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

Munehiro Fukuda
Edited by Jennifer Kowalsky
February 24th, 2015

1. Introduction

This document is written to define our on-going C++ 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. For example, simulations could 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

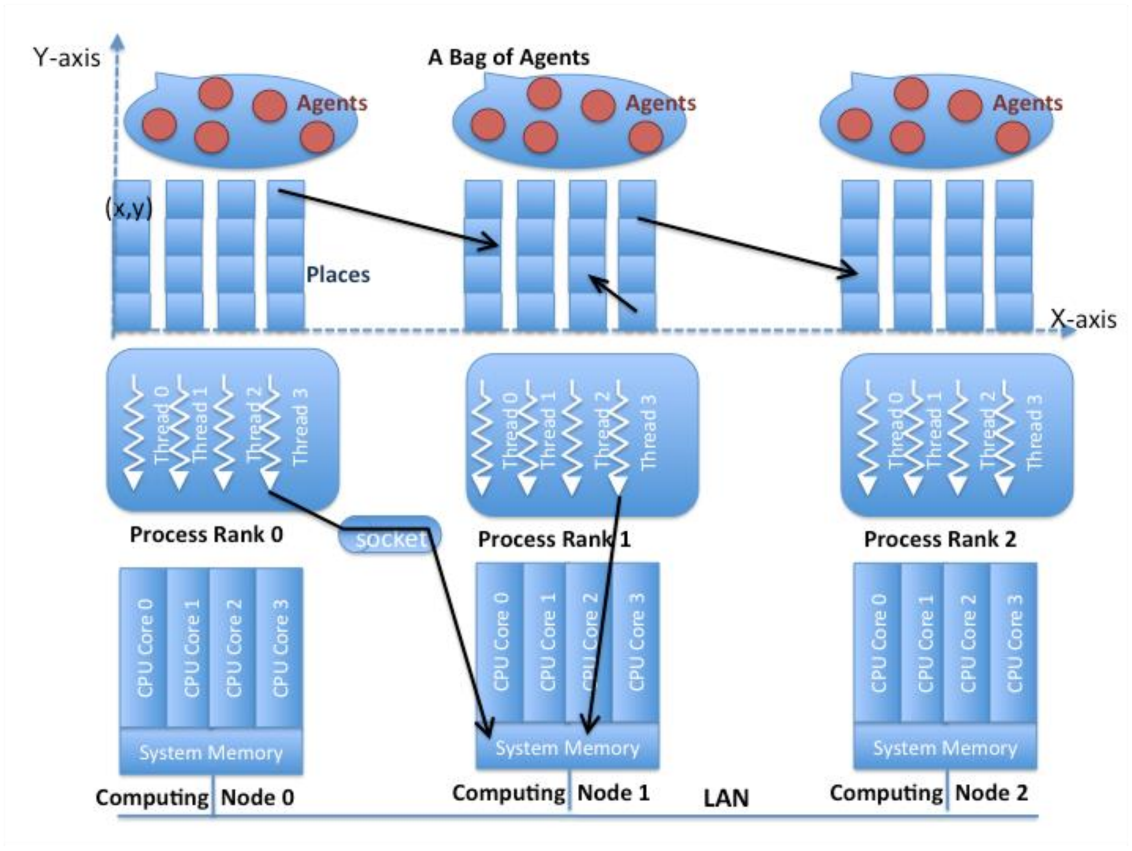
2.1. Components: Places and Agents

"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" are 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. 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 C++ programming framework that uses the MASS library to simulate a multi-agent spatial simulation.

Example 1:

```
#include "MASS.h"
#include "Territory.h"
#include "Troop.h"
#include <vector>
#define MSG "argument\0"

int main( int argc, char *args[] ) {
    char *arguments[4];
    arguments[0] = args[1]; // username
    arguments[1] = args[2]; // password
    arguments[2] = args[3]; // machinefile
    arguments[3] = args[4]; // port
    int nProc = atoi( args[5] ); // # processes
    int nThr = atoi( args[6] ); // # threads

    // start a process at each computing node
    MASS::init( args, nProc, nThr );

    // distribute places and agents over computing nodes
    char *msg = MSG;
    Places *territories
        = new Places( 1, "Territory", msg, sizeof( MSG ), 2, 100, 100 );
23:         Agents *troops
24:             = new Agents( 2, "Troop", msg, sizeof( MSG ), territories, 4000 );
25:         // start cyclic simulation in parallel
26:         vector<int*> destinations;
27:         int north[2] = {0, 1};  destinations.push_back( north );
28:         int east[2]  = {1, 0};  destinations.push_back( east );
29:         int south[2] = {0, -1}; destinations.push_back( south );
30:         int west[2]  = {-1, 0}; destinations.push_back( west );
31:         for ( int time = 0; time < MaxTime - 10; time++ ) {
32:             territories->callAll(Territory::compute_, (void *)&time, sizeof(time));
33:             territories->exchangeAll( Territory::exchange_, &destinations );
34:             troops->callAll( Troop::compute_, (void *)&time, sizeof(time) );
35:             troops->manageAll( );
36:         }
37:
38:         // terminate the processes
39:         MASS::finish( );
40:     }
```

The behavior of the above code is as follows: it synchronizes all processes with MASS::init( ) and has them spawn multiple threads (line 17). The code thereafter maps a matrix of 100 × 100 "Territory" places as well as 4000 "Troop" agents over these processes (lines 19 – 24). Each process falls into a cyclic simulation (lines 31 – 36) 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 status

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 are synchronized together for their termination (line 39).

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::finish( ) 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::finish( ) 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 )`** Is not implemented yet. It 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`	**`finish( )`** Finishes computation.

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, void *argument, int argument_size, int dimension, int size[] )`** Instantiates a shared array with "size[]" from the "className" class as passing an argument to the "className" constructor. This array is associated with a user-given handle that must be unique over machines.
`public`	**`Places( int handle, string className, void *argument, int argument_size, int dimension, ... )`** Is the same as the 1st constructor except dimensions are numerated in the "…" format.
`public`	**`Places( int handle, string className, int boundary_width, void *argument, int argument_size, int dimension, int size[] )`** Instantiates a shared array with "size[]" from the "className" class as passing an argument to the "className" constructor. This array is associated with a user-given handle that must be unique over machines. This constructor also allocates the left and right shadow places whose with is given by boundary_width. These left and right shadows are a copy of the right boundary places of the left neighbor and a copy of the left boundary places of the right neighbor.
`public`	**`Places( int handle, string className, int boundary_width, void *argument, int argument_size, int dimension, ... )`** Is the same as the 1st constructor except dimensions are numerated in the "…" format. This constructor also allocates the left and right shadow places whose with is given by boundary_width. These left and right shadows are a copy of the right boundary places of the left neighbor and a copy of the left boundary places of the right neighbor.
`public int`	**`getDimension( )`** Not yet implemented It returns the dimension of this multi-dimensional array.
`public int*`	**`size( )`** Not yet implemented 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, void *argument, int argument_size)** Calls the method specified with functionId of all array elements as passing an argument to the method. Done in parallel among multi-processes/threads.
public void*	**callAll(int functionId, void *arguments[], int argument_size, int return_size)** 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 (void *)[i] whose element's size is return_size. 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 dim, int index[])** Is not implemented yet. 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, void *argument, int argument_size, int dim, int index[])** Is not implemented yet. Calls the method specified with functionId of one or more selected array elements as passing an argument to the method. The format of index[] is the same as the above callSome( ). Done in parallel among multi-processes/threads.
public void*	**callSome(int functionId, void *arguments[], int argument_size, int dim, int index[])** Is not implemented yet. Calls the method specified with functionId of one or more selected array elements as passing arguments[i] to element[i]'s method, and receives a return value from it into (void *)[i] whose element's size is return_size. 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 is a continuous set of arguments passed to the callee's method. The caller's inMessages[] 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 dim, int index[] )`** Is not implemented yet. It 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 is a contiguous set of arguments passed to the callee's method. The caller's inMessages[] stores values returned from all callees. More specifically, inMessages[i] maintains a set of return values from the ith callee.
`public void`	**`exchangeBoundary( )`** Exchanges the boundary places with the left and right neighboring nodes. The remote boundary places are stored in the left and right shadow spaces.

4.2. public class Place

"Place" is the abstract 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( void *args )`** Is the default constructor. A contiguous space of arguments is passed to the constructor.
`public vector<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 vector<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<MObject*>`	**`agents`** Includes all the agents residing locally on this place.
`public virtual void*`	**`callMethod( int functionId, void *arguments )`** Is called from Places.callAll( ), callSome( ), exchangeAll( ), and exchangeSome( ), and invoke the function specified with functionId as passing arguments to this function. A user-derived Place class must implement this method.
`public void*`	**`outMessage`** 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. The argument size must be specified with outMessage_size.

protected int	**outMessage_size** Defines the size of outMessage.
public vector<void*>	**inMessages** Receives a return value in inMessages[i] from a function call made to the i-th remote cell through exchangeAll( ) and exchangeSome( ). Each element size must be specified with inMessage_size.
public int	**inMessage_size** Defines the size of inMessage.
protected void	***getOutMessage(int handle, int index[])** Returns a pointer to the outMessage of a remote place specified with handle and index[]. The values of index[] must be relative index from the current place.

4.3. A Framework of Application-Specific Place-Derived Class

An application-specific "Place"-derived class, (thus whose objects are instantiated upon a Places instantiation), should have the following programming framework as shown in example 2. First of all, it must include "Place.h" and inherits the Place class (lines 5 and 7). The constructor must be defined to receive a void pointer as its argument (line 13). The place-derived class must then implement callMethod( ) that receives an int-type functionId to invoke the corresponding method and to pass a void pointer to it as its argument (lines 19 – 26). The actual functions invoked from callMethod( ) and should be implemented as private method members (lines 30 – 32). Since this application-specific "Place"-derived class is internally dynamic-linked to the MASS library, using dlopen( ) and dlsym( ) that understand C programs, it must define instantiate( ) and destroy( ) for object creation and deletion (lines 38-44).

Example 2:

```
 1.  #ifndef DERIVEDPLACE_H
 2.  #define DERIVEDPLACE_H
 3.
 4.  #include <iostream>
 5.  #include "Place.h"
 6.
 7.  class DerivedPlace : public Place {
 8.  public:
 9.    // 0: FUNCTION ID
10.    static const int function_ = 0;
11.
12.    // 1: CONSTRUCTOR DESIGN
13.    DerivedPlace( void *argument ) : Place( argument ) {
14.      // START OF USER IMPLEMENTATION
15.      // END OF USER IMPLEMENTATION
16.    }
17.
18.    // 2: CALLALL DESIGN
19.    virtual void *callmethod( int functionId, void *argument ) {
20.      switch( functionId ) {
21.        // START OF USER IMPLEMENTATION
22.      case function_: return function( argument );
23.        // END OF USER IMPLEMNTATION
24.      }
25.      return NULL;
26.    };
private:
  // 3: EACH FUNCTION DESIGN
  // START OF USER IMPLEMENTATION
  void *function( void *argument ) {
    return NULL;
  }
  // END OF USER IMPLEMENTATION
};

#endif

extern "C" Place* instantiate( void *argument ) {
  return new DerivedPlace( argument );
}

extern "C" void destroy( Place *object ) {
  delete object;
}
```

Example 3 shows how to instantiate a 100 by 100 objects from the above DerivedPlace class (line 14) and to call the function( ) of each object in parallel (line 18).

Example 3:

```
 1.  #include "MASS.h"
 2.  #include "DerivedPlace.h"
 3.  #include <vector>
 4.
 5.  int main( int argc, char *argv[] ) {
 6.    int nProc = 4, nThr = 4;  // define the number of processes and threads
 7.    MASS.init( argv, nProc, nThr ); // initialize MASS with a list of args that
 8.    // includes hostname information, username, password, ect; and the number of
 9.    // threads and processes.
10.
11.    // initialize places with our derived class.
12.    // Arguments are, in order:
13.    //    handle, className, boundary_width, argument, argument_size, dim, ...
14.    Places *places = new Places( 1, "DerivedPlace", "args", 4, 2, 100, 100 );
15.
16.    // call the DerivedPlace class's implementation of callAll
17.    // with the message "message" and the length of the message.
18.    places->callAll( DerivedPlace::function_, "message", 7 );
19.
20.    // Finished with MASS.
21.    MASS.finish( ).
22.  }
```

5. Agents

"Agents" is a set of execution instances, each capable of residing on a place, migrating to another place with matrix indices, cloning, and interacting with any other agents indirectly through the currently residing place.

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 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, void *argument, int argument_size, 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( )` Not yet implemented. Returns the handle associated with this agent set.
`public int`	`nAgents( )` Returns the total number of agents over the sytem.
`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, void *argument, int argument_size)` Calls the method specified with functionId of all agents as passing a (void *)argument to the method. Done in parallel among multi-processes/threads.
`public *void`	`callAll( int functionId, void *arguments[], int argument_size, int return_size )` 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 (void *)[i] whose element's size is return_value. 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( ), and kill( ) 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.

5.2 public class Agent

"Agent" is the abstract 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( void *args )`** Is the default constructor. A contiguous space of arguments is passed to the constructor.
`protected Place*`	**`place`** Points to the current place where this agent resides.
`protected vector<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.
`protected 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.
`protected int`	**`parented`** Is the identifier of this agent's parent.
`protected int`	**`newChildren`** Is the number of new children created by this agent upon a next call to Agents.manageAll( ).
`protected vector<void*>`	**`arguments`** Is an array of arguments, each passed to a different new child.
`protected bool`	**`alive`** Is true while this agent is active. Once it is set false, this agent is killed upon a next call to Agents.manageAll( ).
`protected int`	**`agentsHandle`** Maintains this handle of the agents class to which this agent belongs.
`protected int`	**`placeHandle`** Maintains this handle of the agents class with which this agent is associated.
`public int`	**`map(int maxAgents, vector<int> size, vector<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.
`protected bool`	**`migrate( vector<int> index )`** Initiates an agent migration upon a next call to Agents.manageAll( ). More specifically, migrate( ) updates the calling agent's index[].

protected void	**spawn(int numAgents, vector<void*> arguments, int arg_size)** Spawns a "numAgents' of new agents, as passing arguments[i] (with arg_size) to the i-th new agent upon a next call to Agents.manageAll( ). More specifically, spawn( ) changes the calling agent's newChildren.
public void	**kill( )** Terminates the calling agent upon a next call to Agents.manageAll( ). More specifically, kill( ) sets the "alive" variable false.
public Object	**callMethod(int functionId, void *arguments)** Is called from Agents.callAll. It invokes the function specified with functionId as passing arguments to this function. A user-derived Agent class must implement this method.
protected void*	**migratableData** Is a pointer to a user-allocated space that will be carried with the agent when it migrates to a different space.
Protected int	**migratableDataSize** Indicates the size of the migratableData space.

5.3. A Framework of Application-Specific Agent-Derived Class

An application-specific "Agent"-derived class, (thus whose objects are instantiated upon a Agents instantiation), should have the following programming framework as shown in example 4. First of all, it must include "Agent.h" and inherits the Agent class (lines 5 and 7). The constructor must be defined to receive a void pointer as its argument (line 13). The agent-derived class must then implement callMethod( ) that receives an int-type functionId to invoke the corresponding method and to pass a void pointer to it as its argument (lines 19 – 26). The actual functions invoked from callMethod( ) and should be implemented as private method members (lines 31 – 40). They may call the "Agent" base class' migrate( ), spawn( ), and kill( ) methods to control the invoking agents (lines 34 and 38). Note that actual migration, spawning, and termination will be performed with the following Agents.manageAll( ) invocation. Similar to the "Place"-derived class definition, an "Agent"-derived class must define instantiate( ) and destroy() for object creation and deletion (lines 46 – 52).

Example 4:
```
1.   #ifndef DERIVEDAGENT_H
2.   #define DERIVEDAGENT_H
3.
4.   #include <iostream>
5.   #include "Agent.h"
6.
7.   class DerivedAgent : public Agent {
8.   public:
9.     // 0: FUNCTION ID
10.    static const int function_ = 0;
11.
12.    // 1: CONSTRUCTOR DESIGN
13.    DerivedAgent( void *argument ) : Agent( argument ) {
14.      // START OF USER IMPLEMENTATION
15.      // END OF USER IMPLEMENTATION
16.    }
17.
18.    // 2: CALLALL DESIGN
19.    void *callmethod( int functionId, void *argument ) {
20.      switch( functionId ) {
21.        // START OF USER IMPLEMENTATION
```

```
    case function_: return function( argument );
      // END OF USER IMPLEMNTATION
    }
    return NULL;
  };
private:
  // 3: EACH FUNCTION DESIGN
  // START OF USER IMPLEMENTATION
  void *function( void *argument ) {  // Here’s a sample user implementation
    vector<void*> arguments;
    arguments.push_back( "hello" );   // give the message “hello” to each agent.
    spawn( 1, arguments, 5 );  // spawn one child agent, with the message hello.
    vector<int*> destinations;
    int next[2] = { place->index[0] + 1, place->index[1] – 1 } // go NW
    destinations.push_back( next );
    migrate( );
    return NULL;
  }
  // END OF USER IMPLEMENTATION
};
#endif

extern "C" Place* instantiate( void *argument ) {
  return new DerivedAgent( argument );
}
extern "C" void destroy( Agent *object ) {
  delete object;
}
```

Example 5 shows how to uniformly distribute 4000 agents from the above DerivedAgent class over a Places array (line 14), to call the function( ) of each object (line 23), and to control these agents in parallel (line 27)

Example 5:

```
 1.  #include "MASS.h"
 2.  #include "DerivedPlace.h"
 3.  #include <vector>
 4.
 5.  int main( int argc, char *argv[] ) {
 6.    int nProc = 4, nThr = 4;  // define the number of processes and threads
 7.    MASS.init( argv, nProc, nThr ); // initialize MASS with a list of args that
 8.    // includes hostname information, username, password, ect; and the number of
 9.    // threads and processes.
10.
11.    // initialize places with our derived class.
12.    // Arguments are, in order:
13.    //    handle, className, boundary_width, argument, argument_size, dim, ...
14.    Places *places = new Places( 1, "DerivedPlace", "args", 4, 2, 100, 100 );
15.
16.    // initialize agents with our derived class.
17.    // Arguments are, in order:
18.    //    handle, className, *argument, argument_size, initPopulation
19.    Agents *agents = new Agents( 2, "DerivedAgent", "hello", 5, 4000 );
20.
21.    // Perform the Agent's callAll with the
22.    // agent implementation's function, msg, and message size.
23.    agents->callAll( DerivedAgent.function_, "message", 7 )
24.
25.    // Updates each agent’s status, based on each of its latest migrate( ),
26.    // spawn( ), and kill( ) calls.
27.    agents->manageAll( );
28.    MASS.finish( ). // we’re done
29.  }
```

6. Compilation and Execution

MASS C++ is currently available from metis.uwb.edu, the CSS Linux file server at University of Washington Bothell.

6.1 Directory Structure

The MASS directory structure is as follows:

/net/metis/home3/dslab/MASS/c++/source	MASS C++ source code
/net/metis/home3/dslab/MASS/c++/ubuntu	MASS C++ executable library for Ubuntu
/net/metis/home3/dslab/MASS/c++/ubuntu/ssh2	SSH2 library compiled for Ubuntu
/net/metis/home3/dslab/MASS/c++/redhat	MASS C++ executable library for Redhat
/net/metis/home3/dslab/MASS/c++/redhat/ssh2	SSH2 library compiled for Redhat
/net/metis/home3/dslab/MASS/c++/libssh2.tar	SSH2 source code
/net/metis/home3/dslab/MASS/c++/ubuntu/samples	MASS C++ sample test program

Note that, unless you want to install the MASS and SSH2 libraries into your own directory, you do not have to copy any files from the above directories except make a symbolic link to the mprocess daemon and the killMProcess.sh shell script (see below for the details).

6.2 Working Directory Set-Up and Compilation

(1) To develop MASS application programs, set up a working directory and create a symbolic link to the mprocess daemon and the killMProcess.sh.

```
ln –s ~dslab/MASS/c++/ubuntu/mprocess mprocess
ln –s ~dslab/MASS/c++/ubuntu/killMProcess.sh killMProcess.sh
```

or

```
ln –s ~dslab/MASS/c++/redhat/mprocess mprocess
ln –s ~dslab/MASS/c++/redhat/killMProcess.sh killMProcess.sh
```

(2) Create machinefile.txt that lists remote computing nodes you want to use:

```
uw1-320-01
uw1-320-02
uw1-320-03
uw1-320-04
```

Please do not include the local IP name. In other words, you must start your MASS application from any other machine than these four computing nodes, (e.g., uw1-320-00). This in turn means that the above example indicates that you will use five computing nodes, including your local machine.

(3) Set up the following two shell variables:

```
export MASS_DIR=/net/metis/home3/dslab/MASS/c++
export
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/net/metis/home3/dslab/MASS/c++/ubuntu/ssh2/li
b:/net/metis/home3/dslab/MASS/c++/ubuntu
```

You might want to add the above two statements in your ~/.bash_profile or include them in compile.sh and run.sh that you create for automating compilation and execution of your application programs.

(4) Compile your main program as well as all your Agents/Places-derived classes.
To compile your program that includes main( ), say main.cpp, type:
```
g++    -Wall    main.cpp    -I$MASS_DIR/source    -L$MASS_DIR/ubuntu    -lmass    -
I$MASS_DIR/ubuntu/ssh2/include -L$MASS_DIR/ubuntu/ssh2/lib -lssh2 -o main
```

To compile your Agents/Places-derived class, say Land.cpp, type:
```
g++ -Wall Land.cpp -I$MASS_DIR/source -shared -fPIC -o Land
```

Note that you must compile all your Agents/Places-derived classes whose executable is dynamic-linked to mprocess whenever your main program invokes new Places( ) or new Agents( ).

6.3 Execution of Your MASS Program

Simply type your executable file name and arguments. Please note that MASS::init( ) needs to receive three arguments, of which the first argument *char[] must include:
```
  arguments[0] // username
  arguments[1] // password
  arguments[2] // machinefile name
  arguments[3] // port
```
To use CSS Linux machines, you must specify your UNetID and its password in arguments[0] and arguments[1]. Please don't keep these pieces of information in your shell script such as run.sh or type in from your keyboard input without disabling "echo". To disable and enable "echo" of your Unix terminal, type the following commands respectively.

```
stty –echo
stty echo
```

6.4 Abnormal Termination and Clean-up

To stop an execution of your MASS program, just simply type ^c, (i.e., control c). However, please note the following MASS daemon behavior:

Once your program invokes MASS::init( ), all the remote machines you declared in machinefile.txt starts an mprocess daemon. All the daemons then dynamically link your code to it and execute MASS functions such as callAll, exchangeAll, and manageAll. Upon an invocation of MASS::finish( ), these daemon processes will be terminated automatically. This in turn means that they may stay alive if your program get finished without MASS::finish( ), (i.e., a program crash or termination with ^c). In that case, run killMProcess.sh to kill all remote mprocess daemons.

7. Outputs from Places and Agents

Although your main program can use cout and cerr as usual, you cannot use them from each of place/agent objects. This is because they may reside on a remote computing node. All remote processes use their cin and cout/cerr for their communication with the MASS library running on your local machine. Therefore, using cout/cerr in a place or an agent corrupts inter-process communication and hangs the MASS library execution.

To catch outputs from a place or an agent, please use MASS_base::log(string msg) function. If you need to pass any other data types in addition to a string, use ostringstream:

```
ostringstream convert;
convert.str( "" );
convert << "Message from agent[" << agentId << "] = " << message;
MASS_base::log( convert.str( ) );
```

The message is written to the file named MASS_logs/PID_X_IPresult.txt, where X is the remote process ID and IP is the remote IP name. Assuming that you use uw1-320-01, uw1-320-02, and uw1-320-03 remotely from uw1-320-00, all the messages written from uw1-320-01 will be written to MASS_logs/PID_1_uw1-230-01result.txt.

8. Contact Point

For any bug reports or technical questions, please contact Munehiro Fukuda at mfukuda@uw.edu.