Porting Agent-Based benchmarks to MASS CUDA

David Woo

CSS 499 Spring 2023 Term report

Professor Munehiro Fukuda

Jun 1, 2023

1. Introduction **2**

1.1 Motivation 2

1.2 Project Goal 3

2. Background **3**

2.1 What is the “software Agents” 3

2.2 MASS specification and mechanism 5

2.3 MASS CPP vs MASS JAVA vs MASS CUDA 6

3. Implementation **7**

3.1 Financial Market Place as Places 7

3.2 Bank as Agent 8

3.3 Firm as Agent 9

3.4 Owner as Agent 10

3.5 Worker as Agent 11

4. Verification **12**

4.1 Result/Visualization 12

5. Conclusion **12**

5.1 Summary 12

5.2 Future Development 12

1. Introduction

1.1 Motivation

The motivation for this individual research is to explore distribution systems and parallelizing computing. By studying MASS (Multi-agent spatial simulation), I learned and built the concrete concept of agent-based parallelization of micro and spatial simulation. An Agent-based computational framework is crucial in Distribution Data analysis.

MASS is focused on Data discovery for “Big-data computing”. Multi-Agents spatial simulation(MASS) can analyze data patterns and be features for the distributed data sciences. Here dslab team has been working on modeling maga-scale social or biological agents and simulating their emergent collective behavior using parallelization. Therefore, it can increase the performance of the application. This report will cover more details of it in Section 2. My research objective is to port the social agent simulation application (Bank BailIn and Bailout) from MASS CPP to MASS CUDA.

1.2 Project Goal

The work of the research project will be used to compare different library performances. Each version of the MASS library is composed to overcome parallelization challenges such as machine unawareness, ghost space management, and cross-processor agent management (including migration, propagation, and termination). That’s why dslab team has developed MASS, a new parallel-computing library for multi-agent and spatial simulation over a cluster of computing nodes.

This research application will be used to examine the performance of the MASS CUDA simulation with other MASS libraries’ comparison. Unlike CPU-based parallelization, GPU-based parallelization has benefits in the aspect of small cost for thread switching over thousand cores. With that benefit, the GPU-based Cuda library is suited for all types of agent-based model applications.

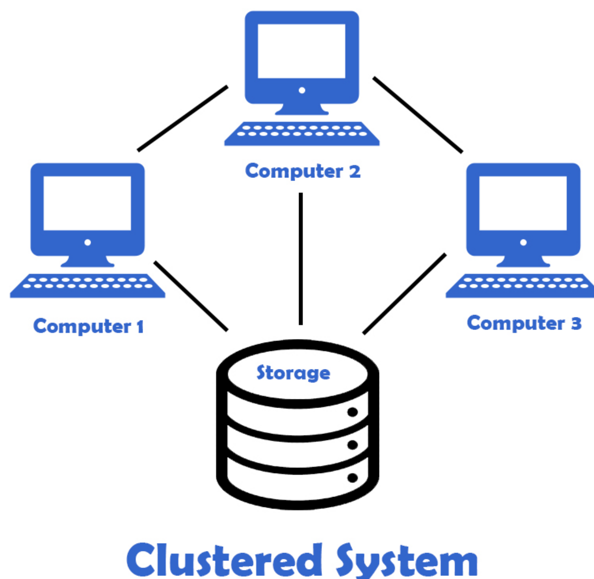
2. Background

2.1 What is the “software Agents”

There are two kinds of software agents in distributed data analysis (cognitive agents and reactive agents)

- Cognitive agents: “Coarse-grain executable entities that achieve a network-administrative and computation-intensive task, based on their behavior intelligence”
- Reactive Agents: “fine-grain entities, each reacting to its environment with simpler rules.

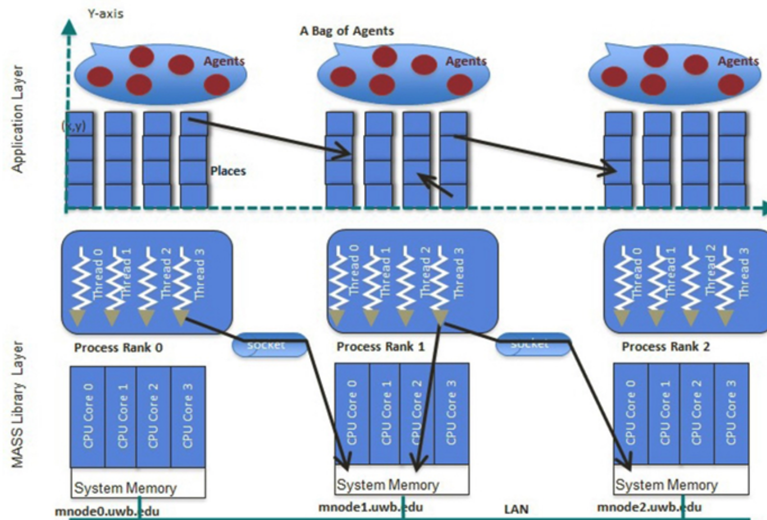
In the Agent-based computational framework, we focus on reactive agents reacting to their environment with simpler rules than cognitive agents for clustered systems.



Multi-Agents’ characteristic

- Reactive agents (fine-grain entities)
- High-level views (Hawk’s view)
- System-level agents management is required(each agent updates their status)
- Automated agent migration
- Agent collision avoidance
- Agent propagation and distributed termination
- The more accuracy is required, the more agent (CPU scalability)
- It allows spatial data analysis, which can improve programmability.

MASS execution model



MASS library, Multi-agent-spatial simulation, facilitates this flow-oriented network parallelization. In the MASS library, they have two main components. Places and Agents. Places are a multi-dimensional array of elements allocated over a cluster of multi-core computing nodes. Each element is called a place and is referred to by a network-independent array of indices capable of exchanging information with others.

Agents are a set of objects that resides in a place. And it migrates to other places and interacts with other agents from their current place. The interaction and data migration occurs on underlying shared memory or socket communication.

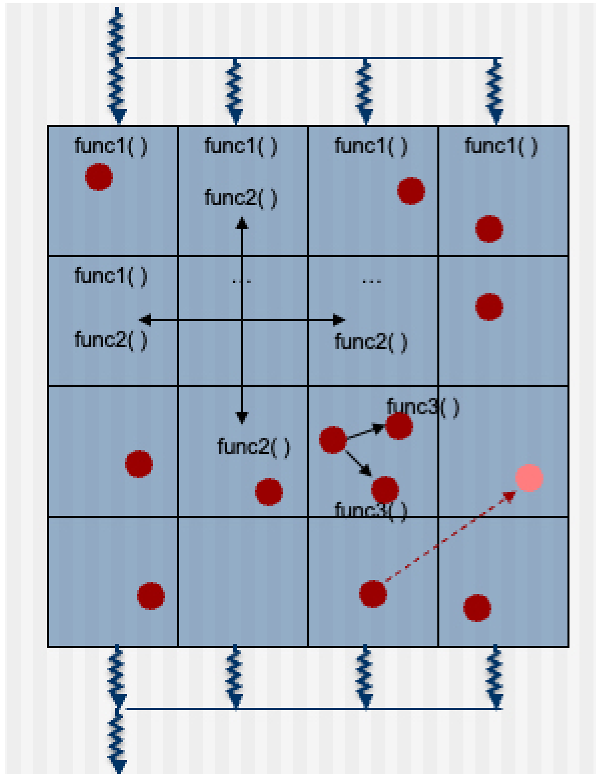
2.2 MASS specification and mechanism

```
Public static void main (String[] args) {
      MASS.init(args); //Start MASS

      Places space = new Places (handle, "MySpace", "params, xSize, ySize);
      // Create a array
      Agents agents = new Agents(handle, "MyAgents", params, space,
population);

      space.callAll( MySpace.func1, params);
      //initialize the simulation space
      space.exchangeAll(MySpace.func2, neighbors);
      // Update each place with neighbor’s information
      agents.exchangeAll(MyAgents.func3);
      // it invokes a given function from each array element to its
neighbor
      agents.manageAll();
```

```
        //it allows agent migration, duplication and termination
        MASS.finish(); // finish MASS
}
```



The MASS program starts with MASS.init() to launch remote processes, each spawning multi threads. A Places multi-dimensional array is created over these processes and partitioned into small stripes, each allocated to a different thread. space.callAll() invokes a given function of each array element in parallel. Likewise, Agents also follow a similar process. For inter-element communication, exchangeAll() invokes a given function(func2, func3) from each array element to its neighbors. Agents.manageAll() allows agent migration, duplication, and termination. Finally, MASS.finish() terminates all the places and agents and consolidates all computation data into the main thread.

2.3 MASS CPP vs MASS JAVA vs MASS CUDA

MASS CPP

- Agent-based microsimulation
- Demonstrate its practicability of parallelizing ABM microsimulation

MASS JAVA

- It cannot outperform native execution and may not be the best option for parallel computing

- Despite that limitation, it facilitates a large global memory space distributed over a cluster system.

MASS CUDA

- It has massive parallel computing capabilities of GPU

3. Implementation

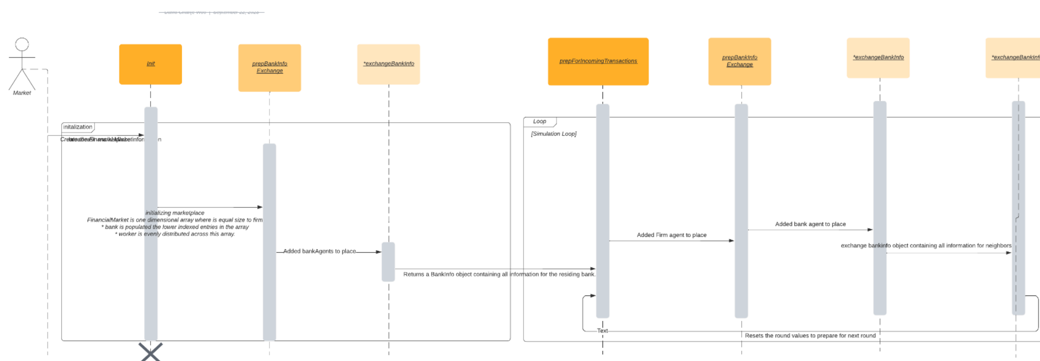
3.1 Financial Market Place as Places

Financial Market

1. The Financial Market is initialized as a one-dimensional array equal in size to the number of Firms in the simulation.

2. The Agents in the simulation use the financial Market which they can reside on as a shared memory space, relying on it to communicate with other agents and send messages to other Financial Market places.

3. Financial Marketplace keeps a local copy of the bank's information in the "bank registry" allowing firms to look up each bank's interest rates and liquidates when they need to take out a loan. Then Firm adds a negative value to their Financial Market's outgoing transactions total to be sent to the bank a subsequent call to exhcangeAll.



Financial Market Function

init():
initializing marketplace FinancialMarket is a one-dimensional array where is equal size to firm
* bank is populated the lower indexed entries in the array
* worker is evenly distributed across this array.

prepBankInfoExchange(): Added bank agents to neighbor place

***ExchangeBankInfo():** Returns a BankInfo object containing all information for the residing bank.

[Simulation loop]

prepForIncomingTransactions(): Added Firm agent to neighbor place

ExchangeOutGoingTransactions(): receive current agent bankId and return total amount of all outgoing transactions which is accumulated in given bank

ManageIncomingTransactions(): calculate all incoming transactions that are coming from other elements.

prepBankInfoExchange(): Added bank agents to neighbor place

ExchangeBankInfo(): exchange bank info object containing all transaction information for neighbor agents.

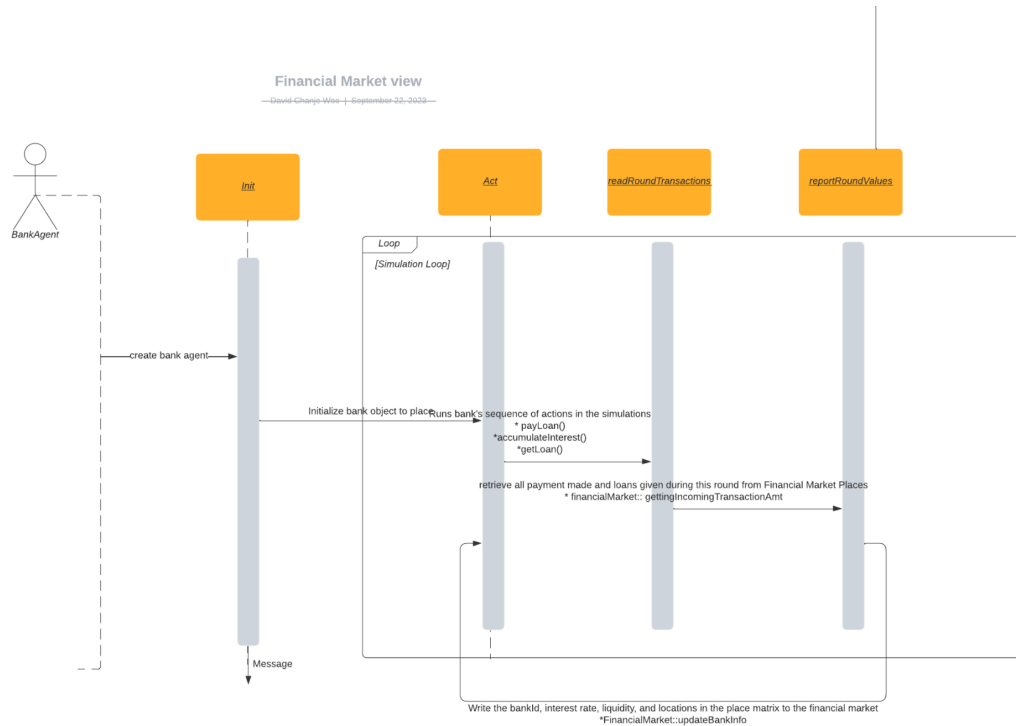
resetRoundValues(): resets the round values to prepare for the next simulation round

3.2 Bank as Agent

Bank State

Bank populates the lower-indexed entries in the array.

1. Banks receive payments, pay back loans, and seek from other banks when they run out of liquidity to their expense to product and worker wages.
2. If a bank cannot secure the loan and has run out of liquidity, it goes bankrupt and the simulation ends.



Bank Function

init(): initialize bank object to place

[Simulation loop]

Act(): runs bank's sequence of actions in the simulation

- payLoan()
- accumulateInterest()
- getLoan()

ReadRoundTransactions(): retrieve all payments made and loans given during this round from current place (*FinancialMarket::gettingIncomingTransactionAmt)

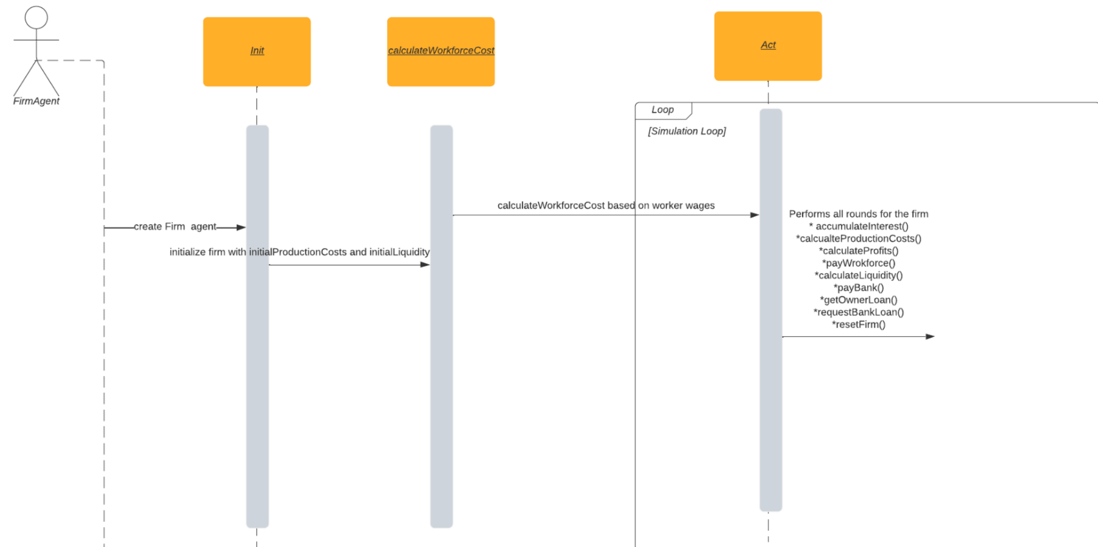
reportRoundValues(): write the bankId, interest rate, liquidity, and locations in the place matrix to the financial market(*FinancialMarket::updateBankInfo)

3.3 Firm as Agent

Firm

Firm calculate their operation costs and profits, and provide worker wage and pay back loan to bank.

1. If operation costs exceed the firm’s liquidity and owner assets, they seek a loan from the bank.
2. If the firm has negative liquidity, it resets to the initial condition and goes back to simulation.



Firm Function

init(): initialize firm with initial production costs and initial liquidity

calculateWorkforceCost(): calculate workforce cost based on worker wages

[Simulation loop]

Act(): perform all rounds for the firm

- accumulateInterest()
- calculate production costs ()
- calculate profits ()
- payWorkforce()
- calculate liquidity ()
- payBank()
- getOwnerLoan()
- requestBankLoan()
- resetFirm()

3.4 Owner as Agent

Owner

Owner agent owns a single firm

- If firm liquidity is positive, the owner gets paid.
- If it is negative, the owner lends the owner's assets to the firm.

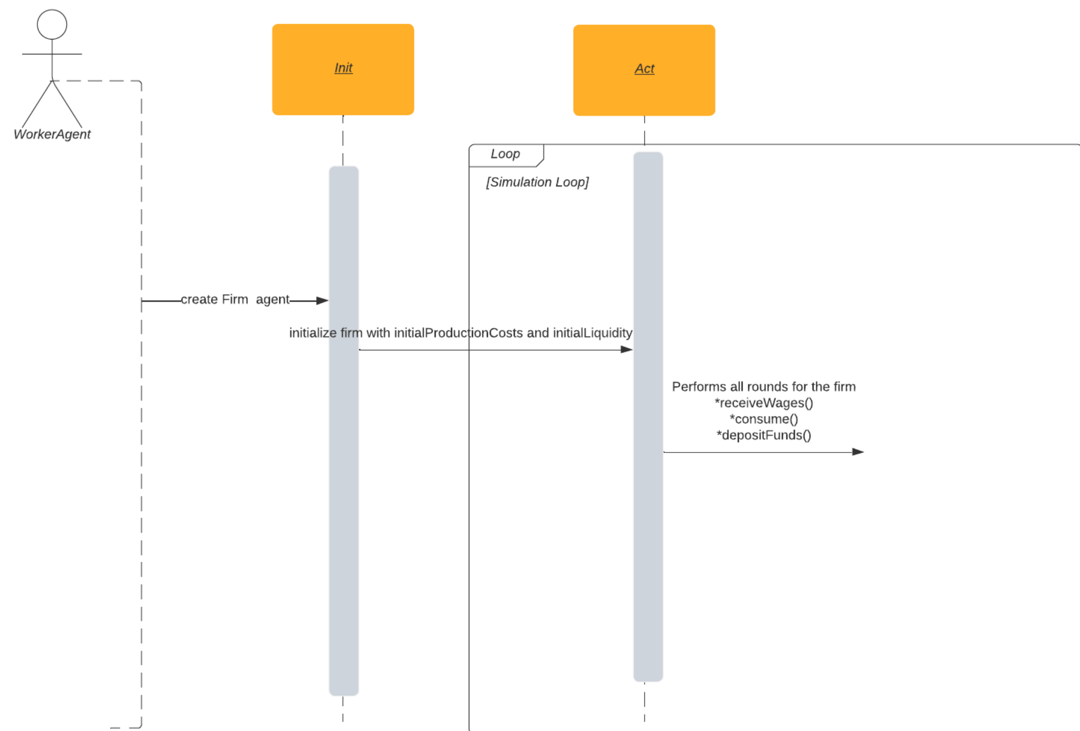
Owner Function

getCapital() : set owner's paid
setCapital() : retrieve owner's assets

3.5 Worker as Agent

Worker

Worker receives wages from their employing firms, spend a certain percentage of their wage in every round, and deposit the rest of their wage at the bank.



Worker Function

init(): initialize worker agent with wage, capital, consumptionBudget, workerCost,

[Simulation loop]

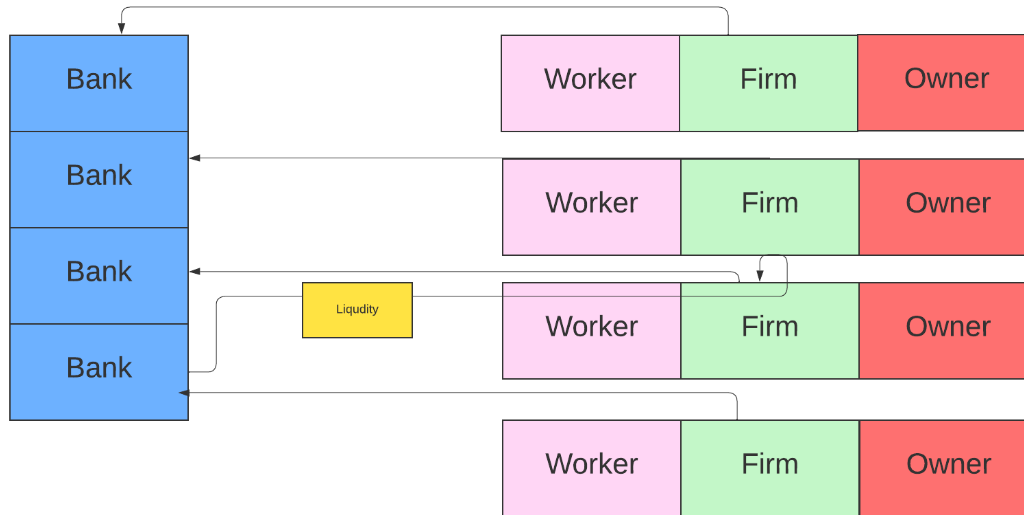
Act(): perform all around for the firm

- receiveWages()
- consume()
- depositFunds()

4. Verification

4.1 Result/Visualization

planned visualization result



5. Conclusion

5.1 Summary

Over the development quarter, I studied and implemented bankbailinbailout. I could learn parallel computing and how it can be used in spatial data analysis. I couldn't complete the implementation for various reasons (first project for parallelization, limited resources). I left unsolved issues here for further development.

5.2 Future Development

1. InMessage and outMessage

   MASS C++: Each commuting node (in each server like csslab 1 ~ 12) has a place and all agents reside in each place. To interact with other nodes, they use a message method.
   InMessage-> incoming data from another node.
   OutMessage-> outgoing data from the current node.

   MASS CUDA: "CUDA" is Unified Device Architecture. Unlike C++, they don't use the message method because there are no other servers to communicate with.

-> Current implementation, it has a Financial marketplace, four bank agents, four worker agents, and four owners. They exchange their liquidity using outgoingTransaction and incomingTransaction between agents in a place.

2. Adding neighbors to places

MASS C++:

```
 * @param functionId
 */
void Places::exchangeAll(int dest_handle, int functionId) {
    // send a PLACES_EXCHANGE_ALL message to each slave
    Message *m = new Message(Message::PLACES_EXCHANGE_ALL, this->handle,
                             dest_handle, functionId, 0, this->dimension);

    if (printOutput == true) {
```

```
    // exchangeall implementation
    Places_base::exchangeAll(MASS_base::destinationPlaces, functionId, 0);

    if (printOutput == true) {
```

```
void Places_base::exchangeAll(Places_base *dstPlaces, int functionId, int tid) {
    int range[2];
    getLocalRange(range, tid);
    ostringstream convert;
    // debugging
    if (printOutput == true) {
        convert << "thread[" << tid
                << "] exchangeAll functionId = " << functionId
                << ", range[0] = " << range[0] << " range[1] = " << range[1];
        MASS_base::log(convert.str());
    }

    DllClass *src_dllclass = MASS_base::dllMap[handle];
    DllClass *dst_dllclass = MASS_base::dllMap[dstPlaces->handle];
    // now scan all places within range[0] ~ range[1]
    if (range[0] >= 0 && range[1] >= 0) {
        for (int i = range[0]; i <= range[1]; i++) {
            // for each place
            Place *srcPlace = (Place *)(src_dllclass->places[i]);

            // check its neighbors
            // for ( int j = 0; j < int( srcPlace->neighbors.size( ) ); j++ ) {
            int inMsgIndex = -1;

            for (vector<int *>::iterator it = srcPlace->neighbors.begin();
                 it != srcPlace->neighbors.end(); ++it) {
                inMsgIndex++;
                // for each neighbor
                int *offset = *it;  //(srcPlace->neighbors)[j];
                int neighborCoord[dstPlaces->dimension];

                // compute its coordinate
                getGlobalNeighborArrayIndex(
                    srcPlace->index, offset, dstPlaces->size,
                    dstPlaces->dimension, neighborCoord);
                if (printOutput == true) {
                    convert.str("");
                    convert << "tid[" << tid << "]: calls from"
                            << "[" << srcPlace->index[0] << "]["
                            << srcPlace->index[1] << "]"
                            << " (neighborCord[" << neighborCoord[0] << "]["
                            << neighborCoord[1] << "]"
                            << " dstPlaces->size[" << dstPlaces->size[0] << "]["
                            << dstPlaces->size[1] << "]";
                }
                if (neighborCoord[0] != -1) {
                    // destination valid
                    int globalLinearIndex =
```

MASS CUDA:

```cpp

#ifndef PLACESTATE_H_
#define PLACESTATE_H_

#include "settings.h"

class Agent; //forward declaration

namespace mass {

class PlaceState {
    friend class Place;

public:
    Place *neighbors[MAX_NEIGHBORS];  // my neighbors
    unsigned index;           // the row-major index of this place
    int size[MAX_DIMS];   // the size of the Places matrix

    Agent *agents[MAX_AGENTS]; //agents residing on this place
    unsigned agentPop; // the population of agents on this place

    Agent* potentialNextAgents[N_DESTINATIONS]; //agents that expressed an intention to migrate into t

};

} /* namespace mass */
#endif /* PLACESTATE_H_ */

```

```cpp
}

void Places::exchangeAll(std::vector<int*> *destinations) {
    dispatcher->exchangeAllPlaces(handle, destinations);
}

/**
 * This function causes all Place elements to call the function specified on all neighboring
 * place elements. In addition to the fuctionality of the exchangeAllPlaces function specified above
 * it also takes functionId as a parameter and arguments to that function.
 * When the data is collected from the neighboring places,
 * the specified function is executed on all of the places with specified parameters.
 * The rationale behind implemening this version of exchangeAllPlaces is performance optimization:
 * the data cached during data collection step can be used for the data calculation and thus minimize
 * the number of memeory fetches and improve performance.
 */
void Places::exchangeAll(std::vector<int*> *destinations, int functionId, void *argument, int argSize) {
    dispatcher->exchangeAllPlaces(handle, destinations, functionId, argument, argSize);
}

```

-> We stored agents in a place's neighbors(one-dimensional array) as we mentioned earlier. ExchangeAll() is for exchanging data with their neighbors in place. In MASS CUDA, exchangeAll() only can take vector<int *> which is a specific coordinate. This feature hasn't been implemented yet.

3. nvLinks warning

```
/usr/include/c++/8/bits/unique_ptr.h:53:25: note: declared here
   template<typename> class auto_ptr;
                            ^~~~~~~~
nvlink warning : Stack size for entry function '_ZN4mass23exchangeAllPlacesKernelEPPNS_5PlaceEiiiPv' cannot be statically determined
nvlink warning : Stack size for entry function '_ZN4mass19callAllAgentsKernelEPPNS_5AgentEiiPv' cannot be statically determined
nvlink warning : Stack size for entry function '_ZN4mass19callAllPlacesKernelEPPNS_5PlaceEiiPv' cannot be statically determined
MASS_BailInBailOut build complete.
```

Presumable cause
-> stack overflow in exchangeAll(places), callAll(Agent), callAll(place)
-> agent is not able to point to a memory address so it raised an illegal memory access error

Cuda debugger tool: cuda-memcheck
Implementation code branch: David_develop