FLAME Benchmarking for MASS C++: An Overview of the Current Benchmarks and Next Steps

University of Washington Bothell
Sarah Panther
CSS 499: Undergraduate Research
November 12, 2019

Table of Contents

Purpose.......... *3*

Completed and In-Progress Benchmarks.......... *3*

Game of Life.......... **4**

Specification.......... 4

Edits made.......... 4

Output.......... 4

Social Networks.......... **6**

Specification.......... 6

Model template.......... 7

Functions.......... 7

Initialization program.......... 7

Output.......... 8

Tuberculosis.......... **9**

Specification.......... 9

Model template.......... 9

Functions.......... 11

Initialization program.......... 12

Output.......... 12

Self-Organizing Neural Network.......... **14**

Specification.......... 14

Model template.......... 14

Functions.......... 15

Initialization program.......... 17

Output.......... 17

MatSim.......... **18**

Specification.......... 18

Initialization program.......... 18

Next Steps.......... *19*

Goals for Winter Break.......... 19

Plan for Winter Quarter 2020.......... 19

References.......... *19*

Purpose

The purpose of my undergraduate research project is to qualitatively and quantitatively compare the three agent-based model (ABM) simulation platforms - MASS C++, RepastHPC, and FLAME.

This will be accomplished by writing the specifications for seven ABM benchmarks; creating, editing, and running the benchmarks; evaluating the platform's performance and runtime; qualitatively analyzing the ported programs' code; and co-authoring a journal paper with my research advisor.

The seven benchmarks are as follows:

- *Game of Life*
- *Tuberculosis*
- *Social Networks*
- *Brain Grid*
- *MatSim*
- *Bail In/Bail Out*
- *VDT*

This quarter and the previous summer session was focused on writing specifications for the benchmarks and re-writing the FLAME benchmark programs.

Completed and In-Progress Benchmarks

Currently, the following benchmarks have been completed for FLAME:

- *Game of Life*
- *Tuberculosis*
- *Social Networks*
- *Brain Grid*

These benchmarks will be unit-tested to check for errors during the end portion of this project.

The benchmark *MatSim* is currently in progress.

Game of Life

This benchmark's specification may be found at the following Bitbucket address on the "*master*" branch:

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

The code for this benchmark can be found at the following Bitbucket address on the "*master*" branch:

https://bitbucket.org/mass_application_developers/mass_cpp_appl/src/master/Benchmarks/FLAME/Game%20of%20Life/

Specification

The specification for this ABM was not student written; the simulation simply follows the rules of Conway's Game of Life.

Edits made

This ABM was edited during the summer quarter and not re-written as it was mostly correct. The changes made were outlined in the term paper for CSS 499: Spring 2019.

There are further changes that could be made to clean up the code (i.e. refactoring the initialization program, moving the logic of checking to see if the message is from a neighbor to a filter in the model template, etc.), and can be made during a future quarter.

Output

The output shown in Figure 1 through 3 was demoed during this quarter's presentation for a grid of 400 cell agents for 5 iterations:

```
~/C/GameOfLife ❯❯❯ ./main 5 0.xml
[libmboard] Version        : 0.3.1 (SERIAL)
[libmboard] Build date     : Sun Mar 10 12:54:00 PDT 2019
[libmboard] Config options :  '--with-mpi=/usr/lib/x86_64-linux-gnu/openmpi/'

[libmboard] +++ This is a DEBUG version +++
[libmboard] <settings> MBOARD_MEMPOOL_RECYCLE = 0 (default)
[libmboard] <settings> MBOARD_MEMPOOL_BLOCKSIZE = 512 (default)
FLAME Application: Game of Life
Debug mode enabled
Iterations: 5
xml: Geometric partitioning
Reading initial data file: 0.xml
Reading environment data from: 0.xml
Reading agent data from: 0.xml
output: type='snapshot' format='xml' location='' period='1' phase='0'
xdiv=1 ydiv=1
Partition 0: -999999.123456, 999999.123456, -999999.123456, 999999.123456
Iteration - 1

 1  0  0  0  0  0  0  0  0  1  1  0  1  1  1  0  1  1  0  0
 1  0  0  0  0  0  1  0  0  0  1  0  0  0  0  0  0  1  1  0
 1  0  0  0  0  0  0  0  1  0  0  0  0  0  0  1  0  1  0  0
 0  1  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 1  0  0  0  0  0  0  0  1  0  0  0  0  0  0  1  0  0  0  1
 1  0  0  0  0  0  0  1  1  0  0  0  0  0  0  0  0  0  0  0
 1  0  0  0  0  0  1  0  0  0  0  0  0  0  0  1  0  0  0  0
 0  0  0  0  0  0  0  0  0  1  1  0  0  0  0  1  0  0  0  1
 0  1  0  0  0  1  1  0  0  0  1  1  0  1  0  1  0  0  0  0
 0  0  0  1  0  1  0  0  0  0  0  0  1  1  0  1  1  0  0  0
 0  1  0  0  0  0  0  0  0  0  0  0  0  1  1  1  1  0  0  1
 1  0  0  0  0  1  0  0  0  0  0  0  0  0  0  0  0  1  0  1
 1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 1  0  1  0  0  1  1  0  0  0  0  0  0  0  0  0  0  0  0  1
 0  1  0  0  0  1  1  0  0  0  0  0  0  0  0  0  0  0  0  1
 0  0  0  0  1  0  1  0  0  0  0  1  1  1  1  0  1  0  0  0
 0  0  0  0  1  1  1  0  0  0  0  0  0  0  1  1  0  0  1  1
 0  0  0  0  0  1  1  0  0  0  0  0  1  0  0  0  0  0  1  1
 1  1  1  0  1  1  0  0  0  0  1  0  1  0  0  0  0  1  1  0
 0  0  0  0  0  0  0  0  0  1  0  1  0  0  1  0  1  1  0  0 Iteration - 2
```

Figure 1. *Output of the Game of Life benchmark for a grid of 400 agents for 5 iterations in FLAME serial mode - Iteration 1*

```
0  0  0  0  0  0  0  0  0  1  1  1  0  1  0  0  1  1  1  0
1  1  0  0  0  0  0  0  0  0  1  1  0  1  1  1  0  0  1  0
1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  1  1  0
1  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0  0
1  0  0  0  0  0  0  1  1  0  0  0  0  0  0  0  0  0  0  0
1  1  0  0  0  0  0  1  1  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  1  1  1  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  1  1  0  0  1  1  1  0  0  0  1  1  0  0  0
0  0  0  0  1  1  1  0  0  1  1  1  0  1  0  1  0  0  0  0
0  0  1  0  1  1  1  0  0  0  0  1  0  0  0  0  0  0  0  0
0  0  0  0  1  0  0  0  0  0  0  0  1  1  0  0  0  1  1  0
1  1  0  0  0  0  0  0  0  0  0  0  0  0  1  1  1  0  1  0
1  0  0  0  0  1  1  0  0  0  0  0  0  0  0  0  0  0  1  0
1  0  0  0  0  1  1  0  0  0  0  0  0  0  0  0  0  0  0  0
0  1  0  0  1  0  0  1  0  0  0  0  1  1  0  0  0  0  0  0
0  0  0  0  1  0  0  1  0  0  0  0  1  1  1  0  0  0  1  1
0  0  0  0  1  0  0  1  0  0  0  1  0  0  1  1  0  1  1  1
0  1  0  1  0  0  0  0  0  0  0  1  0  1  0  0  0  0  0  0
0  1  0  0  1  1  1  0  0  0  1  0  1  1  0  0  1  0  0  1
0  1  0  0  0  0  0  0  0  0  1  1  0  0  0  0  1  1  1  0 Iteration - 3

0  0  0  0  0  0  0  0  0  1  0  1  0  1  0  1  1  1  1  0
1  1  0  0  0  0  0  0  0  1  0  1  0  1  1  1  0  0  0  1
0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  1  0  1  0
1  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0  0
0  0  0  0  0  0  0  1  1  0  0  0  0  0  0  0  0  0  0  0
1  1  0  0  0  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  1  0  0  0  0  0  0  1  1  0  1  1  1  0  0  0
0  0  0  1  0  0  0  1  0  1  0  0  0  0  1  1  1  0  0  0
0  0  0  0  0  0  1  0  0  0  0  1  0  1  1  0  0  0  0  0
0  1  0  1  1  0  0  0  0  0  0  0  1  1  1  1  1  1  1  0
1  1  0  0  0  1  0  0  0  0  0  0  0  1  1  1  1  0  1  1
1  0  0  0  0  1  1  0  0  0  0  0  0  0  0  1  0  1  0  0
1  1  0  0  1  0  0  1  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  1  0  0  1  0  0  0  0  1  0  1  0  0  0  0  0
0  0  0  1  1  1  1  1  1  0  0  1  0  0  0  1  0  1  0  1
0  0  0  1  1  0  0  0  0  0  0  1  0  0  0  1  0  1  0  1
0  0  1  1  0  0  1  0  0  0  1  1  0  1  0  1  1  1  0  1
1  1  0  0  1  1  0  0  0  0  1  0  0  1  0  0  1  0  1  0
0  0  0  0  0  1  0  0  0  0  1  1  1  0  0  0  1  1  1  0 Iteration - 4
```

Figure 2. *Output of the Game of Life benchmark for a grid of 400 agents for 5 iterations in FLAME serial mode - Iterations 2 and 3*

Figure 3. *Output of the Game of Life benchmark for a grid of 400 agents for 5 iterations in FLAME serial mode - Iterations 4 and 5*

Social Networks

This benchmark's specification may be found at the following Bitbucket address on the "*master*" branch:

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

The code for this benchmark can be found at the following Bitbucket address on the "*SARAH-SOCIAL-NETWORKS-V2*" branch:

https://bitbucket.org/mass_application_developers/mass_cpp_appl/src/SARAH-SOCIAL-NETWORKS-V2/Benchmarks/FLAME/Social%20Networks/Version_2/

Specification

The specification for this benchmark was re-written during the summer quarter and was reviewed Dr. Fukuda.

The purpose of this ABM is to model the degrees of friendship between people in a social network.

During initialization, a group of *N* people with *k* number of first degree, bi-directional friendships is deterministically created.

During the simulation, each person checks their first degree friends' friends and adds anybody the current person isn't already friends with to their own list of friends. This expands everyone's social network by one degree every iteration.

Once the desired number of iterations have passed, the friends list of each person is printed out in order of degree of friendship.

Model template

The Social Networks XML model template consists of one type of agent - the Person Agent.

The Person agent consists of the following elements:

- a unique ID
- an array of first degree friends' ID numbers
- an array of connections (a connection element consists of the friend's id and degree of friendship from the current Person)
- the current degree of friendship found and contained in the connections array

The Person agent completes two state functions during one iteration: posting the friends in their social network onto the FLAME message board and finding the next degree of friendship.

The section "Functions" summarizes these functions.

Functions

post_found_friends() is the first state function in the finite state machine (FSM) Person agent. This function simply iterates over each connection in the current Person's connections array and posts their friends and degree of friendship as a found_friend_message to the messageboard for all the other Person agents to read.

find_next_degree_friends() is the last state function. This function reads through the board of found_friend_messages that are filtered to include only messages sent from the current Person's first degree friends list. It then adds the friends in the message to this Person's connections array if the friend is not already in the array and is also not themselves.

Initialization program

The initialization *init_state_SocialNet.java* creates a *k*-connected graph by connecting the current Person to *k*/2 people to their right (if the list of people were viewed as a circular array) and *k*/2 people to their left. If *k* is an odd number, another connection is made with the person across from this current Person. These connections are this Person's first degree friends.

The Person agents are then written to the XML file "0.xml".

Output

The output shown in Figure 4 was demoed during this quarter's presentation for a group of 10 people with *k* = 2 first degree friends to find the 3rd degree of connectedness for three iterations (only two were necessary to find the desired degree) :

```
0> Processor name: sarah-HP-Laptop-14-df0xxx
0> No of agents on node: 10
0> Person agents on node: 10
0> Agent total check: 10
P0:1
P0:2
Person agent 9's friends
              9's degree     1 : 0 8
              9's degree     2 : 7 1
              9's degree     3 : 2 6
Person agent 8's friends
              8's degree     1 : 7 9
              8's degree     2 : 0 6
              8's degree     3 : 5 1
Person agent 7's friends
              7's degree     1 : 6 8
              7's degree     2 : 9 5
              7's degree     3 : 4 0
Person agent 6's friends
              6's degree     1 : 5 7
              6's degree     2 : 8 4
              6's degree     3 : 3 9
Person agent 5's friends
              5's degree     1 : 4 6
              5's degree     2 : 7 3
              5's degree     3 : 2 8
Person agent 4's friends
              4's degree     1 : 3 5
              4's degree     2 : 6 2
              4's degree     3 : 1 7
Person agent 3's friends
              3's degree     1 : 2 4
              3's degree     2 : 5 1
              3's degree     3 : 0 6
Person agent 2's friends
              2's degree     1 : 1 3
              2's degree     2 : 4 0
              2's degree     3 : 9 5
Person agent 1's friends
              1's degree     1 : 0 2
              1's degree     2 : 3 9
              1's degree     3 : 8 4
Person agent 0's friends
              0's degree     1 : 1 9
              0's degree     2 : 8 2
              0's degree     3 : 3 7
P0:3
Execution time - 0:00:001 [mins:secs:msecs]
~/C/SocialNetworks ❯❯❯
```

Figure 4. *Output of the Social Networks benchmark for a 10 people with 2 first degree friends to find the 3rd degree of connectedness for 3 iterations in FLAME serial mode*

Tuberculosis

This benchmark's specification may be found at the following Bitbucket address on the "*master*" branch:

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

The code for this benchmark can be found at the following Bitbucket address on the "*SARAH-TUBERCULOSIS-VERSION2*" branch:

https://bitbucket.org/mass_application_developers/mass_cpp_appl/src/SARAH-TUBERCULOSIS-VERSION2/Benchmarks/FLAME/Tuberculosis/TB_version2/

Specification

The specification for this ABM was written by Brian Tran and I during the summer quarter and reviewed by Dr. Fukuda. We based the model on the paper *Data-Parallel Algorithms for Agent-Based Model Simulation of Tuberculosis On Graphics Processing Units* [1].

In this simulation, a 2-d grid is created that represents human lung tissue. Bacteria and macrophages are then spawned. Bacteria grow at a predetermined rate, and macrophages eat and react to the bacteria and other macrophages if they are infected.

Each time step, macrophages are spawned from four entry points on the grid that represent blood vessels. At a predetermined time step, T-cells enter the simulation via the blood vessel and react to the macrophages, activating the infected ones and bursting the chronically infected.

Model template

The Tuberculosis XML model template consists of three main types of agents - the Place Agent, the Macrophage agent, and the TCell agent.

It also has two helper agent types : the DayTracker - an iteration tracker, and the AgentFactory - an agent spawner that tracks the next available Macrophage and TCell IDs.

The Place agent represents one square of the grid simulation space and contains the state of that area. It consists of the following elements:

- a unique ID number
- an x coordinate

- a y coordinate
- a bacteria flag - whether or not there is a live bacteria on this Place
- a chemokine level - these levels can be 0, 1, or 2 (the max level)
- a blood vessel flag - whether or not this Place is an entry point
- a macrophage flag - whether a macrophage is present
- a t-cell flag - whether a t-cell is present
- the IDs of the Places neighboring it (in its Moore's area)

The Macrophage agent represents a human macrophage immune cell and consists of the following variables:

- a unique ID number (unique to the other macrophages)
- an x coordinate
- a y coordinate
- the current state (i.e. ACTIVE, RESTING, etc.)
- a located-on-bacteria flag - whether or not this macrophage is located on a Place with bacteria
- the day this macrophage became infected (-1 if it's not infected)
- the number of intracellular bacteria it contains
- an array of Place ID's this macrophage is neighboring to which it can move or spread its bacteria or chemokines if necessary

The TCell agent represents a generic human t-cell and is made up of the following characteristics:

- a unique ID number (unique to other T-cells)
- an x coordinate
- a y coordinate
- array of neighboring Place IDs - the area to which this T-cell can move to the next time step

Functions

The Place agent has the following six state functions:

- *write_place()* : writes its state as a placeStart message
- *decay_chemokine_and_grow_bacteria()* : changes its internal state as time progresses
    - chemokine levels decrease
    - bacteria is grown if necessary by checking the time step and neighboring Place's placeStart messages
    - its chemokine level is posted using a placeDecayAndGrow message
- *cell_recruitment()* : TCells and macrophages are spawned at the appropriate time steps if this Place is an entry point
- *approve_macrophage_movement()* :
    - if there are no macrophages on this Place, it iterates through macroMoveReq messages addressed to it
    - approves one request and denies the rest by posting a grantMacroMove message for each request
- *approve_t_cell_movement()* :
    - if there are no t-cells on this Place, it iterates through the tCellMoveReq messages addressed to it.
    - approves one request and denies the rest by posting an tCellState message for each request
- *react_to_macro_and_tcell()* : updates its current state based on any new t-cell or macrophage presence
    - chemokine level may be maxed to 2
    - bacteria may be cleared (by being eaten by the resident macrophage)

The Macrophage agent has the following three state functions:

- *macrophage_request_move()* :

  - looks through the add_placeDecayAndGrow messages from its neighboring Places to find the Place with the highest chemokine level or, if there are no chemokines in the neighboring Places, a random neighboring Place
  - requests the Place by posting a macroMoveReq message
- *macrophage_move() :* reads the grantMacroMove message addressed to it and either updates its x and y coordinates to the new Place it has moved to and re-calculates its neighbors or does nothing
- *macrophage_react() :*
  - checks if its on the same Place as a t-cell by seeing if there is a tCellState message addressed to the same Place it's currently on
  - it then updates its state and intracellular bacteria as needed. If the macrophage dies, it returns 1 for this function, removing it from the simulation.

The TCell agent has the following two state functions:

- *t_cell_request_move()* :
  - looks at the neighboring Place's chemokine levels by reading through their placeDecayAndGrow messages
  - either requests to move to the Place with the highest level or, if all of the levels are 0, requests a random Place from its neighbors to move to by sending a tCellMoveReq message
- *t_cell_move()* : reads the tCellState message addressed to it and updates its location or does nothing

Initialization program

The initialization program *init_state.py* creates an N by N size grid of Places and spawns 100 macrophages on random spaces. It then creates four bacteria in the center of the grid and four entry points, one in the center of each quadrant.

Finally, it writes the spawned agents and Places to the file "0.xml".

Output

The output was visualized using an adapted version of WOut.java. In this visualization, the resting macrophages are a bright green, the t-cells are bright blue, and the bacteria are black. The chemokine levels are shown as red for the max level (level = 2) and yellow for level = 1.

As macrophages transition from resting to the possible states of infected, activated and chronically infected, their colors change to yellow, light blue, and dark purple, respectively.

Figure 5 shows the end state of visualization that was demoed at the end of summer quarter.

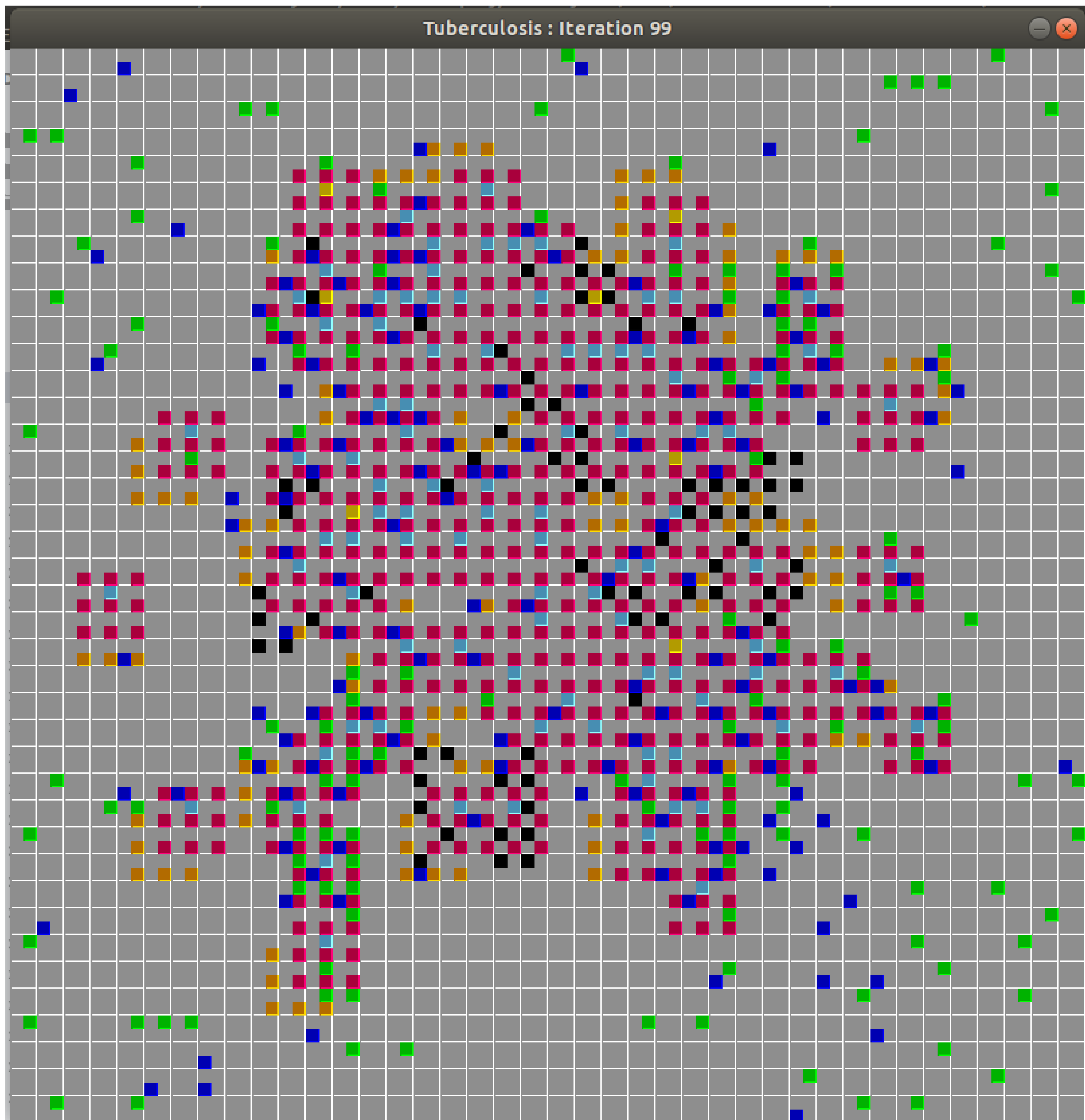

Figure 5. *Output visualization of Tuberculosis using TBOut.java for a grid of 40 by 40 for 100 iterations*

Self-Organizing Neural Network

This benchmark's specification may be found at the following Bitbucket address on the "*master*" branch:

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

The code for this benchmark can be found at the following Bitbucket address on the "*SARAH-BRAIN-GRID-V2*" branch:

https://bitbucket.org/mass_application_developers/mass_cpp_appl/src/SARAH-BRAIN-GRID-V2/Benchmarks/FLAME/Brain%20Grid/Brain_Grid_V2/

Specification

This specification was written by Dr. Fukuda during the summer quarter.

In this simulation, *N* neurons are spawned upon a 2-d grid. Each neuron first only consists of a soma (cell body) and, at random time steps, grows dendrites and an axon.

Once the axon is finished growing , synaptic terminals are spawned. These may make connections with other neurons' dendrites. Likewise, the dendrites of the current neuron are grown and may make connections with other neurons' synaptic terminals.

Once a connection is made, all neurons will sum the received signals and add them to the output sum. Then, the neuron may activate a signal if it is an excitatory type and add it to the output sum it will send to its synaptic terminals.

The output sum is then modified depending on the type of the current neuron. If the value is above the threshold value, it will forward the signal to the other neurons connected to its synaptic terminals.

Model template

The Self-Organizing Neural Network XML model template consists of two main types of agents - the Place Agent and the Neuron agent.

It also has one helper agent type : the IterationTracker.

The Place agent represents one square of the grid simulation space and contains the state of that area. It consists of the following variables:

- a unique ID
- an occupying neuron ID (-1 if a neuron isn't present)
- a neuron part type if there is a neuron occupying this place (synaptic terminal, dendrite, etc.)

The Neuron agent represents a human neuron cell and is made up of the following elements:

- a unique ID
- the type of neuron (excitatory, inhibitory, or neutral)
- sum of signals from the previous iteration. If this is the first iteration, this value is 0.
- array of in-connections (neurons from which this current neuron will receive signals)
- array of out-connections (neurons to which this neuron will send signals)
- Place ID where the soma resides
- array of available directions to grow around the soma - this is where the dendrites and axon will spawn and grow from
- array of Places that neighbor new synaptic terminal growth
    - this is used to filter Place ID's to only track the delta of neuron position and check for new neuronal out-connections
    - cleared and reset at the end of every iteration
- array of Places that neighbor new dendrite growth
    - this is used to filter Place ID's to only track the delta of neuron position and check for new neuronal in-connections
    - cleared and reset at the end of every iteration
- axon state information - position, Place growth requests, branches, etc.
- dendrite array - state and position of this neuron's dendrites
- dendrite Place growth requests

Functions

The Place agent has two state functions:

- *approve_growth_requests()* :
  - reads through growth_request messages
  - if this Place isn't occupied, grants one growth request approval and denies the rest, or ,if this Place is occupied, deny all of the requests
  - post a growth_approval message to approve or deny the requests
- *occupied_post_state()* : if this Place is occupied by a synaptic terminal or a dendrite, post which part is present and the neuron's ID in an occuppied_state message

The Neuron agent has the following six state functions:

- *request_to_grow()* : the neuron attempts to grow by the rules outlined in the specification by requesting the appropriate Place ID's using a growth_request message
- *grow()* : the neuron updates the positions of its parts by reading through the growth_approval messages sent to it and changes the state of its parts accordingly (stop branching, stop growing, etc.)
- *process_connections()* : the neuron adds new connections to its in_connections and out_connections as necessary by reading through its neighboring Place's occupied_state messages and seeing if the correct parts are neighboring each other (its synaptic terminal to another neuron's dendrite or its dendrite to another neuron's synaptic terminal)
- *receive_new_connections()* : add new connections to its in_connections and out_connections as necessary by reading new_connections messages sent to it by other neurons
- *process_and_transmit_signal()* :
  - modulate signals received during the previous time step according to this neuron's type
    - excitatory amplifies the signal
    - inhibitory reduces the signal
    - neutral transmits the signal without a change
  - if this is an excitatory neuron, add 1.0 to the output sum
  - transmit the signal by posting a signal message to this neuron's out_connections
- *receive_signals()* : receive the signal from this neuron's in_connections by reading through the signal messages addressed to this neuron

Initialization program

The initialization program *WriteNeuron.java* creates an *S* by *S* sized grid of Places and populates *N* number of neurons' somas on it randomly. It then assigns a random iteration number for the axon and each dendrite to spawn.

It then writes the neurons and places to the file "0.xml".

Output

The visualization program of this simulation's output was adapted from Dr. Fukuda's WOut.java program.

In the visualization, neurons are displayed as follows:

- somas - dark purple
- excitatory neurons - bright purple/pink
- inhibitory neurons - warm gray
- neutral neurons - light blue

The received and modulated signals are shown as greens that lighten into white as the signal value grows higher.

Figure 6 shows the last iteration of the visualization demoed during this quarter's presentation.

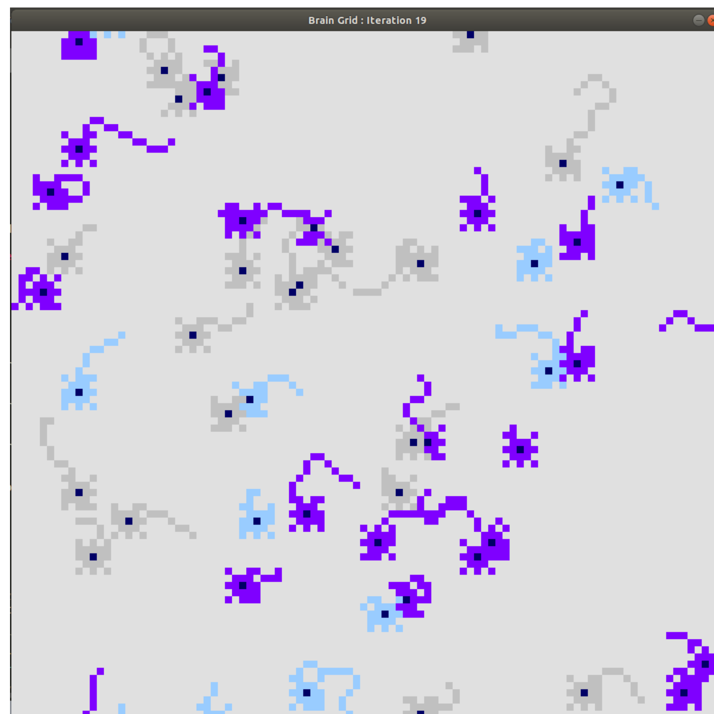

Figure 6. *Visualization of Brain Grid output for a 100 by 100 grid with 50 neurons for 20 iterations*

MatSim

This benchmark is in progress, and its specification and code has not been pushed to Bitbucket currently.

Specification

The rough draft of the specification for this benchmark was written this quarter, and has been reviewed by Dr. Fukuda. Some edits need to be made to the document to reflect what was discussed in the last lab meeting of the quarter.

The purpose of this simulation is to model traffic flow and congestion during morning and evening commutes, when most vehicles have similar origin and destination points.

In the simulation, a graph is created and stored in an adjacency list. Each node represents an intersection, and each unidirectional edge represents a one-way road. The intersections and roads each have a car capacity that they can hold during a single iteration. Roads additionally have a distance weight, or the number of iterations needed for a vehicle to traverse them.

In the morning, the cars will start from a predetermined origin point. They will then proceed to their predetermined destination point located either at the appointed center vertex or nearby as the node's capacities allow. Cars will be aware of and follow their optimal route as computed by an initialization program.

In the evening, they will make the reverse commute from the morning's destination point ("work") to the morning's origin point ("home").

Initialization program

The initialization program to generate the graph/map, cars' origin and destination points, and cars' routes will be a sequential C++ program.

This program will first generate a directed, connected graph with random weights and capacities stored in an adjacency list. Then, the graph will be modified to be strongly connected so the cars will have a guaranteed route between their "home" and their "work". This will be accomplished using Tarjan's algorithm and connecting the strongly connected components.

Then, the shortest paths between each pair of vertices will be calculated using the Floyd-Warshall algorithm - this information will be used to select the cars' origin and destination points as well as compute their routes.

Then, the origin and destination points will be determined by selecting a single node as the center of the graph - vertex 0. The top *X* nodes furthest away from vertex 0 will be selected as the morning origin ("home") points for the car agents. *X* is the number of nodes that will be selected whose capacity will accommodate the desired number of car agents.

Similarly, *Y* number of nodes neighboring vertex 0 will be selected to accommodate the desired number of car agents vertex 0 can not hold - these nodes will be the morning destination point ("work").

Finally, the program will write the graph's adjacency list, the cars' "home" and "work" points, and their morning and evening commute routes to an input file. This input file will be used for the FLAME, MASS C++, and RepsatHPC MatSim simulations.

Next Steps

Goals for Winter Break

My plan for winter break is to finish the MatSim benchmark: the initialization program, the model template, and the function files.

Also, I will add comments to the benchmarks I've written this quarter and push those changes and any additional edits I've made to bring the *mass_cpp_appl* repository up to date

Finally, I plan to work on the VDT and Bail-In, Bail-Out benchmark specifications in preparation for the winter quarter.

Plan for Winter Quarter 2020

During winter quarter, I plan to finish the VDT and Bail-In, Bail-Out benchmark specifications, learn MASS C++, and port all seven benchmarks to MASS C++.

References

[1] D'Souza, R. et al(n.d.). *Data-Parallel Algorithms for Agent-Based Model Simulation of Tuberculosis On Graphics Processing Units*. Michigan Tech. University.