©Copyright 2012
Timothy Chuang

Design and Qualitative/Quantitative Analysis of Multi-Agent Spatial Simulation Library

Timothy Chuang

A thesis

submitted in partial fulfillment of the

requirements for the degree of

Master of Science

University of Washington

2012

Committee:

Munehiro Fukuda, Chair

Charles Jackels

Hazeline Asuncion

Kelvin Sung

Program Authorized to Offer Degree:

Computer Science and Software Engineering

University of Washington

Abstract

Design and Qualitative/Quantitative Analysis of Multi-Agent Spatial Simulation Library

Timothy Chuang

Chair of the Supervisory Committee:
Munehiro Fukuda, Ph.D.
Computing & Software Systems

Integrating sensor networks in cloud computing gives new opportunities of using as many cloud-computing nodes as necessary to analyze real-time sensor data on the fly. However, most cloud services for parallelization such as OpenMP, MPI, and MapReduce are not always suitable for on-the-fly sensor-data analyses that are implemented as model-based, entity-based, and multi-agent simulations. To address this semantic gap between analyzing algorithms and their actual implementations, we have designed and implemented MASS: a library for multi-agent spatial simulation that composes of a user application of distributed array elements and multi-agents, each representing an individual simulation place or an active entity. All computation is enclosed in each of elements and agents that are automatically distributed over different computing nodes. Their communication is then scheduled as periodic data exchanges among those entities using their logical indices. This thesis presents the design, implementation and evaluation of the MASS library.

Acknowledgement

First and foremost, I would like to express my sincerest gratitude for my supervisor, Dr. Munehiro Fukuda, who has motivated me to study parallel and distributed computing, and supported me throughout my research with his knowledge and patience. This work would not have been possible otherwise.

I would also like to extend my gratitude to my supervisory committee members Dr. Hazeline Asuncion, Dr. Charles Jackels, and Dr. Kelvin Sung, who have provided valuable inputs. Dr. Hazeline Asuncion has taught me the fundamental tools to understand how a good initial design can dramatically simplify implementation, and allow me to identify potential problems before they occur. Dr. Kelvin Sung has shaped me into a better student with his rigorous course schedule and infinite incentive to out-perform myself on every single assignment, which has given me the drive to never settle.

The department has provided me with equipment and support I needed to complete my thesis. Especially Megan Jewell, the graduate advisor, who had constantly provided me with reminders and guidance when I was completely overwhelmed by the complex process of writing a master's thesis. I would also like to thank the Linux lab administrator Josh Larios for keeping the lab machines up and running all the time and for putting up with my constantly crashing the lab machines due to the intensive tests I had to perform.

Finally, I thank my parents for supporting me throughout all my studies at the University.

TABLE OF CONTENTS

LIST OF FIGURES iii

LIST OF TABLES iv

Preface v

Chapter 1: Introduction 1

1.1 Background 1

1.2 Research Objective 2

Chapter 2: Methods 3

2.1 Challenge and Solution in Multi-Agent Individual-Based Models 3

2.2 Execution Model 4

2.3 Language Specification 6

2.4 Coding Examples 9

2.5 System Design 12

Chapter 3: Evaluation 22

3.1 Programmability Analysis 22

3.2 Execution Performance Analysis 24

3.3 Summary 33

Chapter 4: Related Work 34

4.1 Distributed Array 34

4.2 Multi-Agents 34

Chapter 5: Conclusion 36

5.1 Result Statement 36

5.2 Problems Encountered 36

5.3 Future Work 37

References 38

Appendix A: Source Code and User Manual 39

Appendix B: Detailed Language Specification 40

Appendix C: Wave2D Source Code 45

Appendix D: RandomWalk Source Code 50

LIST OF FIGURES

Figure Number | Page

1. MASS Execution Model ... 6
2. Wave2D Call Methods ... 9
3. Wave2D Code Snippet ... 10
4. RandomWalk Code and Simulation Space ... 11
5. Internal Classes over a Cluster ... 12
6. MASS Data Structure ... 13
7. Connection Establishment between Computing Nodes ... 15
8. ExchangeAll Local and Remote Process ... 16
9. Multi-Threaded Places Granularity ... 25
10. Multi-threaded and Multi-Process Places Granularity ... 26
11. Performance of CallAll ... 27
12. Performance of CallAll with Return Values ... 28
13. Performance of ExchangeAll with Return Values ... 29
14. Agents Granularity ... 30
15. Total Execution Time of Wave 2D ... 31
16. Total ExchangeAll Execution Time ... 31
17. Execution Performance of RandomWalk ... 33

LIST OF TABLES

Table Number Page

1. MASS Interface 7

2. MASS Thread Types 16

Preface

This thesis contains work done from September 2011 to June 2012. It has been produced in close collaboration with my supervisor, Dr. Munehiro Fukuda.

When professor Fukuda introduced me to the world of parallel and distributed computing during my undergraduate study several years ago, I was intrigued by the idea that utilizing inexpensive hardware can achieve high throughput performance similar to some costly high performance servers. However, all of it comes at the expense of knowing how such parallelization works, which can be burdensome and time consuming.

I immediately jumped on board after learning about the concept of a new library from professor Fukuda that would alleviate the parallelization challenge while allowing the application developers to focus on the design aspect of the application and easily tap into the resources of a computing cluster without having to deal with complex parallelization techniques.

The challenge of designing and implementing such a library has allowed me to improve my understanding of parallel and distributed systems, and I am thankful for the support and the resources the department has provided me to make all of this possible.

Chapter 1: Introduction

MASS (Multi-Agent Spatial Simulation) is a new parallelization library for multi-agent and spatial simulation that facilitates individual cell centered programming of multi-agents and simulation spaces. MASS is composed of a user application of distributed array elements and multi-agents that represent each individual simulation cell or an entity. It contains a set of methods that allows users to easily manipulate the parallel behavior of each individual cell in their simulation space, and is designed to accommodate a wide array of needs for general scientific computing applications such as molecular dynamics, artificial society, and home automation.

1.1 Background

The emergent dissemination of wireless sensor networks and the recent popularity of could computing have brought new opportunities of sensor-cloud integration [4] that will facilitate on-the-fly analysis, simulation, and prediction of physical and environmental conditions by feeding real-time sensor data to cloud jobs. For instance in agriculture, frost protection need to predict the overnight transition of orchard air temperature, which can be done by sending temperature data to prediction polynomials [10] and artificial neural network [9] running in the cloud. Another example is accurate car navigation that finds the best route to drive through a busy metropolitan area by feeding the current traffic data to traffic simulation models such as MatSim [6].

For on-the-fly analysis, these simulation models need to be accelerated with cloud/grid-provided common parallelization tools such as OpenMP, MPI, a hybrid of these two libraries, and MapReduce, all generally suited well to simple task parallelism. However, of concern is a big semantic gap between sensor-data analyzing models and these software tools. Most model-based simulations [1] apply formula-based, spatial, and/or multi-agent models to sensor data, where model designers would prefer to code their algorithms as focusing on each individual simulation entity [3]. Therefore, the designers feel difficulty in mapping their algorithms to the underlying parallelization tools.

Our research goal is to reduce this semantic gap by providing users with MASS, a new parallelization library for multi-agent and spatial simulation that facilitates: (1) individual-centered programming of multi-agents and simulation spaces, each automatically parallelized over the underlying platforms, and (2) utilization of a SMP cluster, (i.e. composed of a collection of communicating multithreaded processes). MASS composes a user application of distributed array elements and multi-agents, each representing an

individual simulation place or an active entity. All computation is enclosed in each element or agent, and all communication is scheduled as periodical data exchanges among those entities using their relative indices. In temperature prediction, an orchard is meshed into two-dimensional array elements over which agents move as air flow. This model unleashes an application from accessing entities in for-loops, and thus eases dynamic allocation of entities to multiple computing nodes and multiple CPU cores.

1.2 Research Objective

MASS library is a part of the Sensor Grid research project that is currently underway at the UWB distributed systems lab. My research goal is to design and implement MASS, gather performance and programmability results and present the findings.

The very initial version of the multi-threaded MASS was already completed by another research student John Spiger [13]. As part of this research, a network platform layer has been designed, implemented and integrated into the multi-threaded MASS library to make use of multiple computing nodes with multi-core CPUs. The evaluation of the library includes:

- Demonstrate the programmability advantage of the MASS library with a simple multi-agent application.
- Demonstrate the competitive performance to be gained from parallel execution of the MASS library when computing nodes are added to the cluster with several performance benchmarks.

The following chapter describes the underlying challenge and solution to the limitations in section 2.1, execution model in section 2.2. Section 2.3 and section 2.4 contain MASS coding examples and language specification. Section 2.5 describes system design in detail, including data structure, algorithm and implementation. Chapter 3 contains the evaluation of the MASS library. Section 3.1 presents the qualitative analysis and section 3.2 presents the execution performance analysis. Chapter 4 details comparisons between similar software tools, and chapter 5 is the conclusion with result statements, encountered problems during my research and future work.

Chapter 2: Methods

This chapter describes MASS library's components, model, language specification, design and implementation.

2.1 Challenge and Solution in Multi-Agent Individual-Based Models

We identify the following three design challenges:

(1) **Naming** is the typical problem in distributed computing when allocating agents and individual objects over multiple computing nodes. In general, objects are maintained in a list or an array, and are scanned in a loop for their method invocation. The key in naming is to hide object-to-processor mapping, to automate loop partitioning, and to make inter-object communication independent from the actual distributed environment.

(2) **Fine granularity** of each object computation is a quite familiar problem in multi-agent individual-based models. These agents and individuals may be spawned and terminated dynamically and frequently. Therefore, we will minimize object context switches by invoking their methods in a batch, reduce their communication overheads by sending their messages in a larger packet, and utilize cache and main memory effectively by aggregating objects whose proximity is logically close.

(3) **Dynamic load balancing** plays an important role, because agents and individuals are not all active and not all uniformly distributed over a simulation space. Therefore, we will use several techniques for reallocating objects to CPUs, moving a simulation job to a faster node, and adding more computing nodes to the job.

To tackle the research challenges enumerated above, we will use MASS: a parallelizing library for multi-agent spatial simulation. The library instantiates multi-agents on a global array, all parallelized over a cluster of multithreaded computing nodes with multi-core CPUs. The library hides the underlying agent/element-to-processor mapping as well as inter-processor communication/synchronization from user programs, so that the model designers are given one consistent programming paradigm regardless of the underlying parallel architecture.

We implement the MASS library based on the following three design principles:

(1) Naming is resolved by two strategies: (1) having each array element communicate with neighbors, using its relative indices, (i.e., a distance from it), and (2) allowing agents to communicate with only those residing on the same array element unless they migrate to a different element.

(2) Fine granularity is addressed by four library features: (1) allocating agents and array elements in one flat contiguous memory space, (2) invoking them at once through the library calls such as *callAll( )*, (3) processing their parallel communication at once through *exchangeAll( )*, and (4) moving agents to the actual CPU that has their next array element to reside on (to keep agent-place proximity).

(3) Dynamic load balancing is supported at the library level by two techniques: (1) repartitioning arrays of agents and elements at run time to balance each partition's computation amount, and (2) increasing the number of partitioned blocks to acquire more CPU cores.

2.2 Execution Model

"Places" and "agents" are keys to the MASS library. "Places" is a matrix of elements that are dynamically allocated over a cluster of computing nodes. Each element is called a place, is pointed to by a set of network-independent matrix indices, and is capable of exchanging information with any other places. On the other hand, "agents" is a set of execution instances that can reside on a place, migrate to any other places with matrix indices (thus as duplicating themselves), and interact with other agents as well as multiple places.

An example of places and agents in a battle game could be territories and military units respectively. Some applications may need only either places or agents. For instance, Schrödinger's wave simulation needs only two-dimensional places, each diffusing its wave influence to the neighbors. Molecular dynamics needs only agents, each behaving as a particle since it must collect distance information from all the other particles for computing its next position, velocity, and acceleration.

Parallelization with the MASS library assumes a cluster of multi-core computing nodes as the underlying computing architecture, and thus uses a set of multi-threaded communicating processes that are forked

over the cluster and managed under the control of typical message-passing software infrastructure such as sockets and MPI. The library spawns the same number of threads as that of CPU cores per node or per process. Those threads take charge of method call and information exchange among places and agents in parallel.

Places are mapped to threads, whereas agents are mapped to processes. Unless a programmer indicates his/her places-partitioning algorithm, the MASS library divides places into smaller stripes in vertical or in the X-coordinate direction, each of which is then allocated to and executed by a different thread. Contrary to places, agents are grouped into bags, each allocated to a different process where multiple threads keep checking in and out one after another agent from this bag when they are ready to execute a new agent. If agents are associated with a particular place, they are allocated to the same process whose thread takes care of this place.

Agents are a set of execution instances that can reside on a place, migrate to any other places with array indices (thus as duplicating themselves), and interact with other agents and places. As shown in Figure 1, parallelization with the MASS library uses a set of multi-threaded communicating processes that are forked over a cluster and are connected to each other through SSH-tunneled TCP links. The library spawns the number of threads as specified by the application developer. Those threads take charge of method call and information exchange among places and agents in parallel. Places are mapped to threads, whereas agents are mapped to processes. Unless a programmer indicates a place-allocation algorithm, the MASS library partitions places into smaller stripes in vertical, each of which is statically allocated to and executed by a different thread (static scheduling). Contrary to places, agents are allocated to a different process, based on their proximity to the places that this process maintains, and are dynamically executed by multiple threads belonging to the process (dynamic scheduling).

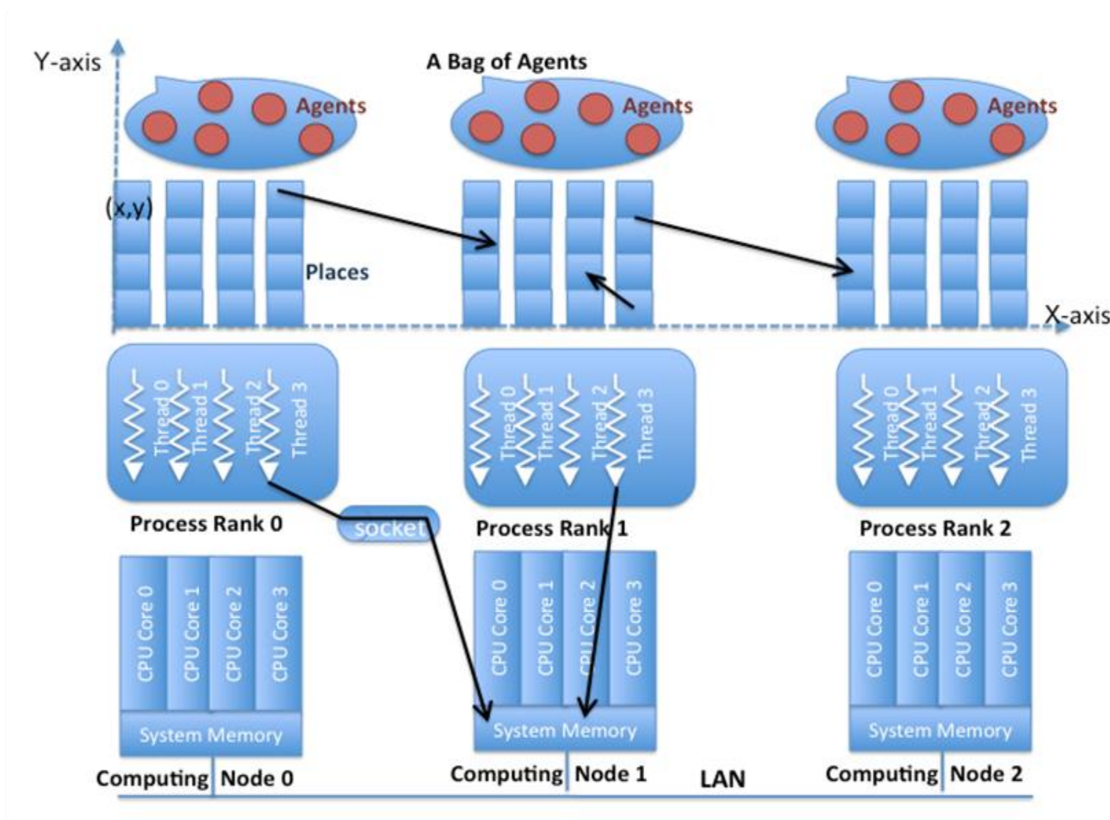


Figure 1. MASS Execution Model

2.3 Language Specification

All processes involved in the same MASS library computation must call MASS.init( ) and MASS.finalize( ) at the beginning and end of their code respectively so as to get started and finished together. Upon a MASS.init( ) call, each process, running on a different computing node, spawns the same number of threads as that of its local CPU cores, so that all threads can access places and agents. Upon a MASS.finalize( ) call, each process cleans up all its threads as being detached from the places and agents objects. A snippet of the MASS programming interface is shown in table 1.

Table 1. MASS Interface

`public static void`	`init( String[] args, int nProc, int nThr )` Involves nProc processes in the same computation and has each process spawn nThr threads. Args include user ID, password, machine file name which includes a list of remote computing nodes and optional parameters such as the designated port to use and arguments to supply to the user application.
`public static void`	`init( String[] args )` Involves as many processes as requested in the same computation and has each process spawn as many threads as the number of CPU cores. Args are the same as the above method.
`public static void`	`finalize( )` Finishes computation.
`public static Places`	`getPlaces( int handle )` Retrieves a "Places" object that has been created by a user-specified handle and mapped over multiple machines.
`public static Agents`	`getAgents( int handle )` Retrieves an "Agents" object that has been created by a user-specified handled and mapped over multiple machines.

2.3.1 Places

"Places" is a distributed matrix whose elements are allocated to different computing nodes. Each element, (termed a "place") is addressed by a set of network-independent matrix indices. Once the main method has called MASS.init( ), it can create as many places as needed, using the following constructor. Unless a user supplies an explicit mapping method in his/her "Place" definition, a "Places" instance (simplified as "places" in the following discussion) is partitioned into smaller stripes in terms of coordinates[0], and is mapped over a given set of computing nodes, (i.e., processes).

Places Class

- *public Places(int handle, [String primitive,] String className, Object argument, int size)* instantiates a shared array with *size* from *className* or a *primitive* data type as passing an *argument* to the *className* constructor. This array receives a user-given *handle*.
- *public Object[] callAll(String functionName, Object[] arguments)* calls the method specified with *function-Name* of all array elements as passing *arguments[i]* to element[i], and receives a return value from it into *Object[i]*. Calls are performed in parallel among multi-processes/threads. In case of a multi-dimensional array, *i* is considered as the index when the array is flattened to a single dimension.

- *public Object[] callSome(String functionName, Object[] argument, int... index)* calls a given method of one or more selected array elements. If *index[i]* is a non-negative number, it indexes a particular element, a row, or a column. If *index[i]* is a negative number, say $-x$, it indexes every x element. Calls are performed in parallel.
- *public void exchangeAll(int handle, String functionName, Vector<int[]> destinations)* calls from each of all elements to a given method of all destination elements, each indexed with a different *Vector* element. Each vector element, say *destination[]* is an array of integers where *destination[i]* includes a relative index (or a distance) on the coordinate *i* from the current caller to the callee element. The caller passes its *outMessage[]* data member to the callee as a set of arguments, and receives return values in its *inMessage[]*.
- *public void exchangeSome(int handle, String functionName, Vector<int[]> destinations, int... index)* calls each of the elements indexed with *index[]*. The rest of the specification is the same as *exchangeAll( )*.

2.3.2 Agents

"Agents" is a set of execution instances, each capable of residing on a place, migrating to any other place(s) with matrix indices, and interacting with other agents as well as multiple places.

Agents Class

- *public Agents(int handle, String className, Object argument, Places places, int population)* instantiates a set of agents from *className*, passes the *argument* 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.
- *public void manageAll( )* updates each agent's status, based on its latest calls of *migrate( )*, *spawn( )*, *kill( )*, *sleep( )*, *wakeup( )*, and *wakeupAll( )*. These methods are defined in the *Agent* base class and may be invoked from other functions through *callAll( )* and *exchangeAll( )*.

The rest of Agent's methods such as *callAll( )* and *exchangeAll( )* are similar to *Places.callAll( )* and *Places.exchangeAll( )* respectively.

2.3.3 CallMethod

Since method names are user-given, it is quite natural to invoke each array element's method through Java reflection, which is however intolerably slow for parallel computing. Thus, a selection of methods to call should be preferably done with switch( ), where we need to identify each method as an integer value. callMethod( ) is a user-provided framework that assists the MASS library in choosing a method to call. Figure 2 illustrates the general code pattern of such method calls.

```
 1.    public class Wave2D extends Place {
 2.        // constants: each array element's methods are identified by an integer
 3.        // rather than its name.
 4.        public static final int init_ = 0;
 5.        public static final int computeNewWave_ = 1;
 6.        public static final int exchangeWave_ = 2;
 7.        public static final int collectWave_ = 3;
 8.        public static final int startGraphics_ = 4;
 9.        public static final int writeToGraphics_ = 5;
10.        public static final int finishGraphics_ = 6;
11.
12.        // automatically called from callAll, callSome, callStatic, exchangeAll, or
13.        // exchangeSome.
14.        // args may be null depending on a calling method.
15.        public static Object callMethod( int funcId, Object args ) {
16.            switch( funcId ) {
17.            case init: return init( args);
18.            case computeNewWave_: return computeNewWave( args );
19.            case exchangeWave_: return exchangeWave( args );
20.            case storeWave_: return exchangeWave( args );
21.            case startGraphics_: return startGraphics( args );
22.            case writeToGraphics_: return writeToGraphics( args );
23.            case finishGraphics_: return finishGraphics( args );
24.            }
25.            return null;
26.        }
27.
28.        public Object init( Object args ) {
29.           ...;
30.        }
31.        public Object computeNewWave( Object args ) {
32.            ...;
33.        }
34.    }
```

Figure 2. Wave2D Call Methods

2.4 Coding Examples

To give more concrete ideas of the MASS library, this section contains two example MASS applications: *Wave2D* (a two-dimensional wave simulation) and *RandomWalk* (an agent-movement simulation over a two-dimensional space).

2.4.1 Wave2D

Figure 3 shows an example of how *Places.CallAll( )* and *Places.ExchangeAll( )* can be easily utilized to develop a parallel spatial simulation application named *Wave2D*. It is a two-dimensional matrix that simulates Schrödinger's wave diffusion.

In this example, a two-dimensional matrix is instantiated on line 14 by simply creating a new instance of *Places*. The application enters a cyclic simulation to calculate wave heights in every cell in parallel (line 28), and every cell exchanges wave height information (line 32) with all of its neighboring cells (defined on line 20 – 23).

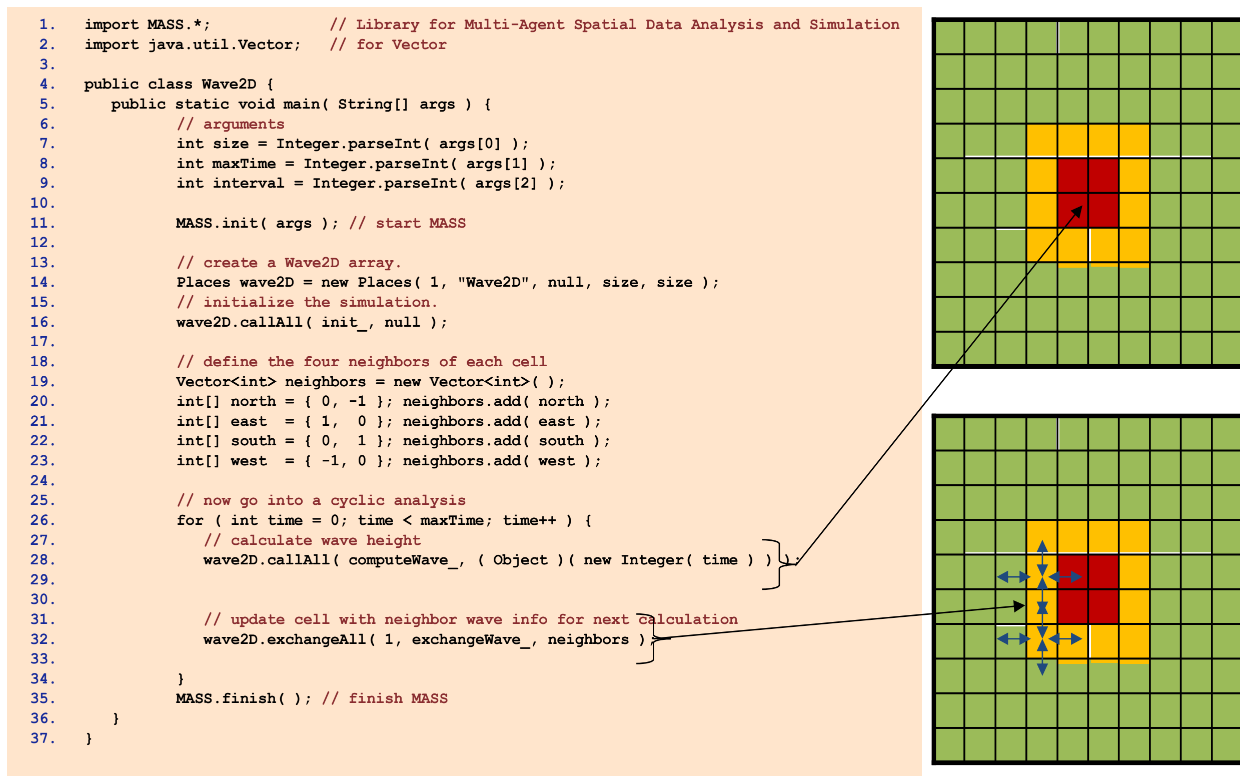


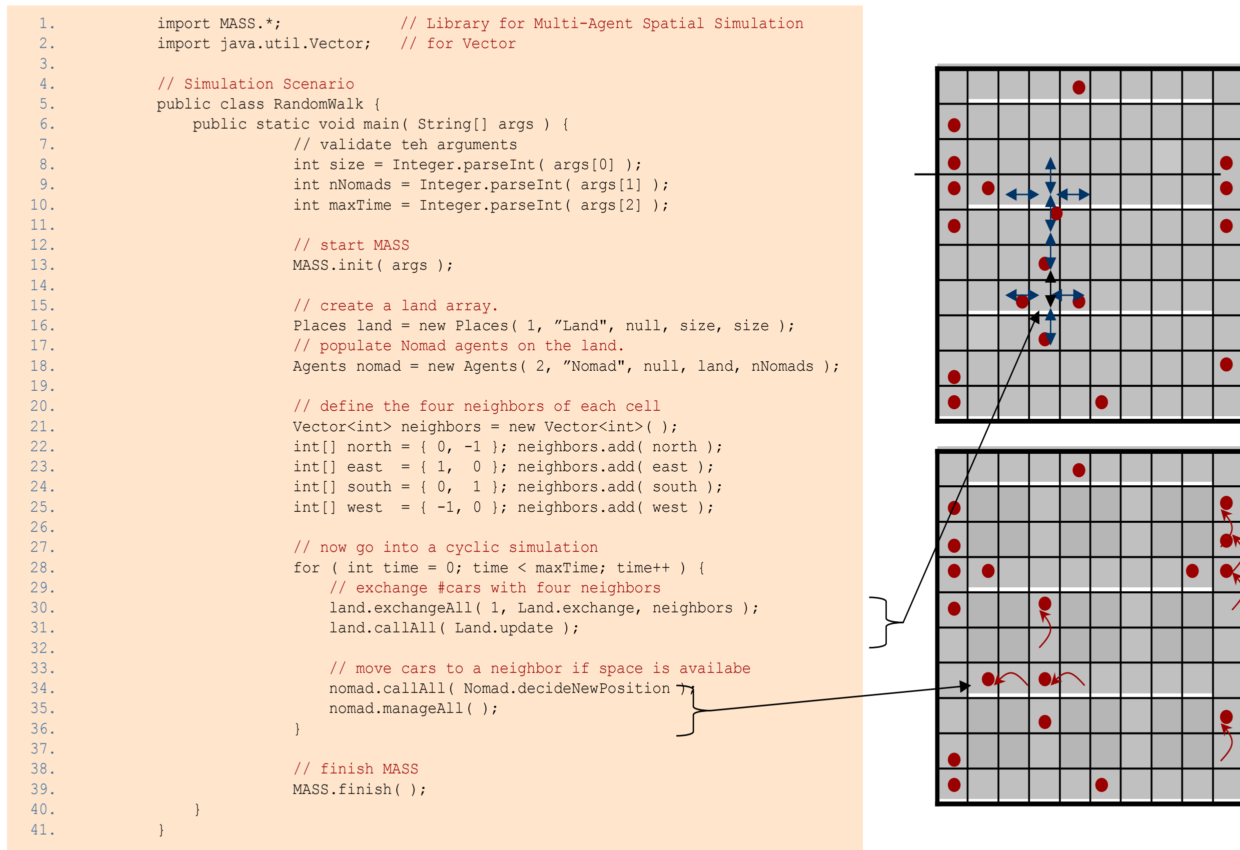
Figure 3. Wave2D Code and Simulation Space

2.4.2 RandomWalk

Figure 4 shows an example of how *Places.callAll( )*, *Places.exchangeAll( )*, *Agents.callAll( )* and *Agents.manageAll( )* can be easily utilized to develop a parallel multi-agent application.

In this example, an args[0] x args[0] (defined on line 8) array over multiple processes that simulates “Land” is created (line 16). “Nomad” agents are created and distributed every four places of this matrix (lines 18), and simulates random walking of these agents over the matrix using multiple processes and threads.

In each simulation iteration, every individual cell exchanges population information with its neighboring cells, which are defined on line 21 to 25, and updates such information with calls on line 30 to 31 as demonstrated in the simulation mockup on the upper right corner of figure 4. The population information is then utilized by “Nomad” agents to determine an un-occupied cell to migrate to (line 34) and perform the actual agent migration in parallel with *nomad.ManageAll( )* (line 35) as demonstrated on the lower right corner of figure 4.



```
1.          import MASS.*;            // Library for Multi-Agent Spatial Simulation
2.          import java.util.Vector;   // for Vector
3.
4.          // Simulation Scenario
5.          public class RandomWalk {
6.              public static void main( String[] args ) {
7.                      // validate teh arguments
8.                      int size = Integer.parseInt( args[0] );
9.                      int nNomads = Integer.parseInt( args[1] );
10.                     int maxTime = Integer.parseInt( args[2] );
11.
12.                     // start MASS
13.                     MASS.init( args );
14.
15.                     // create a land array.
16.                     Places land = new Places( 1, "Land", null, size, size );
17.                     // populate Nomad agents on the land.
18.                     Agents nomad = new Agents( 2, "Nomad", null, land, nNomads );
19.
20.                     // define the four neighbors of each cell
21.                     Vector<int> neighbors = new Vector<int>( );
22.                     int[] north = { 0, -1 }; neighbors.add( north );
23.                     int[] east  = { 1,  0 }; neighbors.add( east );
24.                     int[] south = { 0,  1 }; neighbors.add( south );
25.                     int[] west  = { -1, 0 }; neighbors.add( west );
26.
27.                     // now go into a cyclic simulation
28.                     for ( int time = 0; time < maxTime; time++ ) {
29.                         // exchange #cars with four neighbors
30.                         land.exchangeAll( 1, Land.exchange, neighbors );
31.                         land.callAll( Land.update );
32.
33.                         // move cars to a neighbor if space is availabe
34.                         nomad.callAll( Nomad.decideNewPosition );
35.                         nomad.manageAll( );
36.                     }
37.
38.                     // finish MASS
39.                     MASS.finish( );
40.             }
41.         }
```

Figure 4. RandomWalk Code and Simulation Space

2.5 System Design

This section contains discussions of the design decisions of the MASS library as well as data structure, algorithm and implementation details.

Figure 5 describes the relationship between objects and computing nodes in a cluster by example of inter-process communication.

To facilitate inter-process communication, the master node launches a remote process named *MProcess* on each computing node and establishes a communication channel using Java Secure Channel (JSCH). JSCH is the main communication channel that allows the master node to send various commands to remote nodes. A collection of *Mnode* objects each containing an established JSCH communication channel to a remote host is stored on the master node for direct communication between the master node and remote nodes. An *ExchangeHelper* object is instantiated during the initialization phase on all nodes that is used to establish direct socket connection between two hosts only when such communication is required (i.e. during *exchangeAll( )*).

The remote process *MProcess* sits in a blocking read awaiting commands from the master node. Each remote process and the master node create a pool of worker threads that are ready to perform parallel processing.

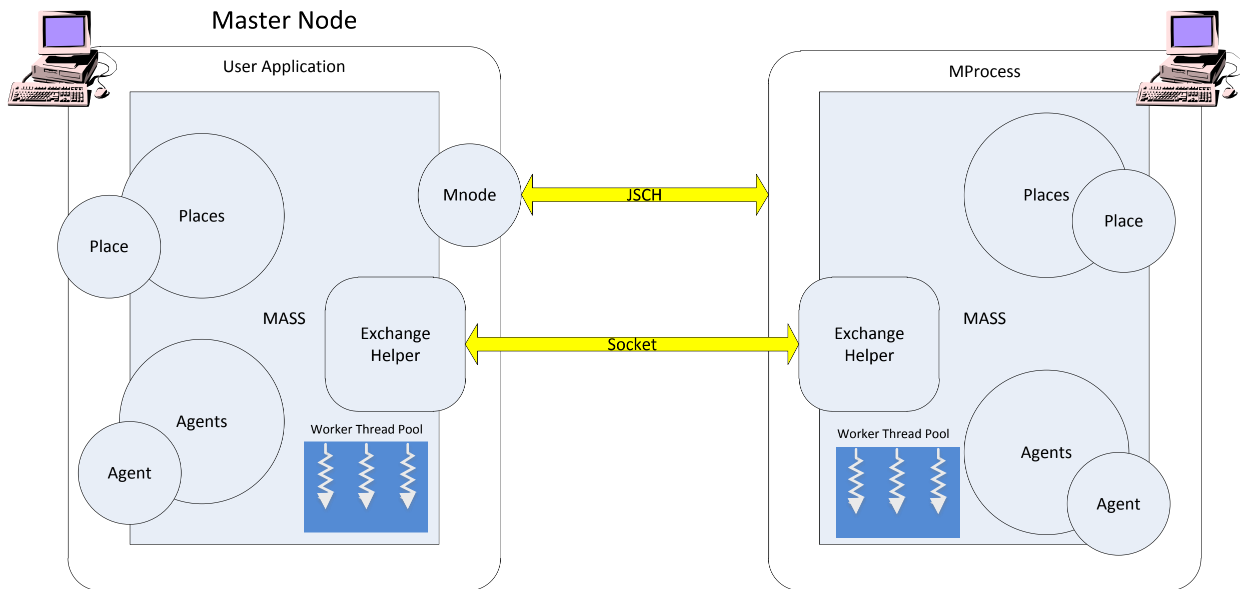


Figure 5. Internal Classes over a Cluster

Log files are created to monitor progress of each remote computing node and to record errors should the application encounter unexpected behavior. The master node simply outputs error messages to the console.

2.5.1 Data Structure

Figure 6 describes the data distribution in the MASS library. The underlying data structure for Places is a distributed array. The array can have n-dimension defined by the application developer. MASS library translates multi-dimension indices into a linear index system for easy management. The application developers can design applications without any knowledge of such index translation taking place.

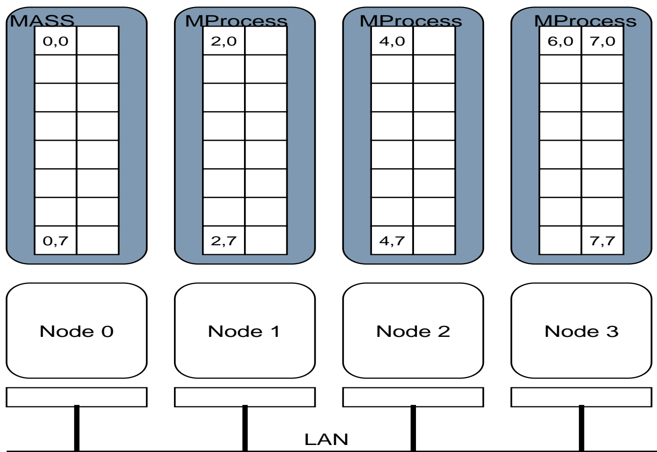


Figure 6. MASS Data Structure

An agent is attached to a specific Place, and a Place object can contain many agents. MASS agents are reactive and have the ability to perform simple tasks that are defined within the confines of a function call by the application developer.

2.5.2 Algorithm

This section details the major functionalities of the MASS library and the underlying algorithm.

(1) Places Call-All

The Call-All method allows the user to perform a user-specified operation on all cells in the simulation space in parallel.

The master node sends the “Call-All” command to all remote computing nodes and every node receives and executes the command in parallel with the main thread. The distributed array elements on each node are then further partitioned into even chunks for the worker threads and the main thread to process in parallel.

If a return value is required, the master node uses the JSCH channel created per remote computing node during the initialization phase to receive the return values, and each remote computing node collects and sends results with the main thread to the master node, which are presented to the user by the main thread on the master node.

(2) Places Exchange-All

The Exchange-All method allows the user to exchange data stored in each cell with all of its user-defined neighboring cells in the simulation space.

An exchange helper *ExchangeHelper* object is instantiated during the initialization phase to act as a server thread that constantly listens to a user-specified port, which is an optional argument provided for the *MASS.init()* call, for client connection requests from other computing nodes. This ensures timely response whenever a connection needs to be established. Figure 7 demonstrates how connections are established between remote processes during the execution of exchange all call. This inter-node connection is dynamically established on demand and the connection objects are reusable throughout the entirety of the application lifespan.

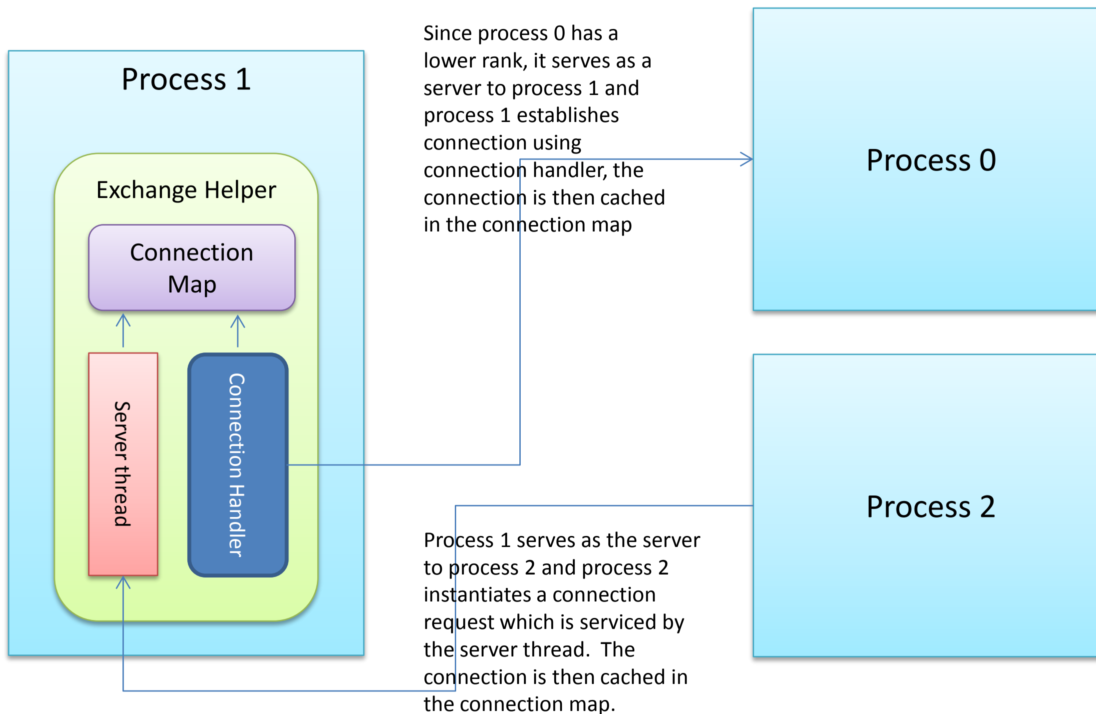


Figure 7. Connection Establishment between Computing Nodes

Figure 8 demonstrates the *exchangeAll* process. During the *exchangeAll* call, local exchanges are performed first, and if the exchange needs to be made with a remote node, each available thread (worker and main threads all participate in the computation) writes the host name of the remote node and the remote exchange request that needs to be sent to an exchange request map with the host name being the key.

After the local exchange is complete, all available threads (the main and worker threads) begin processing the remote exchange requests by selecting a hostname on a first-come-first-serve basis and process the remote exchange with the selected host. This is done to ensure that the connection object, which is cached on a map per remote host using the remote host name as key, is not shared by multiple threads.

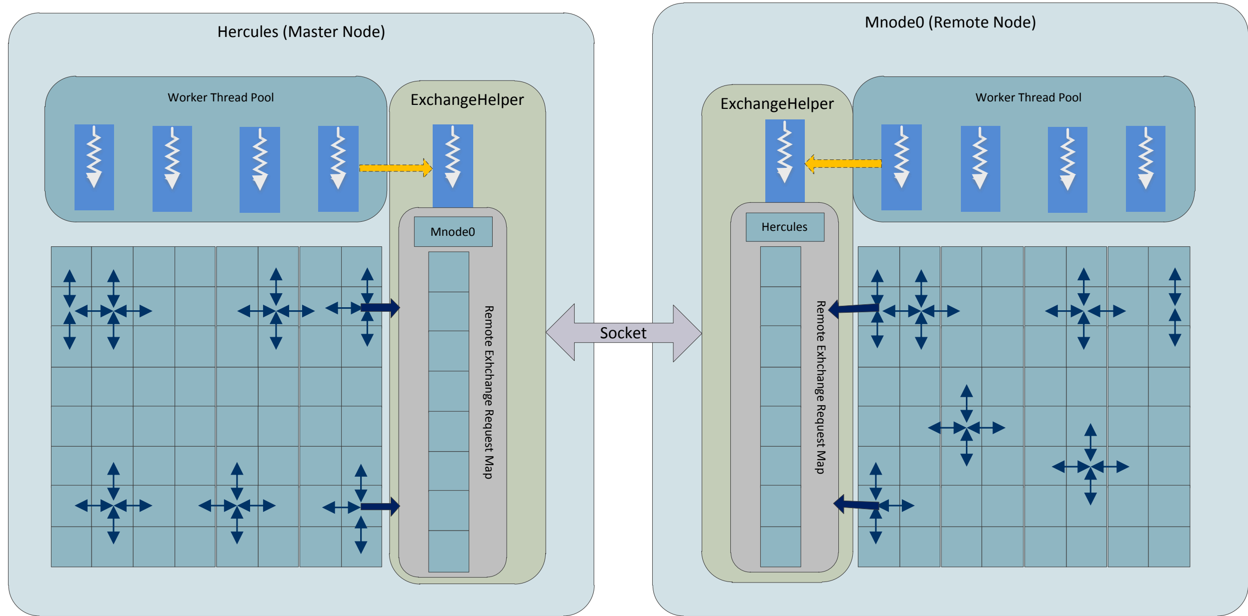


Figure 8. ExchangeAll Local and Remote Process

When a host is selected by a thread, it checks the connection map to see if the connection has already been established between the local host and the remote host. If not, the thread establishes connection and process the remote exchange request.

Ideally, there will be more threads than the computing nodes *exchangeAll( )* call needs to communicate with. However, there are cases where there are more computing nodes stored on the exchange request map than the available threads. In this case, the thread that finishes its assigned workload first picks up the remaining exchange requests and processes them. Table 2 details various threads used throughout the library.

Table 2. MASS Thread Types

Thread Type	Remark
Main Thread	It is the main application thread that controls the pacing of the application and participates in the computation. It also joins worker threads in handling remote exchange requests.
Worker Thread	Worker threads are spawned by the main thread that participate in the computation in parallel and handle remote exchange requests. The application developer specifies the number of total threads to spawn, which includes the one main thread and worker threads.
Server Thread	Server thread is created by the *ExchangeHelper* object that awaits connection requests from remote hosts in a blocking *accept( )* loop.

(3) Agents Initialization

This method allows the application developer to specify the number of agents to create and where agents should be mapped to the simulation space. A map function is provided to the user to specify how agents in the simulation space should be populated.

(4) Agents Call All

This method is similar to the regular Call All method in which each simulation space executes the same segment of code. In the case of mobile agents, each agent executes the same segment of code.

(5) Agents Manage All

After the agents-call all call is executed, each agent can queue up a number of actions to take and the manage-all call executes these actions. The main functionality is the migrate call which makes MASS agents autonomous.

Agent migrate requires each agent to travel from one simulation cell to another. Due to multi-process and multi-threaded nature of the MASS library, agent migration involves moving agents from one process to another. This is done by utilizing the "Exchange-All" infrastructure that is set up for the Places Exchange-All call in which each process establishes a socket communication with one another to send and receive necessary information.

Since user application's agent implementation is an extension of the MASS Agent class, it is against the ease of use design principle to ask the application developer to create the extension as a Java serializable object. As such, MASS agent migration is a weak migration [14]. Only the states of the agent will be transferred to the destination process. Weak migration implementation in MASS offers good performance and does not have any negative impact on user application due to the simple nature of reactive agents.

2.6 Implementation

The MASS library was designed with client server architectural style with implicit barrier synchronization to ensure all processes and threads proceed at the same pace. The client module

MProcess is implemented using an event-driven architectural style. This section contains detailed implementation description for the major components of MASS.

(1) MASS

MASS is the infrastructure of the library on which the user application is executed. It is responsible for construction and deconstruction of remote processes on a cluster of computing nodes, and maintains references to all *Places* and *Agents* instances.

Init( ) identifies all remote hosts as specified in the host file *machinefile.txt* and through JSCH, an *MProcess* is launched on remote hosts, and for every remote host, an Mnode instance is locally created on the master node as a wrapper for maintaining the master and slave JSCH connection. After all the remote processes have been launched, each process creates a pool of worker threads using the *MThread* instance that handles the coordination of multiple threads.

MASS can be initialized on any host running an SSH server and Java Virtual Machine. Throughout the application lifecycle, MASS provides useful methods for manipulating individual cells in the user defined simulation space, including direct control over Place and Agent instances.

Finish( ) is called at the end of an application to deconstruct the cluster. This is done by sending termination commands to all slave processes and closing all connections for a graceful exit.

(2) MNode

MNode is a wrapper for the JSCH connection object that allows direct communication between the master node and the slave nodes. A collection of MNodes will be created during initialization to facililate master-slave communication. Each MNode instance contains a set of wrappers for sending and receiving communication messages. This channel is also utilized for when the slave nodes need to send return values to the master node.

(3) MProcess

MProcess is run on remote hosts as a launching platform for MASS functions. The MProcess facilitates all commands invoked by the master node and manages program flow on its behalf. The MProcess has

three states in its lifecycle, initiation, running and deconstruction. During initialization, the MProcess establishes JSCH communication with the master node, creates a logger file for outputting error messages and instantiates a pool of worker threads. After the initialization has finished, it sits in an infinite loop and is blocked on a read() call awaiting commands from the master node. Once a command is received, it calls the corresponding MASS function and return to the blocking read state for the next command. The master node can terminate all MProcess instances by sending the finish command. When a finish command is received, MProcess closes all existing connections and exits the loop. Should any exceptions be encountered during the lifecycle of MProcess, it executes the finish command to free up all connections and logs the error to the logger file for debugging purposes.

(4) Mthread

Mthread is an extension of Java Thread class that is used to facilitate multi-threaded processing of the MASS library. All threads are synchronized on the MASS.STATUS object that can be changed by the calling process to different states to either wake up the worker threads or put them to sleep.

(5) ExchangeHelper

ExchangeHelper is a utility class used to establish and cache socket connection objects to facilitate inter-node communication. It is an extension of Java Thread class that is run during the MASS initialization phase to start a server thread that is blocked on *accept( )* awaiting client connection requests.

If a connection between two nodes needs to be established, the node with lower process ID acts as a server and the node with higher process ID sends a connection request. The request is picked up by the server thread and a socket connection is established and cached on a connection map with the remote host name as its key and the connection object as its value. This connection will be reused for future requests.

Connection establishment is done on demand. Whenever two nodes need to communicate with each other during an *exchangeAll( )* call, one of the worker threads first attempts to retrieve a connection object from the connection map. If the return value is null, it then calls *establishConnection( )* to establish connection to the remote host. All worker threads are synchronized on the connection map object to ensure that a cached socket connection is not shared by multiple threads.

(6) Places Implementation

Places manages all place elements in the simulation space. Every process maintains a collection of Places instances, each Places instance created by a user program on the master node has a corresponding instance on a number of slave processes in the cluster. There are two major methods for place manipulation in the Places class: callAll and exchangAll. The Places class utilizes ExchangeHelper instance to assist in the implementation of the *exchangeAll* algorithm.

1. *CallAll/Some( )*: These calls are for issuing commands and sending data to all or some place elements. Each MProcess receives a method identifier, place element indices, and arguments from the master node, and invokes the given method of the specified place elements. The return value of this method may be returned to master node after all executions have been completed. If there is a return value, the master node is blocked on receive via the JSCH channel until all return values have been received from the remote hosts.

2. *ExchangeAll/Some( )*: These calls are for exchanging data among place elements. Each MProcess receives a method identifier, and a collection of invoking destinations from the master node. All the local exchanges will be processed first. If a destination is outsides the bounds the local process, this remote exchange request is wrapped in a message and queued up on a hash map with the host name this message needs to be delivered to as the key. At the end of the local exchange process, *ExchangeHelper* instance is utilized to establish connection to the remote hosts, and each of the worker thread processes a set of requests that is stored on the hash map. The remote exchange step is deferred to the end of the local request and organized into key and value pair to avoid worker threads asynchronously sending exchange requests to the remote hosts that could potentially cause performance degradation and network congestion.

(7) Agents Implementation

Agents manages all agent elements in the simulation space. Unlike a place element which is tied to a specific cell in the simulation space, agents are free to migrate to other cells. At any given time in the simulation, a host can contain one to many agents. There are three major methods for Agents manipulation in the Agents class: CallAll, ExchangeAll and ManageAll.

1. *CallAll( )*: This call is similar to the Places.CallAll in a way that it is also used to issue commands to all agents. Each MProcess receives a method identifier and arguments from the master node, and invokes the given method of the specified agent elements.

There are a number of methods inside the Agent class such as migrate, spawn, suicide, sleep and wake up that are exposed to the application developer which can be used in the Agents.CallAll( ) call.

2. *ExchangeAll( )*: Similar to the Places.ExchangeAll( ), this method allows data exchange among all agents in the simulation space. It utilizes the ExchangeHelper instance to complete the exchange request in the same manner as Places.ExchangeAll( ).

3. *ManageAll( )*: This method is invoked to process Agent actions such as migrate, spawn and suicide. Each MProcess utilizes its worker thread pool to divide the agents up and iterate through all the agents in parallel to perform the user specified actions.

> *Migrate( )*: Given a set of coordinates, the agent will migrate to the destination upon the next *ManageAll( )* call. This method utilizes the *ExchangeHelper* instance to establish communication with the destination host and transmit agent states for re-instantiation. As opposed to *Places.ExchangeAll( )* where communications are all two-way, in the case of *Agent.Migrate( )*, it is not atypical for agents to migrate from local host to the destination host but not the other way around. If there are no agents to migrate, each host sends a simple acknowledgement byte to the destination host signaling that no remote migration needs to be performed to avoid being blocked on *read( )*.
>
> *Spawn( )*: This call allows the application developer to dynamically create any number of agents at the current place element.
>
> *Kill( )*: This call allows the application developer to terminate an agent. This is simply done by setting the agent's a*live* flag to false, and the next *ManageAll( )* call de-allocates all the agents with false *alive* flag.

Chapter 3: Evaluation

This chapter details the MASS library evaluation with programmability and performance analyses.

3.1 Programmability Analysis

This section gives the qualitative analysis of the MASS library, using *Wave2D* and *RandomWalk*.

3.1.1 Wave2D

Spatial simulation programmability analysis was conducted using Wave2D whose model and code are represented in figure 3. The steps to develop this spatial simulation program and any program using MASS are as follows. First, create a class that inherits from the Place, in this example *Wave*. Second, create call methods within the classes and assign them function IDs. Lastly, define a main function in which the simulation will be run.

The simulation starts with three parameters such as the *size* of the simulation, *max time* to finish the simulation and the *interval* on which the graphical updates are periodically displayed to the user. Then, it invokes the MASS library, creates a two-dimensional array that represents the simulation space by instantiating a new *Places* object. It sets up communication links from each array element to its four neighbors. Thereafter, the program enters a cyclic simulation where each iteration calculates the current wave height in every cell in parallel with a simple *Places.callAll( )*, and exchanges the wave height information among all array elements with a simple *Places.exchangeAll( )*.

The same application can be coded using a combination of MPI and OpenMP to take advantage of multi-process and multi-core CPUs on a cluster. However, the array representing Wave2D simulation space must be created and distributed by the application developer among multiple computing nodes with several MPI calls. Thereafter, in each iteration, all cells, and therefore, all computing nodes, need to exchange wave height information with neighboring nodes in order to calculate the wave height in the next iteration. This needs to be strategically planned to arrange the order in which *MPI_Read( )* and *MPI_Recv( )* calls are made to prevent processes from being blocked. To utilize multi-core CPUs, the application developer needs to enclose computation iterations in OpenMP compiler directives while identifying potential critical sections to avoid performance degradation.

Compared to the simple interfaces MASS provides to the application developer, MPI and OpenMP combination requires the application developer to possess knowledge of the programming interfaces of MPI and OpenMP and a certain degree of understanding of parallel programming paradigm. On the plus side, this approach allows application developers more fine-grained control over how distributed array elements are managed. MASS, on the other hand, offers simple, automatic solution to utilize multiple computing nodes equipped with multi-core CPUs, but does not allow any performance optimization beyond the confines of *callMethod* function definitions.

3.1.2 RandomWalk

Multi-agent programmability analysis was conducted using a simple multi-agent simulation program RandomWalk whose model and code are represented in figure 4. The program simulates a piece of land that is constructed as a two-dimensional Places array over which a number of nomads, each controlled by an individual agent and denoted as a red dot in the map, chooses the next un-occupied location to migrate to every iteration.

The steps to develop this multi-agent and spatial simulation program is similar to the above Wave2D example with only a few extra steps to add agents to the simulation. First, create a class that inherits from the Place and Agent classes, in this example *Land* and *Nomad* respectively. Second, create call methods within the classes and assign them function IDs. Lastly, create a driver program that will run the simulation.

The simulation starts with three parameters such as a map *size*, a given *nNomads* number of nomads and *max-Time* to finish the simulation. Then, it invokes the MASS library, creates a *land* array and distributes *nomad* objects on it. It also sets up communication links from each array element to its four neighbors. Thereafter, the program enters a cyclic simulation where each iteration exchanges the population status among all array elements with *Places.exchangeAll( )*, decides each nomad's next destination, and moves it there with *Agents.callAll( )* and *Agents.manageAll( ).*

The same application can also be developed using a combination of MPI and OpenMP. With this approach, in addition to instantiating and distributing an array that represents the simulation space array elements across multiple MPI ranks, which involves a series of MPI send/receive function calls, adding multi-agent functionalities requires instantiating a list of agents that represents *Nomad* and distribute list elements across multiple computing nodes in the same manner. It is up to the application developer to keep track of the process each nomad currently resides on and develop inter-process communication

strategies with the available interfaces MPI provides that will facilitate agent migration from one process to another. Once the multi-process parallelization is complete, the application developer then has to utilize OpenMP to take advantage of multi-core CPUs by strategically enclosing the simulation loops with OpenMP compiler directives while identifying critical sections to avoid performance degradation.

Adding agents on top of a distributed array translates to another layer of programming complexity with traditional MPI/OpenMP approach, whose individual behavior can be easily handled in parallel with the MASS library.

3.1.3 Summary

The above two examples demonstrate two programming advantages of the MASS library. One is a clear separation of the simulation scenario from the simulation models. The *main( )* function in both *Wave2D* and *RandomWalk* works as a scenario that introduces necessary models, instantiates/constructs entities and controls their interaction. This separation allows model designers to focus on each model design. The other advantage is automatic parallelization. The MASS library construct the simulation space array over multiple computing nodes, populates mobile agents on it as maintaining the mobile agent to simulation space array proximity, and calls their functions in parallel. These advantages can be applied to other multi-agent spatial simulations such as Fourier's heat simulation and artificial societies.

3.2 Execution Performance Analysis

Performance evaluation was conducted in the following areas: computation granularity, simulation size granularity, and application performance. All tests were conducted on a cluster of nodes each equipped with similarly configured dual 3.2 Xeon Ghz Intel processor with 1 GB of memory. The master node has an Intel Xeon E5520 processor with 6 GB of memory.

3.2.1 Data Size and Computation Granularity Analysis

(1) Places Simulation Size Granularity

First, we evaluated the performance of multi-threaded implementation of the MASS library. Multi-threaded simulation size granularity test was used to determine how large the simulation size needs to be in order to benefit from multi-threaded execution on a single node with up to 6 hardware threads. This

test was conducted for 1000 iterations. Each iteration contains a single *CallAll( )* and *ExchangeAll( )*, each having one iteration of floating point multiplications while the simulation size increases from 100 to 500. The results presented in Figure 9 indicate near linear scaling even with a small simulation size such as a 100x100 grid for up to four threads.

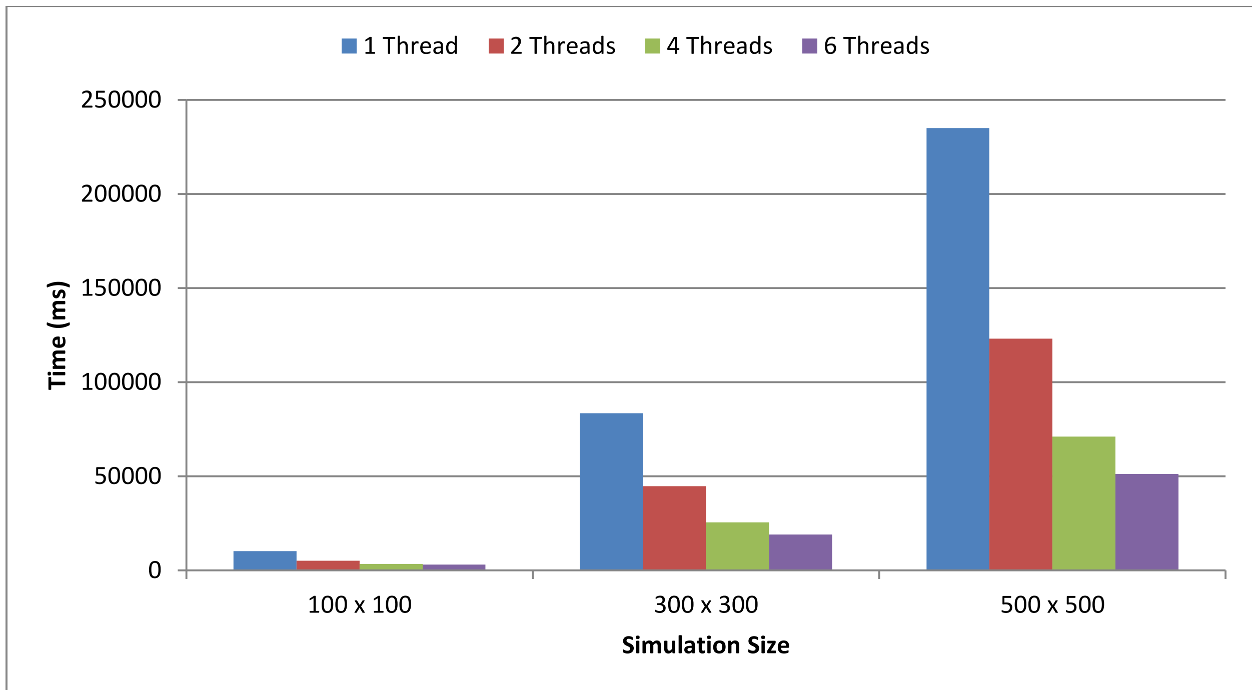


Figure 9. Multi-Threaded Places Granularity

The results demonstrated in figure 9 indicate that performance increases in a near linear fashion with additional threads, and the lab machines are configured to support up to four hardware threads, with the exception of the master node that can support up to six threads. Therefore, all tests from this point on were performed with one main thread and three worker threads for a total of four threads on each computing node for maximum performance.

Multi-threaded and multi-process Simulation size granularity test was performed to determine how large the simulation size needs to be in order to observe tangible performance increase as more computing nodes are added to the simulation. This test was conducted for 1000 iterations. Each iteration contains a single *CallAll( )* and *ExchangeAll( )* with one iteration of floating point multiplications to simulate a real world simulation, and the master node collects results once every 10 iterations while the simulation size

increases from 100 to 500 to provide near real time update to the user. This is particularly straining on the master node and is a very network performance bound operation.

Figure 10 shows that there is no tangible performance gain when the simulation size is smaller than a 300 by 300 grid, and the multi-node performance is bottle-necked by the master node and the network latency, but the performance increase can be observed when the simulation size increases to 500 by 500.

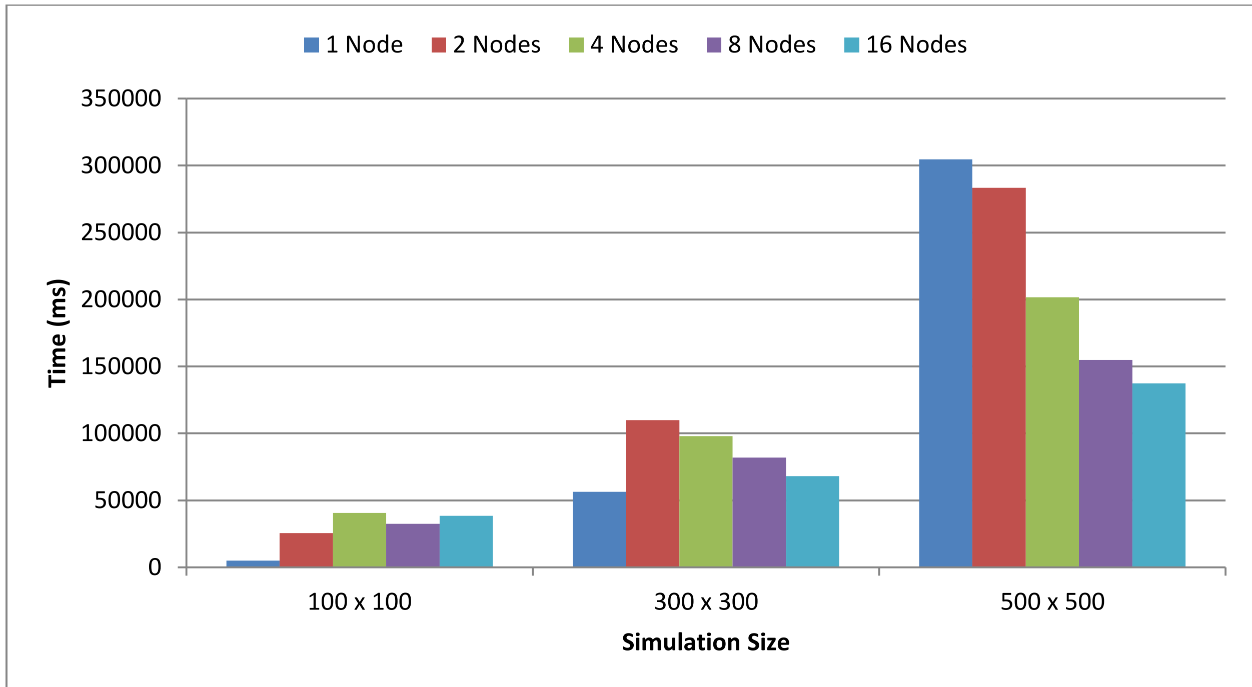


Figure 10. Multi-threaded and Multi-Process Places Granularity

(2) Places Computation Granularity

Based on the size granularity test results shown in figure 10, the following tests were performed using a 500 by 500 simulation grid to meet the minimum size requirement to obtain scalable performance with additional computing nodes. Computation granularity test was conducted separately for *callAll( )* and *exchangeAll( )* to observe their overheads as granularity increases from fine-grained to coarse-grained computation.

Figure 11 shows the execution performance of *callAll( )*. The test was conducted using a 500 by 500 simulation grid with each iteration containing one single *callAll( )* function call. The computation increases from 1 to 100 iterations of floating point multiplication in each simulation iteration, and the result is near linear scaling of performance when more computing nodes are added to the simulation as

expected since this simulates an embarrassingly parallel application where the user is only interested in the end result.

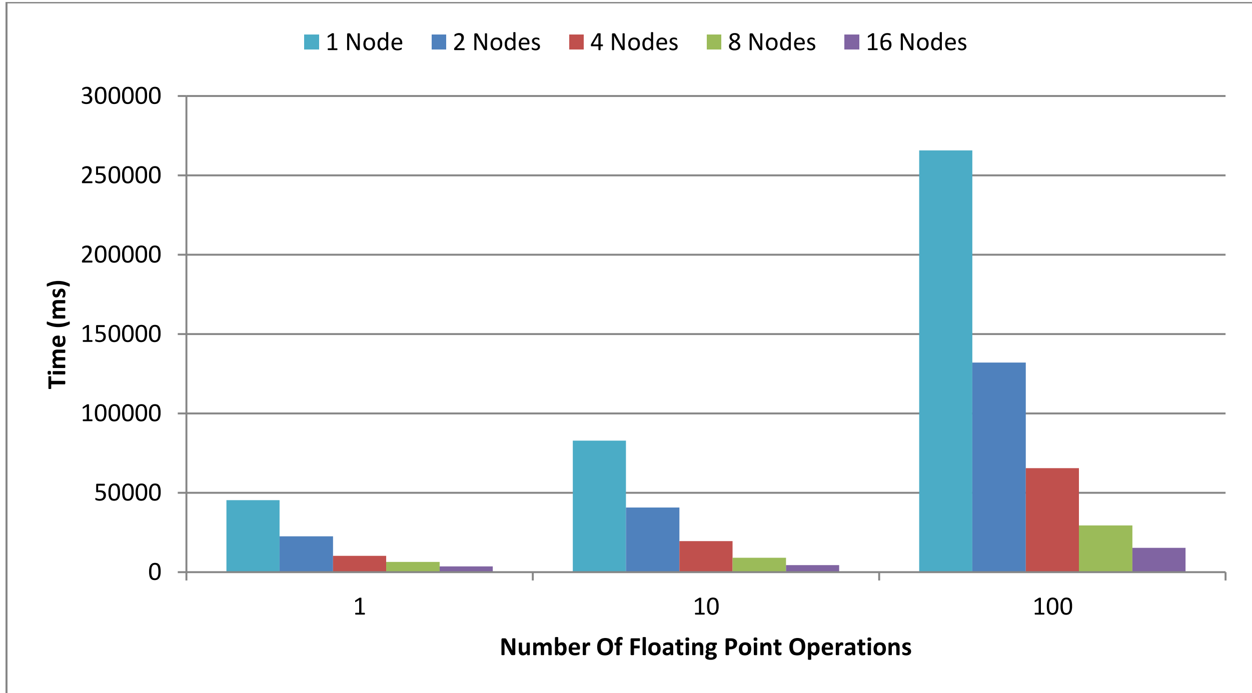


Figure 11. Performance of CallAll

In certain applications, user expects periodic updates of simulation states. Figure 12 shows the performance of *callAll( )* where the intermittent results are collected by the master node once every 10 iterations. The test was conducted using a simulation space of 500 by 500 grid for 1000 iterations. Each iteration contains a single *callAll( )* call while the computation granularity increases from 1 to 500 iterations of floating point operations.

Performance advantage a single node holds over multiple nodes can be observed with fine to medium computation granularity, and the performance gain is minimal with additional computing nodes due to the fact that frequent result collection introduces large network overhead with multiple nodes. However, as the computation granularity increases to 100 iterations of floating point computation, the performance of two nodes starts to outperform single node execution as the network overhead starts to be outweighed by the heavy computation, and the performance of 16 nodes finally shows improvement over 8 nodes when the granularity is increased to 500 iterations of floating point multiplications.

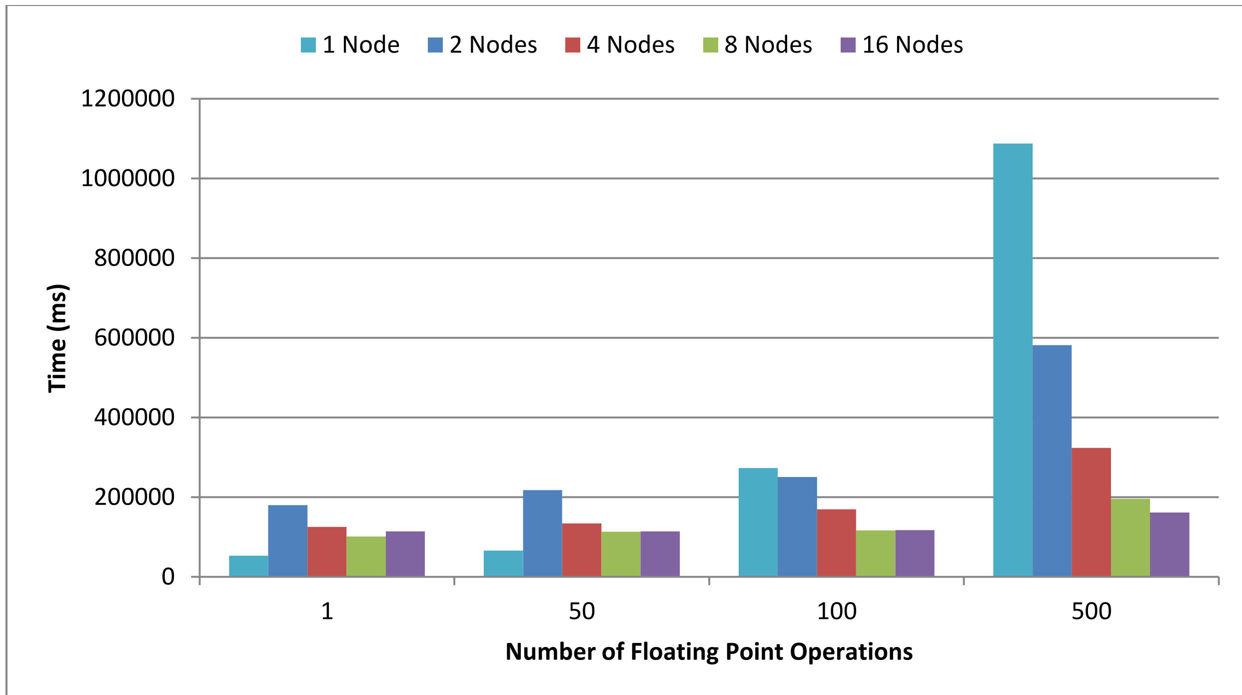


Figure 12. Performance of CallAll with Return Values

exchangeAll( ) performance was evaluated using the same criteria as the test above. Figure 13 shows that the single node outperformed multiple nodes up until the computation granularity increases to 100 iterations of floating point multiplications, and the increase is not as observable as the increase gained from parallel execution of *callAll( )* due to the fact that each *exchangeAll ( )* call involves communication with the nearest neighbors, however, the result indicates favorable performance gain as more computing nodes are added, and computation granularity is at least more than 100 iterations of floating point multiplication in such a network performance bound execution.

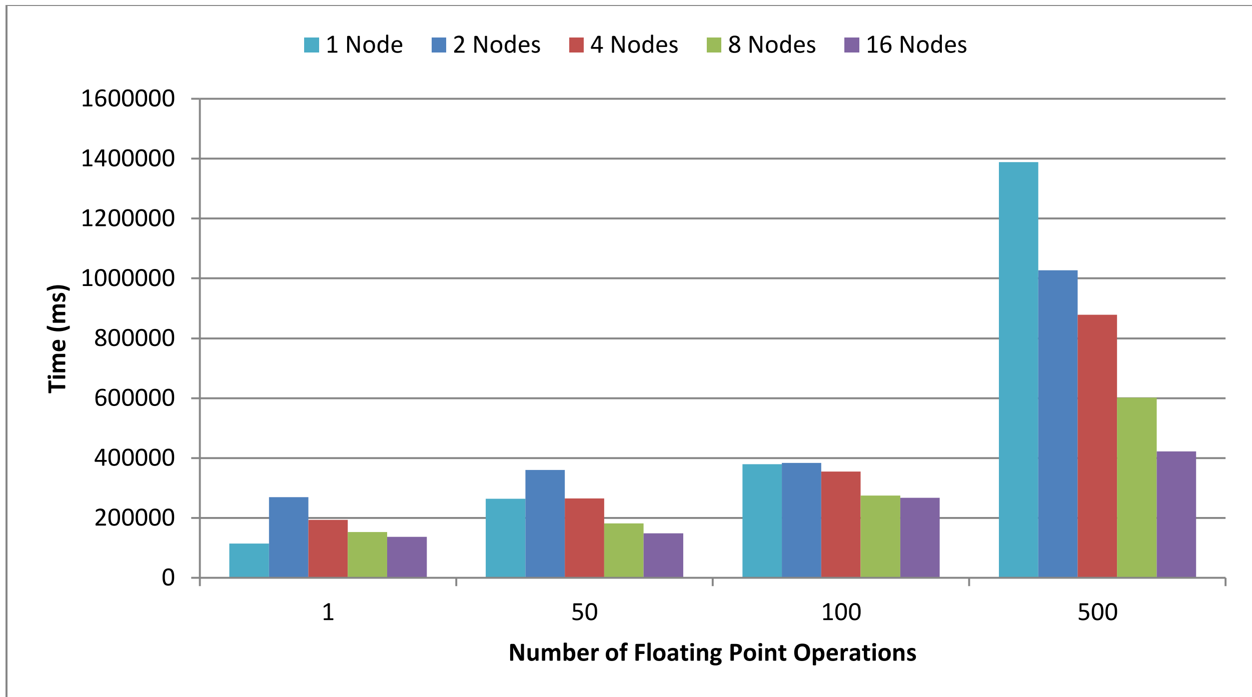


Figure 13. Performance of ExchangeAll with Return Values

(3) Agents Granularity

Agents granularity test was performed to see how many agents need to be created and distributed in order to observe tangible performance increase as more computing nodes are added to the simulation. This test was conducted for 1000 iterations on a 500 by 500 simulation grid with agents evenly distributed. Each iteration contains an *Agents.callAll( )* function call with one iteration of fine-grained floating point multiplications and an *Agent.migrate( )* call. Results are collected by the master node once every 10 iterations.

Figure 14 shows an irregular pattern when the simulation grid size is greater than the number of agents. The performance with 16 nodes is worse than the performance with 8 nodes. As the number of agents increases, the performance increases in an expected pattern similar to *Places.exchangeAll( )* as the underlying algorithms for *Places.exchangeAll( )* and *Agent.migrate( )* are very similar. *Agent.migrate( )* performs better than *Places.exchangeAll( )* because *Places.exchangeAll( )* requires two way communication where each node exchanges information with all of its neighbors a, whereas, in *Agent.migrate ( )*, oftentimes there are agents that migrate from the host node to the destination node, but the destination node does not have any agents to send back to the host node. In this case, a simple

acknowledgement message is sent when there are no agents to send for migrate, which results in a low network overhead and little computation overhead.

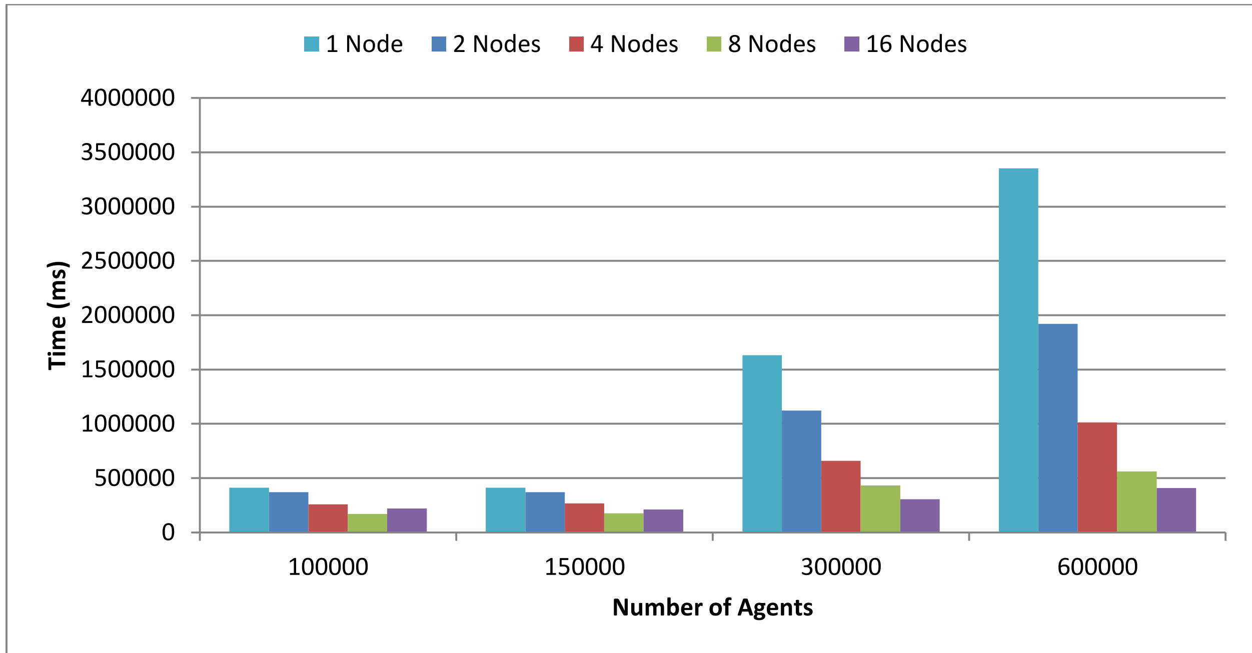


Figure 14. Agents Granularity

3.2.2 Application Performance: Places

This evaluation focuses on the place evaluation, using Wave2D.

Wave2D was chosen for its fine grained computation and nearest neighbor communication requirements. The test was conducted with a simulation grid of 1500 by 1500 in size for 1000 iterations. The much larger grid size was chosen to stress the cluster and to demonstrate performance of the library with a large problem set and all the available computing resources (up to 23 nodes) at the time the test was performed. A centralized result collection call is issued once every 10 iterations. Each iteration contains a *callAll( )* and an *exchangeAll( )* function calls.

In this test, the benefit of distributed memory became apparent as a 1500 by 1500 simulation size is too large for a single and two computing nodes to handle. Figure 15 demonstrates the total execution performance of Wave2D. Although the performance scaling is far from ideal, this demonstrates the performance of a type of application where the user demands near real time update on the simulation,

which is particularly dependent on the network and the master node performance. The performance gain with additional nodes is still desirable.

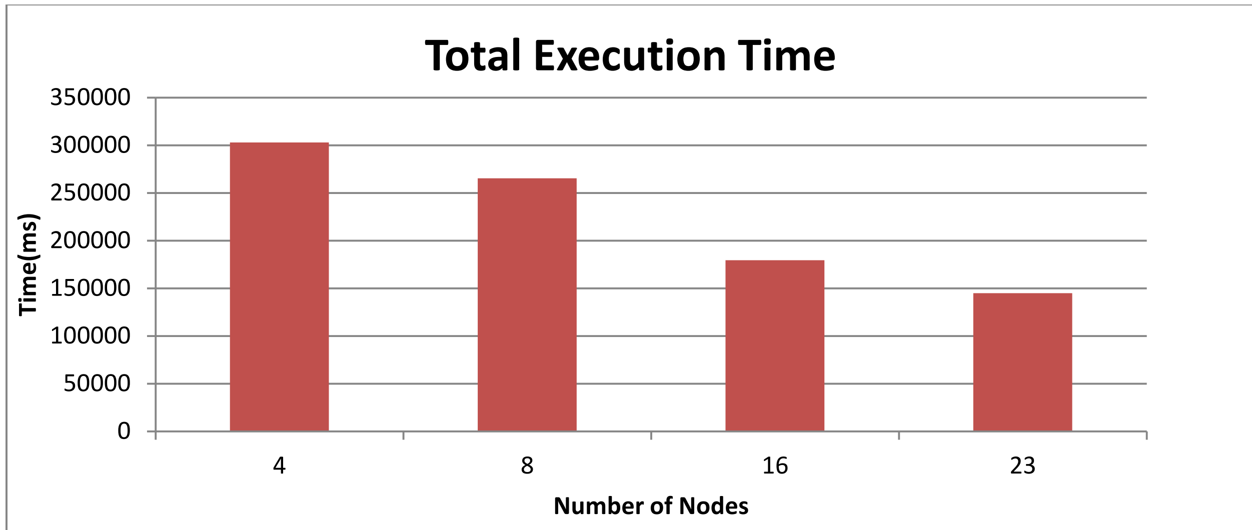


Figure 15. Total Execution Time of Wave 2D

Now we take a closer look at the *ExchangeAll( )* performance where the application spends the majority of the time processing. Figure 16 demonstrates that as more computing nodes are added into the simulation, the time spent exchanging data with nearest neighbors reduces in a near-linear fashion.

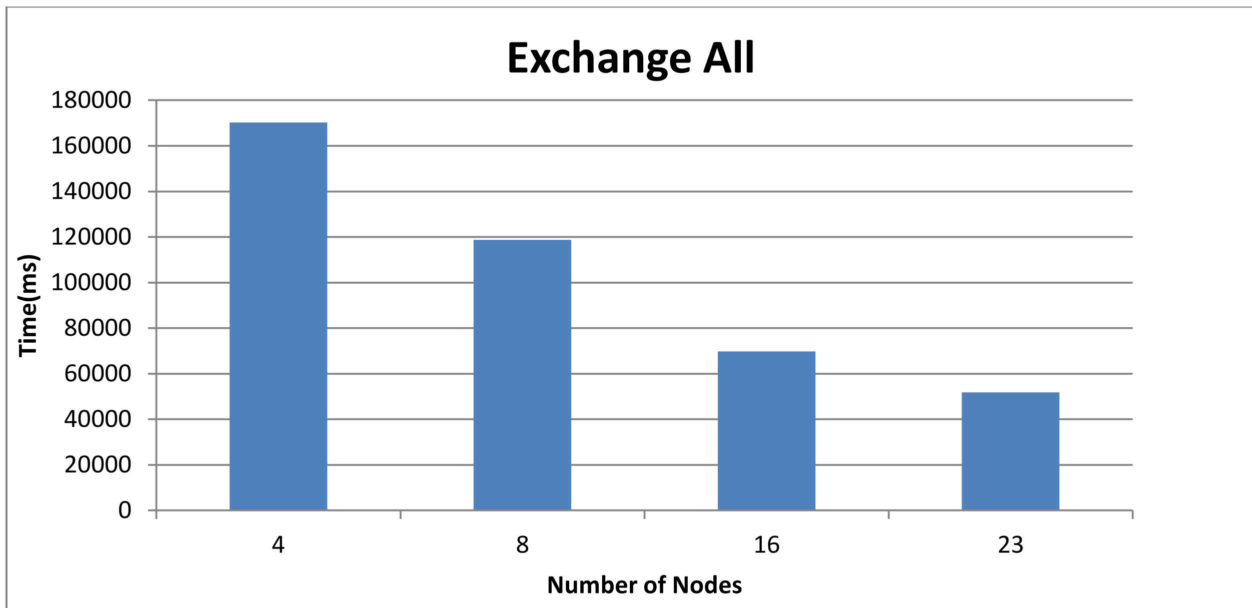


Figure 16. Total ExchangeAll Execution Time

3.2.3 Application Performance: Agents

This evaluation focuses on the agent evaluation, using RandomWalk.

Based on the performance results of agent and simulation size granularity shown in figure 14 and 10 respectively, simulation size of 500 by 500 with 1000 simulation iterations and 300000 agents were selected to meet the minimum performance scalability requirements.

Figure 17 demonstrates the performance. The blue bars (bars on the left) show the execution performance of RandomWalk application where each iteration contains a *Agents.callAll( )* , *Agents.manageAll( )* and *Places.callAll( )* to collect results once every 10 iterations. The performance increase with additional nodes exhibits similar pattern as the *Places.exchangeAll( )* execution performance due to the fact that *Agents.migrate( )* uses the same logic.

The red bars (bars on the right) demonstrate the execution performance where each iteration contains *Agents.callAll( )*, *Agents.manageAll( ), Places.exchangeAll( )* to update neighboring cells and *Places.callAll( )* to collect results once every 10 iterations. This simulates a situation where fine-grained computation is bottlenecked by the master node and the network performance as each iteration requires a node to exchange information with neighboring nodes twice. The results demonstrate that even with frequent periodic result collection by the master node, the performance increase is noticeable when more computing nodes are added to the simulation, and in all situations, multiple computing nodes outperform single computing node performance even with such fine grained computation.

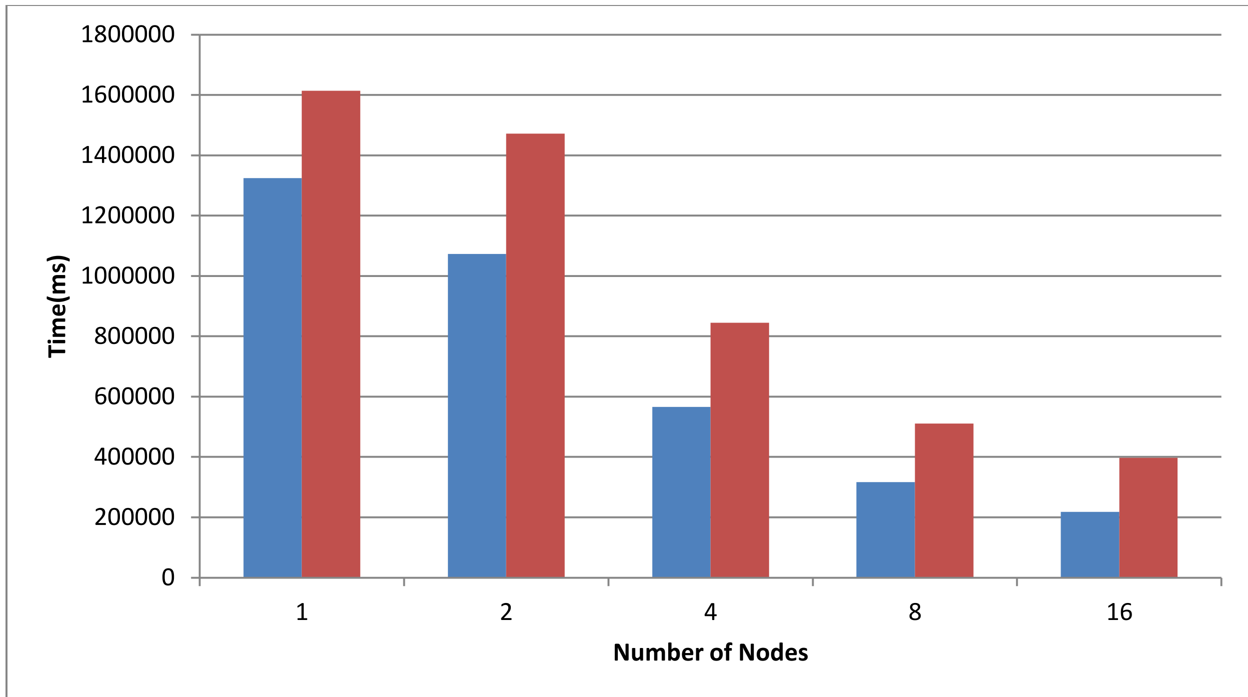


Figure 17. Execution Performance of RandomWalk

3.3 Summary

Execution performance results demonstrate that when the user is only interested in the end result, the performance increase is near linear regardless of computation granularity, and in applications where users require intermittent updates of simulation states, having additional nodes still provides performance enhancement for medium to coarse grained computation that is in-line with network bound, centralized management style where the performance is bottlenecked by the master node.

Based on the performance evaluation results, the minimum condition to benefit from automatic parallel execution using the MASS library is as follows:

- Places (Simulation size): 500 x 500.
- Agents: 300,000 over 500 x 500.
- Computation granularity:
    - If intermittent updates are required: 100 floating point operations.
    - If only end result is required: 1 floating point operation.

Chapter 4: Related Work

This chapter differentiates MASS from its related work in two major aspects: (1) distributed shared arrays and (2) parallel multi-agent simulation environments.

4.1 Distributed Array

Example systems supporting distributed shared arrays include UPC: Unified Parallel C [11], Co-Array Fortran [8] and GlobalArray [7]. UPC allocates global memory space in the sequential consistency model, which is then shared among multiple threads running on different computing nodes. Co-Array Fortran allows “so-called” images, (i.e. different execution entities including ranks, processes and threads) to co-allocated, to perform one-sided operations onto, and to synchronize on shared arrays.

GlobalArray (GA) facilitates not only one-sided but also collective operations onto global arrays that are shared among different computing nodes. Parallelization of user application is done with the use of Message Passing Interface (MPI), and GA provides various degrees of control to the application developer to exploit data locality for increased performance and multiple levels of memory hierarchy for data access optimization. However, all of the performance optimization features GA offers require advanced understanding of parallel programming paradigm, whereas, MASS aims to ease application development by hiding parallel programming complexity by exposing a set of interfaces to the user to easily manipulate the behavior of each individual cell (array element) in parallel with implicit synchronization.

Although MASS has a similarity as these language-based runtime systems in allocating global shared arrays, it is unique in implementing both one-sided and collective operations as the form of user-defined remote method invocations rather than providing users with a system-predefined operations. In particular, exchangeAll/Some operations in MASS do not invoke a method call to each array element, but rather invoke a parallel call from each to other elements (In other words, inter-element parallel calls).

4.2 Multi-Agents

Most multi-agent simulation environments such as PDES-MAS [5] and MACE3J [2] focus on parallel execution of coarse-grained cognitive agents, each with rule-based behavioral autonomy. These systems provide agents with interest managers that work as inter-agent communication media to exchange spatial

information as well as multi-cast an event to agents. From the viewpoints of agent-to-space or agent-to-event proximity, PDES-MAS recursively divides each interest manager into child managers, structures them in a hierarchy, and maps them over a collection of computing nodes for parallelization. MASS is different from these systems in handling fine-grain reactive agents that sustain a partial view of their entire space and interact with other agents in their neighborhood. Although an array element in MASS can be considered as interest manager in PDES-MAS and MACE3J, MASS instantiates a large number of array elements (i.e. interest managers), and define their logical connection with exchangeAll/Some functions.

Another similar framework to consider is Nomadic Threads [12]. Nomadic Threads also places emphasis on parallel execution of coarse-grained cognitive agents that reside on top of a distributed array. Agent's autonomy is achieved through thread migration, and it is able to take advantage of data locality by processing all computation locally before migrating to a different computing node and ultimately migrating back to the user's computer where the results will be stored. Similarly, MASS deploys multiple agents that reside on top of a distributed array. However, agents in MASS are fine-grained reactive agents that can perform tasks in parallel, and whenever migration is required, only the current states of an agent are transferred to the destination host for re-instantiation. In contrast to Nomadic Threads' strong migration where execution states of a thread are required to be transferred to the destination host, MASS agent migration can be quickly performed in parallel with little implication on performance, and have the ability to facilitate inter-agent communication whenever desired by the application developer with the exposed interfaces such as the *exchangeAll* function.

In summary, MASS offers a unique approach in facilitating user-defined inter-element communication in distributed arrays and realizing fine-grain reactive agents, each interacting with others through numerous array elements.

Chapter 5: Conclusion

The MASS library is intended to facilitate entity-based simulation for on-the-fly sensor-data analysis. We have demonstrated the programming advantages in using the MASS library for such simulation as well as its highly scalable performance in parallel execution of fine-to-medium grained computation. This work is a proof of concept that MASS is an excellent parallel execution platform that is centered on the ease of use aspect and provides a scalable solution for entity based simulations. The MASS library has ample room for further performance enhancements to achieve competitive performance compared with similar execution platforms.

5.1 Result Statement

The MASS library provides application developers with an easy-to-use parallelization framework that is highly scalable. Its unique approach is well suited for model based simulations where application developers can focus on the behavior of each individual entity without having any knowledge of complex parallelization techniques.

5.2 Problems Encountered

During the development of the MASS library, it became immediately apparent that the multi-threaded nature of the library would require the use of shared memory, which resulted in an over-abundance of global variable declarations that are needed for thread synchronization. As such, for optimal performance, the master node needs to be the most capable node in the cluster and is oftentimes the bottle neck in the MASS execution performance.

The machines utilized in the performance evaluation are shared among all students on campus. There were times where tests conducted would have very different results using the same computing nodes. To better validate the test results, all tests were run multiple times and any outliers were removed from the results. Conducting large tests with complex simulation applications that would consume all resources on shared computing nodes were prohibitive as all students on campus have equal access to the equipment. The largest test size was restricted to 500 by 500, and any higher would render multiple computing nodes unusable.

5.3 Future Work

MASS is an on-going project and is a part of the Sensor Grid integration. The current implementation of MASS is missing some multi-agent functionality, which will be implemented and tested in the near future. There is also room for more performance optimization in the current incarnation. For instance, the Agent's *migrate( )* is currently locally handled in parallel by the worker thread pool, but the remote migration is handled by one single thread. This can be expanded to multi-thread for enhanced performance similar to Places *exchangeAll/Some* functions. The next step with the Java version is to perform benchmarks using Java-Grande suite and further fine-tune the library.

There are also multiple versions of MASS currently in development to support different platforms such as a version to support CUDA based Nvidia GPU and a version to support C++. In the near future, we hope to combine the C++ and CUDA/OpenCL versions to be able to utilize a cluster of computing nodes each having a high performance GPU to achieve supercomputer-like performance.

References

[1] A. Deshpande, C. Guestrin, S. R. Madden, J. M. Hellerstein, and W. Hong. Model-based approximate querying in sensor networks. The VLDB Journal, Vol.14(No.4):417{443, November 2005.

[2] L. Gasser and K. Kakugawa. MACE3J: fast exible distributed simulation of large, large-grain multi-agent systems. In Proc. of the _rst international joint conference on Autonomous agents and multiagent systems - AAMAS-2002, Bologna, Italy, November 2001. ACM.

[3] V. Grimm and S. F. Railsback. Individual-based Modeling and Ecology. Princeton Univesity Press, Princeton, NJ, 2005.

[4] M. M. Hassan, B. Song, and E.-N. Huh. A framework of sensor-cloud integration opportunities and challenges. In Proc. of the 3rd International Conference on Ubiquitous Information Management and Communication, pages 618{626, Suwon, Korea, January 2009. ACM.

[5] B. Logan and G. Theodoropoulos. The distributed simulation of multi-agent systems. Proceedings of the IEEE, Vol.89(No.2):174{186, January 2001.

[6] Multi-Agent Transport Simulation - MATSim. http://www.matsim.org/.

[7] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Apra. Advances, applications and performance of the global arrays shared memory programming toolkit. International Journal of High Performance Computing Applications, Vol.20(No.2):203{ 231, 2006.

[8] R. W. Numrich and J. Reid. Co-array fortran for parallel programming. ACM SIGPLAN Fortran Forum, Vol.17(No.2):1{31, August 1998.

[9] B. A. Smith, G. Hoogenboom, and R. W. McClendon. Arti_cial neural networks for automated year-round temperature prediction. Computers and Electronics in Agriculture, Vol.68(Issue.1):52{61, August 2009.

[10] UC Davis Biometeorology Program - Frost Protection. http://biomet.ucdavis.edu/frost-protection.html.

[11] UPC Consortium. UPC Language. Language specifications 1.2, The High Performance Computing Laboratory, George Washington University, http://upc.gwu.edu/, May 2005.

[12] Stephen Jenks and Jean-Luc Gaudiot. Nomadic Threads: A Migrating Multithreaded Approach to Remote Memory Accesses in Multiprocessors. Conference on Parallel Architectures and Compilation Techniques, October 1996.

[13] John Spiger. Multi-Threaded MASS Library. Multi-Threaded MASS Library Report, UWB Distributed Systems Lab, http://depts.washington.edu/dslab/SensorCloud/report/JohnS_su10.pdf, September 2010.

[14] David R.A. de Groot, Frances M.T. Brazier and Benno J. Overeinder. Cross-Platform Generative Agent Migration. Intelligent Interactive Distributed Systems Group, Computer Science, Faculty of Sciences Vrije Universiteit Amsterdam, http://www.iids.org/publications/CPGAM_DRAdeGroot.pdf

Appendix A: Source Code and User Manual

The MASS library source code and user manual are available for download on the UWB distributed systems lab website at
http://depts.washington.edu/dslab/SensorCloud/index.html

Appendix B: Detailed Language Specification

public class Places

`Public`	`Places( int handle, String className, Object argument, int… size )` Instantiates a shared array with "size" from the "className" class as passing an Object argument to the "className" constructor. This array is associated with a user-given handle that must be unique over machines.
`public int`	`getHandle( )` Returns the handle associated with this array.
`public int[]`	`size( )` Returns the size of this multi-dimensional array.
`public void`	`callAll( int functionId )` Calls the method specified with functionId of all array elements. Done in parallel among multi-processes/threads.
`public void`	`callAll( int functionId, Object argument )` Calls the method specified with functionId of all array elements as passing an Object argument to the method. Done in parallel among multi-processes/threads.
`public Object[]`	`callAll( int functionId, Object[] arguments )` Calls the method specified with functionId of all array elements as passing arguments[i] to element[i]'s method, and receives a return value from it into Object[i]. Done in parallel among multi-processes/threads. In case of a multi-dimensional array, "i" is considered as the index when the array is flattened to a single dimension.
`public void`	`callSome( int functionId, int... index )` Calls the method specified with functionId of one or more selected array elements as passing. If index[i] is a non-negative number, it indexes a particular element, a row, or a column. If index[i] is a negative number, say –x, it indexes every x element. Done in parallel among multi-processes/threads.
`public void`	`callSome( int functionId, Object argument, int... index )` Calls the method specified with functionId of one or more selected array elements as passing an Object argument to the method. The format of index[] is the same as the above callSome( ). Done in parallel among multi-processes/threads.
`public Object[]`	`callSome( int functionId, Object[] argument, int... index )` Calls the method specified with functionId of one or more selected array elements as passing argument[i] to element[i]'s method, and receives a return value from it into Object[i]. The format of index[ ] is the same as the above callSome( ). Done in parallel among multi-processes. In case of a multi-dimensional array, "i" is considered as the index when the array is flattened to a single dimension.
`public void`	`exchangeAll( int handle, int functionId, Vector<int[]> destinations )`

	Calls from each of all cells to the method specified with functionId of all destination cells, each indexed with a different Vector element. Each vector element, say destination[] is an array of integers where destination[i] includes a relative index (or a distance) on the coordinate i from the current caller to the callee cell. The caller cell's outMessage, (i.e., an Object) is a set of arguments passed to the callee's method. The caller's inMessages[], (i.e., an array of Objects) stores values returned from all callees. More specifically, inMessages[i] maintains a set of return values from the ith callee.
`public void`	**`exchangeSome( int handle, int functionId, Vector<int[]> destinations, int... index)`** Calls from each of the cells indexed with index[ ] (whose format is the same as the above callSome( )) to the method specified with functionId of all destination cells, each indexed with a different Vector element. Each vector element, say destination[ ] is an array of integers where destination[i] includes a relative index (or a distance) on the coordinate i from the current caller to the callee cell. The caller cell's outMessages[], (i.e., an array of Objects) is a set of arguments passed to the callee's method. The caller's inMessages[], (i.e., an array of Objects) stores values returned from all callees. More specifically, inMessages[i] maintains a set of return values from the ith callee.

public abstract class Place

`public`	**`Place( Object args )`** Is the default constructor. No primitive data types can be passed to the methods, since they are not derivable from the "Object" class.
`public final int[]`	**`size`** Defines the size of the matrix that consists of application-specific places. Intuitively, size[0], size[1], and size[2] correspond to the size of x, y, and z, or that of i, j, and k.
`public final int[]`	**`index`** Is an array that maintains each place's coordinates. Intuitively, index[0], index[1], and index[2] correspond to coordinates of x, y, and z, or those of i, j, and k.
`public Vector`	**`agents`** Includes all the agents residing locally on this place.
`public boolean[]`	**`eventIds`** includes eventIds[0] through to eventIds[9], each corresponding to event 1 through to 10. If eventIds[i] is set true, Agents.manage( ) wakes up all agents sleeping on event i +1. After a call from Agents.mange( ), eventIds[i] is resent false.

public static Object	**callMethod(int functionId, Object[] arguments)** Is called from Places.callAll( ), callSome( ), callStatic( ), exchangeAll( ), and exchangeSome( ); and invokes mass_0, mass_1, mass_2, mass_3, or mass_4 whose postfix number corresponds to functionId. An application may override callMethod( ) so as to direct Places to invoke an application-specific method
public Object	**outMessages** Stores a set arguments to be passed to a set of remote-cell functions that will be invoked by exchangeAll( ) or exchangeSome( ) in the nearest future.
public Object[]	**inMessages** Receives a return value in inMessages[i] from a function call made to the i-th remote cell through exchangeAll( ) and exchangeSome( ).

public class Agents

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 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.
public int	**getHandle( )** Returns the handle associated with this agent set.
public int[]	**nAgents( )** Returns the total number of agents.
public void	**callAll(int functionId)** Calls the method specified with functionId of all agents. Done in parallel among multi-processes/threads.
public void	**callAll(int functionId, Object argument)** Calls the method specified with functionId of all agents as passing an Object argument to the method. Done in parallel among multi-processes/threads.
public Object[]	**callAll(int functionId, Object[] arguments)** Calls the method specified with functionId of all agents as passing arguments[i] to agent[i]'s method, and receives a return value from it into Object[i]. Done in parallel among multi-processes/threads. The order of agents depends on the index of a place where they resides, starts from the place[0][0]…[0], and gets increased with the right-most index first and the left-most index last.
public void	**manageAll( )**

	Updates each agent's status, based on each of its latest migrate( ), spawn( ), kill( ), sleep( ), wakeup( ), and walekupAll( ) calls. These methods are defined in the Agent base class and may be invoked from other functions through callAll and exchangeAll. Done in parallel among multi-processes/threads.
public void	**sortAll(boolean descending)** Sorts agents within each place in the descending order of their "key" values.
public void	**exchangeAll(int handle, int functionId)** Allows each agent to call the method specified with functionId of all the other agents residing within the same place as where the calling agent exists as well as belonging to the agent group with "handle". The caller agent's outMessage, (i.e., an Object) is a set of arguments passed to the callee's method. The caller's inMessages[], (i.e., an array of Objects) stores values returned from all callee agents. More specifically, inMessages[i] maintains a set of return values from the ith callee.

public abstract class Agent

public	**Agent(Object args)** Is the default constructor. No primitive data types can be passed to the methods, since they are not derivable from the "Object" class.
public Place	place Points to the current place where this agent resides.
private int[]	**index** Is an array that maintains the coordinates of where this agent resides. Intuitively, index[0], index[1], and index[2] correspond to coordinates of x, y, and z, or those of i, j, and k.
Public final int	**agentId** Is this agent's identifier. It is calculated as: the sequence number * the size of this agent's belonging matrix + the index of the current place when all places are flattened to a single dimensional array.
Public final int	**parented** Is the identifier of this agent's parent.
private int	**newChildren** Is the number of new children created by this agent upon a next call to Agents.manageAll( ).
private Object[]	**arguments** Is an array of arguments, each passed to a different new child.
private boolean	**alive** Is true while this agent is active. Once it is set false, this agent is killed upon a next call to Agents.manageAll( ).
private int	**eventId** indicates which event this agent sleeps on. The eventId should be between 1 and 10. All the other numbers mean that the agent does not sleep.

private int	**Key** Is the value used to sort agents.
public static int	**map(int maxAgents, int[] size, int[] coordinates)** Returns the number of agents to initially instantiate on a place indexed with coordinates[]. The maxAgents parameter indicates the number of agents to create over the entire application. The argument size[] defines the size of the "Place" matrix to which a given "Agent" class belongs. The system-provided (thus default) map( ) method distributes agents over places uniformly as in: maxAgents / size.length The map( ) method may be overloaded by an application-specific method. A user-provided map( ) method may ignore maxAgents when creating agents.
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[].
public void	**create(int numAgents, Object[] arguments)** Spawns a "numAgents' of new agents, as passing arguments[i] to the i-th new agent upon a next call to Agents.manageAll( ). More specifically, create( ) changes the calling agent's newChildren.
public void	**kill( )** Termiantes the calling agent upon a next call to Agents.manageAll( ). More specifically, kill( ) sets the "alive" variable false.
public boolean	**sleep(int eventId)** Puts the calling agent to sleep on a given eventId whose value should be 1 through to 10. If eventId is not in the range of 1 through to 10, the agent will not be suspended. The sleep( ) function returns true if the agent is suspended successfully.
public void	**wakeup(int eventId)** Wakes up only one agent that is sleeping on a given eventId within the same place as where this calling agent resides.
public void	**wakeupAll(int eventId)** Wakes up all agents that are sleeping on a given eventId within the same place as where this calling agent resides.
public void	**setKey(int value)** Substitutes the calling agent's "key" variable a given value. It is used for sorting agents within the same place.
public Object	**callMethod(int functionId, Object[] arguments)** Is called from Agents.callAll( ) and exchangeAll( ), and invokes mass_0, mass_1, mass_2, mass_3, or mass_4 whose postfix number corresponds to functionId. An application may override callMethod( ) so as to direct Agents to invoke an application-specific method
public Object	**outMessages** Stores a set arguments to be passed to a set of remote-cell functions that will be invoked by exchangeAll( ) in the nearest future.
public Object[]	**inMessages** Receives a return value in inMessages[i] from a function call made to the i-th remote cell through exchangeAll( ).

Appendix C: Wave2D Source Code

```
1.    import MASS.*;              // Library for Multi-Agent Spatial Simulation
2.    import java.util.*;         // for Vector
3.    import java.awt.*;          // uses the abstract windowing toolkit
4.    import java.awt.event.*;    // also uses key events so we need this
5.
6.    public class Wave2D extends Place {
7.        // constants
8.        public static final int init_ = 0;
9.        public static final int computeWave_ = 1;
10.       public static final int exchangeWave_ = 2;
11.       public static final int collectWave = 3;
12.       public static final int startGraphics_ = 4;
13.       public static final int writeToGraphics_ = 5;
14.       public static final int finishGraphics_ = 6;
15.
16.       // wave height at each cell
17.       // wave[0]: current, wave[1]: previous, wave[2]: one more previous height
18.       double[] wave = new double[3];
19.
20.       int time = 0;
21.       int interval = 0;
22.
23.       // wave height from four neighbors: north, east, south, and west
24.       private final int north = 0, east = 1, south = 2, west = 3;
25.       double[] neighbors = new double[4];
26.
27.       // simulation constants
28.       private final double c  = 1.0; // wave speed
29.       private final double dt = 0.1; // time quantum
30.       private final double dd = 2.0; // change in system
31.
32.       // the array size and my index in (x, y) coordinates
33.       private int sizeX, sizeY;
34.       private int myX, myY;
35.
36.       /**
37.        * Is the constructor of Wave2D.
38.        * @param interval a time interval to call writeToGraphics( )
39.        */
40.       public Wave2D( Object interval ) {
41.          this.interval = ( ( Integer )interval ).intValue( );
42.       }
43.
44.       public static Object callMethod( int funcId, Object args ) {
45.          switch( funcId ) {
46.          case init_: return init( args );
47.          case computeNewWave_: return computeNewWave( args );
48.          case exchangeWave_: return ( Object )exchangeWave( args );
49.          case collecdtWave_: return ( Object )collectWave( args );
50.          case startGraphics_: return startGraphics( args );
51.          case writeToGraphics_: return writeToGraphics( args );
52.          case finishGraphics_: return finishGraphics( args );
53.          }
54.          return null;
55.       }
56.
57.       /**
58.        * Since size[] and index[] are not yet set by
59.        * the system when the constructor is called, this init( ) method must
60.        * be called "after" rather than "during" the constructor call
```

```
61.          * @param args formally declared but actually not used
62.          */
63.         public Object init( Object args ) {
64.            sizeX = size[0]; sizeY = size[1]; // size  is the base data members
65.            myX = index[0];  myY = index[1];  // index is the base data members
66.
67.            // reset the neighboring area information.
68.            neighbors[north] = neighbors[east] = neighbors[south] =
69.                neighbors[west] = 0.0
70.
71.            return null;
72.         }
73.
74.         /**
75.          * Compute this cell's wave height at a given time.
76.          * @param arg_time the current simulation time in Integer
77.          */
78.         public Object computeWave( Object arg_time ) {
79.            // retrieve the current simulation time
80.            time = ( Integer )arg_time.intValue( );
81.
82.            // move the previous return values to my neighbors[].
83.            if ( inMessage != null ) {
84.                for ( int i = 0; i < 4; i++ )
85.                   neighbors[i] = ( Double )inMessage[i].doubleValue( );
86.            }
87.
88.            if ( myX == 0 || myX == sizeX - 1 || myY == 0 || myY == sizeY ) {
89.                // this cell is on the edge of the Wave2D matrix
90.                if ( time == 0 )
91.                   wave[0] = 0.0;
92.                if ( time == 1 )
93.                   wave[1] = 0.0;
94.                else if ( time >= 2 )
95.                   wave[2] = 0.0;
96.            }
97.            else {
98.                // this cell is not on the edge
100.               if ( time == 0 ) {
101.                 // create an initial high tide in the central square area
102.                  wave[0] =
103.                       ( sizeX * 0.4 <= myX && myX <= sizeX * 0.6 &&
104.                         sizeY * 0.4 <= myY && myY <= sizeY * 0.6 ) ? 20.0 : 0.0;
105.                  wave[1] = wave[2] = 0.0;
106.               }
107.               else if ( time == 1 ) {
108.                  // simulation at time 1
109.                  wave[1] = wave[0] +
110.                       c * c / 2.0 * dt * dt / ( dd * dd ) *
111.                       ( neighbors[north] + neighbors[east] + neighbors[south] +
112.                         neighbor[west] - 4.0 * wave[0] );
113.               }
114.               else if ( time >= 2 ) {
115.                  // simulation at time 2
116.                  wave[2] = 2.0 * wave[1] - wave[0] +
117.                       c * c * dt * dt / ( dd * dd ) *
118.                       ( neighbors[north] + neighbors[east] + neighbors[south] +
119.                         neighbors[west] - 4.0 * wave[1] );
120.               }
121.           }
122.           wave[0] = wave[1]; wave[1] = wave[2];
123.           return null;
124.        }
```

```
125.
126.        /**
127.         * Exchange the local wave height with all my four neighbors.
128.         * @param args formally declared but actually not used.
129.         */
130.        public Double exchangeWave( Object args ) {
131.          return new Double( ( ( time == 0 ) ? wave[0] : wave[1] ) );
132.        }
133.
134.        /**
135.         * Return the local wave height to the cell[0,0]
136.         * @param args formally declared but actually not used.
137.         */
138.        public Double collectWave( Object args ) {
139.          return new Double( wave[2] );
140.        }
141.
142.        // Graphics
143.        private static final int defaultN = 100; // the default system size
144.        private static final int defaultCellWidth = 8;
145.        private static Color bgColor;            //white background
146.        private static Frame gWin;               // a graphics window
147.        private static int cellWidth;            // each cell's width in the window
148.        private static Insets theInsets;         // the insets of the window
149.        private static Color wvColor[];          // wave color
150.        private static int N = 0;                // array size
151.        private static int interval = 1;         // graphic interval
152.
153.        // start a graphics window
154.        public Object startGraphics( Object args ) {
155.          // define the array size
156.          N = size[0];
157.
158.          // Graphics must be handled by a single thread
159.          bgColor = new Color( 255, 255, 255 );//white background
160.
161.          // the cell width in a window
162.          cellWidth = defaultCellWidth / ( N / defaultN );
163.          if ( cellWidth == 0 )
164.              cellWidth = 1;
165.           // initialize window and graphics:
166.           gWin = new Frame( "Wave Simulation" );
167.           gWin.setLocation( 50, 50 );  // screen coordinates of top left corner
168.           gWin.setResizable( false );
169.           gWin.setVisible( true );     // show it!
170.           theInsets = gWin.getInsets();
171.           gWin.setSize( N * cellWidth + theInsets.left + theInsets.right,
172.                         N * cellWidth + theInsets.top + theInsets.bottom );
173.
174.           // wait for frame to get initialized
175.           long resumeTime = System.currentTimeMillis() + 1000;
176.           do {} while (System.currentTimeMillis() < resumeTime);
177.
178.           // paint the back ground
179.           Graphics g = gWin.getGraphics( );
180.           g.setColor( bgColor );
181.           g.fillRect( theInsets.left,
182.                       theInsets.top,
183.                       N * cellWidth,
184.                       N * cellWidth );
185.
186.          // prepare cell colors
187.           wvColor = new Color[21];
```

```
188.           wvColor[0] = new Color( 0x0000FF );   // blue
189.           wvColor[1] = new Color( 0x0033FF );
190.           wvColor[2] = new Color( 0x0066FF );
191.           wvColor[3] = new Color( 0x0099FF );
192.           wvColor[4] = new Color( 0x00CCFF );
193.           wvColor[5] = new Color( 0x00FFFF );
194.           wvColor[6] = new Color( 0x00FFCC );
195.           wvColor[7] = new Color( 0x00FF99 );
196.           wvColor[8] = new Color( 0x00FF66 );
197.           wvColor[9] = new Color( 0x00FF33 );
198.           wvColor[10] = new Color( 0x00FF00 );  // green
199.           wvColor[11] = new Color( 0x33FF00 );
201.           wvColor[12] = new Color( 0x66FF00 );
202.           wvColor[13] = new Color( 0x99FF00 );
203.           wvColor[14] = new Color( 0xCCFF00 );
204.           wvColor[15] = new Color( 0xFFFF00 );
205.           wvColor[16] = new Color( 0xFFCC00 );
206.           wvColor[17] = new Color( 0xFF9900 );
207.           wvColor[18] = new Color( 0xFF6600 );
208.           wvColor[19] = new Color( 0xFF3300 );
209.           wvColor[20] = new Color( 0xFF0000 );  // red
210.
211.           System.out.println( "graphics initialized" );
212.          return null;
213.       }
214.
215.       // update a graphics window with new cell information
216.       public Object writeToGraphics( Object arg_waves ) {
217.          Double[] waves = ( Double[] )arg_waves;
218.
219.          Graphics g = gWin.getGraphics( );
220.
221.          for ( int i = 0; i < sizeX; i++  )
222.              for ( int j = 0; j < sizeY; j++ ) {
223.                 // convert a wave height to a color index ( 0 through to 20 )
224.                 int index = ( int )( wave[i * sizeY + j ] / 2 + 10 );
225.                 index = ( index > 20 ) ? 20 : ( ( index < 0 ) ? 0 : index );
226.
227.                 // show a cell
228.                 g.setColor( wvColor[index] );
229.                 g.fill3DRect( theInsets.left + myX * cellWidth,
230.                              theInsets.top  + myY * cellWidth,
231.                              cellWidth, cellWidth, true );
232.          }
233.          return null;
234.       }
235.
236.       // finish the graphics window
237.       public Object finishGraphics( Object args ) {
238.          Graphics g = gWin.getGraphics( );
239.          g.dispose( );
240.          gWin.removeNotify( );
241.          gWin = null;
242.
243.          return null;
244.       }
245.
246.       /**
247.        * Starts a Wave2 application with the MASS library.
248.        * @param receives the array size, the maximum simulation time, the graphic
249.        *        updating time, the number of processes to spawn, and the
250.        *        number of threads to create.
251.        */
```

```
252.       public static void main( String[] args ) {
253.          // validate the arguments.
254.          if ( args.length != 5 ) {
255.                System.err.println( "usage: " +
256.                                    "java Wave2D size time graph_interval" +
257.                                    "#processes #threads" );
258.                System.exit( -1 );
259.           }
260.          int size = Integer.parseInt( args[0] );
261.          int maxTime = Integer.parseInt( args[1] );
262.          int interval = Integer.parseInt( args[2] );
263.          int nProcesses = Integer.parseInt( args[3] );
264.          int nThreads = Integer.parseInt( args[4] );
265.
266.          // start MASS
267.          MASS.init( args, nProcesses, nThreads );
268.
269.          // create a Wave2D array
270.          Places wave2D = new Places( 1, "Wave2D",
280.                                   ( Object )( new Integer( interval ) ),
281.                                   size, size );
282.          wave2D.callAll( init_, null );
283.
284.          // start graphics
285.          if ( interval > 0 )
286.              wave2D.callSome( startGraphics_, null, 0, 0 );
287.
288.          // define the four neighbors of each cell
289.          Vector<int[]> neighbors = new Vector<int[]>( );
290.          int[] north = { 0, -1 }; neighbors.add( north );
291.          int[] east  = { 1,  0 }; neighbors.add( east );
292.          int[] south = { 0,  1 }; neighbors.add( south );
293.          int[] west  = { -1, 0 }; neighbors.add( west );
294.
295.          Date startTime = new Date( );
296.
297.          // now go into a cyclic simulation
298.          for ( int time = 0; time < maxTime; time++ ) {
299.             wave2D.callAll( computeWave_, ( Object )( new Integer( time ) ) );
300.             wave2D.exchangeAll( 1, exchangeWave_, neighbors );
301.             // at every given time interval, display the array contents
302.             if ( time % interval == 0 ) {
303.                Object[] waves = wave2D.callAll( collectWave_, null );
304.                wave2D.callSome( writeToGraphics_, waves, 0, 0 );
305.             }
306.          }
307.
308.          Date endTime = new Date( );
309.          System.out.println( "elapsed time = " +
310.                              ( endTime.getTime( ) - startTime.getTime( ) ) );
311.
312.          // stop graphics
313.          if ( interval > 0 )
314.              wave2D.callSome( finishGraphics_, null, 0, 0 );
315.
316.          MASS.finalize( );
317.       }
318.    }
```

Appendix D: RandomWalk Source Code

```
 1.   import MASS.*;             // Library for Multi-Agent Spatial Simulation
 2.   import java.util.Vector;   // for Vector
 3.
 4.   // Simulation Scenario
 5.   public class RandomWalk {
 6.       /**
 7.        * Starts a RandomWalk application with the MASS library
 8.        * @param receives the Land array size, the number of initial agents, and
 9.        *                 the maximum simaution time.
10.        */
11.       public static void main( String[] args ) {
12.          // validate teh arguments
13.          if ( args.length != 3 ) {
14.              System.err.println( "usage: " +
15.                                  "java RanodomWalk size nAgents maxTime" );
16.              System.exit( -1 );
17.          }
18.          int size = Integer.parseInt( args[0] );
19.          int nAgents = Integer.parseInt( args[1] );
20.          int maxTime = Integer.parseInt( args[2] );
21.
22.          // start MASS
23.          MASS.init( args );
24.
25.          // create a Land array.
26.          Places land = new Places( 1, "Land", null, size, size );
27.
28.          // populate Nomda agents on the land.
29.          Agents nomad = new Agents( 2, "Nomad", null, land, nAgents );
30.
31.          // define the four neighbors of each cell
32.          Vector<int> neighbors = new Vector<int>( );
33.          int[] north = { 0, -1 }; neighbors.add( north );
34.          int[] east  = { 1,  0 }; neighbors.add( east );
35.          int[] south = { 0,  1 }; neighbors.add( south );
36.          int[] west  = { -1, 0 }; neighbors.add( west );
37.
38.          // now go into a cyclic simulation
39.          for ( int time = 0; time < maxTime; time++ ) {
40.              // exchange #agents with four neighbors
41.              land.exchangeAll( 1, Land.exchange, neighbors );
42.              land.callAll( Land.update );
43.
44.              // move agents to a neighbor with the least population
45.              nomad.callAll( Nomad.decideNewPosition );
46.              nomad.manageAll( );
47.          }
48.
49.          // finish MASS
50.          MASS.finalize( );
51.       }
52.   }
53.
54.   // Land Array
55.   public class Land extends Place {
56.       // function identifiers
57.       public static final int exchange_ = 0;
58.       public static final int update_ = 1;
```

```
59.
60.       /**
61.        * Is called from callAll( ) or exchangeAll( ), and forwards this call to
62.        * update( ) or exchange( ).
63.        * @param funcId the function Id to call
64.        * @param args argumenets passed to this funcId.
65.        */
66.       public static Object callMethod( int funcId, Object args ) {
67.          siwtch ( funcId ) {
68.          case exchange_: return exchange( args );
69.          case update_: return update( args );
70.          }
71.          return null;
72.       }
73.
74.       int[][] neighbors = new neighbors[2][2]; // keep my four neighbors' #agents
75.
76.       /**
77.        * Is called from exchangeAll( ) to exchange #agents with my four neighbors
78.        * @param args formally requested but actuall not used.
79.        */
80.       public Object exchange( Object args ) {
81.          return new Integer( agents.size( ) );
82.       }
83.
84.       /**
85.        * Is called from callAll( ) to update my four neighbors' #agents.
86.        * @param args formally requested but actuall not used.
87.        */
88.       public Object update( Object args ) {
89.          int index = 0;
90.          for ( int x = 0; x < 2; x++ )
91.              for ( int y = 0; y < 2; y++ )
92.                 neighbors[x][y] = ( inMessages[index] == null ) ?
93.                     Integer.MAX_VALUE ?
94.                     ( Integer )inMessages[index].intValue( );
95.          return null;
96.       }
97.   }
98.
99.   // Nomad Agents
100.  public class Nomad extends Agent {
101.      /**
102.       * Instantiate an agent at each of the cells that form a square
103.       */ in the middle of the matrix
104.      public static int map( int maxAgents, int[] size, int[] coordinates ) {
105.
106.         sizeX = size[0], sizeY = size[1];
107.         int populationPerCell = maxAgents / ( sizeX * 0.6 * sizeY * 0.6 );
108.         currX = coordinates[0], currY = coordinates[1];
109.         if ( sizeX * 0.4 < currX && currX < sizeX * 0.6 &&
110.               sizeY * 0.4 < currY && currY < sizeY * 0.6 )
111.             return populationPerCell;
112.         else
113.             return 0;
114.      }
115.
116.      // function identifiers
117.      public static final int decideNewPosition = 0;
118.
119.      /**
120.       * Is called from callAll( ) and forwards this call to
121.       * decideNewePosition( )
```

```
122.        * @param funcId the function Id to call
123.        * @param args argumenets passed to this funcId.
124.        */
125.       public static Object callMethod( int funcId, Object args ) {
126.          siwtch ( funcId ) {
127.          case decideNewPosition_: return decideNewPosition( args );
128.          }
129.          return null;
130.       }
131.
132.       /**
133.        * Computes the index of a next cell to migrate to.
134.        * @param args formally requested but actually not used
135.        */
136.       public Object decideNewPositioin( Object args ) {
137.          int newX = 0;                 // a new destination's X-coordinate
138.          int newY = 0;                 // a new destination's Y-coordinate
139.          int min = Integer.MAX_VALUE;  // a new destination's # agents
140.
141.          int currX = place.index[0], currY = place.index[1]; // the curr index
142.          int sizeX = place.size[0]; sizeY = place.size[1];   // the land size
143.
144.          for ( int x = 0; x < 2; x++ )
145.              for ( int y = 0; y < 2; y++ ) {
147.                 if ( currY < 0 )
148.                     continue; // no north
149.                 if ( currX >= sizeX )
150.                     continue; // no east
151.                 if ( currY >= sizeY )
152.                     continue; // no south
153.                 if ( currX < 0 )
154.                     continue; // no west
155.                 if ( place.neighbors[x][y] < min ) {
156.                     // found a candidate cell to go.
157.                     newX = x;
158.                     newY = y;
159.                     min = ( Land )place.neigbors[i];
160.                 }
161.              }
162.
163.          // let's migrate
164.          migrate( newX, newY );
165.
166.          return null;
167.       }
168.   }
```