Benchmarking, Evaluating and Improving ABM Programming Features in Social, Behavioral and Economic Sciences

Craig Shih

Spring Term Paper 2017

University of Washington

06/2017

Project Committee:

Prof. Munehiro Fukuda, Committee Chair

Prof. Michael Stiber, Committee Member

Prof. Erika Parsons, Committee Member

------- Introduction

Simulation of social, behavioral, and economic (SBE) activities have drastically increased as scientists discover ways to analyze and study data to improve the world around us. As such, there is a growing need allow users to conduct analysis with increased performance and efficiency. Agent-based models (ABMs) have gained popularity to simulate these activities. MASS: Multi-Agent Spatial Simulation is a library that has been developed by Professor Fukuda and his research group as a parallelization platform for spatial simulation applications.

MASS can be partitioned from a high-level point of view into three branches: a C++ version, a JAVA version, and a CUDA version with C++ as the primary branch to focus on simulation SBE activities. There are currently platforms available for spatial simulation applications such as NetLogo and MASON (for sequential execution), and RepastHPC and FLAME (for parallel execution); however, RepastHPC and FLAME have difficult programming frameworks. The goal of MASS is to provide parallelization of spatial simulation while providing a manageable programming framework.

------- Background

Professor Fukuda's overall goal with MASS in assisting in SBE activities is to provide a framework that allows high-speed ABM simulation using C++ and GPU to assist real-time cyber-physical computation. Caleb Yang has begun this process by providing an initial categorization of spatial simulation computing patterns into three groups: Biology, Business/Industry, and Economics/Social Sciences. Sequential benchmark applications have also been developed for each of these groupings as an initial method to measure performance statistics.

My work for this project is expand on what Caleb has done by verifying the existing categories to ensure that the above-mentioned categories encompasses all general agent based spatial simulation computing patterns. Once clear categories are established, parallel benchmark applications for each category will be developed under RepastHPC, FLAME, and MASS so that programmability and performance can be analyzed and compared between the three frameworks. Utilizing those numbers, I will design additional features that can be added to the MASS C++ library that would assist in increasing its performance speeds.

The project will be broken down into six phases: Literature Review, Categorization of ABM Simulation Computing Patterns, Exploration of Multi-Agent Spatial Simulation Libraries, Parallelization and Implementation of Benchmark Tests under different Spatial Simulation Libraries, Programmability and Performance Evaluation, and Defense Preparation.

------- Current Considerations

At the beginning of the quarter, our goal was to submit a paper into PacRim's Conference. As a result, the original project plan was revised and Literature Review, Categorization of ABM Simulation Computing Patterns, and Exploration of Multi-Agent Spatial Simulation Libraries were pushed back into summer quarter's work with parallelization of sequential benchmark

tests with MASS being the main focus of this quarter work. Jumping into parallelization with MASS resulted in my unfamiliarity in a lot of the benchmark tests and majority of the work was done consulting Caleb on the sequential program's logic at which point the benchmark tests were ported over to MASS C++.

Brain Grid, Social Network, and Game of Life benchmark tests were completed with Financial Modeling still being a work in progress.

------- Design

Brain Grid:

The Brain Grid benchmark is a test that covers agents with dynamic communication utilizing the Places Object of the MASS library. The program has an orthogonal grid of square cells where each is either an active, inactive, or neutral neuron.

Active neurons grow in all 8 directions (Moore neighborhoods), neutral neurons grow in the direction it was grown into, and inactive neurons don't grow, but prohibit growth.

PseudoCode for Brain Grid Main:

```
neuron->callAll(Neuron::init_);          //initialize Neuron

timer.start();

for(int turn = 0; turn < numTurns; turn++)
{

   //display the current Type status of the neuron
   neuron->callAll(Neuron::displayTypeStatus_);
   neuron->exchangeBoundary();  // exchange shadow space information
   //exchange information on neighboring neuron's type (Active, Inactive, Neutral).
   neuron->callAll(Neuron::getBoundaryTypeStatus_);

   //display the current stage of the neuron(ENACTED,STOPPED,INDEF)
   neuron->callAll(Neuron::displayStageStatus_);
   neuron->exchangeBoundary();  // exchange shadow space information
   //exchange information on the neighboring neuron's stage.
   neuron->callAll(Neuron::getBoundaryStageStatus_);

   //display the current incomingDirection of neuron
   neuron->callAll(Neuron::displayIncomingDirectionStatus_);
   neuron->exchangeBoundary();  // exchange shadow space information
    //exchange information on neighboring neuron's incoming direction
   neuron->callAll(Neuron::getBoundaryIncomingDirectionStatus_);

   neuron->callAll(Neuron::sendSynapses_);
   neuron->exchangeBoundary();  // exchange shadow space information
   //// need to compute synapses
   neuron->callAll(Neuron::computeSynapses_);
}
```

Neuron->callAll for displayTypeStatus, displayStageStatus, and displayIncomingDirectionStatus are all ways to exchange the necessary information that each Place needs in its computeSynapses logic.

The following shows how those methods are written using displayTypeStatus and getBoundaryTypeStatus as examples.

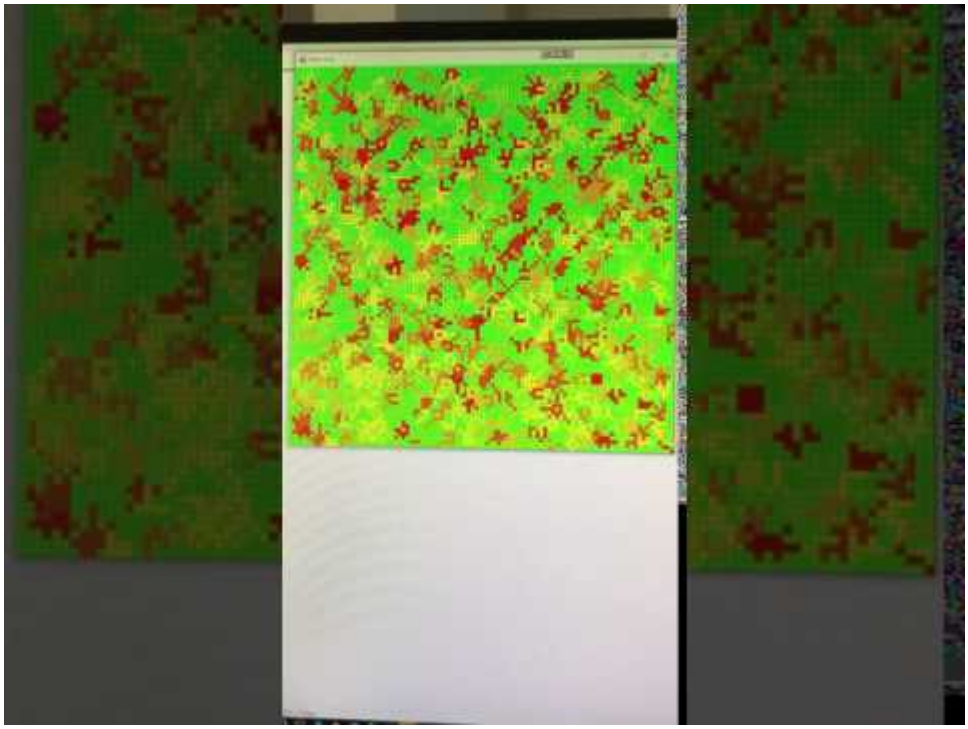
```
void *Neuron::displayTypeStatus()
{
    outMessage_size = sizeof(int);
    outMessage = new int();
    *(int *)outMessage = (int)type;

    return NULL;
}
void *Neuron::getBoundaryTypeStatus()
{
    //Record neighbor status as before
    int North[2] = { 0, 1 };
    int East[2] = { 1, 0 };
    int South[2] = { 0, -1 };
    int West[2] = { -1, 0 };
    int northwest[2] = { -1, 1 };
    int northeast[2] = { 1, 1 };
    int southwest[2] = { -1, -1 };
    int southeast[2] = { 1, -1 };

    int *ptr = (int *)getOutMessage(1, northwest);
    neighborTypeStatus[0] = (ptr == NULL) ? 0 : *ptr;
    ptr = (int *)getOutMessage(1, North);
    neighborTypeStatus[1] = (ptr == NULL) ? 0 : *ptr;
    ptr = (int *)getOutMessage(1, northeast);
    neighborTypeStatus[2] = (ptr == NULL) ? 0 : *ptr;
    ptr = (int *)getOutMessage(1, West);
    neighborTypeStatus[3] = (ptr == NULL) ? 0 : *ptr;
    ptr = (int *)getOutMessage(1, East);
    neighborTypeStatus[4] = (ptr == NULL) ? 0 : *ptr;
    ptr = (int *)getOutMessage(1, southwest);
    neighborTypeStatus[5] = (ptr == NULL) ? 0 : *ptr;
    ptr = (int *)getOutMessage(1, South);
    neighborTypeStatus[6] = (ptr == NULL) ? 0 : *ptr;
    ptr = (int *)getOutMessage(1, southeast);
    neighborTypeStatus[7] = (ptr == NULL) ? 0 : *ptr;

    return NULL;
}
```

Once each Place has the necessary information, computeSynapses() utilizes that information to determine whether a connection between the current neuron and one of its neighbors should be established using the following logic:

```
int direction = -1;
int randStartDir = rand() % 8;
for (int i = 0; i < 8; i++) {
   if (neighborSynapseStatus[(randStartDir + i) % 8] != -1) {
      if ((neighborTypeStatus[(randStartDir + i) % 8] == NEUTRAL
            && neighborIncomingDirection[(randStartDir + i) % 8] == (randStartDir + i) % 8)
                                                || neighborTypeStatus[(randStartDir + i) % 8] == ACTIVE) {
         incomingDirection = (randStartDir + i) % 8;
         direction = (randStartDir + i) % 8;
         break;
      }
   }
}

if (direction != -1) {
   if (type == ACTIVE) {
      //growActive
      if (stage != ENACTED) {
         neuronSynapseNum = neighborSynapseStatus[direction];
         stage = ENACTED;
      }
   }
   else if (type == NEUTRAL) {
      //growNeutral
      //set to Stage of Neutral to currStage
      if (stage == INDEF) {
         neuronSynapseNum = neighborSynapseStatus[direction];
         stage = ENACTED;
      }
      else {

      }
   }
   else { //INACTIVE (inhibitory)
            //stop growth.
      if (stage == INDEF) {
         neuronSynapseNum = neighborSynapseStatus[direction];
         stage = STOPPED;
      }
      else {

      }
   }
}
```

As you can see, a connection is made if the neighbor that is attempting a connection is active or if the neutral neighbor's direction is set to move into the current neuron.

Demo:



Social Networks:
The Social Network benchmark is a test that covers network of agents utilizing the Places Object of the MASS library. The program has an orthogonal grid of square cells where each is a "person" with a random set of neighbors.

Each Place has a random set of "alive" people, up to the total number of Places. Each "person" has a vector of its own friends (1 degree away). Degrees of separation is then calculated by traversing through each neighbor's vectors up to the number of degrees of freedom. An assumption is made that there is at least 1 degree of separation. Each "person" in their neighbor is known through some ambiguous friend.

PseudoCode for Social Networks:
The following is the logic in main (with the initialization of MASS removed).

```
//calculates all destination cells based on the current cell's location.
calculateDestinations();

for(int turn = 0; turn < numTurns; turn++)
{
   int degree = rand() % sizeX;
   if (degree == 0) {
      degree = 1;
   }

   printf("\n%i degrees away \n", degree);

   //exchanges each "persons" 1st degree of separation friends
   snetwork->exchangeAll(1, SNetwork::exchangeMessage_, &destinations);

   //places each corresponding neighboring person's 1st degree of separation
   //friends into inMessage[]
   //proceeds to calculate the friends from n degree of freedoms away
   snetwork->callAll(SNetwork::checkInMessage_, (int *)degree, sizeof(int));

   //resets the inMessage[]
   snetwork->callAll(SNetwork::resetInMessages_);

   destinations.clear();
}
```

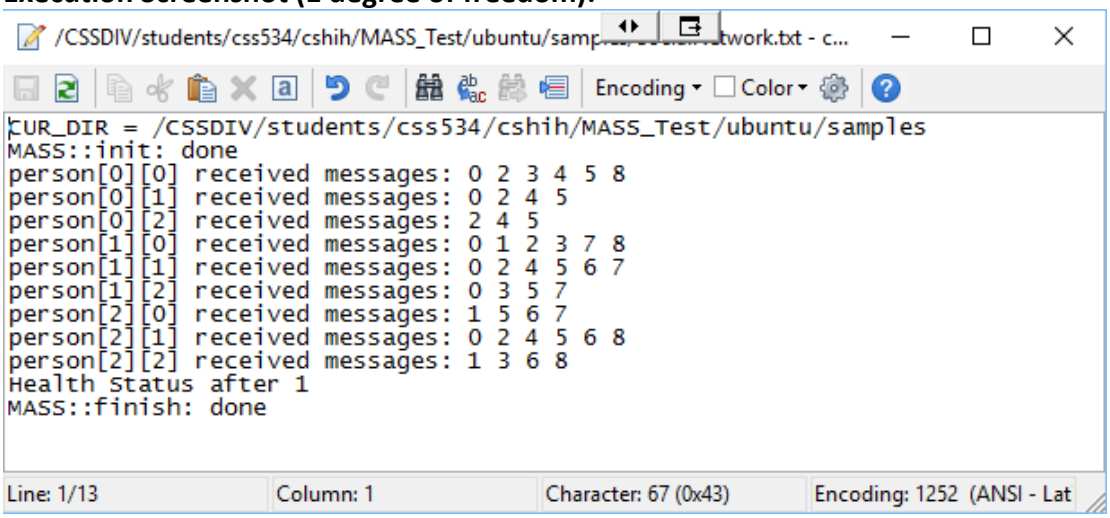
exchangeMessage() and checkInMessage() is as follows:

```
void *SNetwork::exchangeMessage()
{
   vector<int> *totNumNeighbors = &totalNumNeighbors;

   return totNumNeighbors;
}
```

```
void *SNetwork::checkInMessage(void *argument) {
   // ...
   ostringstream convert;

   int degreesOfSeparation = static_cast<int>(reinterpret_cast<std::uintptr_t>(argument));

   // ...
   vector<int> totalDegreeNeighbors;
   int count = 0;
   int tempCount = 0;
   for (int i = 0; i < totalNumNeighbors.size(); i++) {
      if (totalNumNeighbors.at(i) == 1) {
         totalDegreeNeighbors.push_back(i);
      }
   }
   for (int i = 1; i < degreesOfSeparation; i++) {
      tempCount = count;
      for (int j = count; j < totalDegreeNeighbors.size(); j++) {
         vector<int> neighborsNeighbors = *(vector<int> *)inMessages[j];
         for (int k = 0; k < neighborsNeighbors.size(); k++) {
            if (neighborsNeighbors.at(k) == 1) {
               totalDegreeNeighbors.push_back(k);
            }
         }
         tempCount++;
      }
      count += tempCount;
   }

   convert << "person[" << index[0] << "][" << index[1]
      << "] received messages: ";
   for (int i = count; i < totalDegreeNeighbors.size(); i++) {
      convert << totalDegreeNeighbors.at(i) << " ";
   }

   MASS_base::log(convert.str());

   return NULL;
}
```

As can be seen above, a vector is used to store all “friends” up to X degrees of freedom away. The for loop appends each friend found at each degree of freedom up to the degrees of freedom specified in main. A counter is incremented so that when the degrees of freedom specified is reached, a for loop can be looped from count up to the end of the vector to retrieve all friends within that degree of freedom.

Execution Screenshot (1 degree of freedom):



Conway's Game of Life:
Conway's Game of Life benchmark is a test that covers cellular automata utilizing the Places Object of the MASS library. The program has an orthogonal grid of square cells where each is categorized as dead or alive.

In this program initial setup is done using the rand() function which generates either a 1 (alive) or 0 (dead). Each cell will abide by the following rules:

- Any live cell with fewer than two live neighbors dies
- Any live cell with two or three live neighbors lives
- Any live cell with more than three live neighbors dies
- Any dead cell with exactly three live neighbors becomes a live cell.

The above 4 statements can be summarized by the following algorithm:

- If the sum of all nine cells is 3, the current cell's next generation will be life.
- If the sum of all nine cells is 4, the current cell's next generation will be its current health status of life or dead.
- Every other sum, the current cell's next generation will be death.

PseudoCode for Game of Life:

```
for(int turn = 0; turn < numTurns; turn++)
{

   //display the current health status of the life
   life->callAll(Life::displayHealthStatus_);

   life->exchangeBoundary();  // exchange shadow space information

   //exchange information on whether curr life is dead or alive.
   life->callAll(Life::getBoundaryHealthStatus_);

   // need to compute dead or alive based on neighbors
   life->callAll(Life::computeDeadOrAlive_);
}
void *Life::computeDeadOrAlive()
{
   int totalAlive = alive;
   for (int i = 0; i < (int)cardinals.size(); i++)
   {
      totalAlive += neighborHealthStatus[i];
   }

   if (totalAlive == 3) { //next generation will be life
      alive = 1;
   }
   else if (totalAlive > 4 || totalAlive < 3) { //next generation will be the same as current state
      alive = 0;
   }
```

The following two code snippets show the most important aspects of the program: the general structure of the program and the logic to determine whether a cell is alive or dead.

To give you an idea of the general structure, the above screenshot first has each cell perform displayHealthStatus.

displayHealthStatus sets the outmessage of each cell as the current health status of that cell (either dead or alive). This allows other cells to retrieve this status in order to help them determine what their next health status will be in its next generation.

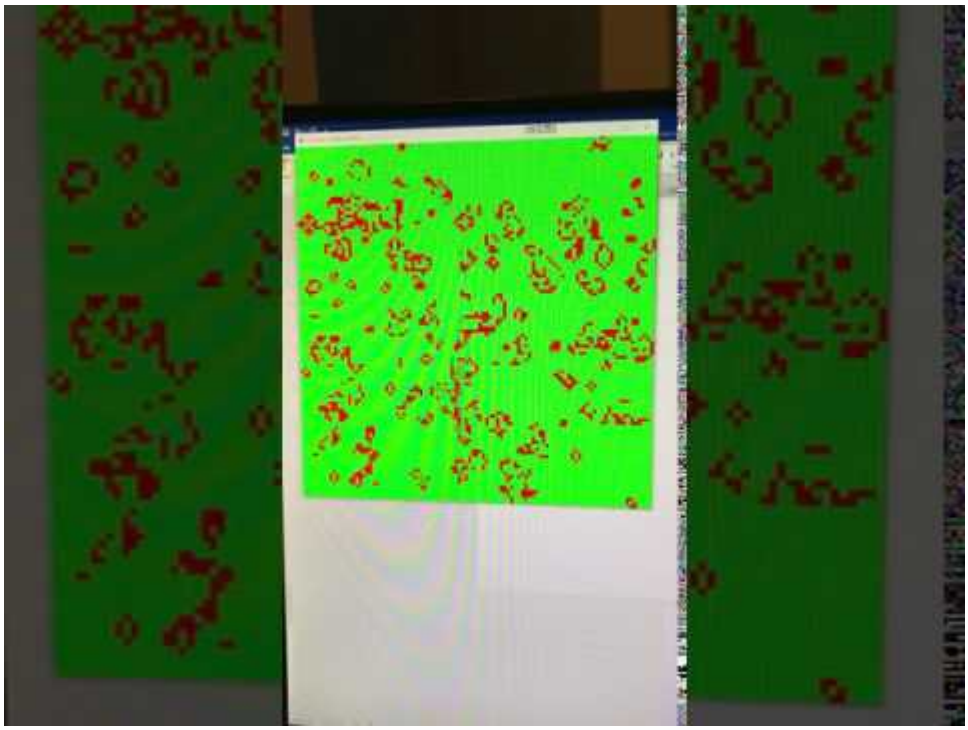
exchangeBoundary() is used as it states above: to exchange shadow space information. This is needed for when Place objects need to communicate across nodes.

getBoundaryHealthStatus is then called by all cells again. This function allows each cell to pull its 9 adjacent cells' health statuses. Using this information, the cell's next generation's status is calculated by performing computeDeadOrAlive(), which is shown above.

To reiterate, the following above logic performs the following statements:

- Any live cell with fewer than two live neighbors dies
- Any live cell with two or three live neighbors lives
- Any live cell with more than three live neighbors dies
- Any dead cell with exactly three live neighbors becomes a live cell.

Demo:



Project Plan

The estimated timeline of events is as follows:

Phase	Tasks	Expected Deadline
Parallelization and Implementation of Benchmark Tests under different Spatial Simulation Libraries	Parallelize benchmark tests using MASS for BrainGrid, SocialNetwork, GameOfLife,	5/27/17
	Parallelize benchmark tests for MatSim, Tuberculosis, Financial Modeling	6/30/2017
Literature Review	Agent Based Computing Patterns for Spatial Simulation	7/8/17
	Agent Based Computing Libraries (RepastHPC, FLAME)	
	MASS C++ library	
	Development environment tools	
Categorization of ABM Simulation Computing Patterns	Verify and establish general agent based computing patterns	7/15/17
	Add/Reorganize computing patterns	7/22/17
Exploration of Multi-Agent Spatial Simulation Libraries	Setup RepastHPC, FLAME, MASS	7/29/17
	Explore API for RepastHPC, FLAME, MASS	7/29/17
	Practice using methods within the three APIs	7/29/17
	Implement test program utilizing three libraries	7/29/17
Parallelization and Implementation of Benchmark Tests under different Spatial Simulation Libraries	Parallelize benchmark tests using FLAME, RepastHPC	8/31/17
Programmability & Performance Evaluation	Evaluate performance between FLAME, RepastHPC, and MASS	9/14/17
	Suggest and design additional features for MASS C++ that could improve performance	9/30/17
	Report Writing	10/15/17
Defense Preparation	Draft Report	11/1/17
	Share Results	12/1/17
	Finalize Report	12/7/17