A Parallel Multi-Agent Spatial Simulation Environment for Cluster Systems

Timothy Chuang and Munehiro Fukuda
Computing & Software Systems
University of Washington Bothell
18115 NE Campus Way, Bothell, WA 98011
{ctc, mfukuda}@uw.edu

***Abstract*—For more than the last two decades, multi-agent simulations have been highlighted to model mega-scale social or biological agents and to simulate their emergent collective behavior that may be difficult only with mathematical and macroscopic approaches. A successful key for simulating mega-scale agents is to speed up the execution with parallelization. Although many parallelization attempts have been made to multi-agent simulations, most work has been done on shared-memory programming environments such as OpenMP, CUDA, and Global Array, or still has left several programming problems specific to distributed-memory systems, such as machine unawareness, ghost space management, and cross-processor agent management (including migration, propagation, and termination). To address these parallelization challenges, we have been developing MASS, a new parallel-computing library for multi-agent and spatial simulation over a cluster of computing nodes. MASS composes a user application of distributed arrays and multi-agents, each representing an individual simulation place or an active entity. All computation is enclosed in each array element or agent; all communication is scheduled as periodic data exchanges among those entities, using machine-independent identifiers; and agents migrate to a remote array element for rendezvousing with each other. This paper presents the programming model, implementation, and evaluation of the MASS library.**

I. Introduction

The multi-agent model, sometimes coupled with individual-based model, gained in popularity when Swarm [1] was developed to model a large number of social or biological agents and to simulate their emergent collective behavior that may be difficult only with mathematical and macroscopic approaches. A successful key is how to handle mega-scale agents within an acceptable time, which is generally facilitated by parallel computing. For instance, FluTE, a multi-agent influenza epidemic simulation with 10 million people takes about 2 hours, and therefore uses OpenMP and MPICH for a parallel simulation of the entire continental US with 280 million people, using 32 cluster nodes [2].

Although many parallelization attempts have been made to multi-agent simulations, most parallelization work has been limited to shared-memory programming environments such as multithreading, OpenMP, and CUDA [2], [3], [4], [5]. It is much more challenging to parallelize multi-agent simulation on distributed-memory systems. The challenges can be characterized into (1) programming and (2) execution performance problems. The former problems are mainly brought from partitioning and allocating a simulation space and a collection of agents to different cluster nodes, each with an independent memory space. They include *machine unawareness* to refer to a remote simulation space and name each agent transparently, *ghost space management* to allow agents to read and modify the boundary of adjacent remote simulation space, *guarded agent migration* to avoid agents from colliding each other at the same logical location, and *agent propagation and distributed termination* to spawn and terminate agents dynamically. The latter problems are resulted from handling mega-scale agents for more accurate simulation with a larger problem. More specifically, we need to facilitate *fine-grained scalable computation* of agents by reducing their inter-processor communication overheads.

To address these problems, we have been developing MASS, a new parallel-computing library for multi-agent and spatial simulation. 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 array element or agent; all communication is scheduled as periodic data exchanges among those entities using their machine-independent identifiers; and agents migrate to a different array element (on a remote cluster node) for rendezvousing with each other. These MASS features can therefore allow a user application to intuitively mimic actual phenomenon such as epidemic among people in FluTE [2] and traffic jams in MatSim [6].

The contribution of this paper is two-fold: (1) giving new solutions to the above programming problems in multi-agent parallelization and (2) empirically demonstrating competitive performance of fine-grained scalable computation of agents. The rest of the paper is organized as follows: Section II examines the current problems in parallelizing multi-agent simulations; Sections III and IV discusses the MASS programming model and its current implementation; Sections V and VI evaluate the MASS programmability and execution performance; and Section VII concludes our discussions.

II. Related Work

Frequent interaction among mega-scale agents via their simulation space is a challenge for parallelizing a multi-agent simulation over a distributed-memory system. The successful key is how to visualize remote sub-spaces to distributed agents. One solution is to use a distributed shared array whose element is intended to describe a simulation sub-space and maintain agents residing on it. The other is to use an existing multi-agent simulation environment to develop an application, based on its API. This section considers these two programming platforms.

A. Distributed Array

Example systems supporting distributed shared arrays include UPC: Unified Parallel C [7], Co-Array Fortran [8], and GlobalArray [9]. 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 share, to perform one-sided operations onto, and to synchronize on distributed arrays. They are all based on the distributed shared memory concept. If an entire array is made visible to all agents, the simulation scalability is limited due to their memory consistency enforcement. Therefore, GlobalArray facilitated ghost spaces that are adjacent boundaries of remote spaces, visible to local agents. However, ghost spaces are updated by their home, (thus remote) computing node but not by local agents. Although this read-only feature is sufficient for some spatial simulations such as computational fluid dynamics and Schrödinger's wave diffusion, many agent-based applications (including traffic and epidemic simulations [6], [2]) need to move local agents onto ghost spaces, thus by allowing them to update the ghost spaces. Since agents are described as data members in each array element, model designers are responsible for manually implementing agent management such as duplication, migration, termination, and collision avoidance. Finally, array elements and agents are indexed and executed by iterative loops, which makes model designers aware of which machines maintain which simulation objects.

B. Multi-Agents

Most multi-agent systems such as PDES-MAS [10] and MACE3J [11] 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 to multicast an event to agents. However, their nature of coarse-grained agents and multicast-based interest managers obstructs mega-scale agent simulations. Another similar framework to consider is Nomadic Threads [12]. These threads migrate over a distributed array to make all data accesses local to the threads for realizing thread-memory proximity. Similarly to the above systems, Nomadic Threads are coarse-grained cognitive agents that are not suitable for a mega-scaled agent simulation. Repast HPC [13] is a large-scale agent-based modeling system that was implemented on top of MPI and tested on Argonne National Laboratory's Blue Gene/P. The system gives a C++ based agent framework, shared contexts as inter-agent communication media, and ghost spaces visible to adjacent processes, which facilitated basic requirements for agent parallelization. However, it has been designed from the hawk's viewpoint or in the top-down strategy where a user must launch a Repast process at each MPI rank, update agent status with iterative loops, construct logical network spaces with MPI ranks, and use read-only ghost spaces similar to those implemented in distributed arrays.

In many cases, mega-scale agent simulations are modeled from the individual-based viewpoint, (i.e., in the bottom-up strategy), and used for observing their emergent collective behavior or self-organization. Therefore, to assist model designers, (i.e, non-computing specialists) in agent parallelization, we need to address (1) individual-based design, (2) machine unawareness, (3) readable/writable ghost spaces, and (4) system-level agent management.

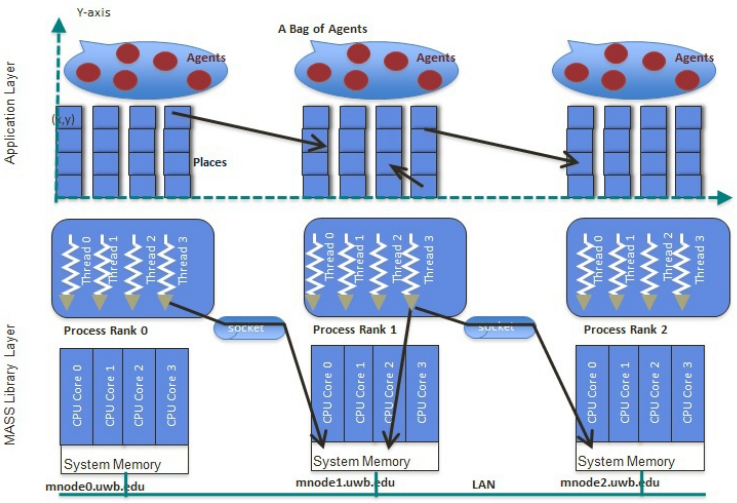

Fig. 1. MASS execution model.

III. Programming Model

To address the above problems in agent parallelization, we have first developed the MASS library in Java, and are currently porting it to C++ and CUDA platforms. Its design principles are three-fold: (1) *individual-based design*: allows simulation users to focus on agent and space design by automating underlying agent and (ghost) space management; (2) *machine-unaware transparent environment*: relieves simulation users from model parallelization by automating parallel agent execution and migration; and (3) *scalable and fine-grain parallelization*: facilitates mega-scale agent simulation by implementing MASS with low-level system calls rather than additional middleware libraries such as MPI.

A. Execution Model

Places and *Agents* are keys to the MASS library. *Places* is a multi-dimensional array of elements that are dynamically allocated over a cluster of multi-core computing nodes. Each element is called a *place*, is pointed to by a set of network-independent array indices, and is capable of exchanging information with any other *place*s. *Agents* are a set of execution instances that can reside on a *place*, migrate to any other *place*s with array indices (thus as duplicating themselves), and interact with other *agent*s as well as multiple *place*s. As shown in Figure 1, parallelization with the MASS library uses a set of multi-threaded communicating processes that are forked over a cluster of multi-core computing nodes and are connected to each other through ssh-tunneled TCP links. The library spawns the same number of threads as that of CPU cores per node. Those threads take charge of method call and information exchange among *place*s and *agent*s in parallel.

B. Library Specification

A user designs a behavior of a *place* and an *agent* by extending the *Place* and *Agent* base classes respectively.

They are populated through the *Places* and *Agents* classes. Actual computation is performed between *MASS.init()* and *MASS.finish()*, using the following major methods [14]. (Note that the following discussions focus on the Java specification for simplicity.)

MASS Class

- ***public init(String[] args, int nProc, int nThr)*** uses *nProc* processes, each with *nThr* threads if specified.
- ***finish()*** finishes computation.

Places Class

- ***public Places(int handle, String className, Object argument, int boundary, int size...)*** instantiates a shared array with *size* from *className* as passing an *argument* to the *className* constructor. The array has a shadow space with *boundary* and receives a user-given *handle*.
- ***public Object[] callAll(int functionId, Object[] arguments)*** calls the method specified with *functionId* 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. We also have the *callSome()* method to call a given method of one or more selected array elements.
- ***public void exchangeAll(int handle, int functionId, 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[]*. The *exchangeSome()* method is a variation to call from one or more selected array elements to their destinations. Another variation is *exchangeBoundary()* to exchange node-boundary elements with neighboring nodes as a ghost space.

Place Class

- ***private size[]; private index[]*** maintains the size of the shared array that each element belongs to and the index of each array element.
- ***public Object callMethod(int functionId, Object argument)*** is invoked from *Places.callAll/Some()* and *exchangeAll/Some()* so as to call a function specified with *functionId*.

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 in the *Agent* class.
- ***public Object callAll(int functionId, Object[] arguments)*** is the same as *Places.callAll()*.
- ***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()*.

Agent Class

- ***migrate(int[] index...)*** allows a calling *Agent* to migrate to the *Place* specified with *index* upon *Agents.manageAll()*. The *Agent* can also propagate itself to all neighboring or multiple *Place* elements with no or multiple *index* arguments.
- ***spawn(int nChildren, Object arguments)*** spawns children as passing *arguments* to them.
- ***kill()*** terminates a calling *Agent*.
- ***public Object callMethod(int functionId, Object argument)*** is the same as *Place.callMethod()*.

It is natural for *callAll/Some()* and *exchangeAll/Some()* to call each element's function with a user-given string-class name. However, Java reflection is intolerably slow for parallel computing, and C/C++ dynamic loading does not resolve a method within a given object. Thus, a selection of methods to call should preferably be done with *switch()*, where we need to identify each method with an integer value. For this reason, a user must provide *callMethod()* 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: all methods are identified by an integer
3     public static final int init_         = 0;
4     public static final int computeWave_  = 1;
5     public static final int exchangeWave_ = 2;
6     // automatically called from callAll/Some and exchangeAll/Some
7     public static Object callMethod( int funcId,
8                                      Object args ) {
9       switch( funcId ) {
10      case init_:         return init( args );
11      case computeWave_:  return computeWave( args );
12      case exchangeWave_: return exchangeWave( args );
13      }
14    public Object init( Object args ) {
15      ...;
16    }
17 }
```

Fig. 2. Function calls from callAll/Some() and exchangeAll/Some().

C. Coding Examples

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

1) Wave2D: Figure 3 shows an example use of *Places.callAll()* and *Places.exchangeAll()* for developing a parallel spatial simulation program 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 9 by simply creating a new instance of *Places*. The application enters a cyclic simulation to calculate the wave height at every *Place* element in parallel (line 24), and to exchange wave height information (line 26) between each element and all its neighbors (as defined on lines 16-19).

```
1  import MASS.*;// Library for Multi-Agent Spatial Simulation
2  public class Wave2D extends Place {
3     public static void main( String[] args ) {
4        // validate the arguments
5        int size = Integer.parseInt( args[0] );
6        int maxTime = Integer.parseInt( args[1] );
7        MASS.init( args );// start MASS
8        // create a Wave2D array.
9        Places wave2D = new Places( 1, "Wave2D", null,
10                                   size, size );
11       // initialize the Simulation space.
12       wave2D.callAll( init_, null );
13
14       // define the four neighbors of each cell
15       Vector<int[]> neighbors = new Vector<int[]>( );
16       int[] north = { 0, -1 }; neighbors.add( north );
17       int[] east  = { 1,  0 }; neighbors.add( east );
18       int[] south = { 0,  1 }; neighbors.add( south );
19       int[] west  = { -1, 0 }; neighbors.add( west );
20
21       // now go into a cyclic simulation
22       for ( int time = 0; time < maxTime; time++ ) {
23          // calculate each place's wave height
24          wave2D.callAll( computeWave_ );
25          // update each place with neighbor's wave information.
26          wave2D.exchangeAll( exchangeWave_, neighbors );
27       }
28       MASS.finish( );// finish MASS
29    }
30 }
```

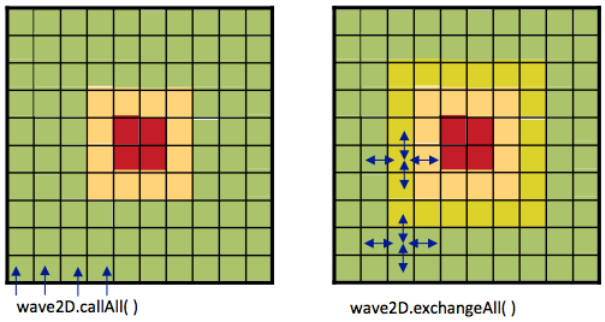

Fig. 3. Wave2D code and simulation space.

```
1  import MASS.*;// Library for Multi-Agent Spatial Simulation
2  public class RandomWalk {
3     public static void main( String[] args ) {
4        // validate the arguments
5        int size = Integer.parseInt( args[0] );
6        int nNomads = Integer.parseInt( args[1] );
7        int maxTime = Integer.parseInt( args[2] );
8        MASS.init( args );start MASS
9        // create a Land array.
10       Places land  = new Places( 1, "Land", null,
11                                  size, size );
12       Agents nomad = new Agents( 2, "Nomad", null,
13                                  land, nNomads );
14       // define the four neighbors of each cell
15       ...;
16       // now go into a cyclic simulation
17       for ( int time = 0; time < maxTime; time++ ) {
18          // decide the next destination
19          nomad.callAll( computePosition_ );
20          // move agents
21          nomad.manageAll( );
22       }
23       MASS.finish( );// finish MASS
24    }
25 }
26 public class Nomad extends Agent {// the Nomad agent
27    Random rand = new Random( );
28    Object computePosition( Object arg ) {
29       // decide a neighboring place to visit
30       int[] dest = new int[index.length];
31       dest[0] = index[0] + rand( 2 );//from current x to new x
32       dest[1] = index[1] + rand( 2 );//from current y to new y
33       migrate( dest );// actual migration invoked by nomad.manageAll
34 } }
```

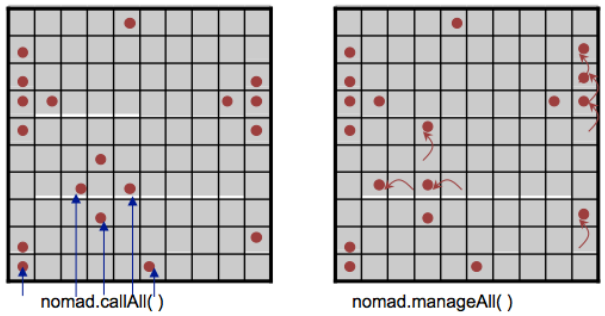

Fig. 4. Random walk code and simulation space.

2) RandomWalk: Figure 4 shows an example of multi-agent migration over a two-dimensional space with *Agent.callAll()* and *manageAll()*. This example instantiates a *Land* array with $args[0] \times args[0]$ (line 10), and populates *Nomad* agents on it (line 12). In each simulation cycle (lines 17-22), agents compute their next position to visit (line 19) and migrate there (line 21). Defined in lines 26-34, the *Nomad* agent randomly chooses one of the four neighboring places in *computePosition()* and schedules its migration on line 33.

IV. Implementation

Both Java and C++ versions of the MASS library is based on the master-slave architecture where the master process spawns slaves, each running at a different computing node until the end of the *main()* function and executing a given method of all places and agents with multi-threads in parallel. Therefore, the master process does not become a system bottleneck.

A. MASS Implementation

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*, launches a slave process on each remote host through JSCH in Java or libssh2 in C++. After all the remote processes have been launched, each process (including the master process that runs a user's *main()* function) creates a pool of worker threads. *Finish()* sends a termination message to all slave processes that terminate their worker threads and close all TCP connections for a graceful exit.

B. Places Implementation

For each of the following *Places* functions, the master process sends the corresponding command to all slaves, each waiting on a different remote node.

Places constructor: sends a different sub-range of a new shared array to each slave. All master and slave nodes then create their own array partition with the given range.

callAll/Some(): sends a function identifier, place element indices, and arguments to each slave process. All master and slave nodes then invoke the given method of the specified place elements. The return value of this method may be sent back to the master node after all executions have been completed.

exchangeAll/Some(): sends each slave a function identifier and a collection of destinations to invoke the function. All master and slave nodes execute local exchanges first. If a destination is outside of the local node, this exchange request is wrapped in a message and queued up on a hash map with the host name to which this message must be delivered. At the end of the local data exchange, each slave sends a different hash map to the corresponding remote slave that handles these remote exchange requests.

C. Agents Implementation

In the same as the *Places* implementation, the master process sends all slaves a command to invoke *callAll()* and *manageAll()*.

callAll(): is similar to *Places.callAll()* in a way that a slave at each remote node receives a function identifier and arguments from the master node, and invokes the given method of the specified *Agents*.

manageAll(): is invoked to complete *Agent*'s actions such as *migrate()*, *spawn()*, and *kill()* in a batch. At each of the master and slave processes, the worker threads pick up one after another agent from the *Agents*' pool to complete its action.

- *migrate():* given a set of coordinates, the agent will migrate to the destination upon the next manageAll() call. This method transmits agent states for re-instantiation to the destination host. As opposed to *Places.exchangeAll()* where communications are two-way, *agent.migrate()* moves agents from their local host to destination host but not the other way around. If there are no agents to migrate, each slave node 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():* dynamically creates any number of duplicates of the calling *Agent* at the current *Place* element.
- *kill():* simply sets a calling *Agent*'s *alive* flag to false, so that the next *manageAll()* call de-allocates all the agents with the false *alive* flag.

V. Programmability Analysis

This section analyzes the programmability of the MASS library in both spatial and multi-agent simulations.

A. Spatial Simulation

In addition to Wave2D shown in Figure 3, we consider a more practical spatial simulation. BrainGrid is a neural network simulator that imitates the growth of electrical activation and synapses among neurons that are placed in a given square space [15]. The simulation ultimately examines the neural spike and radius history of many different layouts, each mixing endogenously active, inhibitory, and neutral neurons, A simulation run proceeds as a sequence of 100 second segments (epoch), during each of which synaptic strengths are computed for all pairs of neurons based on an overlap of their round-shaped connectivity regions, and the status of neurons and synapses is updated. At the end of each epoch, each neuron's average firing rate is calculated based on its spike history and used to adjust its neurite outgrowth in 300-600 epochs long (i.e., 30,000-60,000 seconds). A single-threaded 60,000-second simulation of 100×100 network takes 2,000 hours (83 days). BrainGrid can be parallelized with only MASS places as shown below to distribute a network of $N \times N$ neurons over multiple computing nodes. Each "neuron" place maintains synapses dynamically emanating from it, using a vector of neighbors to be passed to *exchangeAll()* at run time. The following MASS code abstracts synapse inputs converged onto adjacent neurons.

```
1 Places neurons = new Places(1, "Neuron", arguments, N,N);
2 Vector<int[]> neighbors = new Vector<[]>( );
3 int[] synapse = { ... }; neighbors.add( synapse );
4 neurons.exchangeAll( 1, Neuron.SynapseInput, neighbors );
```

Based on the code inspection of both Wave2D and BrainGrid, the MASS library has the following four merits: (1) *machine-unaware data distribution and collection* to map the Wave2D and BrainGrid spaces over cluster nodes automatically, (2) *no explicit ghost space management* to allow each Wave2D element to receive its neighbors' information through exchangeAll(), (3) *machine-unaware logical network creation* to support the dynamic growth of BrainGrid's synapses with the "neighbors" parameter given to exchangeAll(), and (4) *no explicit for-loop parallelization* to invoke a given function of all array elements in parallel.

B. Multi-Agent Simulation

In addition to RandomWalk described in Figure 4, we examine FluTE, a stochastic influenza epidemic simulation model [2] as a publicly available multi-agent simulation. It synthesizes populations, each corresponding to the year-2000 Census tract that is evenly subdivided into communities of 500-3000 individuals. One to seven individuals form a household where influenza is transmitted most often. Individuals are also categorized into preschool-age/school-age children and non-working/working adults. Workers are divided into non-migrant and migrant workers, the latter of whom travel across communities. An unmitigated epidemic simulation with 10 million people requires 800MB memory and takes about 2 hours [2]. The following shows a MASS-parallelized programming framework of FluTE: a *Places* forms a collection of communities (line 1); an *Agents* populates people over the communities (line 2); and thereafter MASS starts a cycle simulation (lines 5-11). In each cycle, the agents calculate their daytime susceptibility and infection in *day()* (line 6), migrate to a different community in *manageAll()* (lines 7 and 9), and transmit infection among their family in *night()* (line 8). The places respond to epidemic such as school closure and vaccination in *response()* (line 10).

```
1  Places comm = new Places(1, "Community", size);
2  Agents people=new Agents(2, "Person", comm, population);
3  Vector<int[]> neighbors = new Vector<int[]>( );
4  int[] traffic = { ... }; neighbors.add( traffic );
5  for ( int time = 0; time < maxTime; time++ ) {
6    people.callAll( Person.day );
7    people.manageAll( );
8    people.callAll( Person.night );
9    people.manageAll( );
10   comm.callAll( Community.response );
11   comm.exchangeAll( 1, Community.exchange, neighbors );
12 }
```

For multi-agent simulations such as RandomWalk and FluTE, the MASS library facilitates the following features:

(1) *automated agent migration* to packetize and transfer agents with *Agnets.manageAll()*, (2) *guarded agent migration* to avoid agent collision on the same place in RandomWalk or community in FluTE with *Agent.migrate()* that receives a list of multiple destinations and choose an available place, and (3) *agent propagation and distributed termination* to duplicate an agent, propagate its copies to all the neighboring array elements, and clean up all terminated agents with just one *Agents.manageAll()*.

C. Summary

The above code analysis demonstrates that the MASS library successfully addresses the four problems we listed in Section III: (1) *individual-based design*: facilitating bottom-up agent and space design; (2) *machine unawareness*: relieving developers from parallelization; (3) *ghost space management*: reading and updating adjacent boundaries of remote spaces; and (4) *system-level agent management*: allowing application developers to focus on designing agent behavior. Furthermore, an additional advantage of the MASS library 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 application developers to focus on each model design.

VI. Execution Performance

At present we completed the MASS library in Java and verified its functionalities through a pilot use in our regular course at University of Washington Bothell [16] where students developed simple applications including Wave2D, RandomWalk, and Conway's game of life. Since we are currently porting MASS to C++ platforms, the following performance evaluation was conducted with the Java version in terms of the following five test items: (1) identification of the minimum problem size necessary for parallelization, (2) the minimum place computation granularity, (3) the minimum agent group size, (4) spatial simulation performance using Wave2D and Conway's game of life, and (5) multi-agent simulation performance using RandomWalk. Unless otherwise specified, the following evaluation was conducted on a Gigabit-Ethernet cluster of nodes, each equipped with dual 3.2 Xeon GHz Intel processor with 1 GB memory. The master node has an Intel Xeon E5520 processor with 6 GB of memory.

A. Problem Size

Figure 5 determines the minimum MASS simulation space that can benefit from multithreaded multi-process simulation on top of a cluster system. This test was conducted for 1000 iterations while the simulation size increases from 100^2 to 500^2. Each iteration contains a single *callAll()* and *exchangeAll()*, each computing a floating-point multiplication. For mimicking many applications that check on-going computation, the test collected intermittent computation at the *main()* program every 10 iterations. The results showed that a 500×500 or larger space was necessary to compensate repetitive data collection overheads.

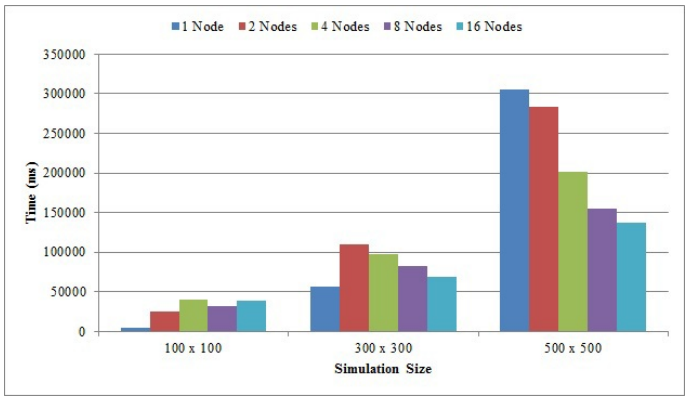

Fig. 5. Problem size for multi-threaded and multi-process computation

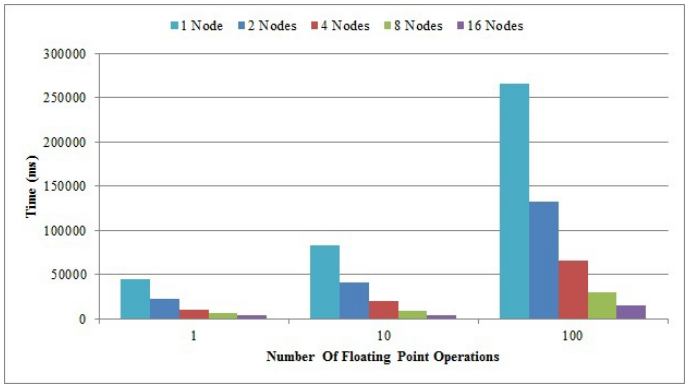

Fig. 6. Computation granularity of callAll without return values.

B. Place Granularity

Based on the previous test, we used a 500×500 grid for investigating the minimum computation granularity of each *Place* to obtain scalable performance. We observed *callAll()* and *exchangeAll()* separately. Figure 6 shows the execution performance of *callAll()*. The computation increased from 1 to 100 floating-point multiplications in *callAll()*. The result demonstrated typical embarrassingly-parallel performance.

Figure 7 shows the *callAll()* performance where the intermittent results are collected by the *main()* program every 10 iterations. We increased *callAll()* computation granularity from 1 to 500 floating-point operations. Obviously, due to repetitive data collection by the *main()* program, we need 100 computation in each *callAll()* when returning intermittent results to the master node.

The *exchangeAll()* performance was evaluated using the same criteria as *callAll()*. Figure 8 shows that performance increase was not as observable as the increase gained from *callAll()* due to the fact that each *exchangeAll()* involves communication with the nearest neighbors. However, the result indicates that *exchangeAll()*'s minimum computation granularity is 100 floating-point multiplications as well.

C. Agent Group Size

We examined to see how many agents are necessary for parallel simulation. The test included 1000 iterations on a 500×500 grid with agents evenly distributed. Each iteration

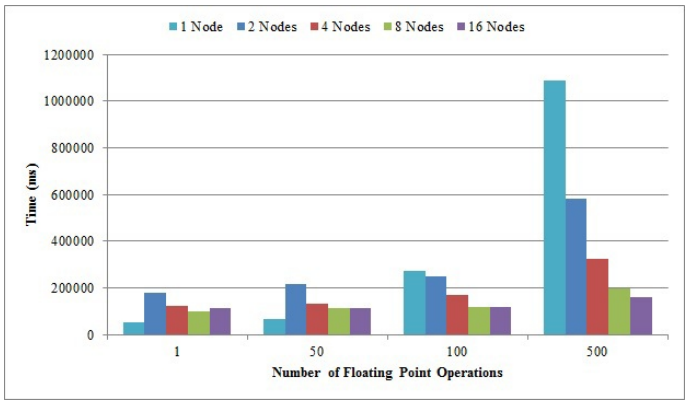

Fig. 7. Computation granularity of callAll with return values.

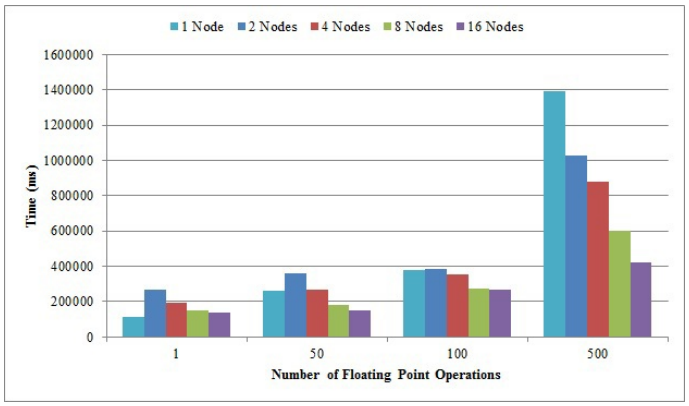

Fig. 8. Computation granularity of exchangeAll with return values

contains an *Agents.callAll()* with only one floating-point multiplication and an *Agent.migrate()* call. Intermittent results were collected at the master node once every 10 iterations. Figure 9 shows that the MASS library needs $300,000$ fine-grained agents, (i.e., 1.2 agents per place) to yield scalable computation with up to 16 cluster nodes. This means that *Agent.migrate()* is much lighter than *Places.exchangeAl()* that requires 100 floating-point computation per place. This is because agent migration involves only one-way communication from a source to a destination.

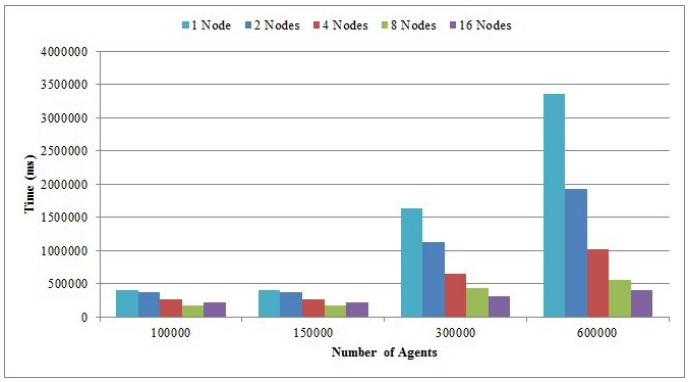

Fig. 9. Agent group size.

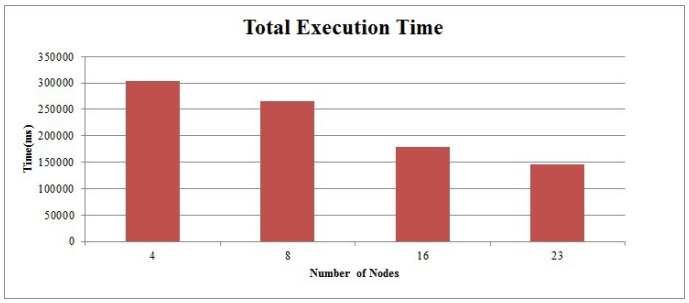

Fig. 10. Total execution time of Wave2D.

Procs \ Threads	#	1	2	3	4
	1	464.2	432.1	442.1	436.7
	2	74.86	55.85	56.60	56.42
	3	63.04	47.19	45.86	44.16
	4	64.82	43.94	43.47	40.92

Fig. 11. Performance of Conway's game of life

D. Performance of Spatial Simulation

The first application is Wave2D. The test was conducted with a 1500×1500 grid for 1000 iterations, (each containing *callAll()* and *exchangeAll()* as shown in Figure 3). The master node collected intermittent results of the simulation every 10 iterations. Figure 10 demonstrates the total execution performance of Wave2D with 4 to 23 cluster nodes. In this application, the benefit of distributed memory became apparent as a 1500×1500 simulation size requires a huge memory space. Although the performance scaling is slower than ideal, the performance gain with additional nodes is still observed. This demonstrates the competitive performance of a typical application where the user demands near real-time update on the simulation.

The second application is Conway's game of life (simplified as Life in the discussions below) that creates a 2D grid of square cells, (i.e., places in MASS), each alive or dead based on its neighbors: live cells under 2 live neighbors die, live cells with 2 or 3 live neighbors live on, live cells over 3 live neighbors die, and dead cells with 3 live neighbors come alive. The code structure is the same as Wave2D in containing *callAll()* and *exchangeAll()*. We used four computing nodes, each with two 1.8GHz dual-core AMD Opterons and 1MB memory to simulate 100 generations of Life over a 600×600 grid. Despite that Life's computation granularity was much smaller than Wave2D, Figure 11 shows that its execution performance was scalable with up to 4 nodes with 4 cores, (i.e., 16-way parallelization).

E. Performance of Multi-Agent Simulation

We used RandomWalk (see Figure 4) for evaluating the execution performance of multi-agent simulation. The test was conducted by iterating migration of $300,000$ agents over a 500×500 grid for 1000 simulation cycles to identify scalability requirements. Figure 12 demonstrates the performance with

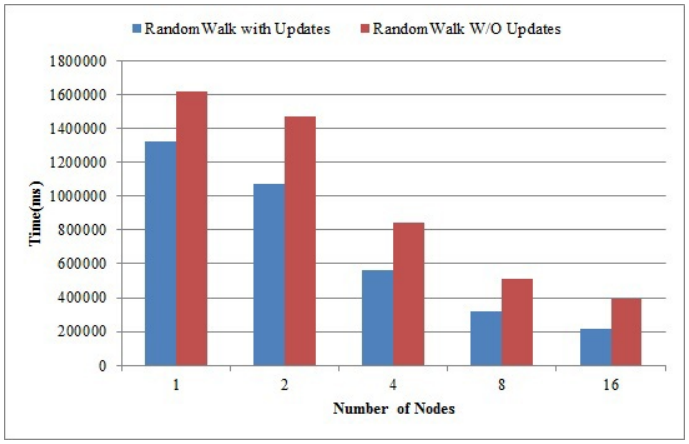

Fig. 12. Execution performance of RandomWalk.

1 to 16 cluster nodes. The blue bars (i.e., bars on the left) show the execution performance of RandomWalk where each iteration executes *Agents.callAll()*, *Agents.manageAll(),* and *Places.callAll()* as collecting results every 10 iterations. On the other hand, the red bars, (i.e., bars on the right) show the performance when we added *Places.exchangeAll()* to each iteration for updating neighboring place information, which thus incurs additional communication overheads. The results demonstrate that even with frequent information exchange among cluster nodes, the performance increase was noticeable when more computing nodes were added to the simulation, and in all cases, multiple computing nodes outperformed single computing-node performance even with such fine grained computation.

F. Summary and Comparison with Other Systems

Based on all the performance results, the minimum conditions to benefit from automatic parallel execution using the MASS library are: (1) 500×500 places, (2) $300,000$ agents, and (3) 100 floating-point operations required per *exchangeAlll()* or *callAll()*, both returning values to the calling place.

Once the MASS C++ version is complete, we will be able to compare it with GlobalArray [9] and Repast HPC [13], each representing a distributed array and a multi-agent simulation system. Both systems are implemented on top of MPI that is an additional communication layer and thus can be a substantial overhead. In contrast, the MASS C++ version is being implemented, directly using low-level system calls, and therefore it will be able to run with less overheads.

VII. Conclusions

In this paper, we elaborated the programming advantages in using the MASS library for spatial and multi-agent simulation: individual-based design, machine unawareness, readable/writeable ghost spaces, and system-level agent management. Our performance evaluation of the Java version of the MASS library demonstrated the CPU scalability of MASS applications with 300,000 fine-grain agents on a 500×500 grid. While the Java version (coupled with the JSCH jar file) is portable to any JVM-available machines, we understand that it does not outperform native execution. Therefore, we are currently developing two versions of the MASS library to support C++ and CUDA-based Nvidia GPUs. 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. We are also planning to port BrainGrid and FluTE to MASS for purpose of analyzing its practical execution performance.

Acknowledgment

We extend our appreciation to Mr. Daniel Lewis for his parallelization and evaluation work of Conway's game of life.

References

[1] Swarm main page, "http://www.swarm.org/index.php/swarm_main_page."

[2] FluTE, an influenza epidemic simulation model, "http://www.cs.unm.edu/∼dlchao/flute/."

[3] M. Lysenko and R. M. D'Souza, "A framwork for megascale agent based model simulations on graphics processing units," *Journal of Artificial Societies and Social Simulation*, vol. Vol.11, no. No.4 10, October 2008.

[4] B. G. Aaby, K. S. Perumalla, and S. K. Seal, "Efficient simulation of agent-based models on multi-gpu and multi-core clusters," in *Proc. of 3rd International ICST Conference on Simulation Tools and Techniques - SIMUTools 2010*. Malaga, Spain: ICST, pp. 29–38.

[5] R. M. D'Souza, S. Marino, and D. Kirschner, "Data-parallel algorithms for agent-based model simulations of tuberculosis on graphics processing units," in *Proc. of Agent-Directed Symposium - ADS09*. San Diego, CA: Publisher, March 2009.

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

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

[8] R. W. Numrich and J. Reid, "Co-array fortran for parallel programming," *ACM SIGPLAN Fortran Forum*, vol. Vol.17, no. No.2, pp. 1–31, August 1998.

[9] 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. Vol.20, no. No.2, pp. 203–231, 2006.

[10] B. Logan and G. Theodoropoulos, "The distributed simulation of multi-agent systems," *Proceedings of the IEEE*, vol. Vol.89, no. No.2, pp. 174–186, January 2001.

[11] L. Gasser and K. Kakugawa, "MACE3J: fast flexible distributed simulation of large, large-grain multi-agent systems," in *Proc. of the first international joint conference on Autonomous agents and multiagent systems - AAMAS-2002*. Bologna, Italy: ACM, November 2001.

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

[13] Repast HPC Tutorial, "http://repast.sourceforge.net/hpc_tutorial/toc.html."

[14] M. Fukuda, "MASS: Parallel-Computing Library for Multi-Agent Spatial Simulation," Distributed Systems Laboratory, Computing & Software Systems, University of Washington Bothell, Bothell, WA, http://depts.washington.edu/dslab/SensorGrid/doc/MassSpec.pdf, May 2010.

[15] F. Kawasaki, "Accelerating large-scale simulations of coortical neuronal network development," Master's thesis, Master of Science in Computing and Software Systems, University of Washington, 2012.

[16] CSS 534/490 Final Project, "http://courses.washington.edu/css534/prog /prog4.pdf."