Benchmarking MASS C++: An Overview of the Current Programs and Current Plan for Improvement

Table of Contents

Purpose

The purpose of my research project is to qualitatively and quantitatively compare the three multi-agent simulation platforms MASS C++, RepastHPC, and FLAME by accomplishing the following tasks:

- investigating, correcting, and rewriting as necessary the following seven benchmarks:
  - *Game of Life*
  - *Brain Grid*
  - *Tuberculosis*
  - *Social Networks*
  - *Bail In/Bail Out*

    - *MatSim*
    - *VDT*

- measuring the execution performance and runtime of the benchmark programs across the three simulation platforms

- conducting a quantitative analysis of the benchmarks using the following criteria:
    - Lines of code for agent descriptions, spatial descriptions, and overall benchmark programs
    - Percentage of parallelization boilerplate code
    - User involvement/level of needed knowledge to create new simulations

- co-authoring a journal paper with my advisor

My goal for winter quarter was to correct and rewrite the benchmarks for MASS C++.

Summary of Progress

Currently, the following benchmarks have been completed for MASS C++:
- *Game of Life*
- *Tuberculosis*

The following benchmarks are in progress for MASS C++:
- *Social Networks*
- *Brain Grid*

These programs will be unit-tested near the end of the research project.

Concerning the specifications for the benchmarks, the MatSim specification was altered to include details for the random road system generation/initialization program, and the updated version was sent to Dr. Fukuda for review.

MASS C++ During the Time of This Work

The following outline details the version of MASS C++ used to edit, create, run, and test the benchmarks:
- Game of Life - ‘dev’ branch version
- Tuberculosis - ‘master’ branch version
- Brain Grid (in-progress) - ‘dev’ branch version

During this quarter, a single node with varying numbers of threads was used, as MASS C++'s multi-node capabilities were being repaired, having been lost due to an OS change at the beginning of the quarter.

A Brief Comparison of the Programmability of MASS C++ and FLAME

From the user perspective, writing an agent-based model (ABM) for MASS C++ and for FLAME differ the most when considering the design of the model. This is because FLAME only utilizes Agents, and MASS C++ uses both Agents and Places.

In FLAME, a major difficulty is translating a simulation that considers space and relative location to its model. Additionally, tracking state changes such as growth and distance between Agents becomes an expensive transaction as Agents, whether dynamic or static, can only communicate using messages. On top of this, Agents will look at all existing messages of the specified type that exists globally if filters are not used.

For non-computing scientists, using FLAME to simulate certain models that need strong spatial awareness, such as Tuberculosis, may find it difficult to translate certain aspects of their simulation to a purely Agent-based platform.

In MASS C++, the use of Places simplifies spatial consideration and state changes. Growth and movement are easily tracked by storing data on a derived Place class and checking the state of other Places using the current Place as a medium. Agents are used to traverse Places and collect and modify data.

One possible difficulty that could be foreseen for non-computing scientists could be determining which part of their models should be Agents and which part should be Places. Because Agents aren't able to communicate with each other, the user must use Places to store any state or message that must be communicated to other Places or Agents.

A non-computing scientist may assume that any live entity in their simulation (a neuron, a bacteria, an animal, etc.) should be represented by a derived Agent class, when in reality, MASS C++'s programming model would be more adept to handle a different organization of information - for example, instead of the entire neuron being an Agent, using Agents only to represent the growing ends of a neuron and using these Agents to change the state of the Places it traverses over.

Completed and In-Progress Benchmarks

Game of Life

The specification for this benchmark is the rules for Conway's Game of Life.

The simulation program may be found at the following address in the repository *mass_cpp_appl* on the "SARAH-MASS-GAME-OF-LIFE" branch:

https://bitbucket.org/mass_application_developers/mass_cpp_appl/src/SARAH-MASS-GAME-OF-LIFE/Benchmarks/MASS/MASS_GameOfLife/

This program was originally written by Craig Shih and edited by me.

Simulation Program

In this program, a 2-D, square simulation grid of size 'n' by 'n' is initialized, where each square in the grid is represented by a *Life* Place. Each *Life* Place can be considered a static agent; each *Life* does not change position but does change state as each executes a sequence of state functions for each iteration.

'n' side length of the grid and number of iterations to run the simulation are specified by the user as input parameters.

Each *Life* is initialized to be alive or dead, with a random, deterministic 50% chance of being initialized as alive. Then, each *Life* checks its Moore neighbors in the state function *getBoundaryHealthStatus()* and sets its next state (alive or dead) based on Conway's Game of Life rules in the function *computeDeadOrAlive()*.

The *Life* Places use the MASS C++ Place function *getOutMessage()* to check their neighbors' statuses, and Places.*exchangeBoundary()* was used to update the shadow space with a boundary width of 1.

Output

The output for the simulation of a 10 by 10 grid for 10 iterations using a single node is shown in Figure 1.

Figure 1. Output for MASS C++ Game of Life for a grid size of 100 for 10 iterations

Runtime Comparison to FLAME Game of Life

This program and the FLAME version were both used to simulate a grid size of 10,000 for 10 iterations, and the runtimes are as follows:

- MASS C++ (1 node, 1 thread)
    - 0.431231 seconds

- FLAME (1 node)
    - 12.066 seconds

The FLAME version takes an order of magnitude longer to run as compared to the MASS C++ version.

An important note to make for this result is that, for the FLAME version, filters were not used in the XML model template. This probably made the FLAME version unnecessarily slower.

Tuberculosis

The specification for this benchmark can be found at the following address in the *mass_cpp_appl* repository on the "master" branch:

https://bitbucket.org/mass_application_developers/mass_cpp_appl/src/master/Benchmark_Specifications/

The code for this program can be found at the following address in the same repository on the "SARAH-MASS-TUBERCULOSIS" branch:

https://bitbucket.org/mass_application_developers/mass_cpp_appl/src/SARAH-MASS-TUBERCULOSIS/Benchmarks/MASS/MASS_Tuberculosis/Sarah_V2/

This program was entirely rewritten by me to follow the specification.

Simulation Program

This program was designed to be as similar as the FLAME version as possible.

In this simulation, a 2-D square simulation grid of size 'n' by 'n' is initialized, with each grid unit representing a *TB_Place*, or place in human lung tissue. 'n' grid side length and number of iterations to run are specified by the user as input parameters.

Initialization

Four *TB_Places* are chosen to be blood vessels (entry point from which macrophages and t-cells can enter the simulation space). These Places are located in the center of each quadrant in the grid. One *TCell* Agent and one *Macrophage* agent are spawned for each blood vessel and are migrated to those Places in order to spawn the necessary Agents. Then, 100 *Macrophage* Agents are spawned deterministically randomly over the grid.

Simulation

For 'day' number of user specified iterations, the following sequence of events occurs each iteration:

1. Bacteria grow every bacterialGrowth number of iterations. On these iterations, the shadow space of width 1 is updated. Then, each *TB_Place* checks their neighbors' out_messages for their bacterial state.
2. Time is advanced by decrementing each *TB_Place's* chemokine level by 1 if greater than 0 and updating this *TB_Place*'s bacterial state to reflect that of its neighbors. If their neighbor has bacteria, this *TB_Place* updates its own state to containing bacteria.
3. Which immune cells to spawn at the end of the iteration through the blood cells is then determined. These *TB_Places* are marked to spawn either a *TCell* or *Macrophage* depending on the day number and the current state of the *TB_Place*.

   a. The blood vessel may spawn a *TCell* if the day is greater than or equal to tCellEntrance and if there are no *TCells* currently on that particular *TB_Place*.

   b. The blood vessel may spawn a *Macrophage* if there currently are no *Macrophages* on that particular *TB_Place*.

4. *TCell* and *Macrophage* Agents move to the neighboring Place with the highest chemokine level by updating the shadow space using Places.*exchangeBoundary()* and each *TB_Place* checking their neighbors' out_message for their chemokine levels.

   a. Only one *TCell* Agent and *Macrophage* Agent each may be contained by a *TB_Place* at a time. Therefore, *TB_Places* approve one request for movement for each type of immune cell and deny the rest.

5. Next, *TCell* Agents kill any chronically infected *Macrophage* Agents. *Macrophage* Agents kill any bacteria present on the current *TB_Place* and change state accordingly to infected, activated, chronically infected, or dead depending on their internal bacterial state, whether they just killed a bacteria, and if there is *TCell* collocated on this Place.

   a. Dead *Macrophages* are removed from the simulation and *macrophages*.manageAll() is then called.

6. The chemokine levels for each *TB_Place* is updated. If a *Macrophage* that is infected, activated, or chronically infected is present on this *TB_Place* or any of its neighbors, this Place updates its chemokine level to the maxChemokine level.

   a. The neighboring Places' chemokine level is checked by swapping the shadow space of width 1 using *TB_Place*.exchangeBoundary() and checking the neighbors' out_messages.

7. If a *Macrophage* has died because of its internalBacteria exceeding the bacterialDeath number, bacteria is spread to the neighboring *TB_Places* using *TB_Place*.exchangeBoundary() and checking the neighbors' out_messages.

8. Finally, new *TCell* and *Macrophage* Agents are spawned at the blood vessels as determined at Step (3). These Agents are spawned by the invisible spawner Agents that are located at the blood vessels that do not move, change state, or occupy space.

If graphical output is needed, all *TB_Places*, *Macrophages*, and *TCells* write out their state to XML for the TBOut.java program to display at the end of each iteration.

Runtime Comparison to FLAME Tuberculosis

For the simulation of a grid size of 10,000 TB_Places for 10 iterations, the runtime for MASS C++ (1 node, 1 thread) was 1.863309 seconds. The runtime for FLAME (1 node) was 2.964 seconds.

The MASS C++ port took about 63% of the time that FLAME needed to run the same simulation - quite significantly faster.

An important note regarding this data - the FLAME Tuberculosis used filters to limit the messages received by each Place to its Moore's area. Although the FLAME Tuberculosis program was about an order of magnitude longer than the FLAME Game of Life program, it was about an order of magnitude faster.

Even with the use of filters, the MASS C++ simulation port was significantly faster than the FLAME version of Tuberculosis.

Self-Organizing Neural Network

The specification for this agent-based model is located in the mass_cpp_appl repository on the "master" branch:

https://bitbucket.org/mass_application_developers/mass_cpp_appl/src/master/Benchmark_Specifications/

This program is currently in-progress, and has not been finished yet.

Simulation Program

Currently for this program, a square simulation grid of 'n' by 'n' size is generated, with each square in the grid represented by a *Brain_Place* Place.

Initialization

Each *Brain_Place* has a pointer to a Neuron object. During initialization, each Brain_Place spawns the cell body (soma) of a neuron with a probability of E + I + N.

If the Place does not spawn a soma, its Neuron object is null, and it is considered empty (no neuron or cell body parts occupying this Place). If the Place does spawn a soma, this Neuron is assigned to be an excitatory Neuron with E probability, an inhibitory Neuron with I probability, or a neutral Neuron with N probability.

If the Place has the soma of a Neuron, a GrowingEnd Agent is sent to that Place. This Agent will spawn up to 7 GrowingEnd Agents for dendrites at random iterations and 1 GrowingEnd Agent for the axon at a random iteration. This Agent resides on the Brain_Place that contains the soma and does not move, existing only to spawn new GrowingEnds as needed.

Looking Forward: Plan for Spring Break and Spring Quarter

Goals for Spring Break

My plan for spring break is to complete the MASS C++ Self Organizing Neural Network program and visualization and to send a video clip of the demo to Dr. Fukuda documenting its performance.

Additionally, I plan to work on the sequential C++ initialization program for the MatSim benchmark so I may complete the MASS C++ MatSim port and test my FLAME MatSim port.

Goals for Spring Quarter

My plan for spring quarter is to write the specification documents for the following ABMS:

- *Bail-In, Bail-Out*
- *VDT*

I also plan to complete the following benchmarks for MASS C++:

1. *MatSim*
2. *Bail-In, Bail-Out*
3. *VDT*

Finally, I plan to verify the MASS C++ Social Networks port completed by Josh Landron this past quarter.