MASS C++ Development

Jennifer Kowalsky
University of Washington Bothell

Introduction

The Multi-Agent Spatial Simulation (MASS) library allows users to easily create distributed programs without worrying about distribution details in their client programs.

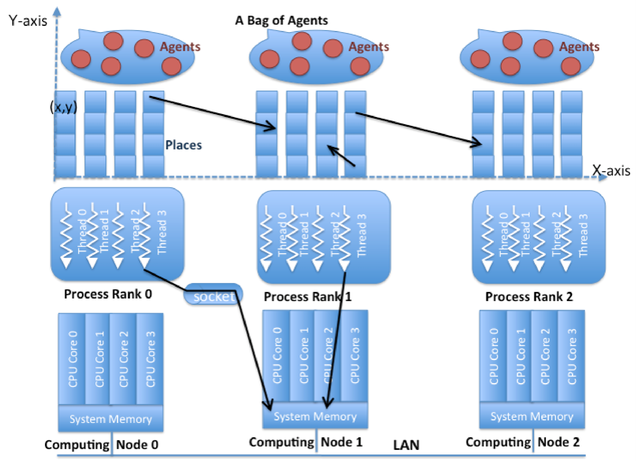

Figure 1: The MASS library's programming paradigm of Agents executing over Places distributed through a network of computing nodes.

Client programs implement Places and Agents. Places can communicate with each other and store information. Agents can communicate with each other and with Places, move to new Places, and spawn new Agents or terminate themselves. A simple example use would be to model Troops (Agents) fighting over Land (Places.)

My project was to update the C++ version of MASS to implement unique neighbors for Places and provide performance testing for the MASS C++ library.

Problem

In order for Places to communicate with each other, MASS asks the user to provide a list of neighboring Places. For example, the user might declare the cardinal directions as a particular Place's neighbors. Any arbitrary pattern the user chose could be used. Unfortunately, MASS originally used the same neighbor's pattern for all Places.

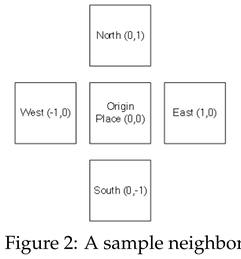

Figure 2: A sample neighbors setup using the cardinal directions.

Certain applications, such as geographic maps or neural simulations, require each Place to have unique neighbors. In a geographic map, neighbors can represent roads connecting locations. In a neural network, neighbors represent the dynamically formed connections between neurons.

The problem my project solved was introducing a method for easily adding unique neighbors for each Place in a MASS simulation.

Solution Design

The following changes were made to add the new neighbors functionality:

- Maintain a set of neighbors in each Place
  - Sets are similar to vectors, but each value in a set is unique.
  - This prevents Places from being "double counted" in calculations.
- Added methods to the Place class to
  - Add neighbors to a given Place
  - Remove neighbors from a given Place
  - Check if a Place is currently a neighbor to a given Place
- Modified the exchangeAll method to use each Place's individual neighbors list instead of a passed in neighbors list
- Maintain backwards compatibility by providing an exchangeAll method with the original method signature by
  - First setting the neighbors for all Places in a simulation
  - Then automatically calling the new exchangeAll method.

Major Code Changes

```
/**
 * Allows the user to add a list of neighbors to this specific place.
 * @param destinations - The neighbors to be added to the current neighbors list
 */
void Place::addNeighbors(vector<int*> *destinations) {
  int *tmp; // to hold elements from destinations

  // for each place in destinations, get the coordinate pair
  for ( int i = 0; i < int( destinations->size( ) ); i++ ) {
    tmp = (*destinations)[i];

      // using a set eliminates duplicates and guarantees items are sorted.
      // A speed boost can be achieved by providing a 'hint' as to the location
      // where you want to insert next.  This is currently not implemented.
      neighbors.insert(tmp);
  }
}

 /**
  * Set the neighbors for a set of places.
  * @param dstPlaces - places whose neighbors we are setting
  * @param destinations - coordinates of the neighboring places
  * @param tid
  */
 void Places_base::setAllPlacesNeighbors(Places_base *dstPlaces,
          vector<int*> *destinations, int tid) {

 /**
  * Replicates the original exchangeAll which acted upon destinations included
  * in the parameters.  This version now performs sanity checks on the destinations.
  * @param dstPlaces
  * @param functionId
  * @param destinations
  * @param tid
  */
 void Places_base::exchangeAll( Places_base *dstPlaces, int functionId,
              vector<int*> *destinations, int tid ) {

   // add our neighbors to each place.
   this->setAllPlacesNeighbors(dstPlaces, destinations, tid);
   // now call exchangeAll to act on those neighbors.
   this->exchangeAll(dstPlaces, functionId, tid);
 }
 /**
  * ExchangeAll method that relies on neighbors contained in each individual place.
  * @param dstPlaces
  * @param functionId
  * @param tid
  */
 void Places_base::exchangeAll( Places_base *dstPlaces, int functionId,
              int tid) {
```

Performance Testing of the MASS C++ Library

The new implementation of neighbors caused a slight performance hit of 1.7%, which was offset by the new functionality introduced.

Times Neighbors Was Faster:	
48	30.00%
Times Unmodified Was Faster:	
112	70.00%
Average Difference in Microseconds:	
-3572.18	
Average Ratio of Neighbors Runtimes to Unmodified Runtimes	
1.01693	

Figure 3: Performance data comparing MASS C++ with and without neighbors functionality.

Performance data was gathered using Jay Hennen's test programs. His tests call all of MASS's main functions with varying computational loads, repeats, and Agent movement complexities. Running each set of tests 10 times gives a statistically significant look at the performance of the MASS C++. This data was used both to calculate the performance hit of the new Neighbor's feature and to provide baseline execution data for the MASS research group.

The following factors were varied for these tests:

- Simulation Size: 64, 256
- Number of Computing Nodes: 1, 2, 4, 8, 16
- Repeats: 1, 50 100, 150, 200, 250
- Agent Test Types: 1, 2, 3, 4, 5, 6, 7
- Place Test Types: 1, 2, 3, 4

The total number of tests run was 6,600.

To automate these tests, I used shell scripts to run the tests multiple times and scrape performance times from the output of Jay Hennon's test programs.

```
#!/bin/bash
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/net/metis/home3/dslab/MASS/c++/ubuntu/ssh2/lib:/net/metis/home3/dslab/MASS/c++/ubuntu
size=256
iters=10
#strace -tt

# for a given number of nodes
for hosts in 1 2 4 8 16
do
  # run each of the 4 test types
  for test in 1 2 3 4
  do
    # with an increasing number of repeats
    for t in 1 50 100 150 200 250
    do
      echo 'PerfTest: Places Test iters='$iters' size='$size' t='$t' test='$test' hosts='$hosts
      ./TestMain dslab dslab-302 machinefile$hosts.txt 43334 $hosts 4 $test $size $t $iters
      wait $!          # Wait for the last test to finish
      sh killMProcess.sh # Make sure all MASS execution instances are terminated
      wait $!
      echo 'done'
    done
  done
done
```

Figure 4: Shell script used to run a set of tests, using waits to prevent collision of MASS instances.