Porting Agent-Based Benchmarks to MASS CUDA

Christopher Sumali

CSS 497 Winter 2023 Term Report

Professor Munehiro Fukuda

March 3, 2023

Table of Contents

1. Introduction

1.1 Motivation

The motivation for this research is to create more complex benchmark applications for MASS CUDA. The reason is because previously developed applications only dealt with static agents, or places, and simple agent movement. Thus, the main objective of this research is to develop two benchmark applications for MASS CUDA: MATSim and Tuberculosis. For a detailed explanation of the differences between these applications and the previous ones, please refer to Section 2.2 and Section 2.3.

Additionally, the newly developed benchmark applications will be used to test the Multi-GPU version of MASS CUDA. These applications will be used to ensure that the Multi-GPU version functions properly and to compare its execution performance with the Single-GPU version, which is expected to be slower.

1.2 Project Goal

The purpose of this research is to verify the accuracy of the MASS CUDA library and measure its performance against other agent-based models using the two benchmark applications. The goal is to identify and resolve any bugs related to the implementation of these applications in the MASS CUDA library.

The performance of these benchmarks will be compared to the MASS C++, RepastHPC, and FLAME versions of the benchmarks. MASS CUDA simulations are expected to have the best performance since it utilizes the processing power of hundreds of GPU cores, compared to the few CPU cores used in MASS C++. The results of the two benchmark applications will be used to support this claim.

2. Background

2.1 MASS CUDA

“CUDA (or Compute Unified Device Architecture) is a parallel computing platform and application programming interface (API) that allows software to use certain types of graphics processing units (GPUs) for general purpose processing, an approach called general-purpose computing on GPUs (GPGPU).”

One way to improve the performance of MATSim is by using CUDA, which is a software library that allows for the parallelization of computations on NVIDIA graphics processing units (GPUs). This can significantly speed up the simulation process, allowing for larger and more complex simulations to be run in less time.

Using MASS CUDA with MATSim requires a system with an NVIDIA GPU and the appropriate drivers installed. The MASS CUDA library can then be integrated into the MATSim codebase, allowing for the parallelization of certain computational tasks. This can lead to significant performance improvements, particularly for simulations involving large numbers of agents.

2.2 Previous Applications in MASS CUDA

There are several benchmark applications that have been developed previously in MASS CUDA. However, these applications are too simple and not complex enough to verify the functionality of MASS CUDA. As a result, this research seeks to develop more complex benchmark applications. The existing applications are Heat2D, BrainGrid, and Sugarscape.

Heat2D simulates the dispersion of heat across a 2D grid. Each unit in the 2D grid is a place which communicates and exchanges information with neighboring places. The problem is that this application only involves places and does not have any complex agent behavior.

BrainGrid simulates a neural network wherein different types of neurons grow axons and dendrites used to communicate with each other. Same as Heat2D, BrainGrid only utilizes places to simulate this neural network. Thus, it has the same problem that it only involves places and no complex agent behavior.

Sugarscape simulates the interaction between ants and a sugar mound. Ants are dynamic agents that traverse mounds, consume sugar, and look for areas in which there is more sugar. They migrate from place to place involving some simple agent movement. This is one of the simplest applications that involve static and dynamic agents.

2.3 Reasons for Selected Applications

MATSim the flow of traffic and congestion formed during the morning commute when the majority of single household cars have a similar origin and destination points. This requires synchronization to maintain the capacity of each intersection and involves the congestion of agents on a single place. Unlike Sugarscape, the agents in MATSim have predetermined routes, instead of random reward-based movement.

Tuberculosis simulates the interaction between *Mycobacterium Tuberculosis* and the immune system's response inside a human lung, tracking the formation of granuloma (small areas of inflammation). What distinguishes this application from other existing applications is the use of two types of agents and the spawning and termination of agents during the simulation.

3. Implementation

3.1 MATSim

3.1.1 Intersections as Places

Intersection State

The Intersection class manages all outgoing roads to other intersections and keeps track of how many cars reside on it at one time, ensuring its capacity is not exceeded. `IntersectionState.h` contains the intersection's id, coordinates, capacity, number of cars present, whether it is a road (for visualization), several arrays representing roads linking to other intersections, and three arrays used for synchronizing agents. An excerpt of this can be seen below:

```
class IntersectionState : public mass::PlaceState {
    public:
        int id, x, y, capacity, numCars;
        bool isRoad;
        int roadId[4], length[4], numLanes[4], destination[4], capacityRoad[4], numCarsRoad[4], direction[4], visualizeCar[4];
        int roadAgents[4], requestedRoad[4];
        bool terminateAgents[4];
};
```

Unlike the MASS C++ version where intersections have a map that points to road objects, this version contains arrays for each attribute of a road. This is due to agents being unable to point to a memory address which would return an illegal memory access error. Thus, the road's id, length, number of lanes, destination id, capacity, number of cars, and direction is stored in an int array of size 4. The index represents the road's direction with 0 being North, 1 – East, 2 – South, and 3 – East.

Another difference from the MASS C++ version is that the C++ version uses mutex locks, while the CUDA version cannot use mutex locks or anything from the std library. So, three arrays are used to maintain synchronization of agents, which will be further discussed in the Simulation section.

Intersection Initialization

Intersections are created using the code:

```
// create Intersection places to represent our simulation space.
mass::Places* intersectionPlaces = mass::Mass::createPlaces<IntersectionPlace, IntersectionState>(
    0,                  // handle
    NULL,               // args
    0,                  // arg size
    dims,               // dimensions
    simSpace            // size of sim space
);
```

Intersection data is read on the host (CPU) from a text file generated by a C++ sequential program. This data is then sent to the device (GPU) to each intersection using a `callAll()`.

On initializing an intersection, each intersection's data received from the host is assigned to the IntersectionState. The capacity of each intersection is calculated here by number of roads multiplied by number of lanes. The capacity of a road is calculated by the number of lanes multiplied by the road's length divided by 15 (assumed car's length). In this implementation, the capacity for both intersections and roads is 1, following MASS C++ MATSim's implementation.

Intersection Functions

When a car agent migrates to the intersection, the agent calls the `putCarOnLane()` function which attempts to move the car to its road. If there is available space on the road, the car will be placed on the road and start travelling on the road, and the length of the road will be returned to the agent. If there is no available space on the road, the car will wait and call the function again on the next loop.

When a car agent leaves its current intersection to migrate to another intersection, the agent calls the `depart()` function. It removes the car from the road and returns true if the destination intersection has available space. If the destination intersection does not have available space, it will only return false to the car agent.

Intersections do not exchange any information during the simulation loop. An `exchangeAll()` is called only once before the loop for each intersection to retrieve its neighboring places.

3.1.2 Cars as Agents

Car State

The CarAgent class manages an agent that can move to an intersection, move from an intersection to a road, or move along a road. The CarAgent class consists of a unique ID, its starting intersection's ID, its destination intersection's ID, a route which is an array of road IDs, the size of the route, its current position in the route array, its distance left to travel if it's on a road, its current moving direction, a boolean for whether it's on a road, a boolean if it's migrating. Below is an excerpt from `CarState.h`:

```
class CarState: public mass::AgentState {
public:
    int id, start, end;  // ID, starting point, ending point of route
    int* route;          // The route of roads to be traversed
    int routeSize;       // Size of route array
    int routeOffset;     // Offset for combined route array
    int routePos;        // Position in route vector
    int distToGo;        // Distance left to be travelled on road
    int movingDirection; // Stores the direction the car is moving
    bool onLane;         // True when on a lane
    bool migrating;  // Indicates if the car has finished its route
};
```

Car Initialization

On initialization, the cars are spawned at their starting intersections by passing an array of each car's starting intersection ID, placeIdxs, to `createAgents()` as shown in the excerpt below:

```
mass::Agents* carAgents = mass::Mass::createAgents<CarAgent, CarState>(
    1,                  // handle
    NULL,               // args
    0,                  // arg size
    opts.numCars,       // number of agents
    0,                  // places handle
    opts.numCars,       // max agents
    placeIdxs           // initial starting places
);
```

The car also initializes its values with the car data received from the host. Since the device cannot access pointers to the host's memory, a 2D array cannot be used for the cars' routes. Instead, the routes will be combined into a single dimensional array and each car will be provided with a route size and their offset in the combined array. So, a different method needs to be called from the `callAll()` to pass in the array, which is `initCarRoute()`. In this

method, cars access their route data from the combined array by their offset up to their offset + route size.

Car Functions

At the start of every simulation loop, all cars are called to execute `moveOnLane()`. First, this method checks whether the car has reached its destination. If it has, then it will be removed from its current intersection and it will be terminated. If not, then it will check if the car is travelling on a road. If it is on a road, then it will move by 1 feet towards the end of the road. If it is not on a road, then the intersection will place the car on a road if there is available space.

After moving on the road, cars will attempt to migrate to an intersection if they have reached the end of their road. If there is available space on the intersection they are migrating to, then they will migrate to that intersection. If not, they will try again in the next simulation loop.

3.1.3 Simulation

This implementation uses direction-based collision handling to prevent intersections from exceeding their capacity. This means the simulation will only allow one direction of movement at a time, then pausing to update the number of open spots each intersection has. An excerpt of this implementation can be seen below inside the simulation loop:

```
for (int i = 0; carAgents->getNumAgents() > 0; i++) {
    carAgents->callAll(CarFunctions::MOVE_ON_LANE);
    intersectionPlaces->callAll(IntersectionFunctions::RESOLVE_LANE);
    carAgents->callAll(CarFunctions::MOVE_TO_LANE, new int(Directions::NORTH), sizeof(int));
    carAgents->callAll(CarFunctions::MOVE_TO_LANE, new int(Directions::EAST), sizeof(int));
    carAgents->callAll(CarFunctions::MOVE_TO_LANE, new int(Directions::SOUTH), sizeof(int));
    carAgents->callAll(CarFunctions::MOVE_TO_LANE, new int(Directions::WEST), sizeof(int));

    carAgents->callAll(CarFunctions::MOVE_TO_INTERSECTION, new int(Directions::NORTH), sizeof(int));
    carAgents->callAll(CarFunctions::MOVE_TO_INTERSECTION, new int(Directions::EAST), sizeof(int));
    carAgents->callAll(CarFunctions::MOVE_TO_INTERSECTION, new int(Directions::SOUTH), sizeof(int));
    carAgents->callAll(CarFunctions::MOVE_TO_INTERSECTION, new int(Directions::WEST), sizeof(int));

    carAgents->manageAll();

    carAgents->callAll(CarFunctions::SETTLE_ON_NEW_PLACE);
    intersectionPlaces->callAll(IntersectionFunctions::RESOLVE_TERMINATE);
}
```

As can be seen above, the simulation stops when all agents have been terminated; meaning that all agents have reached their destination place.

The first line in the loop is a `callAll()` to all agents passing in a message to make all the agents call their `moveOnLane()` function. This makes all agents move along a road if they are already on one, otherwise, cars will send a request with their id and requested road id to their place. The next line calls all intersections to `resolveLane()` which checks for the received requests and only allows one car to move to each direction at a time. If multiple cars request the same direction, the car with the lowest index on the roadAgents array will be chosen to move to the road. The next four lines calls all cars to `moveToLane()` by one direction at a time. This is to maintain synchronization when an intersection calls `putCarOnLane()` so that when decreasing the intersection's number of cars (`state->numCars--`), it cannot get overlapped by multiple car agents calling it at the same time.

The next four lines is where we can see the direction-based collision handling. The agents are being called by a `callAll()` which is called four times and is being passed in the `moveToIntersection()` function with directions: north, east, south, and west. First, this causes all agents that are ready to migrate to an intersection north of them to attempt the migration. After the agents migrating north have finished exchanging values, the agents migrating east are called next. This repeats until all agents have performed this in all directions.

This ensures that agents cannot migrate to the same place from different directions at the same time which may cause an intersection to unintentionally hold more agents than its intended capacity. However, this does not ensure synchronization when there are multiple agents migrating to an intersection from the same direction, which is why the intersection's `depart()` function uses a mutex lock to maintain synchronization.

3.2 Tuberculosis

3.2.1 EnvironmentPlace as Places

EnvironmentPlace State

The environment place is responsible for bacteria growth which grows radially by one grid unit every 10 days. It also keeps track of chemokine levels emitted by macrophages which decay by 1 level per day. Each place is constrained to have a maximum of one macrophage and one t-cell present at a time. An excerpt of the `EnvironmentPlaceState.h` is shown below:

```
class EnvironmentPlaceState : public mass::PlaceState {
public:
  // Whether the bacterium exists.
  bool bacteria;

  // Whether the current EnvironmentPlace is an entry point for simulation agents.
  bool bloodVessel;

  // The chemokine level.
  int chemokine;

  // Whether the macrophage exists.
  bool macrophage;

  // Whether a macro spawner is trying to spawn on the place
  bool shouldSpawnMacro;

  // The macrophage state.
  int macrophageState;

  // The current T-Cell occupying the EnvironmentPlace.
  bool tcell;

  // Whether a tcell spawner is trying to spawn on the place
  bool shouldSpawnTCell;

  // Random value (randomized on init)
  int randState;
};
```

EnvironmentPlace Initialization

EnvironmentPlaces are created using the code:

```
mass::Places* environmentPlaces = mass::Mass::createPlaces<EnvironmentPlace,
EnvironmentPlaceState>(
    0,                  // handle
    NULL,               // args
    0,                  // arg size
    dims,               // dimensions
    simSpace            // size of sim space
);
```

On initialization, the four environmentPlaces at the center of each quadrant is chosen to be a blood vessel where new macrophage and t-cell agents can enter the simulation or where the agent spawning occurs. The randVal of each place is generated on the host side and is then passed to the places. The random values are generated using a seed which allows the simulation to remain deterministic.

EnvironmentPlace Functions

When an agent is ready to migrate, it will call the place's `getHighestChemokine()` to determine where it wants to migrate. This function returns the index of the neighboring place with the highest level of chemokine. If there are more than one place with the same, highest level of chemokine, the agent will randomly choose one of the maximum chemokine level places.

Every simulation day, `chemokineDecay()` is called. This decays the chemokine level by 1, and if a macrophage is present at the place, it will set its own and its neighbors' chemokine level to 2.

Every 10 days, `bacteriaGrowth()` is called. This causes places containing bacteria to spread bacteria to its neighbors.

Agent spawning is decided by using `cellRecruitment()` which grants a 50% for each agent to spawn. Tcells can only spawn after the 10th day.

3.2.2 Macrophages as Agents

Macrophage State

The macrophage dynamic agent represents a human immune system macrophage cell. A macrophage agent may be in one of the following five states: resting, infected, activated, chronically infected, or dead. It also keeps track of its internal bacteria and infected time which are used in these states. An excerpt of its state can be seen below:

```
class MacrophageState : public mass::AgentState {
public:
  // The current state, whether it is a spawner, number of internal bacteria,
  // the number of days since the macrophage was first infected.
  int state, isSpawner, internalBacteria, infectedTime;
};
```

Macrophage Initialization

The number of initial macrophages is determined by the simulation parameter `init_macro_num` plus the four spawner agents which results to the totalMacro. Each of these macrophages is assigned a random place and the spawners are assigned to the blood vessels. They are then created using the code below:

```
mass::Agents* macrophageAgents = mass::Mass::createAgents<Macrophage,
MacrophageState>(
    1,                                  // handle
    NULL,                               // args
    0,                                  // arg size
    totalMacro,                         // number of agents
    0,                                  // places handle
    numCells,                           // max agents
    macroSpawn                          // initial starting places
);
```

Macrophage Functions

Every simulation day, the macrophages `migrate()` to one of the neighboring places with the highest chemokine level by calling the place's `getHighestChemokine()` function.

Its state is then updated using the `updateState()` which calls another function based on its state to perform the state's rules.

Spawners will `spawnMacro()` if the place decided that it should spawn a macrophage.

3.2.3 Tcells as Agents

Tcell State

The tcell only keeps track of whether it is a spawner:

```
class TCellState : public mass::AgentState {
public:
  bool isSpawner;
};
```

Tcell Initialization

At the beginning of the simulation, only the tcell spawners are created. They are initialized at the blood vessels using the code below:

```
mass::Agents* tcellAgents = mass::Mass::createAgents<TCell, TCellState>(
    2,                                  // handle
    NULL,                               // args
    0,                                  // arg size
    4,                                  // number of agents
    0,                                  // places handle
    numCells,                           // max agents
    tcellSpawn                          // initial starting places
);
```

Tcell Functions

Tcells `migrate()` the same way as macrophages do by choosing one of the neighbors with the highest level of chemokine. Spawners also will `spawnTCell()` if the place decides that it should spawn a tcell.

3.3 Simulation

The simulation ends after total_days which is a simulation parameter that is set when running the program or 100 by default. An excerpt of the simulation loop is shown below:

```
for (int i = 0; i < opts.total_days; i++) {

    environmentPlaces->callAll(EnvironmentFunctions::CHEMOKINE_DECAY);

    if (i > 0 && i % 10 == 0)   // grows every 10 days
        environmentPlaces->callAll(EnvironmentFunctions::BACTERIA_GROWTH);

    environmentPlaces->callAll(EnvironmentFunctions::CELL_RECRUITMENT, new
int(i), sizeof(int));

    macrophageAgents->callAll(MacrophageFunctions::MIGRATE_MACRO);
    tcellAgents->callAll(TCellFunctions::MIGRATE_TCELL);

    macrophageAgents->manageAll();
    tcellAgents->manageAll();

    macrophageAgents->callAll(MacrophageFunctions::UPDATE_MACRO_PLACE_STATE);
    tcellAgents->callAll(TCellFunctions::UPDATE_TCELL_PLACE_STATE);

    macrophageAgents->callAll(MacrophageFunctions::UPDATE_STATE);

    macrophageAgents->callAll(MacrophageFunctions::SPAWN_MACRO);
    tcellAgents->callAll(TCellFunctions::SPAWN_TCELL);
```

```
    macrophageAgents->manageAll();
    tcellAgents->manageAll();

    macrophageAgents->callAll(MacrophageFunctions::UPDATE_MACRO_PLACE_STATE);
    tcellAgents->callAll(TCellFunctions::UPDATE_TCELL_PLACE_STATE);
}
```

At the beginning, chemokine levels in each place decay by 1 and is spread to places and their neighbors if a macrophage is present. Every 10 days, bacteria on each place will grow radially. Each blood vessel will then decide whether it should spawn macrophage and tcell agents and reserve a spot if they choose to spawn.

After the places have done the chemokine decay and bacterial growth, macrophage and tcell agents will migrate accordingly.

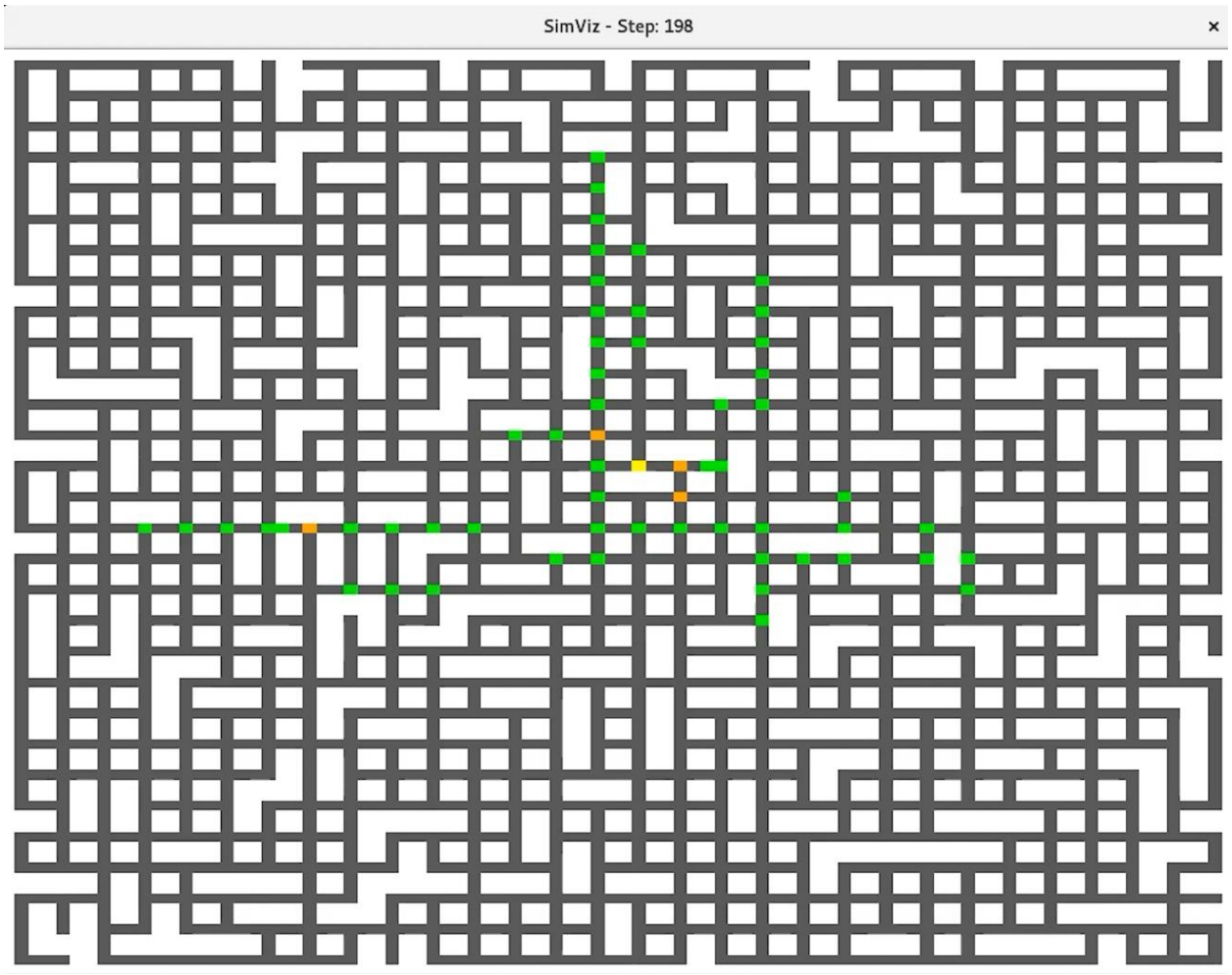
Macrophage and tcell agents will then update their place state to signal their arrival on the new place. Macrophages will then be able to update their state and perform its state's rules.

Next, macrophage and tcell spawners will spawn based on the decision made earlier in the simulation. Once the agents have been spawned, they will finally update their place once more to adjust to the macrophage state rules' changes and the newly spawned agents.

4. Verification

4.1 Results/Visualization

MATSim Visualization



A snapshot of a MATSim simulation at step 198 with 30 by 30 intersections is shown in the figure above.

Each intersection is represented by a 3 by 3 in the grid. The center is the intersection itself and the edges (north, east, south, or west) are the roads connected to it. So, with a simulation of 30 by 30 intersections, the size of the whole grid is 90 by 90.

Cars are indicated by a pixel that changes color on a gradient from green to red based on the density of the car. Red indicates that the intersection or road is at its maximum capacity with 4 cars in this case. An orange color indicates 3 cars, yellow indicates 2 cars, and a light green color indicates that there is only one car.

In the snapshot, cars can be observed to be converging to the center, and some have already reached their destination. Some congestion can be seen at the center.

MATSim Results

During the development of this application, I discovered several bugs in the MASS CUDA library.

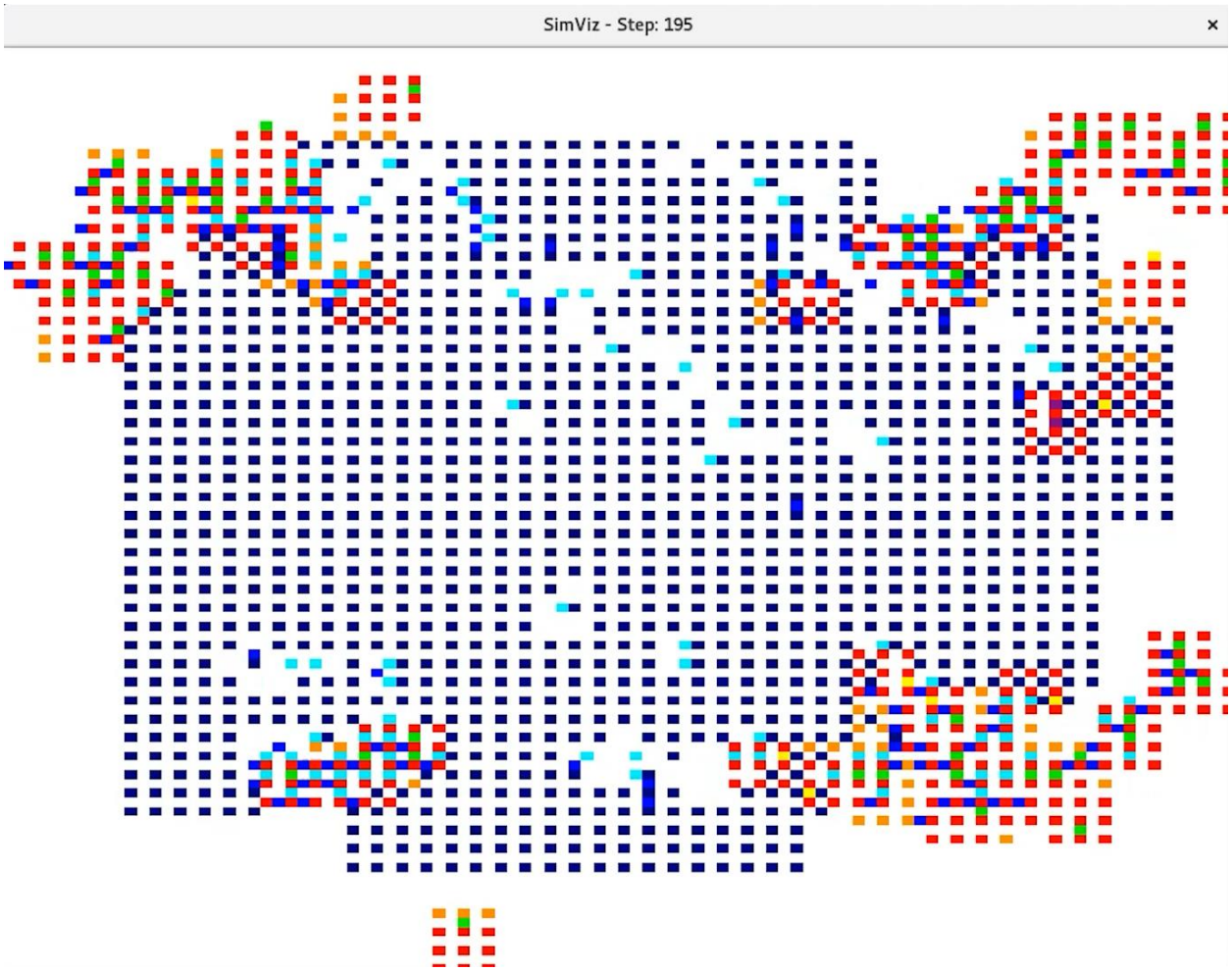
First, I was trying to calculate the capacity of a road with:
`std::max((int)(state->numLanesN * state->lengthN) / 15, 1);`
While I had this code, I noticed that some values weren't being assigned to the state variables and it was causing weird interactions. For example, when I explicitly assigned 10 to the road length, it didn't get changed when using it in later code. I eventually found that `std::max` was causing this issue. I chose to avoid the issue by not using any std library functions.

Another issue I discovered was that when the MAX_AGENTS config is set to greater than 1, when an agent attempts to migrate to a place that another agent has been to, the agent fails to migrate to the place. This issue was not discovered previously because all the applications did not use more than one MAX_AGENTS. I reported this issue to Brian and he found that it happens because when MAX_AGENTS is set to more than one, an agent that migrates to a place will fill the whole array and won't erase itself after migrating out of it. This causes other agents trying to migrate to the place to think that the place is full, preventing migration. Brian later fixed this issue.

A small functionality issue I found while trying to use `getNumAgents()` to get the number of alive agents was that the function doesn't update when an agent is terminated. It will always return the number of agents on initialization. This was reported to Brian and was quickly addressed.

Another migration issue I discovered is when two agents are trying to migrate to a place at the same time. Only one agent will successfully migrate to that place while the other fails to migrate. It turned out that it was a race issue in the library and was quickly fixed by Brian.

Tuberculosis Visualization



A snapshot of a Tuberculosis at step 195 with size 40 is shown in the figure above.

Each environmentPlace is represented by a 2 by 2 grid. The top left pixel is for bacteria (black if present). The top right pixel is for the macrophage (resting - green, infected - yellow, activated - light blue, chronically infected - dark purple). The bottom left pixel is for tcells (bright blue if present). The bottom right pixel represents the chemokine level (1 - orange, 2 - red).

In the snapshot, it seems like the bacteria is winning. Macrophage and tcells can be seen to be spawning at the center of each quadrant. Some macrophages have been activated and are killing bacteria around the center.

Another issue was found and has not been fixed at the time that this paper was written, so it will be discussed in Section 4.3.

Tuberculosis Results

At the time, the race issue in the MATSim application has not been fixed, and so, Tuberculosis agents encountered the same issue where they were unable to migrate to a place that was previously visited by another agent.

At the time that this paper was written, another issue was found. It will be discussed in Section 4.3.

4.2 Execution Performance

MATSim

At this moment, MATSim is unable to provide any execution performance due to current problems.

Tuberculosis

Due to a current bug, the max number of initial macrophages is 2.

With a size of 40, total days of 100, and initial macro number of 2, the average execution performance is 239.936 ms. This simulation can be run by using:

```
./bin/Tuberculosis –init_macro_num=2
```

With a larger size of 100 and same parameters , the average execution performance is 252.899 ms. This simulation can be run by using:

```
./bin/Tuberculosis –init_macro_num=2 –size=100
```

With a longer total days of 500 and same parameters , the average execution performance is 338.169 ms. This simulation can be run by using:

```
./bin/Tuberculosis --init_macro_num=2 --total_days=500
```

4.3 Current Problems

For MATSim, some agents do not reach their destination and do not move after initialization. This is because in environmentPlace's `initAgent()`, it uses the state->agents[] to assign to the roadAgents[]. However, when two or more agents are initialized in the same

place, only one appears in the state->agents[] array. Thus, the other agents aren't assigned to roadAgents and is not detected by the place for requesting a lane.

For Tuberculosis, the highest initial number of macrophages is 2. If the parameter is set to higher than 2, it will return a misaligned address error. Before implementing tcell agents, it was possible to initialize any number of macrophage. However, after implementing tcell agents, it can only be set up to 2. Therefore, the suspicion is that there is a bug in the MASS CUDA library when creating two types of agents.

5. Conclusion

5.1 Summary

Over two quarters of development, I have implemented MATSim and Tuberculosis. I completed my goal of verifying the accuracy of the MASS CUDA library. However, comparing its performance to other agent-based models like MASS C++ and RepastHPC is not possible yet due to some bugs. Though, I discovered several bugs when the MAX_AGENTS config is set to more than one which were resolved by Brian.

I was able to produce a visualization of the simulation using SimViz, and Tuberculosis was able to show some execution results when initial macrophages is less than or equal to 2.

5.2 Future Development

For future plans, the multiple agent initialization at one place issue in MATSim needs to be fixed. For Tuberculosis, it needs to be able to run with any number of initial macrophages. Once these applications are fully able to produce results, the execution performance needs to be compared to the benchmark measurements that Kevin Wang has done with MASS C++, RepastHPC, and FLAME. These applications can also be used to verify the new Multi-GPU version of MASS CUDA.

APPENDIX

Code Location

The code for MATSim and Tuberculosis is located under the csumali_develop branch. The link to the branch is:
https://bitbucket.org/mass_application_developers/mass_cuda_appl/src/csumali_develop/

How To Run

MATSim

To build and run the MATSim application, first run `make develop`. Once the development environment is setup, run `make build` and then `make test` to ensure all tests are passing properly. Once built, a MATSim executable will be placed within the ./bin directory, and can be used to run the simulation.

Running the application with the `--help` flag will provide a description of the application and options that can be used to define simulation parameters.

Specifying an output interval, by using the `--interval` flag will tell the program to output the simulation to a .viz file every <interval> steps. This simulation file can be played using the MASS SimViz application for a graphical visualization of the simulation space. To build the simviz application, run `make build-simviz`.

To run the default input files, run `./bin/MATSim`. This will run the simulation with the files “src/Node_Links.txt” and “src/Car_Agents.txt” with the sizeX and sizeY of 30 and numCars of 90.

To create the viz file, run the simulation with `./bin/MATSim --interval=1`. This will create a matsim.viz file.

To show the visualization, first you would need to RDP into JUNO using a remote desktop client. Then, you would need to assign the program to display the visualization to the display monitor you are connected to. To do so, run `echo $DISPLAY` on the remote client. That will tell you the display you need to configure in your terminal. You do that using the following (using :11.0 as an example): export DISPLAY=:11.0. Once the display has been set, you can run the viz file using `./bin/simviz matsim.viz` from the remote client.

To run the simulation with different input files, for example Node_Links40.txt and Car_Agents40.txt, you would need to specify the location of the files, size of the simulation, and number of cars. For example with a size of 40 by 40, 160 cars, you will run using:

```
./bin/MATSim --sizeX=40 --sizeY=40 --numCars=160 --
roadFile=test_files/Node_Links40.txt --
routeFile=test_files/Car_Agents40.txt
```

Tuberculosis

To build and run the Tuberculosis application, first run `make develop`. Once the development environment is setup, run `make build` and then `make test` to ensure all tests are passing properly. Once built, a Tuberculosis executable will be placed within the ./bin directory, and can be used to run the simulation.

Running the application with the `--help` flag will provide a description of the application and options that can be used to define simulation parameters.

Specifying an output interval, by using the `--interval` flag will tell the program to output the simulation to a .viz file every <interval> steps. This simulation file can be played using the MASS SimViz application for a graphical visualization of the simulation space. To build the simviz application, run `make build-simviz`.

To run the simulation with the default simulation configuration, run `./bin/Tuberculosis`. This will run the simulation with a size of 40 by 40, total days of 100, initial macrophages of 100, and the current time will be used as the seed.

To create the viz file, run the simulation with `./bin/Tuberculosis --interval=1`. This will create a tuberculosis.viz file.

To show the visualization, first you would need to RDP into JUNO using a remote desktop client. Then, you would need to assign the program to display the visualization to the display monitor you are connected to. To do so, run `echo $DISPLAY` on the remote client. That will tell you the display you need to configure in your terminal. You do that using the following (using :11.0 as an example): export DISPLAY=:11.0. Once the display has been set, you can run the viz file using `./bin/simviz tuberculosis.viz` from the remote client.

To run the simulation with custom configurations, you would need to specify the size of the simulation, number of days, and number of initial macrophages. For example with a size of 100

by 100, 300 total days, and 2 initial macrophages, you will run using:

```
./bin/Tuberculosis --size=100 --total_days=300 --
init_macro_num=2
```