MASS: Parallel-Computing Library for Multi-Agent Spatial Simulation

Munehiro Fukuda
May 7th, 2010

1. Introduction

This document is written to define the second draft version of the MASS library, a parallel-computing library for **m**ulti-**a**gent **s**patial **s**imulation. As envisioned from its name, the design is based on multi-agents, each behaving as a simulation entity on a given virtual space. The library is intended to parallelize a simulation program that particularly focuses on multi-entity interaction in physical, biological, social, and strategic domains. The examples include major physics problems (including molecular dynamics, Schrödinger's wave equation, and Fourier's heat equation), neural network, artificial society, and battle games.

2. Programming Model

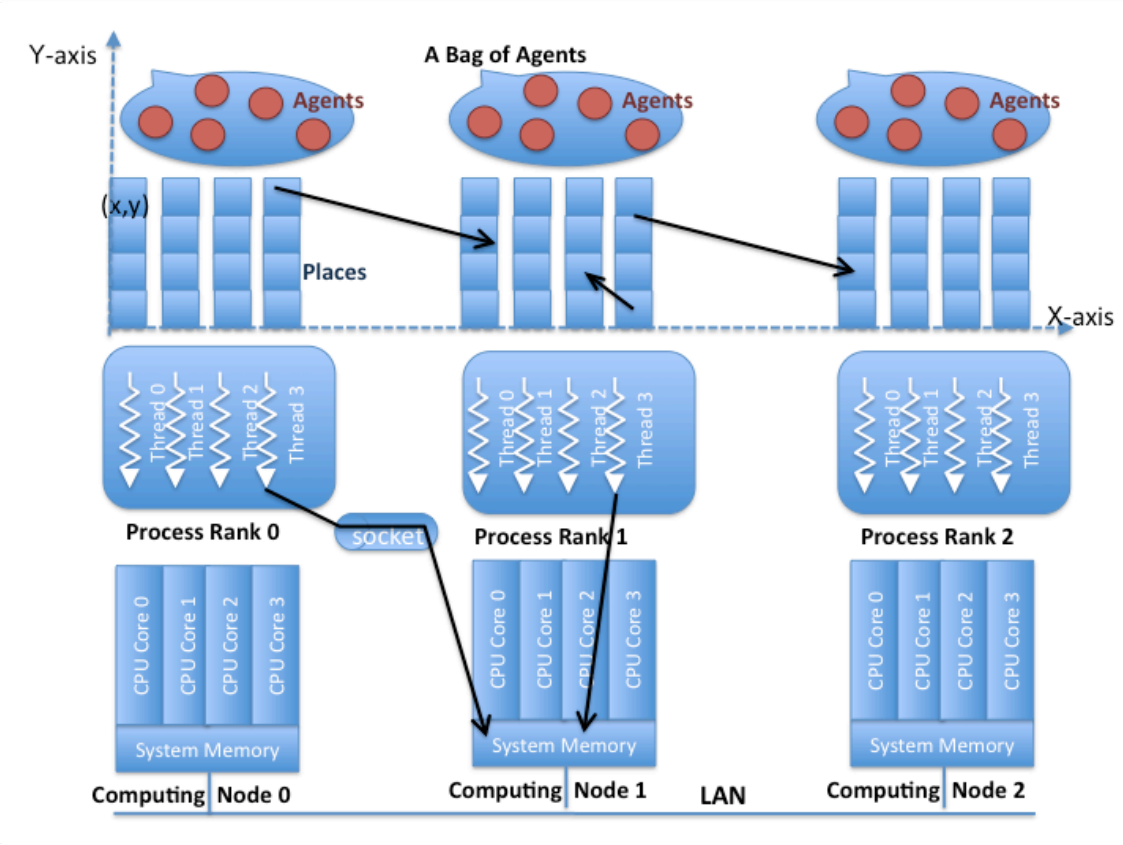
2.1. Components: Places and Agents

"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.



2.2. Programming Framework

The following code shows a Java programming framework that uses the MASS library to simulate a multi-agent spatial simulation.

```
 1:    import MASS;
 2:
 3:    class Application {
 4:
 5:        public void static main( String[] args ) {
 6:            // get the max simulation time
 7:            int maxTime = Integer.parseInt( args[0] );
 8:
 9:            // start a process at each computing node
10:            MASS.init( args );
11:
12:            // distribute places and agents over computing nodes
13:            Places territories
14:                = new Places( 1, "Territory", null, 100, 100 );
15:            Agents troops
16:                = new Agents( 2, "Troop", null, territories, 40000 );
17:
18:            // start cyclic simulation in parallel
19:            int time = 0;
21:            int[] destination = new int[2]; dest[0] = 0; dest[1] = 1;
22:            for ( ; time < MaxTime - 10; time++ ) {
23:                Object arg = ( Object )( new Integer( time ) );
24:                territories.callAll( Territory.compute, arg );
25:                territories.exchangeAll( Territory.exchange, dest );
26:                troops.callAll( Troop.compute, arg );
```

```
27:                troops.exchangAll( 2, Troop.exchange );
28:                troops.manageAll( );
29:            }
30:
31:            // terminate the processes
32:            MASS.finalize( );
33:        }
34:    }
```

The behavior of the above code is as follows: it synchronizes all processes with MASS.init( ) and has them spawn multiple threads (line 11). The code thereafter maps a matrix of 100 × 100 "Territory" places as well as 4000 "Troop" agents over these processes (lines 13 – 16). Each process falls into a cyclic simulation (lines 22 – 29) where all its threads repeat calling the following four functions in a parallel fashion:

- compute( ) of the "Territory" places to update each place object's status
- exchange( ) of the "Territory" places to exchange data among place objects
- compute( ) of the "Troop" agents to update each agent's satus
- exchange( ) of the "Troop" agents to exchange data among agents

as well as control the "Troop" agents in manageAll( ) so as to move, spawn, terminate, suspend, and resume agents. At the end, all the processes get synchronized together for their termination (line 32).

In the following sections, we will define the specification of "MASS", "Places", "Place", "Agents", and "Agent"

3. MASS

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.

public static void	**init(String[] args, int nProc, int nThr)** Involves nProc processes in the same computation and has each process spawn nThr threads.
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.
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.

4. 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 (see 4.2 Place Class), 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).

4.1. public class Places

The class instantiates an array shared among multiple processes. Array elements are accessed and processed by multi-processes in parallel.

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

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 i^{th} 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 i^{th} callee.

4.2. public abstract class Place

"Place" is the base class from which a user can derive his/her application-specific matrix of places. An actual matrix instance is created and maintain within a "Places" class, so that the user can obtain parallelizing benefits from Places' callAll( ) , callSome( ), exchangeAll( ), and exchangeSome( ) methods that invoke a given method of each matrix element and exchange data between each element and others.

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( ).

4.3. 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:

Example:

```
 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.     }
```

4.4. Example 1: Wave2D

The following Wave2D class is a two-dimensional matrix that simulates Schrödinger's wave diffusion. Each "Wave2D" matrix element maintains the six instance variables listed below:

wave[2]: the current wave height at each matrix element
wave[1]: the previous wave height
wave[0]: even the one more previous wave height
neighbor[0] through to neighbor[3]: the wave height of north/east/south/west neighbors
time: the current simulation time
interval: time interval to display the ongoing simulation results

In this Wave2D class, callMethod( ) maps each functionId to the corresponding function (lines 44 – 53); init( ) initializes instance variables of each place or cell (lines 63 – 72); computeWave( ) simulates wave diffusion per time unit (lines 78 – 124); exchangeWave( ) receives wave heights from all the four neighbors (lines 130 – 132); collectWave( ) collects wave heights from all places into place[0][0] that then displays them in graphics (lines 138 – 140); and place[0][0] is in charge of calling startGraphics( ), writeToGraphics( ), and finishGraphics( ) to handle 2D graphics (lines 143 – 244).

```
 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.   }
```

5. 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.

5.1 public class Agents

Once the main method has called MASS.init( ), it can create as many agents as needed, using the Agents( ) constructor. Unless a user supplies an explicit mapping method in his/her "Agent" definition (see 5.2 public abstract class Agent), "Agents" distribute instances of a given "Agent" class (simplified as agents in the following discussion) uniformly over different computing nodes.

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

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 i^{th} callee.

The following example of code fragment creates an args[0] x args[0] matrix over multiple processes (line 11), distributes a "RandomWalk" agent every four places of this matrix (lines 12 – 13), and simulates random walk of these agents over the matrix using many threads (lines 16 – 19).

```
 1: import MPI;
 2: import MASS;
 3:
 4: class RandomWalkDriver {
 5:     public void static main( String[] args ) {
 6:         MASS.iinit( args );
 7:
 8:         int size = Integer.parseInt( args[0] );
 9:         int maxTime = Integer.parseInt( args[1] );
10:
11:         Places matrix = new Place( 1, "Matrix", null, size, size );
12:         Agents walkers = new Agents( 2, "RandomWalk", null, matrix,
13:                                      size*size / 4 );
14:
```

```
16:         for ( int time = 0; time < maxTime; time++ ) {
17:             walkers.callAll( RandomWalk.newLocation );
18:             walkers.manageAll( );
19:         }
20:         MASS.finalize( );
21:     }
22: }
```

5.2 public abstract class Agent

"Agent" is the base class from which a user can derive his/her application-specific agent that migrates to another place, forks their copies, suspends/resumes their activity, and terminate themselves.

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-

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( ).

5.3. Example 2 RandomWalk

The following RandomWalk simulates the walk of agents, each searching for and moving to the least crowded place over a two-dimensional matrix. Starting from RandomWalk.main( ), the program instantiates a two dimensional "Land" array, and populates agents over a small square in the middle of this land (lines 11 – 29). Thereafter, the main( ) function initializes the relative X and Y coordinates of each cell's four neighbors (lines 31 – 37), and finally goes into a cyclic simulation (lines 38 – 47) where each cell exchanges #agents with its four neighbors and each agent migrates to a neighbor with the least agents. This cyclic simulation is performed in parallel with multiple processes, each with multiple threads. The class "Land" defines this two-dimensional simulation space (lines 54 – 98) and the class "Nomad" describes each agent's behavior (lines 99 – 168).

```
 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.   }
```

6. Implementation Plan

Although we used Java to draft the specification of the MASS library and its example code, the actual library implementation can be carried out in not only Java but also C/C++ so as to cover various applications domains.

The following NSF funding programs are possible options to ask support for this library implementation and extension to any existing and new projects such as UW Bothell's AgentTeamwork and UWB-Shizuoka Sensor-Grid.

1. Computer and Network Systems (CNS) Core Programs
   Deadline: mid December annually

2. Computing and Communication Foundations (CCF) Core Programs
   Deadline: mid December annually

7. Credits

The very first draft specification was written in support from PDM&FC, Lisbon, Portugal during my sabbatical in 2008 – 2009 and may be used for their grant proposal submitted European Commission.