Agents Implementation for C++ MASS Library

CSS 497 – Spring 2014 – Term Report

Chris Rouse

Contents

Work Summary

Initial Goals

Initially, I was tasked to finish the implementation of Agents in the C++ MASS Library. This would include the creation of the Agents_base methods CallAll and ManagaAll. These functions would need to pass basic functionality tests to prove their correct implementation, and would need to be finished by the end of spring quarter 2014. A write up would accompany this code, as well as a presentation in both winter and spring quarter. Written write ups would be sent at the end of the respective quarters to give an update on the progress made.

Current Work Progress

As the project moved forward, the deadline was shortened so as to be used in conjunction with the CSSE 534 class. They would use it as homework and report bugs to the professor. To accomplish this, Professor Fukuda and myself pair-programmed the remaining code that was not complete as of the fourth week of spring quarter.

CallAll was complete, and ManageAll was in progress. As of the eighth week of spring quarter, MASS C++ was finished and correctly passing basic tests. Most files in the C++ directory were altered within the course of the project. This project was uploaded onto the school's Git repository, which was created this quarter by another student.

Detailed Specification

Agents should work across multiple nodes without problem, and be able to carry with them information that will be preserved during transit. They will also need to be able to be placed into a correct Place location after migration. Below is the outline of the Agent process.

Agent Diagram

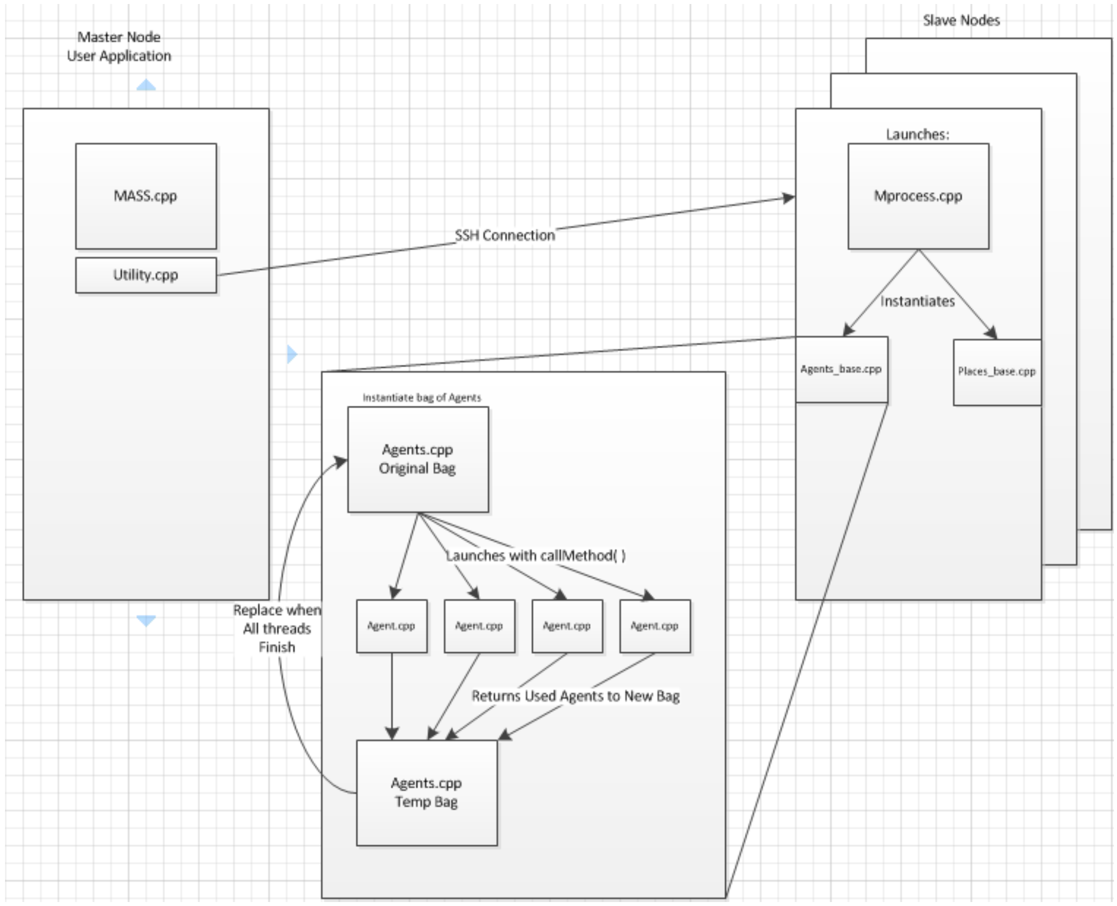


Figure 1 - Agent Flow Diagram

Within the Agents class, the methods of callAll() and manageAll() were needed to be implemented and tested. These would work as follows:

callAll()

- This would allow for Agents to be instantiated, and their contained Agent pieces to be given commands and told to execute them.
- The individual Agent pieces need to be managed, removed from Agents bags, and then re-added to bags upon completion of thread execution.
- Needs to make sure that the system is synchronized so that we can avoid race conditions when we remove from bags for execution.

manageAll()

- This code will need to manage the kill(), spawn() and migrate() methods
    - Kill() will need to destroy an Agent which is specified.
    - Spawn() needs to create an Agent in a local node/place and be added to the Agents bag which exists locally.
    - Migrate() will need to allow for an Agent to move between places and/or nodes at will, changing bags as necessary.
- After the above flags are checked, if kill or migrate are not called, the agent is placed in a temporary bag to be reassigned after the process completes.
- Once the node finishes execution of the Agents, it needs to send migrating agents to remote nodes
- Once all migrating Agents are sent to remote nodes, node must read in all incoming agents.
- While reading in Agents, node must place incoming Agent into correct Place and into bag of executed Agents
- Once finished, must reassign executed Agents bag to old Agents bag location, and update Agent bag count

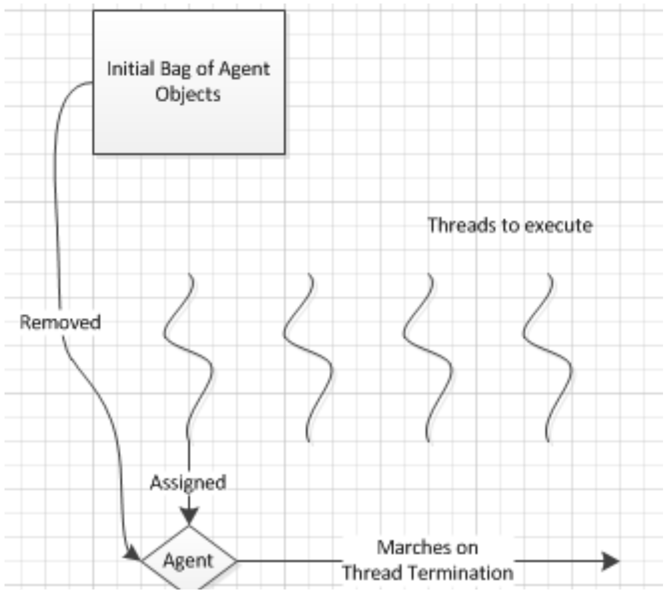


Figure 2 - Places Agent Launching

Design of Methods

Using Two Bags of Agents:

The decision to use two bags of agents was to allow for the maximum number of threads to be running at a given time. If we followed the Java implementation, the threads would travel down a column of places, executing along the way. Any holdup in a place 'above' them would cause another thread to need to wait to execute.

Two bags allows us to assign all the agents out as quickly as possible, and when they finish their execution, they will be placed into a separate bag of Agents as to not interfere with the allocation timing of new threads grabbing Agent objects from the initial bag.

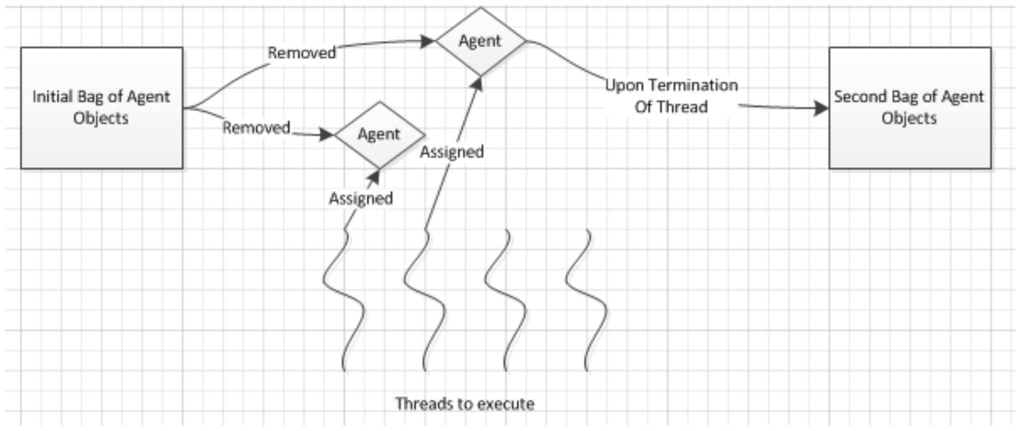

Initial Bag of Agent Objects
Removed
Removed
Agent
Agent
Assigned
Assigned
Upon Termination Of Thread
Second Bag of Agent Objects
Threads to execute

Figure 3 - Multi-Bag Agent Allocation

Removing Elements from the End of Agents:

Since we are using vectors to hold the Agent pointers, we had to implement the method to pull agents from the end of the vector. Vectors don't naturally allow us to pull from the front in C++ (and even if they did, it would mean changing the head pointer every time we called pop_back()). Instead, we simply take from the end to avoid needing to change any pointers through the assignment process.

However, unlike in Java, C++ vectors do not return an object when they are popped from the end of a vector. Because of this, the Agent needs to be saved prior to calling the pop_back() command. This is an important step, as we could lose the Agent if we do not always keep a reference to it somewhere.

Using a Mutex Lock for the Iterator:

While using two bags of Agents allows us to assign Agent objects to threads quickly and run as many as possible, we have to be careful to avoid race conditions that would lead to two Threads wanting to lay claim to the same Agent. Although Vectors are thread-safe in C++, we may end up with Threads trying to pull an Agent from a bag in which the Agent does not exist anymore.

To solve this problem, we use a Mutex lock over the iterator value itself. The iterator becomes the critical section, and only one thread at a time can come and request an Agent for pop_back() purposes. This will ensure that we only allocate as many Agent objects as the bag has available, and once this state has been reached, we will barrier all other threads until the Agent objects become available again, which will be done upon the bags being swapped.

Changes to the Bcopy Function

Through the quarter, during my attempt to adapt the C++ MASS library to the Eclipse IDE platform, I ran into a number of problems. One major problem was the deprecated function bcopy(). Eclipse threw errors over the deprecated functions and refused to run. It was decided to update this function with the more modern function memcpy, which does the same thing as bcopy, but is not deprecated itself.

Once I made these changes, I ran the library to compare output and noticed that the values were all incorrect. Upon later inspection, I noticed that there was a difference in the order that memcpy took its arguments. This was easily remedied, and once again the library was tested. This time, the values matched those from before the changes were implemented.

This was an important change to make, as we want our MASS library to one day be compatible with a multitude of IDE's for students to develop applications using. While it currently still does not run within Eclipse IDE, this is a step towards making that progress. Other issues remain, but I was unable to address those within the current timeframe. Time pending Spring Quarter 2014, I would like to once again attempt code integration.

ManageAll Mutual Exclusion

In ManageAll, it was decided that we would need an order to check the flags (Kill, Spawn, Migrate) to ensure that we are using resources to the best of our abilities. Some flags could render others useless, and some might waste resource usage on an Agent that would die in the same turn. I decided to have it check in the order of: Spawn, Kill, Migrate. This was done to minimize the amount of Migrations that might occur.When an Agent begins its flag checks, the first check should be to spawn. Likely, when a user sets the Agent to spawn, they would mean it to be in the current Place the Agent is residing in. Since Spawning has no effect on the other flags, every Agent will first check this flag and execute a spawn if the

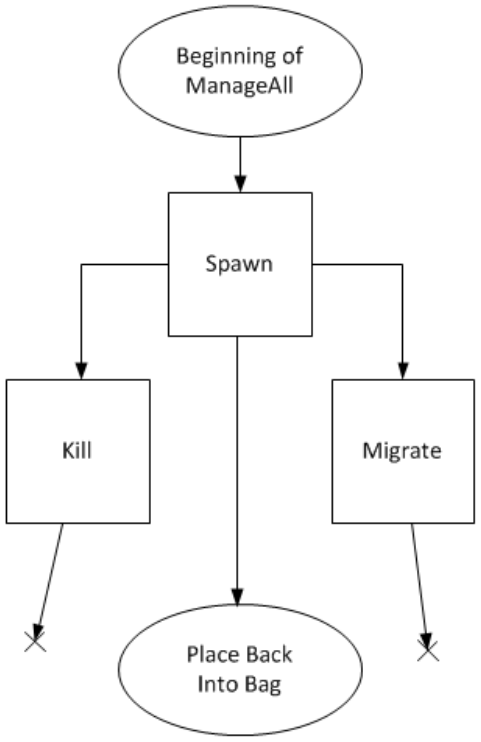


Figure 4 - Mutual Exclusion of Kill & Migrate

number of children is anything above 0.

Afterwards, it would check the kill flag. This is done to ensure that, before migration, we kill off any Agents designated for execution. Killing Agents before they migrate will reduce the migration times/resources, and allow it to only migrate Agents that will be surviving into the next cycle. When an Agent is removed, it is not only deleted, but the pointer its Place has access to will also be removed.

Finally, mutually exclusive with Kill, it will check the Migrate flag. If the Agent was killed, it will never hit this step, as it has already been removed. If Kill is not triggered, it will check the index of the Agent vs. the pointer it contains to its Place. If they match, it will not migrate. If they are different, it will be flagged to migrate, and placed into the migration queue.

Post ManageAll Behavior

Once the three flags are checked, the Node needs to check to see if they have outgoing Agents. If so, they will calculate the global index of the Agent's destination. When this is found, we locate the Node the Place resides on, package the Agent into a Message, and send it to the destination node. During this process, we spawn children threads so we can send and receive Agents from all nodes simultaneously. This prevents the system from deadlocking if too many Agents are coming or being sent.

Received Agents are unpackaged and pointers to them are created in their destination Place. They are then placed into the spent Agents bag, and the Agent count is incremented. This incrementation, as well as the vector of used Agents, is locked with the mutex lock so that we do not invalidate the count. As found by trial and error in the course of this research, C++ vectors are **NOT** thread-safe, and also needed to be locked upon adding and removal of Agents.

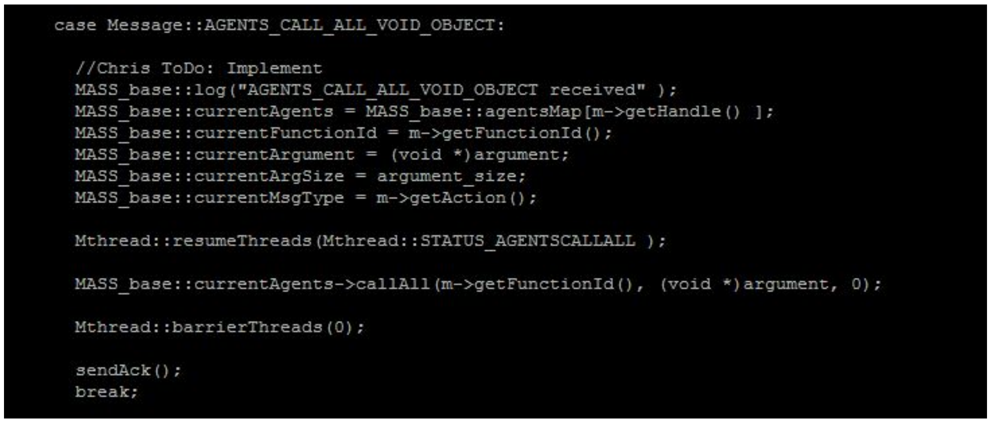
Direction on Where and How to Run Programs:

Currently, all files that were modified are being stored in the dslab directory on UWB's servers. I have placed my own code into a directory called C++, which houses the finished code. No original files were created by me, as I was working with existing files or preexisting skeleton code. All files also exist on the school's Git repository, and can be checked out with the permission of professor Fukuda.

To run these files, the MASS library needs to be started. Since I was not working on an application, there is no pre-defined file to use. However, using the run.sh script, you can test the viability of the library using the wave2d program that the script defaults to. However, within the C++ folder, there are two additional folders, one for compiling and running on Linux Redhat, and one for compiling and running on Linux Ubuntu. To do this, go to the dslab location, and into the C++ file, find the correct version of Linux, and run the run.sh script to start the default test.

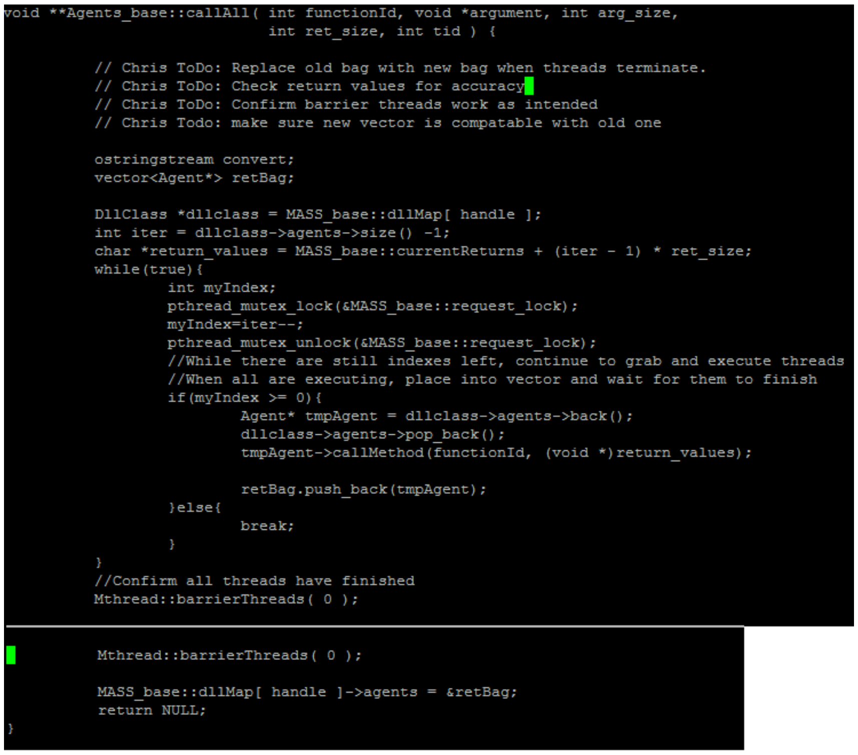
Source Code:

Below are the screen shots of code that was created by me throughout the Winter and Spring 2014 quarter. Each is labeled with its corresponding file it is located within.



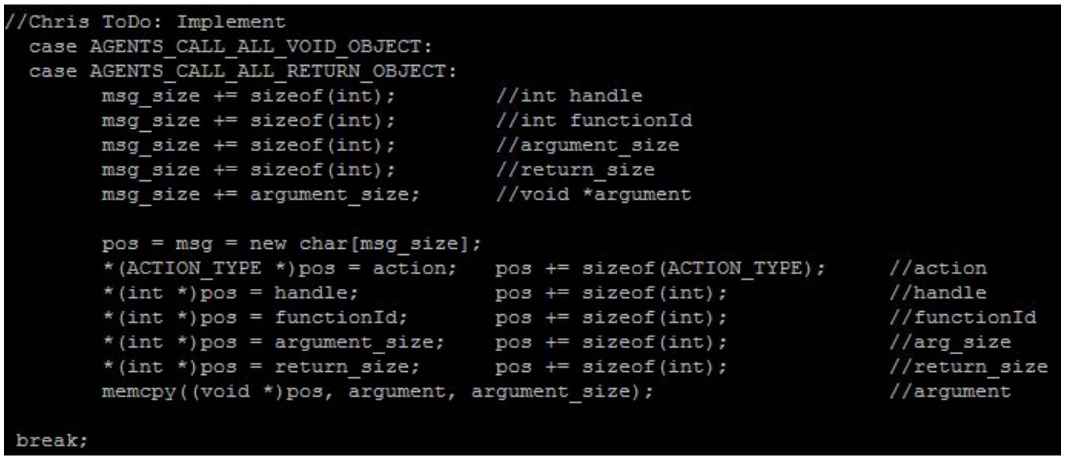
```
    case Message::AGENTS_CALL_ALL_VOID_OBJECT:

      //Chris ToDo: Implement
      MASS_base::log("AGENTS_CALL_ALL_VOID_OBJECT received" );
      MASS_base::currentAgents = MASS_base::agentsMap[m->getHandle() ];
      MASS_base::currentFunctionId = m->getFunctionId();
      MASS_base::currentArgument = (void *)argument;
      MASS_base::currentArgSize = argument_size;
      MASS_base::currentMsgType = m->getAction();

      Mthread::resumeThreads(Mthread::STATUS_AGENTSCALLALL );

      MASS_base::currentAgents->callAll(m->getFunctionId(), (void *)argument, 0);

      Mthread::barrierThreads(0);

      sendAck();
      break;
```

Figure 5 - Mprocess.cpp CallAll void return



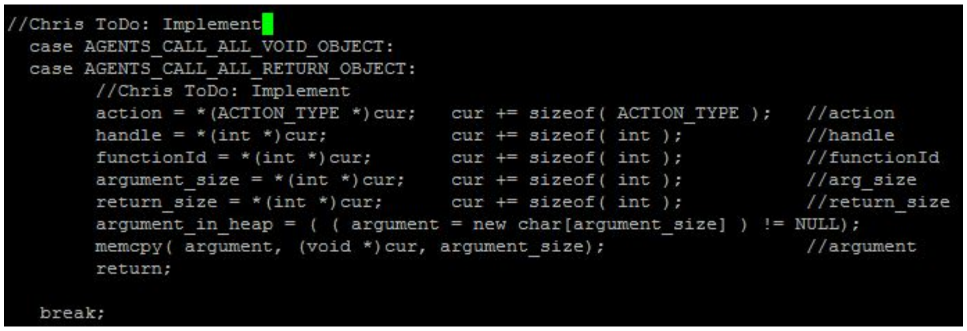
```
void **Agents_base::callAll( int functionId, void *argument, int arg_size,
                             int ret_size, int tid ) {

        // Chris ToDo: Replace old bag with new bag when threads terminate.
        // Chris ToDo: Check return values for accuracy
        // Chris ToDo: Confirm barrier threads work as intended
        // Chris Todo: make sure new vector is compatable with old one

        ostringstream convert;
        vector<Agent*> retBag;

        DllClass *dllclass = MASS_base::dllMap[ handle ];
        int iter = dllclass->agents->size() -1;
        char *return_values = MASS_base::currentReturns + (iter - 1) * ret_size;
        while(true){
                int myIndex;
                pthread_mutex_lock(&MASS_base::request_lock);
                myIndex=iter--;
                pthread_mutex_unlock(&MASS_base::request_lock);
                //While there are still indexes left, continue to grab and execute threads
                //When all are executing, place into vector and wait for them to finish
                if(myIndex >= 0){
                        Agent* tmpAgent = dllclass->agents->back();
                        dllclass->agents->pop_back();
                        tmpAgent->callMethod(functionId, (void *)return_values);

                        retBag.push_back(tmpAgent);
                }else{
                        break;
                }
        }
        //Confirm all threads have finished
        Mthread::barrierThreads( 0 );

        Mthread::barrierThreads( 0 );

        MASS_base::dllMap[ handle ]->agents = &retBag;
        return NULL;
}
```

Figure 6 – Agents_base.cpp



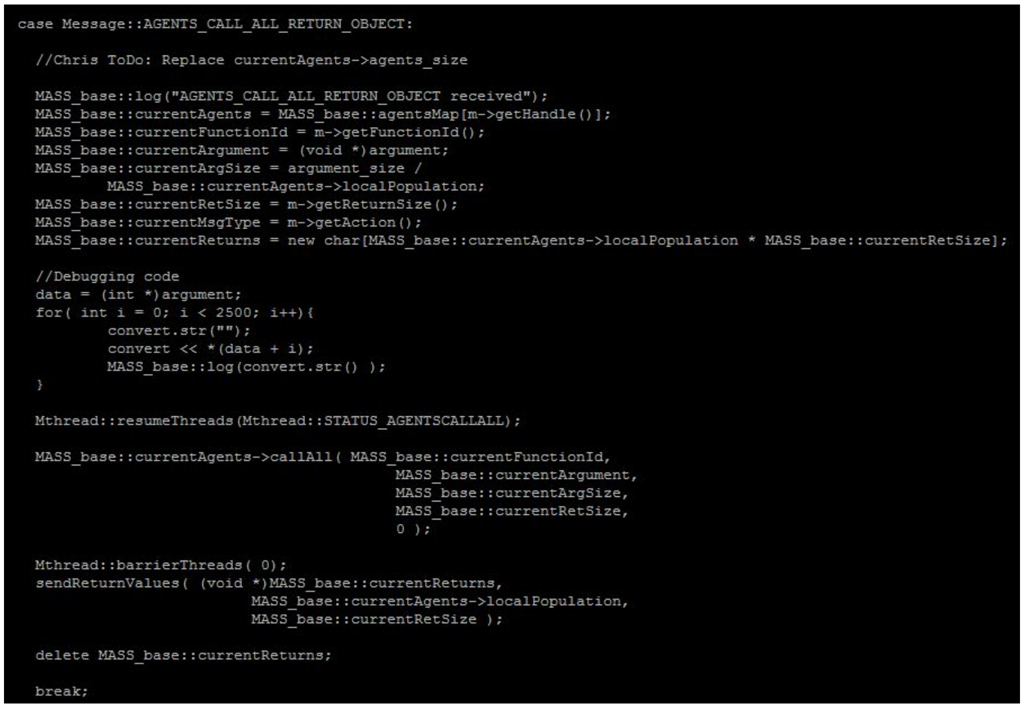
```
//Chris ToDo: Implement
  case AGENTS_CALL_ALL_VOID_OBJECT:
  case AGENTS_CALL_ALL_RETURN_OBJECT:
        msg_size += sizeof(int);        //int handle
        msg_size += sizeof(int);        //int functionId
        msg_size += sizeof(int);        //argument_size
        msg_size += sizeof(int);        //return_size
        msg_size += argument_size;      //void *argument

        pos = msg = new char[msg_size];
        *(ACTION_TYPE *)pos = action;   pos += sizeof(ACTION_TYPE);     //action
        *(int *)pos = handle;           pos += sizeof(int);             //handle
        *(int *)pos = functionId;       pos += sizeof(int);             //functionId
        *(int *)pos = argument_size;    pos += sizeof(int);             //arg_size
        *(int *)pos = return_size;      pos += sizeof(int);             //return_size
        memcpy((void *)pos, argument, argument_size);                   //argument

 break;
```

Figure 7 – Message.cpp Serialize



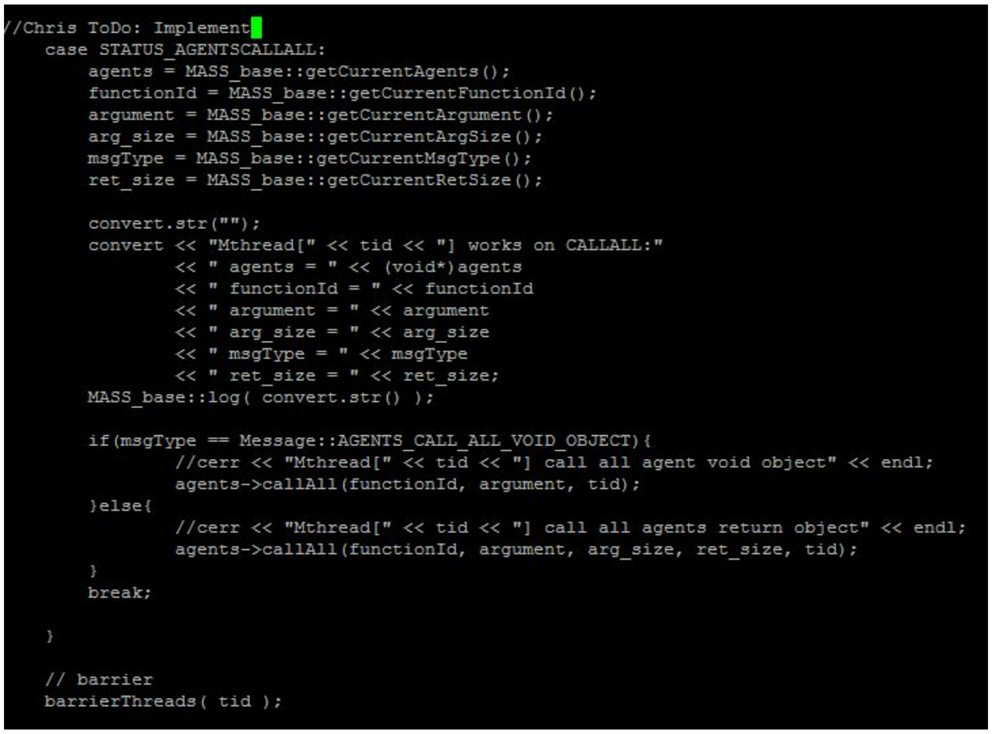
```
//Chris ToDo: Implement
  case AGENTS_CALL_ALL_VOID_OBJECT:
  case AGENTS_CALL_ALL_RETURN_OBJECT:
        //Chris ToDo: Implement
        action = *(ACTION_TYPE *)cur;   cur += sizeof( ACTION_TYPE );   //action
        handle = *(int *)cur;           cur += sizeof( int );           //handle
        functionId = *(int *)cur;       cur += sizeof( int );           //functionId
        argument_size = *(int *)cur;    cur += sizeof( int );           //arg_size
        return_size = *(int *)cur;      cur += sizeof( int );           //return_size
        argument_in_heap = ( ( argument = new char[argument_size] ) != NULL);
        memcpy( argument, (void *)cur, argument_size);                  //argument
        return;

   break;
```

Figure 8 – Message.cpp De-serialize



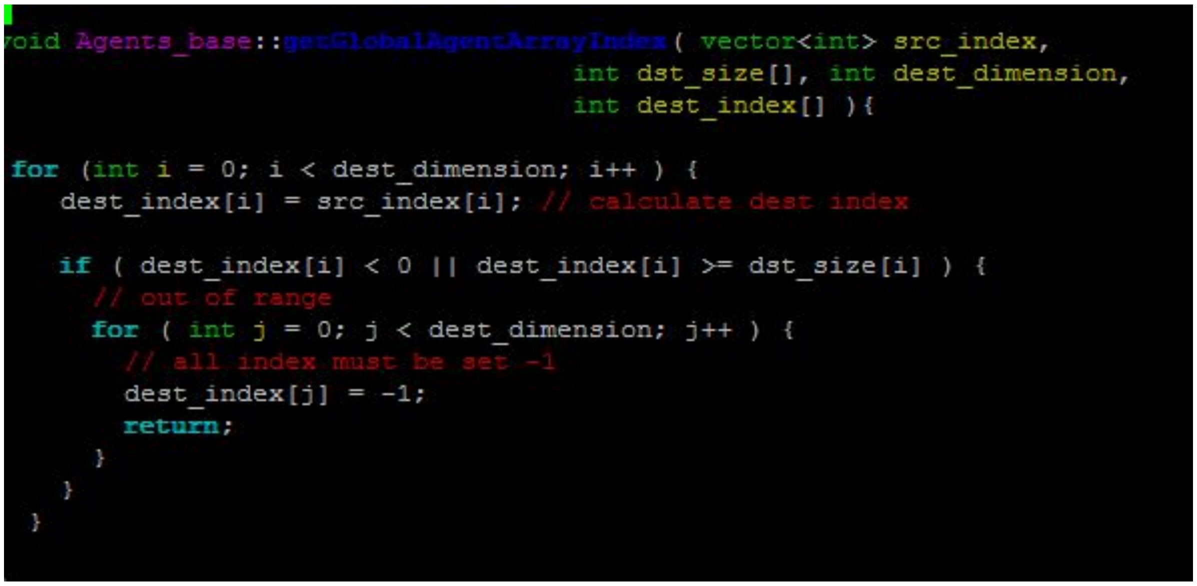
```
case Message::AGENTS_CALL_ALL_RETURN_OBJECT:

  //Chris ToDo: Replace currentAgents->agents_size

  MASS_base::log("AGENTS_CALL_ALL_RETURN_OBJECT received");
  MASS_base::currentAgents = MASS_base::agentsMap[m->getHandle()];
  MASS_base::currentFunctionId = m->getFunctionId();
  MASS_base::currentArgument = (void *)argument;
  MASS_base::currentArgSize = argument_size /
          MASS_base::currentAgents->localPopulation;
  MASS_base::currentRetSize = m->getReturnSize();
  MASS_base::currentMsgType = m->getAction();
  MASS_base::currentReturns = new char[MASS_base::currentAgents->localPopulation * MASS_base::currentRetSize];

  //Debugging code
  data = (int *)argument;
  for( int i = 0; i < 2500; i++){
          convert.str("");
          convert << *(data + i);
          MASS_base::log(convert.str() );
  }

  Mthread::resumeThreads(Mthread::STATUS_AGENTSCALLALL);

  MASS_base::currentAgents->callAll( MASS_base::currentFunctionId,
                                     MASS_base::currentArgument,
                                     MASS_base::currentArgSize,
                                     MASS_base::currentRetSize,
                                     0 );

  Mthread::barrierThreads( 0);
  sendReturnValues( (void *)MASS_base::currentReturns,
                          MASS_base::currentAgents->localPopulation,
                          MASS_base::currentRetSize );

  delete MASS_base::currentReturns;

  break;
```

Figure 9 – Mprocess.cpp CallAll Return Object



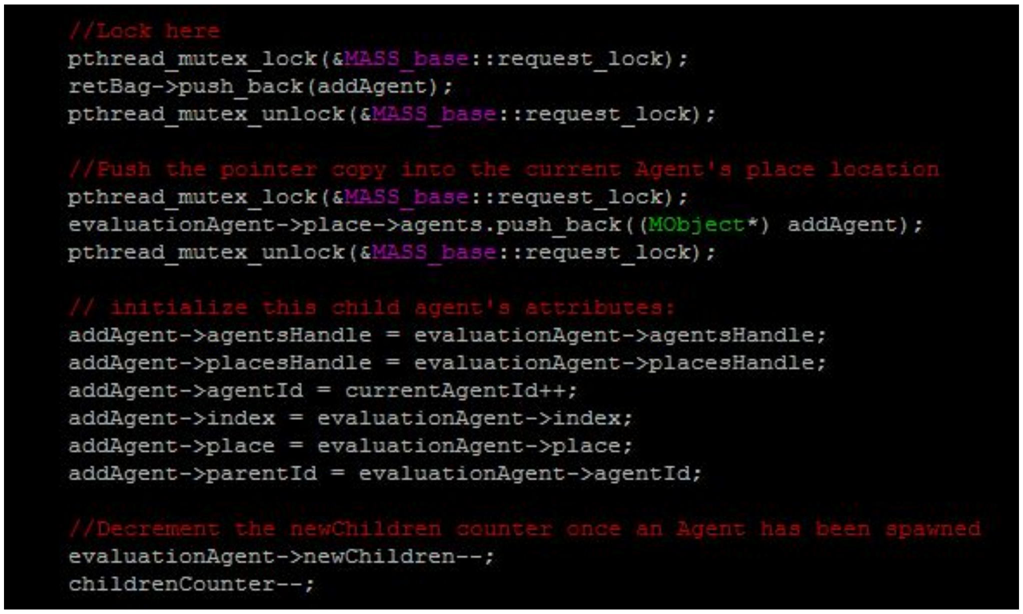
```
/Chris ToDo: Implement
  case STATUS_AGENTSCALLALL:
      agents = MASS_base::getCurrentAgents();
      functionId = MASS_base::getCurrentFunctionId();
      argument = MASS_base::getCurrentArgument();
      arg_size = MASS_base::getCurrentArgSize();
      msgType = MASS_base::getCurrentMsgType();
      ret_size = MASS_base::getCurrentRetSize();

      convert.str("");
      convert << "Mthread[" << tid << "] works on CALLALL:"
              << " agents = " << (void*)agents
              << " functionId = " << functionId
              << " argument = " << argument
              << " arg_size = " << arg_size
              << " msgType = " << msgType
              << " ret_size = " << ret_size;
      MASS_base::log( convert.str() );

      if(msgType == Message::AGENTS_CALL_ALL_VOID_OBJECT){
              //cerr << "Mthread[" << tid << "] call all agent void object" << endl;
              agents->callAll(functionId, argument, tid);
      }else{
              //cerr << "Mthread[" << tid << "] call all agents return object" << endl;
              agents->callAll(functionId, argument, arg_size, ret_size, tid);
      }
      break;

  }

  // barrier
  barrierThreads( tid );
```

Figure 10 – Mthread.cpp



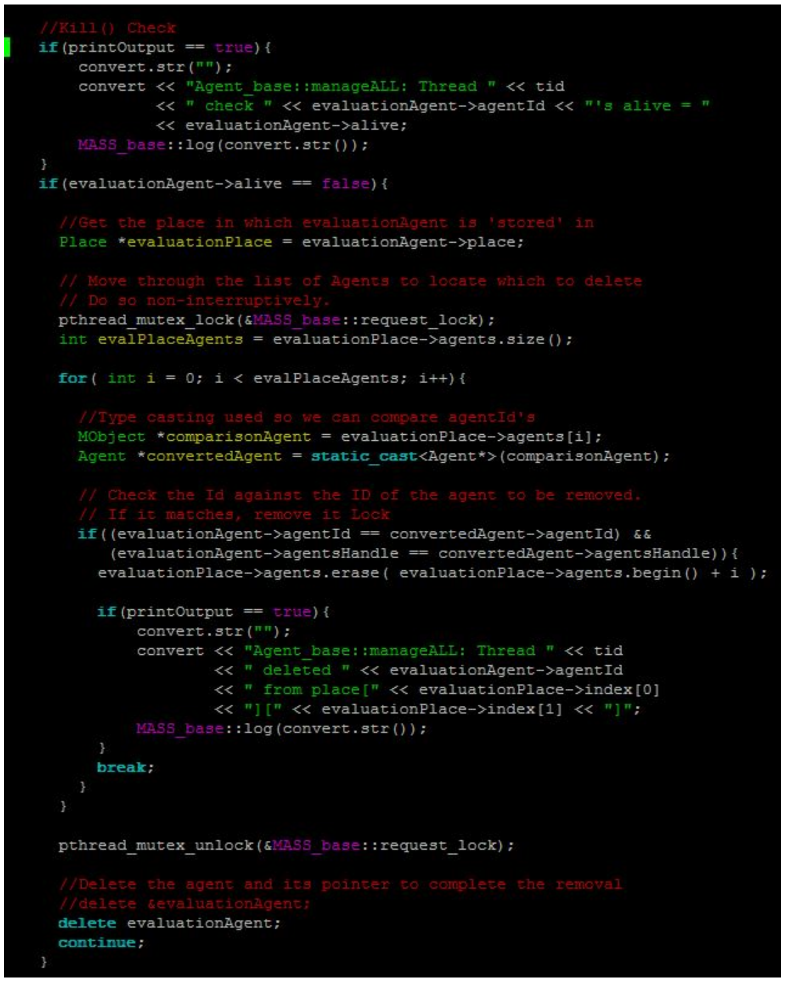
```
void Agents_base::getGlobalAgentArrayIndex( vector<int> src_index,
                                            int dst_size[], int dest_dimension,
                                            int dest_index[] ){

 for (int i = 0; i < dest_dimension; i++ ) {
   dest_index[i] = src_index[i]; // calculate dest index

   if ( dest_index[i] < 0 || dest_index[i] >= dst_size[i] ) {
     // out of range
     for ( int j = 0; j < dest_dimension; j++ ) {
       // all index must be set -1
       dest_index[j] = -1;
       return;
     }
   }
 }
```

Figure 10 – Get Global Array Index, Agents_base.cpp



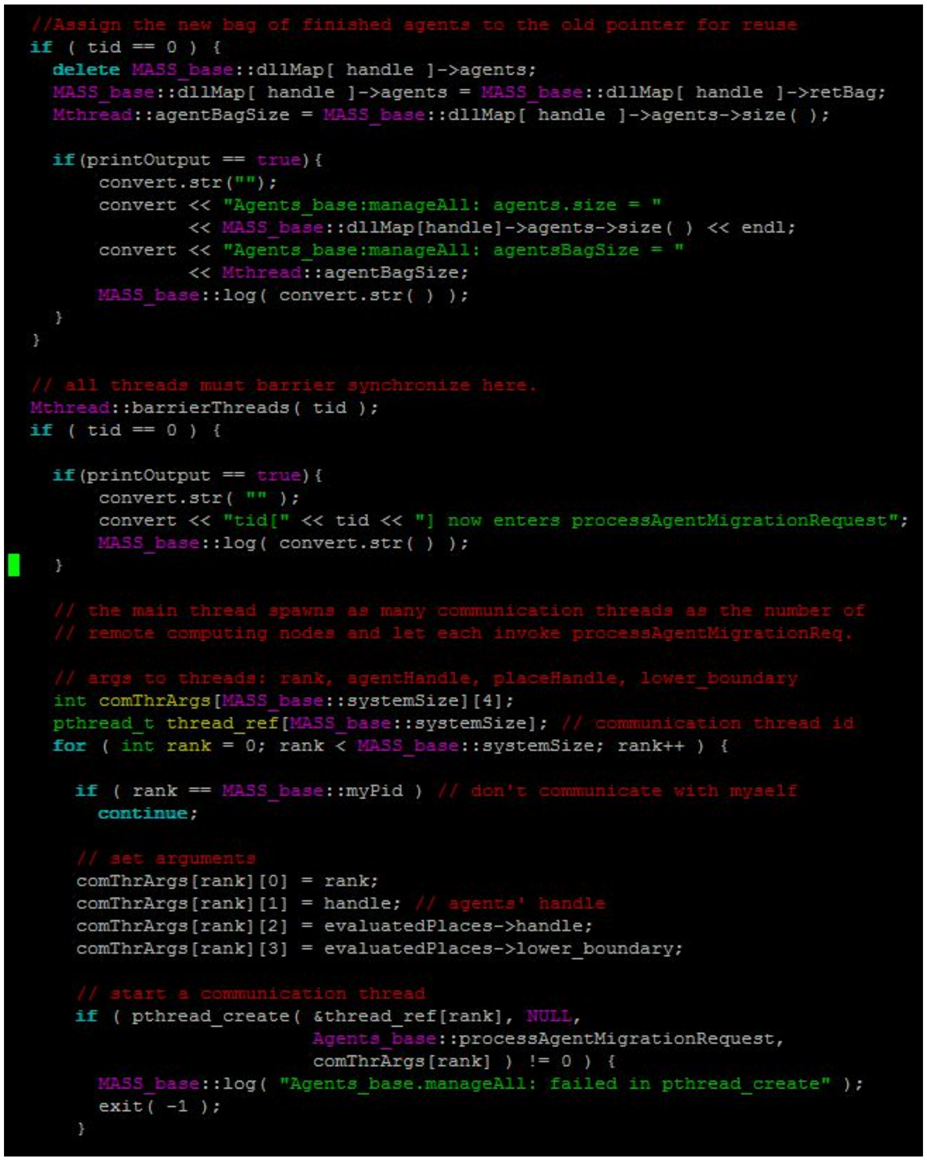
```cpp
//Lock here
pthread_mutex_lock(&MASS_base::request_lock);
retBag->push_back(addAgent);
pthread_mutex_unlock(&MASS_base::request_lock);

//Push the pointer copy into the current Agent's place location
pthread_mutex_lock(&MASS_base::request_lock);
evaluationAgent->place->agents.push_back((MObject*) addAgent);
pthread_mutex_unlock(&MASS_base::request_lock);

// initialize this child agent's attributes:
addAgent->agentsHandle = evaluationAgent->agentsHandle;
addAgent->placesHandle = evaluationAgent->placesHandle;
addAgent->agentId = currentAgentId++;
addAgent->index = evaluationAgent->index;
addAgent->place = evaluationAgent->place;
addAgent->parentId = evaluationAgent->agentId;

//Decrement the newChildren counter once an Agent has been spawned
evaluationAgent->newChildren--;
childrenCounter--;
```

Figure 10 – Spawn, Agents_base.cpp



```cpp
//Kill() Check
if(printOutput == true){
    convert.str("");
    convert << "Agent_base::manageALL: Thread " << tid
            << " check " << evaluationAgent->agentId << "'s alive = "
            << evaluationAgent->alive;
    MASS_base::log(convert.str());
}
if(evaluationAgent->alive == false){

  //Get the place in which evaluationAgent is 'stored' in
  Place *evaluationPlace = evaluationAgent->place;

  // Move through the list of Agents to locate which to delete
  // Do so non-interruptively.
  pthread_mutex_lock(&MASS_base::request_lock);
  int evalPlaceAgents = evaluationPlace->agents.size();

  for( int i = 0; i < evalPlaceAgents; i++){

    //Type casting used so we can compare agentId's
    MObject *comparisonAgent = evaluationPlace->agents[i];
    Agent *convertedAgent = static_cast<Agent*>(comparisonAgent);

    // Check the Id against the ID of the agent to be removed.
    // If it matches, remove it Lock
    if((evaluationAgent->agentId == convertedAgent->agentId) &&
       (evaluationAgent->agentsHandle == convertedAgent->agentsHandle)){
      evaluationPlace->agents.erase( evaluationPlace->agents.begin() + i );

      if(printOutput == true){
          convert.str("");
          convert << "Agent_base::manageALL: Thread " << tid
                  << " deleted " << evaluationAgent->agentId
                  << " from place[" << evaluationPlace->index[0]
                  << "][" << evaluationPlace->index[1] << "]";
          MASS_base::log(convert.str());
      }
      break;
    }
  }

  pthread_mutex_unlock(&MASS_base::request_lock);

  //Delete the agent and its pointer to complete the removal
  //delete &evaluationAgent;
  delete evaluationAgent;
  continue;
}
```

Figure 11 – Kill, Agents_base.cpp



```
//Assign the new bag of finished agents to the old pointer for reuse
if ( tid == 0 ) {
  delete MASS_base::dllMap[ handle ]->agents;
  MASS_base::dllMap[ handle ]->agents = MASS_base::dllMap[ handle ]->retBag;
  Mthread::agentBagSize = MASS_base::dllMap[ handle ]->agents->size( );

  if(printOutput == true){
      convert.str("");
      convert << "Agents_base:manageAll: agents.size = "
              << MASS_base::dllMap[handle]->agents->size( ) << endl;
      convert << "Agents_base:manageAll: agentsBagSize = "
              << Mthread::agentBagSize;
      MASS_base::log( convert.str( ) );
  }
}

// all threads must barrier synchronize here.
Mthread::barrierThreads( tid );
if ( tid == 0 ) {

  if(printOutput == true){
      convert.str( "" );
      convert << "tid[" << tid << "] now enters processAgentMigrationRequest";
      MASS_base::log( convert.str( ) );
  }

  // the main thread spawns as many communication threads as the number of
  // remote computing nodes and let each invoke processAgentMigrationReq.

  // args to threads: rank, agentHandle, placeHandle, lower_boundary
  int comThrArgs[MASS_base::systemSize][4];
  pthread_t thread_ref[MASS_base::systemSize]; // communication thread id
  for ( int rank = 0; rank < MASS_base::systemSize; rank++ ) {

    if ( rank == MASS_base::myPid ) // don't communicate with myself
      continue;

    // set arguments
    comThrArgs[rank][0] = rank;
    comThrArgs[rank][1] = handle; // agents' handle
    comThrArgs[rank][2] = evaluatedPlaces->handle;
    comThrArgs[rank][3] = evaluatedPlaces->lower_boundary;

    // start a communication thread
    if ( pthread_create( &thread_ref[rank], NULL,
                         Agents_base::processAgentMigrationRequest,
                         comThrArgs[rank] ) != 0 ) {
      MASS_base::log( "Agents_base.manageAll: failed in pthread_create" );
      exit( -1 );
    }
```

Figure 12 – MigrateAll's send and receiving of Agents, Agents_base.cpp

Git Backup

One of the advantages this quarter was the implementation of the Git backup service when running the C++ library. This allows us to 'check out' copies of the C++ library, work on them, and then check them back into the system, which allows for many developers to be working on code at the same time, and with the built-in revision control that Git uses, there will be no problem with newer code being overwritten.

To use the library, one must first check out a copy. By designating the current working directory as the source, you can then access the origin with simple commands. To check out the library to your local machine, refer to Matt Sell's guide in the dslab folder. Once you have established your checkout directory and have a local copy, committing them to the origin are simple. To see what files are in need of being pushed, type 'git status.' This will show all the files that need to be committed. To add one of those files to the commit, type 'git add ' and the file name after it. Do this for every file that needs to be updated.

Once all files are added (that were shown in git status), then you need to commit these changes. To do this, type 'git commit –m " ",' where the inside of the quotation marks are messages detailing what was changed. For example, to commit a change which altered the test program, you would type 'git commit –m " Moved Nomad spawning to line 34 to check for spawn bugs" '. This allows the system to have a record of what was changed, and why.

Finally, when you are ready to push these to the origin, simply type 'git push origin source'. This will push all the commits you have done to the origin, and will give you any errors that may arise (if someone pushed a new change before you, you will need to merge the code before pushing). Note that if you and another person worked on the same lines of code, and changed it in different ways, a merge will not help, and you may need to work together to figure out which code will be saved and pushed.

Analysis and Results:

As of the writing of this paper, the code created will now completely compile and run the basic Wave2d program with no errors. All implemented methods have been tested against basic unit tests in Nomad and Wave2d. Output for CallAll and ManagaAll that return values have been checked, to be sure that the program is returning the correct values. Spawn, Migrate, and Kill were tested individually, and the minor combinations of the tests were performed. More advanced testing will be needed to determine if any bugs exist.

As mentioned above, bcopy was a depreciated function call, and I went ahead and made the change to the memcpy function instead. Before doing so, I ran the MASS library as-is and stored the results. After the copy (and argument switch), I ran the library again and used a diff on the output. No notable changes were detected, so performance was unchanged with the new updated function call.

Though we have used basic unit tests, more will be needed to confirm all bugs have been dealt with. The CSS 534 class is currently working with the code in one of their homework assignments, and all feedback will be used to further enhance the library. Another student is working with performance testing, and will be providing feedback about our MASS library vs. an MPI library.