple per node)
Vector<MaxClimateData> results = new Vector<MaxClimateData>();
tempArgs = new Object[nDays];

temp = agents.callAll(MaxFinderPerAPI.collectPerAPI_, tempArgs);

for (int i = 0; i < temp.length; i++) {
    if (temp[i] != null) {
        MaxClimateData temp3 = (MaxClimateData) temp[i];
        results.add(temp3);
    }
}

stopPlacesCollectTime = new Date();

long timePlacesCollect = 0;
timePlacesCollect += (stopPlacesCollectTime.getTime()
            - stopAgentsLoopTime.getTime());
System.out
        .println("\nt--> Time spent to collect values from all Agents: "
                + timePlacesCollect + " ms");

// now go thru max results from each proc/node & find the real max
// and print to console
Iterator<MaxClimateData> iter = results.iterator();
overallMax = new MaxClimateData();
overallMax.mcdFlux = 0; // set to very low value
MaxClimateData tempCD = new MaxClimateData();

while (iter.hasNext()) {
    tempCD = iter.next();
    if (overallMax.mcdFlux < tempCD.mcdFlux) {
        // code assumes all values are unique, and therefore only 1 max value
        overallMax = tempCD;
    }
}

System.out.println("\n\nOverall Max value found at day="
   + (overallMax.mcdDay +1)
        + " and time=" + ( overallMax.mcdTime +1)
        + ", with flux=" + overallMax.mcdFlux
```

```
                    + ", direction=" + overallMax.mcdDir
                    + ", at x=" + (overallMax.mcdX + 1) + ", y="
                    + (overallMax.mcdY + 1) );
            if ( ( ( overallMax.mcdDay + 1 ) == nDays ) &&
                    ( ( overallMax.mcdTime + 1 ) == nTimeSlots ) &&
                    ( ( overallMax.mcdX + 1 ) == ( gridXrange ) ) &&
                    ( ( overallMax.mcdY + 1 ) == ( gridYrange ) ) ) {
             MASS.printResult("$.$.$.$.$.$.$.$.$.$.$.$.$.$      CORRECT "
                 + "!!!  :-)     $.$.$.$.$.$.$.$.$.$.$.$.$.$");
            } else {
             MASS.printResult("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~   NOT correct"
                    + "...  :-(  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
                    }

            stopUserSearchTime = new Date();

            timeUserSearch = stopUserSearchTime.getTime()
                    - stopPlacesCollectTime.getTime();
            System.out.println("\nt--> Time spent in user MAIN of searching for "
                     + "max collected: "    + timeUserSearch + " ms");
            break;

        default:
            System.out.println("\n$$$ that runMode=" + runMode
                    + " not yet supported, come back later !! ");
            break;
        }

        startMassFinishTime = new Date();

        // Gracefully shut-down MASS
        MASS.finish();

        stopMassFinishTime = new Date();

        long timeMassFinish = (stopMassFinishTime.getTime()
                            - startMassFinishTime.getTime());
        System.out.println("t--> Time for MASS.finish: "
                                   + timeMassFinish + " ms");
        long timeOverall = (stopMassFinishTime.getTime()
                            - stopMassInitTime.getTime());
        System.out.println("\n\nt--> Overall Total Time for whole "
                + "simulation after MASS.init: " + timeOverall + " ms");
        System.out
                .println("\n$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$"
                    + "$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$");

        // Terminate the JVM
        System.exit(0);
    }
}
```

A.2 User extends Agent

```
// MaxFinderEnh.java   -- Cherie Wasous 3.5.2014
// for enhanced agent, injection central axes

import java.util.Iterator;
import java.util.Vector;

import MASS.Agent;

public class MaxFinderEnh extends Agent {

    public static Vector<MaxClimateData> agentMaxSeen;

    private long myMaxFlux;
    private double maxDir;
    private int maxTime, maxDay;
    private int myMaxX, myMaxY;

    private int nTimeSlots;
    private int nDays;
    private int nProcs;

    // Constructor
    // ------------
    public MaxFinderEnh(Object arg) {
        super();

        Object[] args = (Object[]) arg;
        nTimeSlots = (int) args[1];
        nDays = (int) args[2];
        nProcs = (int) args[3];
    }

    // function identifiers - For injection type of central axes
    public static final int find_ = 0;
    public static final int collect_ = 1;

    // this is called from callAll( ) and forwards this call
    // to the appropriate function, based on funcId.
    // ------------------------------------------------------
    @Override
    public Object callMethod(int funcId, Object args) {
        switch (funcId) {
            case find_:
                return find(args);
            case collect_:
                return collect(args);

            default:
                return null;
        }
    }

    //
    // ---------------------------------------------------
    public Object find(Object arg) {

        if (iAmCollector) {

            if ( agentMaxSeen == null ) {
                // Create the vector for all agents on this node.
                // This requires the place elements on this node be
                //   5 elements long on each size minimum.  This
                //   requirement needs to be checked & user given
                //    err message if it is not met.
                // Set some initial capacity so that Vector does
                //   not have to grow too many times.  As each
                //   agent kills-self it does dump, make it that big
                agentMaxSeen = new Vector<MaxClimateData>( ( 2 * nDays ) +
                                ( 2 * ( ( nTimeSlots / nProcs )
                                + ( nTimeSlots % nProcs ) ) ) );
                // add this place's data into my local vector
                this.myMaxFlux = ((ClimateData) place).cd_maxFlux;
                this.myMaxX = ((ClimateData) place).cd_maxX;
                this.myMaxY = ((ClimateData) place).cd_maxY;
                this.maxDir = ((ClimateData) place).
                        cd_direction[myMaxX][myMaxY];
                this.maxTime = ((ClimateData) place).cd_time;
                this.maxDay = ((ClimateData) place).cd_day;

                MaxClimateData temp = new MaxClimateData(this.myMaxFlux,
                        this.maxDir, this.maxTime, this.maxDay,
                        this.myMaxX, this.myMaxY);
                agentMaxSeen.add(temp);
            }
        } else {  // get here when I am not the Collector Agent
            // collect data if this place has better values
            if (this.myMaxFlux < ((ClimateData) place).cd_maxFlux) {
                this.myMaxFlux = ((ClimateData) place).cd_maxFlux;
                this.myMaxX = ((ClimateData) place).cd_maxX;
                this.myMaxY = ((ClimateData) place).cd_maxY;
                this.maxDir = ((ClimateData) place).
                        cd_direction[myMaxX][myMaxY];
                this.maxTime = ((ClimateData) place).cd_time;
                this.maxDay = ((ClimateData) place).cd_day;
            }

            if (atLastLocation) {
                // this agent is now at edge of node, so
                // add this agent's collection into the Vector for node
                MaxClimateData temp = new MaxClimateData(this.myMaxFlux,
                        this.maxDir, this.maxTime, this.maxDay,
                        this.myMaxX, this.myMaxY);
                agentMaxSeen.add(temp);
                // and prepare to kill this agent
                kill();
            } else {
                byte injectionType = (byte) 2; // controlled central axes
                migrate(injectionType);
            }
        }
        return null;
    }

    //
    // -----------------------------------------------------
    public Object collect(Object arg) {
        // this function is only called when there is only one agent
        // remaining on the node, the Collector
        MaxClimateData thisNodeMax = new MaxClimateData();
        // iterate over contents of agentMaxSeen, find the maximum
        // flux entry and return it
        Iterator<MaxClimateData> iter = agentMaxSeen.iterator();
        thisNodeMax = iter.next();
        while (iter.hasNext()) {
            MaxClimateData temp = iter.next();
            if (thisNodeMax.mcdFlux < temp.mcdFlux) {
                thisNodeMax = temp;
            }
        }
        return thisNodeMax;
    }

}
```

Appendix B: Source Code – Enhanced Agents in MASS library

B.1 Enhanced Agents Constructor code inside MASS.Agents.java

```
/**
 * add by Cherie 2-25-2014: Agents Enhancements Constructor
 *
 * Is the constructor that instantiates "className" objects as a collection
 * of multi-agents with characteristics of "inject" mode and
 * "guardedMigration" mode.
 *
 * @throws Exception
 *            if there is a problem with a parameter
 * @param inject
 *           a user-given non-negative number to uniquely identify the
 *           injection technique to use = 1 means "controlled"
 * @param guardedMigration
 *           a user-given non-negative number to uniquely identify the
 *           guarded migration technique to use = 1 means "greedy" (only
 *           one agent per place element at a time)
 * @param handle
 *           a user-given non-negative number to uniquely identify this
 *           collection of distributed multi-agents over the system.
 * @param className
 *           the name of the class from which each agent is instantiated.
 * @param argument
 *           an argument passed to each agent.
 * @param places
 *           a distributed array where new agents are instantiated.
 * @param initPopulation
 *           the total number of agents to be created.
 */
public Agents(int inject, int guardedMigration, int handle,
        String className, Object argument, Places places, int initPopulation)
        throws Exception {
    if (handle < 0) {
        throw new Exception(
                "The handle must be an integer of zero or more.");
    } else if (className == null || className.trim().length() == 0) {
        throw new Exception("The class name must be a valid class name");
    } else if (places == null) {
        throw new Exception("The MASS.Places object cannot be null.");
    } else if (initPopulation < 0) {
```

```
    throw new Exception("The starting population must be zero or more.");
} else if (!((inject == 0) || (inject == 1) || (inject == 2))) {
    throw new Exception("The injection mode must be either 0, 1, or 2.");
} else if (!((guardedMigration == 0) || (guardedMigration == 1) ||
        (guardedMigration == 2))) {
    throw new Exception(
            "The guarded migration mode must be either 0, 1, or 2.");
}

// master rank sends all initialization parameters to other ranks
if (MASS.myPid == 0 && MASS.systemSize > 1) {
    MASS.log("------------------Beginning Agent Init Enhanced sequence for " +
        + "agent handle " + handle + "-----------------");
    Message m = new Message();
    m.createAgentInitEnhancedMessage(inject, guardedMigration, handle,
            className, argument, places.getHandle(), initPopulation);
    for (MNode node : MASS.mNodes) {
        node.sendMessage(m);
    }
    MASS.printResult("Agent Information sent! Awaiting Acknowledgement... ");
}

this.inject = inject;
this.guardedMigration = guardedMigration;
this.handle = handle;
this.places = places;
this.agentArgument = argument;

File curDir = new File(MASS.CUR_DIR);
klas = Class.forName(className, true, Places.loader);
ctor = klas.getConstructor(Object.class);

// ****************************************************************
// inject mode Switch statement
//
// ****************************************************************
switch (inject) {

// ****************************************************************
// AGENTS_INJECT_ORIGINAL
//
// .do the same as the original MASS Agents constructor
//
// ****************************************************************
case Constants.AGENTS_INJECT_ORIGINAL: // initialization
    MASS.printResult("Agents injection:  Original");
```

```
setTotalAgents(initPopulation);

// ----- agents added to MASS.MASS here
if (MASS.addAgents(this)) {
    // make all the agents first
    this.bag = new Vector<Agent>();
    Vector<Agent> tempBag = new Vector<Agent>();
    for (int i = 0; i < initPopulation; i++) {
        createAgent(argument, -1);
    }
    tempBag.addAll(bag);
    bag.clear();

    // get ready to distribute agent objs among place objs
    Places.Iterator placesIter = places.iterator();
    java.util.Iterator<Agent> agentsIter = tempBag.iterator();
    int[] placesSize = places.size();
    Place currentPlace = null;
    Agent currentAgent = null;
    int placeIndex = -1; // incremented at start so will be 0
    int colonistsNum = -1;
    boolean needNewAgent = true;

    // do the distributing, iterate through each MASS.Place
    // every step thru iteration get a new place, but not always a
    // new agent
    while (placesIter.hasNext()
            && (agentsIter.hasNext() || !needNewAgent)) {
        placeIndex++;// first value in loop is 0
        currentPlace = placesIter.next();
        // will stick with the same agent if previous place had 0
        // colonists
        // need to always have an agent to work with first to do
        // agent.map()
        if (needNewAgent) {
            currentAgent = agentsIter.next();
            needNewAgent = false;
        }

        colonistsNum = currentAgent.map(initPopulation, placesSize,
                currentPlace.index);
        // fill place with each agent colonist
        while (colonistsNum > 0
                && (!needNewAgent || agentsIter.hasNext())) {
```

```
                if (needNewAgent) {
                    // happens whenever > 1 colonists
                    currentAgent = agentsIter.next();
                    needNewAgent = false;
                }
                currentAgent.index = currentPlace.index.clone();
                currentAgent.place = currentPlace;

                // agent added to place
                currentPlace.agents.add(currentAgent);

                // an agent is added to the bag
                bag.add(currentAgent);
                needNewAgent = true;
                colonistsNum--;
            }
        }

    } else {
        throw new Exception("That handle is already in use.");
    }
    break;

// ***************************************************************
// AGENTS_INJECT_CENTRAL_POINT
//
// .create and put One Collector agent at center point
// .create and put agents at N, S, E, W of the collector agent
// .the non-collector agents will then diffuse outwards as go on,
//  marching in their original direction and spawning child 90-degrees to left
// .only works with 2-dim for now
//
// ***************************************************************
case Constants.AGENTS_INJECT_CENTRAL_POINT:
    MASS.log("Agents injection:  Center Point Outwards");

    // the following agent creation technique is modeled after
    // the original Agents constructor

    // ---- agents added to MASS.MASS here
    if (MASS.addAgents(this)) {

        // get ready to distribute agent objs among place objs
        Places.Iterator placesIter = places.iterator();
        int[] placesSize = places.size();
        Place currentPlace = null;
```

```
Agent currentAgent = null;
int placeIndex = -1; // incremented at start so will be 0
int colonistsNum = -1;
boolean needNewAgent = true;

// TODO: Generalize, as it is hard coded for 2 dimensions right
// now.
// determine the Places center point on this node
// and set the following variables:
// TODO: extend this down to one dimension, and up to 3-dim
int chunk0size = placesSize[0] / MASS.systemSize;
int remainder0size = placesSize[0] % MASS.systemSize;

int thisNode_MinX = MASS.getPid() * chunk0size;
int thisNode_MaxX = thisNode_MinX + chunk0size - 1;
if (MASS.getPid() == MASS.systemSize - 1) {
    thisNode_MaxX = thisNode_MaxX + remainder0size;
}

int thisNode_MinY = 0;
int thisNode_MaxY = placesSize[1] - 1;

int centerX = thisNode_MinX + ((thisNode_MaxX - thisNode_MinX + 1) / 2);
int centerY = (thisNode_MaxY - thisNode_MinY + 1) / 2;

// ignore user input of initial population of agents
// because for this injection method it can be calculated

initPopulation = 5; // TODO: only good for 2-dim

// make all the agents first
this.bag = new Vector<Agent>();
Vector<Agent> tempBag = new Vector<Agent>();
for (int i = 0; i < initPopulation; i++) {
    createAgent(argument, -1);
}
tempBag.addAll(bag);
bag.clear();

// finish getting ready to distribute agent objs among place objs
java.util.Iterator<Agent> agentsIter = tempBag.iterator();

boolean makeAgentCollector = false;
boolean makeAgentAtLastLocation = false;
int[] makeMoveDirection = new int[3];
```

```
// do the distributing, iterate through each MASS.Place
// on this node.
// every step thru iteration get a new place, but not always a
// new agent
while (placesIter.hasNext()
        && (agentsIter.hasNext() || !needNewAgent)) {
    placeIndex++;// first value in loop is 0
    currentPlace = placesIter.next();
    // will stick with the same agent if previous place had 0
    // colonists
    // need to always have an agent to work with first to do
    // agent.map()
    if (needNewAgent) {
        currentAgent = agentsIter.next();
        needNewAgent = false;
    }

    colonistsNum = 0;
    makeAgentCollector = false;
    makeAgentAtLastLocation = false;
    makeMoveDirection[0] = 0;
    makeMoveDirection[1] = 0;
    makeMoveDirection[2] = 0;

    if (currentPlace.index[1] == centerY) {

        if (currentPlace.index[0] == ( centerX - 1 )) {
            // create West-bound agent
            makeMoveDirection[0] = -1;
            colonistsNum = 1;

        } else if (currentPlace.index[0] == ( centerX + 1 ) ) {
            // create East-bound agent
            makeMoveDirection[0] = +1;
            colonistsNum = 1;

        } else if (currentPlace.index[0] == ( centerX ) ) {
            // create the Collector agent because at the center
            // point of this node
            makeAgentCollector = true;
            colonistsNum = 1;

        }

    } else if (currentPlace.index[0] == centerX) {
        if (currentPlace.index[1] == ( centerY + 1 ) ) {
                // create South-bound agent
                makeMoveDirection[1] = +1;
                colonistsNum = 1;

            } else if (currentPlace.index[1] == ( centerY - 1 ) ) {
                // create North-bound agent
                makeMoveDirection[1] = -1;
                colonistsNum = 1;
            }

        }

        // this handles a 3x3 places or bigger
        boolean condA = (makeMoveDirection[0] == +1 )
                && (currentPlace.index[0] == thisNode_MaxX);
        boolean condB = (makeMoveDirection[0] == -1 )
                && (currentPlace.index[0] == thisNode_MinX);
        boolean condC = (makeMoveDirection[1] == +1 )
                && (currentPlace.index[1] == thisNode_MaxY);
        boolean condD = (makeMoveDirection[1] == -1 )
                && (currentPlace.index[1] == thisNode_MinY);
        if ( condA || condB || condC || condD ) {
            makeAgentAtLastLocation = true;
        }

        // fill place with each agent colonist
        while (colonistsNum > 0
                && (!needNewAgent || agentsIter.hasNext())) {
            if (needNewAgent) {
                // happens whenever > 1 colonists
                currentAgent = agentsIter.next();
                needNewAgent = false;
            }
            currentAgent.index = currentPlace.index.clone();
            currentAgent.place = currentPlace;
            currentAgent.iAmCollector = makeAgentCollector;
            currentAgent.atLastLocation = makeAgentAtLastLocation;
            currentAgent.directionToMove[0] = makeMoveDirection[0];
            currentAgent.directionToMove[1] = makeMoveDirection[1];

            // (future:  handle multi-dim)
            currentAgent.thisNode_MinX = thisNode_MinX;
            currentAgent.thisNode_MaxX = thisNode_MaxX;
            currentAgent.thisNode_MinY = thisNode_MinY;
            currentAgent.thisNode_MaxY = thisNode_MaxY;
```

```
                // agent added to place
                currentPlace.agents.add(currentAgent);

                // an agent is added to the bag
                bag.add(currentAgent);
                needNewAgent = true;
                colonistsNum--;
            }

            // update total number of current agents for MASS
            setTotalAgents(initPopulation * MASS.systemSize);
        }

    } else {
        throw new Exception("That handle is already in use.");
    }


    break;

// **********************************************************************
// AGENTS_INJECT_CENTRAL_AXES
//
// .place one Collector agent at center point, and other agents along
// all "center axes", and then march "NSEW" as go on
//
// **********************************************************************
case Constants.AGENTS_INJECT_CENTRAL_AXES:
    MASS.log("Agents injection:  Controlled (central axes)");

    // the following agent creation technique is modeled after
    // the original Agents constructor
    //
    // (future: instead of iterating over Places, just create & place the
    //  sparse number of agents directly into their place elements)

    // ---- agents added to MASS.MASS here
    if (MASS.addAgents(this)) {

        // get ready to distribute agent objs among place objs
        Places.Iterator placesIter = places.iterator();
        int[] placesSize = places.size();
        Place currentPlace = null;
        Agent currentAgent = null;
        int placeIndex = -1; // incremented at start so will be 0
        int colonistsNum = -1;
```

```
boolean needNewAgent = true;

// determine the Places center point on this node
// and set the following variables:
// (future: extend this down to one dimension, and up to 3-dim)
int chunk0size = placesSize[0] / MASS.systemSize;
int remainder0size = placesSize[0] % MASS.systemSize;

int thisNode_MinX = MASS.getPid() * chunk0size;
int thisNode_MaxX = thisNode_MinX + chunk0size - 1;
if (MASS.getPid() == MASS.systemSize - 1) {
    thisNode_MaxX = thisNode_MaxX + remainder0size;
}

int thisNode_MinY = 0;
int thisNode_MaxY = placesSize[1] - 1;

int centerX = thisNode_MinX + ((thisNode_MaxX - thisNode_MinX + 1) / 2);
int centerY = (thisNode_MaxY - thisNode_MinY + 1) / 2;

// (future: give error message if too small sized Places)
MASS.log("EnhAgents constructor:  for myPid = " + MASS.getPid()
        + ", thisNode_MinX = " + thisNode_MinX + ", thisNode_MaxX = "
        + thisNode_MaxX + ", thisNode_MinY = "
        + thisNode_MinY + ", thisNode_MaxY = "
        + thisNode_MaxY + ", centerX = " + centerX
        + ", centerY = " + centerY);

// ignore user input of initial population of agents
// because for this injection method it can be calculated
initPopulation = (thisNode_MaxY + 1) + (thisNode_MaxX - thisNode_MinX);

// update total number of current agents for MASS
setTotalAgents(initPopulation * MASS.systemSize);

// make all the agents first
this.bag = new Vector<Agent>();
Vector<Agent> tempBag = new Vector<Agent>();
for (int i = 0; i < initPopulation; i++) {
    createAgent(argument, -1);
}
tempBag.addAll(bag);
bag.clear();

// finish getting ready to distribute agent objs among place objs
java.util.Iterator<Agent> agentsIter = tempBag.iterator();
```

```
boolean makeAgentCollector = false;
boolean makeAgentAtLastLocation = false;
int[] makeMoveDirection = new int[3];

// do the distributing, iterate through each MASS.Place
// on this node.
// every step thru iteration get a new place, but not always a
// new agent
//
// ( future: make this more efficient: don't need to iterate
//          over every place since know where agents are to be placed)
while (placesIter.hasNext()
        && (agentsIter.hasNext() || !needNewAgent)) {
    placeIndex++;// first value in loop is 0
    currentPlace = placesIter.next();
    // will stick with the same agent if previous place had 0
    // colonists
    // need to always have an agent to work with first to do
    // agent.map()
    if (needNewAgent) {
        currentAgent = agentsIter.next();
        needNewAgent = false;
    }

    colonistsNum = 0;
    makeAgentCollector = false;
    makeAgentAtLastLocation = false;
    makeMoveDirection[0] = 0;
    makeMoveDirection[1] = 0;
    makeMoveDirection[2] = 0;

    if (currentPlace.index[0] == centerX) {
        colonistsNum = 1;
        if (currentPlace.index[1] < centerY) {
            // create West-bound agent
            makeMoveDirection[0] = -1;
        } else if (currentPlace.index[1] > centerY) {
            // create East-bound agent
            makeMoveDirection[0] = +1;
        } else {
            // create the Collector agent because at the center
            // point of this node
            makeAgentCollector = true;
        }
        } else if (currentPlace.index[1] == centerY) {
            colonistsNum = 1;
            if (currentPlace.index[0] < centerX) {
                // create South-bound agent
                makeMoveDirection[1] = +1;
            } else {
                // create North-bound agent
                makeMoveDirection[1] = -1;
            }

        }

        // This should handle a very small places.
        // Only set atLastLocation if the agent is set to move in that direction!
        boolean condA = (makeMoveDirection[0] == +1 )
              && (currentPlace.index[0] == thisNode_MaxX);
        boolean condB = (makeMoveDirection[0] == -1 )
              && (currentPlace.index[0] == thisNode_MinX);
        boolean condC = (makeMoveDirection[1] == +1 )
              && (currentPlace.index[1] == thisNode_MaxY);
        boolean condD = (makeMoveDirection[1] == -1 )
              && (currentPlace.index[1] == thisNode_MinY);
        if ( condA || condB || condC || condD ) {
            makeAgentAtLastLocation = true;
        }

        // fill place with each agent colonist
        while (colonistsNum > 0
                && (!needNewAgent || agentsIter.hasNext())) {
            if (needNewAgent) {
                // happens whenever > 1 colonists
                currentAgent = agentsIter.next();
                needNewAgent = false;
            }
            currentAgent.index = currentPlace.index.clone();
            currentAgent.place = currentPlace;
            currentAgent.iAmCollector = makeAgentCollector;
            currentAgent.atLastLocation = makeAgentAtLastLocation;
            currentAgent.directionToMove[0] = makeMoveDirection[0];
            currentAgent.directionToMove[1] = makeMoveDirection[1];

            // (future:  handle multi-dim)
            currentAgent.thisNode_MinX = thisNode_MinX;
            currentAgent.thisNode_MaxX = thisNode_MaxX;
            currentAgent.thisNode_MinY = thisNode_MinY;
            currentAgent.thisNode_MaxY = thisNode_MaxY;
```

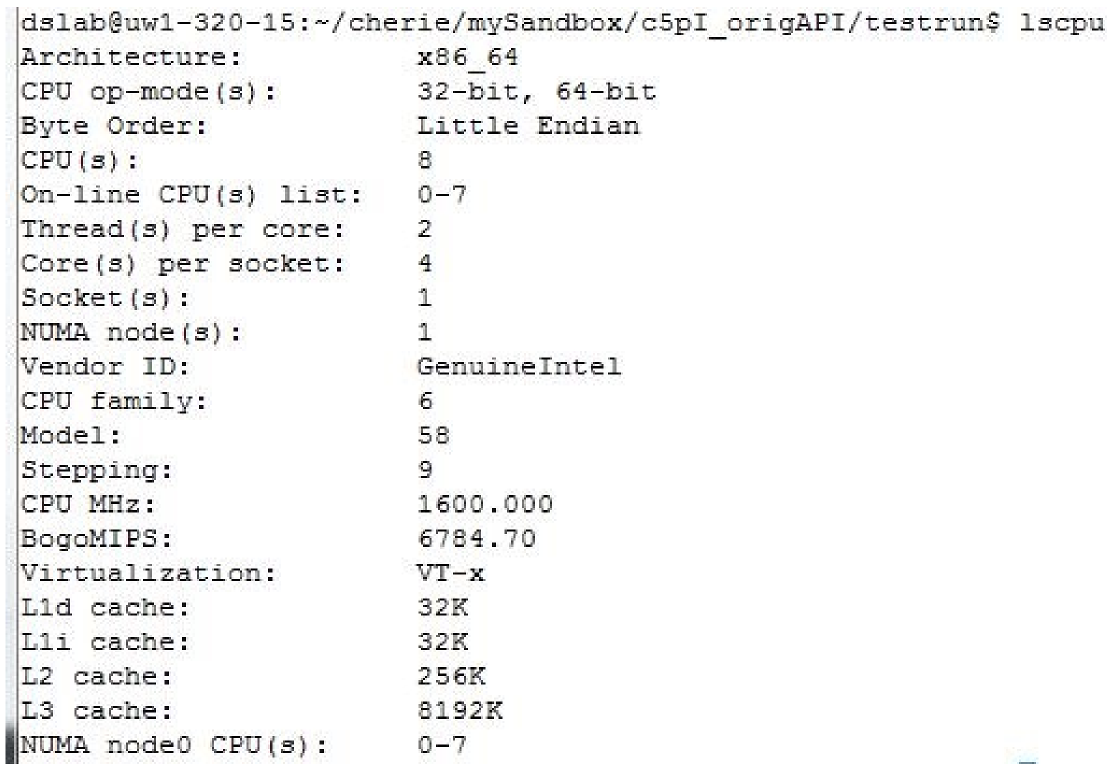
```
                    // agent added to place
                    currentPlace.agents.add(currentAgent);
                    // an agent is added to the bag
                    bag.add(currentAgent);
                    needNewAgent = true;
                    colonistsNum--;
                }
            }

        } else {
            throw new Exception("That handle is already in use.");
        }
        break;

    default:
        MASS.log("Agents injection:  Received unknown agents injection command ");
        break;
    } // end of switch

    /*
     *  if this is the master node, wait until all other remote nodes complete
     */
    if (MASS.myPid == 0 && MASS.systemSize > 1) {
        for (MNode node : MASS.mNodes) {
            node.receiveMessage();
        }
        System.err.println("Received all Acknowledgement... ");
    }
}
```

B.2 Enhanced Agents Migrate code inside MASS.Agent.java

```
/**
 * Enhanced agent migration Initiates an agent migration upon a next call to
 * MASS.Agents.manageAll( ). More specifically, migrate( ) updates the
 * calling agent's index[] by adding the enhanced agent's "directionToMove"
 * to it's current coordinates.
 *
 * @param migrationMethod
 *   add directionToMove to current place and then check for boundary, etc. Also,
 *   update flag to user: atLastLocation (if now migrating to the edge of this node)
 *
 * @return true if a migration was scheduled in success, false if error
```

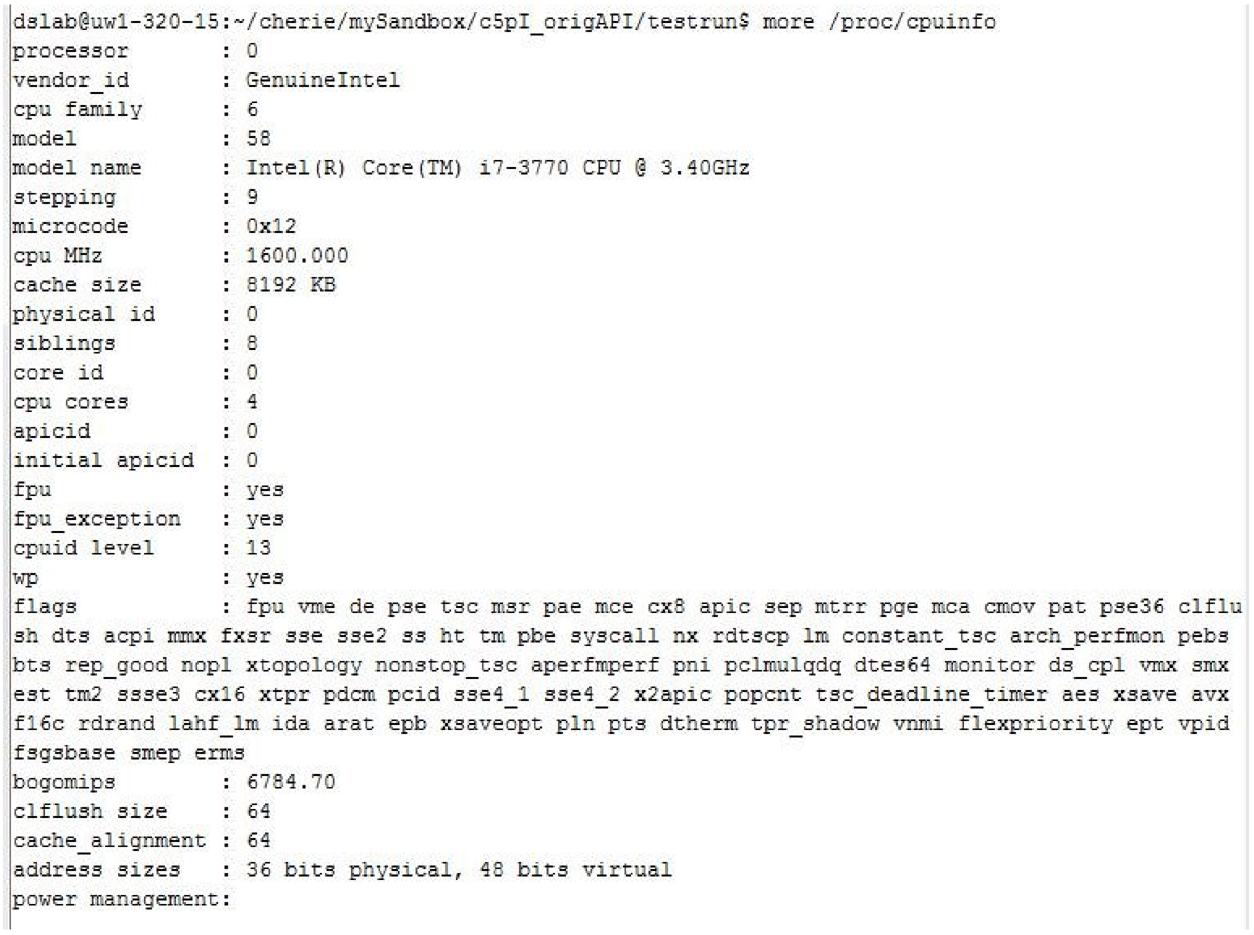
```
*/
public boolean migrate(byte migrationMethod) {
    boolean retVal = false;
    if (this.iAmCollector) { // don't migrate the Collector agent !!
        return true;
    }
    switch ( migrationMethod ) {

    case (byte) Constants.AGENTS_INJECT_CENTRAL_AXES:
    case (byte) Constants.AGENTS_INJECT_CENTRAL_POINT:

        if (atLastLocation) {
            MASS.log(("~~~ err: Collector trying to migrate from [ " + this.index[0]
                    + ", " + this.index[1] + " ], when already atLastLocation"));

            retVal = false;
        } else {

            // future:  make more general to more dimensions
            int newX = this.index[0] + this.directionToMove[0];
            int newY = this.index[1] + this.directionToMove[1];
            // check for being at boundary now & if so, then set atLastLocation
            boolean cond1 = (this.directionToMove[0] != 0 )
                   && ( (newX == this.thisNode_MinX)
                   || (newX == this.thisNode_MaxX) );
            boolean cond2 = (this.directionToMove[1] != 0 )
                   && ( (newY == this.thisNode_MinY)
                   || (newY == this.thisNode_MaxY) );
            if ( cond1 || cond2 ) {
                this.atLastLocation = true;
            }
            // assign the new index
            this.index[0] = newX;
            this.index[1] = newY;
            retVal = true;
        }
        break;

    default:
        MASS.log("Agent.migrate:  Received unknown agents injection method ");
        break;
    } // end of switch
    return retVal;
}
```

Appendix C: Machines Specification Details



```
dslab@uw1-320-15:~/cherie/mySandbox/c5pI_origAPI/testrun$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                8
On-line CPU(s) list:   0-7
Thread(s) per core:    2
Core(s) per socket:    4
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 58
Stepping:              9
CPU MHz:               1600.000
BogoMIPS:              6784.70
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              8192K
NUMA node0 CPU(s):     0-7
```

Figure 15: lscpu command results showing machine information



```
dslab@uw1-320-15:~/cherie/mySandbox/c5pI_origAPI/testrun$ more /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 58
model name      : Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz
stepping        : 9
microcode       : 0x12
cpu MHz         : 1600.000
cache size      : 8192 KB
physical id     : 0
siblings        : 8
core id         : 0
cpu cores       : 4
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 13
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflu
sh dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx rdtscp lm constant_tsc arch_perfmon pebs
bts rep_good nopl xtopology nonstop_tsc aperfmperf pni pclmulqdq dtes64 monitor ds_cpl vmx smx
est tm2 ssse3 cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic popcnt tsc_deadline_timer aes xsave avx
f16c rdrand lahf_lm ida arat epb xsaveopt pln pts dtherm tpr_shadow vnmi flexpriority ept vpid
fsgsbase smep erms
bogomips        : 6784.70
clflush size    : 64
cache_alignment : 64
address sizes   : 36 bits physical, 48 bits virtual
power management:
```

Figure 16: /proc/cpuinfo file showing machine information