Proceedings of the 2015 Winter Simulation Conference
L. Yilmaz, W. K V. Chan, I. Moon, T. M. K. Roeder, C. Macal, and M. D. Rossetti, eds.

A Multi-Agent Spatial Simulation Library for Parallelizing Transport Simulations

Zhiyuan Ma
Munehiro Fukuda

Division of Computing and Software Systems
University of Washington Bothell
18115 Campus Way NE
Bothell, WA 98011, USA

ABSTRACT

One of the major trends in traffic simulations is to take into account microscopic aspects of traffic flows at the street level. Multi-agent models such as MATSim (multi-agent transport simulation) have been highlighted for recent years as a solution to address these complex and microscopic simulation requirements. They are viewed as an emergent and collective behavior of agents, (i.e., vehicles). However, as the simulations scale up, their computational requirements could get increased beyond the capability of a single CPU and thus should be fulfilled with parallelization. Multithreading can partially contribute to parallelization by utilizing multi-cores, but cannot give full scalability of both CPU power and memory space. To support distributed-memory parallelization for multi-agent models, we have developed the MASS (multi-agent spatial simulation) library. This paper presents how to parallelize MATSim using the MASS library and demonstrates the library's portability and execution performance in practical transport simulations.

1 INTRODUCTION

Traffic simulation has always been a difficult task. Each traveller wants to have access to any transportation with his/her own free will to act and behave. Such microscopic and unpredictable behaviors make simulation tougher, because they cannot be formulated by mathematical models. From this microscopic and dynamic viewpoint, any software, designed on the basis of the fact that travellers are intelligent, should be capable and expected to adapt and to learn (Nagel and Marchal 2003). One solution to such complex problems is using multi-agent models. Instead of simulating transport systems with traditional mathematical or traffic flow models that must be statically given before the simulations, multi-agent simulations provide a dynamic environment in which individual agents, in particular travelers will keep taking actions based on their internal rules. Therefore, this approach can perform simulations as an emergent and collective group behavior of agents, thus in the similar fashion of real-world scenarios.

MATSim is a research-based framework to implement large-scale agent-based transport simulations (MATSim Homepage 2012). The framework consists of several modules that can be combined or used independently. Modules can be replaced with user-developed implementations to test new aspects of users' own modules. Currently, MATSim offers a framework to support (1) demand modeling, (2) agent-based mobility simulation, (i.e., traffic flow simulation), (3) re-planning, and (4) methods to analyze their outputs. However, as the scale of simulation increases, the computational requirements could become large beyond the capability of a single computing node. For instance, MATSim takes approximately 15 minutes on average to run each iteration of moving 188,000 agents under its Greater Zurich scenario (Balmer, Meister, Nagel, and K.W.Axhausen 2008).

Therefore, parallelization should be considered to complete simulation runs within an acceptable time range. Multithreading can partially contribute to parallelization by utilizing multi-cores, but cannot give full scalability of both CPU power and memory space. To support distributed-memory parallelization for

multi-agent models, we have developed the MASS (multi-agent spatial simulation) library (Chuang and Fukuda 2013). This paper presents how to parallelize MATSim using the MASS library and demonstrates the library's portability and execution performance in practical transport simulations. Although the MASS library has been examined for its programmability and execution performance, our previous work was based on small test programs such as a two-dimensional wave simulation and agents' random walk over a two-dimensional space. MATSim is the very first practical application for the MASS library to parallelize. Through the parallelization, we explore the potential of and issues in the MASS library's practicability.

The rest of the paper is organized as follows: Section 2 clarifies the challenges in the related parallelization work; Section 3 presents the MASS library as a solution and explains MATSim parallelization using MASS; Section 4 evaluates the programming efforts and the execution performance of the parallelized MATSim; and Section 5 concludes our discussions.

2 RELATED WORK

The most common approach to parallelization of traffic simulation is to directly incorporate parallel algorithms into the source code itself so that the computational workload is shared among different processors. In the following, we review three parallel implementations of the most popular traffic simulations: TRANSIMS, AIMSUN, and Paramics, with respect to their data structures, data decomposition, time management, and parallelization strategies.

TRANSIMS is an integrated system of travel forecasting models, which was designed in C++ to provide transportation planners with information on traffic impacts, congestion, and pollution (Barrett et al. 1999). It models individual travelers and multi-modal transportations as simulation entities, describes a traffic network as cellular automata, populates these entities on the automata, synthesizes their activities, and executes a micro-simulation of a given scenario continuously, (i.e., second by second). Parallelization of TRANSIMS is based on a domain decomposition principle, where its cellular automata are partitioned into small domains in terms of their proximity. Each domain is then mapped to a different CPU of a cluster system and handled in parallel in support of PVM and MPI functions (Rickert and Nagel 2001).

AIMSUN is a microscopic simulation program that was originally developed for sequential execution but was later ported to shared-memory multi-processors (Barcelo, Ferrer, Garcia, Florian, and Saux 1992). The system models a traffic network as a set of links connected to each other through nodes, and runs an event-driven simulation of the traffic flow over a given network. For its parallelization, AIMSUN decomposes a traffic network into blocks, each with proximate links and nodes, and performs their parallel execution on a shared memory computing platform such as a SUN Sparc server, using its OS-provided multithreading library. AIMSUN's parallelization of a small-scale scenario performed 3.5 times faster with eight CPUs than the corresponding sequential execution.

Paramics models a traffic network as a system of junctions connected by roads, each known as nodes and links (Cameron and Duncan 1996). These links are unidirectional and opposing half roadways. Each link has a queue of vehicle objects that are currently passing through the corresponding road segment. Paramics uses continuous time management to update the status of simulation objects and move them from one to another links. For parallel simulation, Paramics uses domain decomposition to distribute links and their queues over parallel processors while the link connectivity information is replicated on every processor. Parallel execution was implemented in C*/C and executed by CM-200 (which is based on the SIMD architecture).

These systems are based on microscopic simulation. Although their internal data structure takes the form of cellular automata or a network of links and nodes, they all use domain decomposition where proximate cells or links/nodes are grouped together and mapped to a different computing node for parallel execution. These simulators are parallelized using the conventional low-level libraries such as MPI and OS-provided multi-threads. Vehicle objects are passively exchanged as messages among domains, thus among different computing nodes.

From the viewpoint of traffic planners and simulation modelers, the major challenges of their parallelization approaches are three-fold: (1) multithreading and message passing require a plenty of knowledge and skills in parallel programming; (2) platform-dependent direct parallelization aggravates the portability of parallelized traffic simulators; and (3) domain decomposition needs a spatial software such as PartitionNet in TRANSIMS or must be hard-coded.

To address these challenges for parallelizing simulations, we have developed the MASS (multi-agent spatial simulation) library in Java and C++. MASS particularly focuses on easy parallelization and quick mock-up for improving the productivity of agent-based models. As summarized in Table 1, the MASS library gives a high-level abstraction of parallelizable data structures - mobile agents over a distributed array. It facilitates MIMD parallelization of traffic simulators such as MATSim and runs it over a general cluster system.

Internal implementation	TRANSIMS	AIMSUN	Paramics	MASS MATSim
Data structures	cellular automata	a network	a network	agents over a distributed array
Data decomposition	domain-based	domain-based	domain-based	depending on data input order
Time management	continuous	event-driven	continuous	continuous
Parallelization strategies	MPI, MIMD	multithreading, MIMD	CM-200, SIMD	distributed array, MIMD

Table 1: A comparison between MASS-parallelized MATSim and related work

In summary, the MASS library can be differentiated from the above related work from the following two perspectives:

1. The MASS library provides agent/entity-based simulations with a high-level data abstraction that eases parallelization and porting of traffic simulations.
2. The MASS library gives an architecture-independent cluster-computing environment to parallelize transport micro-simulation including MATSim.

3 Parallelization of MATSim

Since transportation simulation is a specific type of spatial simulation and MATSim supports multiple agents in its Java implementation, we feel that MASS-Java is a suitable choice for parallelizing MATSim.

3.1 MASS Library

Within MASS library, two key concepts are introduced: *places* and *agents*. *Places* are a matrix of elements that are dynamically allocated over a cluster of computing nodes, and each element is called a *Place* object. On the other hand, *agents* are a set of execution instances that can reside on a place, migrate to any other places with matrix indices, (thus as duplicating themselves), and interact with other agents through their local place. As illustrated in Figure 1, places are mapped to threads, whereas agents are mapped to processes. Unless a programmer indicates his/her places-partitioning algorithm, the MASS library divides places into smaller stripes in vertical or in the X-coordinate direction, each of which is then allocated to and executed by a different thread. Contrary to places, agents are grouped into bags, each allocated to a different process where multiple threads keep checking in and out one after another agent from this bag when they are ready to execute a new agent.

Parallelization with the MASS library uses a set of multi-threaded communicating processes that are forked over a cluster of multi-core computing nodes with JSCH in Java or libssh2 in C++ and are connected to each other through TCP sockets. Multi-threads take charge of method calls and information exchanges among *places* and *agents* in parallel. A user designs a behavior of a *place* and an *agent* by extending the *Place* and *Agent* base classes respectively. They are populated through the *Places* and *Agents* classes. Actual computation is performed between *MASS.init()* and *MASS.finish()*, using the following major methods, each performed in parallel (Fukuda 2010). (Note that the following discussions focus on the Java specification for simplicity.)

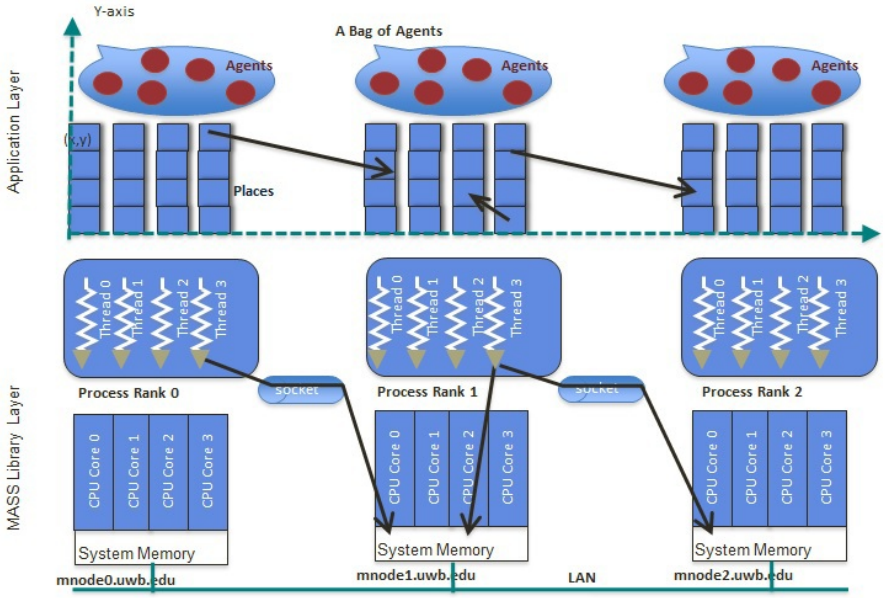

Figure 1: The MASS library's execution model

Places Class

public Places(int handle, String className, Object argument, int boundary, int size...) instantiates a shared array with *size* and a shadow space with *boundary* from *className* as passing an *argument* to the *className* constructor.
public Object[] callAll(int functionId, Object[] arguments) calls the method specified with *functionId* of all elements as passing *arguments[i]* to element[i], and receives a return value into *Object[i]*.
public void exchangeAll(int handle, int functionId, Vector<int[]> destinations) calls from each element to a given method of all destination elements, each indexed with a different *Vector* element. Each vector element, *destination[]* is an array of integers where *destination[i]* includes a relative index on the coordinate *i* from the current caller to the callee element. The caller passes its *outMessage[]* data member to the callee as a set of arguments, and receives return values in its *inMessage[]*.
public void exchangeBoundary( ) exchanges node-boundary's *outMessage[]* data with neighboring nodes as a ghost space.

Place Class

private size[]; private index[] maintains the size of the shared array that each element belongs to and the index of each array element.
public Object callMethod(int functionId, Object argument) is invoked from *Places.callAll()* and *exchangeAll()* so as to call a function specified with *functionId*.

Agents Class

public Agents(int handle, String className, Object argument, Places places, int population) instantiates agents from *className*, passes *argument* to their constructor, and distributes them over a given *Places*, based on *Agent.map()*.
public Object callAll(int functionId, Object[] arguments) is the same as *Places.callAll()*.
public void manageAll() updates each agent's status, based on its latest calls of *migrate()*, *spawn()*, and *kill()*. These methods are invoked within *callAll()*.

Agent Class

migrate(int[] index...) allows a calling *Agent* to migrate or propagate itself to one or more *Place*s specified with *index* upon *Agents.manageAll()*.
spawn(int nChildren, Object arguments) spawns children as passing *arguments* to them.
kill() terminates a calling *Agent*.
public Object callMethod(int functionId, Object argument) is the same as *Place.callMethod()*.

```
1  public class MeshedTrafficSimulation {
2   public static void main(String[] args) {
3     // validate the arguments
4     int size=Integer.parseInt(args[0]);
5     int nCars=Integer.parseInt(args[1]);
6     int maxTime=Integer.parseInt(args[2]);
7     MASS.init(args);// start MASS
8     // create meshed streets.
9     Places streets=new Places(1, "Mesh",
10                          null, size, size);
11    // populate vehicles.
12    Agents cars=new Agnets(2, "Vehicles",
13                         null, streets, nCars);
14    // define the four neighbors (nbrs) of each place
15    Vector<int[]> nbrs = new Vector<int[]>();
16    int[] north = {0, -1}; nbrs.add(north);
17    int[] east  = {1,  0}; nbrs.add(east);
18    int[] south = {0,  1}; nbrs.add(south);
19    int[] west  = {-1, 0}; nbrs.add(west);
20    // now go into a cyclic simulation
21    for (int t = 0; t < maxTime; t++) {
22      // exchange neighboring traffic flows
23      streets.exchangeAll(Mesh.getFlow_,nbrs);
24      // allow cars to drive to the next street segment
25      cars.callAll(Vehicles.migrateNext_);
26      cars.manageAll();
27 } } }
```

```
1  public class Mesh extends Place {
2   static final int getFlow_ = 1;// function ID
3   private Integer myFlow;// a street segment's flow
4   Object callMethod(int funcId, Object arg){
5     swith(funcId) {
6       case getFlow_: return getFlow();
7   } }
8   private Object trafficInfo() {// pass myFlow
9     return (Object)myFlow;       // to neighbors
10 } }
11
12 public class Vehicles extends Agent {
13  private Object migrateNext() {
14    // decide a neighboring street segment to go
15    int[] dest = new int[index.length);
16    dest[0]=index[0] + ...;// from current to new x
17    dest[1]=index[1] + ...;// from current to new y
18    migrate(dest);// migration upon cars.manageAll
19 } }
```

Figure 2: MASS code snippets of the main program (left) and simulation components (right)

Figure 2 shows MASS-based code framework of generalized traffic simulation: the left snippet describes the main program, (i.e., a simulation scenario), whereas the right defines the *Mesh* and *Vehicles* classes, each modeling two-dimensional meshed streets and vehicles driving over them. The main function on the left instantiates *Mesh* over multi-processors (line 9) and populates *Vehicles* agents over the *Mesh* (line 12). The meshed street segments declare their north, east, south, and west neighbors (lines 15-19). After this set-up, the main program enters a cyclic simulation where all street segments exchange their traffic flows (line 23) and vehicles drive to a next street segment (lines 25-26).

It is natural for *callAll()* and *exchangeAll()* to call each element's function with a user-given string-class name. However, Java reflection is intolerably slow for parallel computing, and C/C++ dynamic-linking library does not resolve a method within a given object. Thus, a selection of methods to call should preferably be done with *switch()*, where we need to identify each method with an integer value. For this reason, a user must implement *callMethod()* that assists the MASS library in choosing a method to call (see lines 4-7 in Figure 2's right snippet). Each instance of *Mesh* and *Vehicles* class calls *trafficInfo()* (lines 8-10) or *migrateNext()* (lines 13-18) respectively via *callMethod* upon an invocation of *callAll*.

3.2 Parallelized MATSim's System Overview

We parallelized MATSim by extracting and maintaining its traffic network as the MASS library's *Places* array that is automatically distributed over a cluster of computing nodes, as shown in Figure 3. MATSim's simulation algorithm does not change at all but accesses this *Places* distributed array whenever it needs to manipulate the original traffic network. To be more precise, MATSim iterates the following sequence of four steps: (1) network parameter reloading, (2) vehicles' route assignment, (3) traffic simulation (which moves vehicles from one to another street segment), and (4) parameter exchange among adjacent street segments. These steps are applied to the traffic network that is actually distributed over multiple nodes. Therefore, all the cluster nodes perform each of these four steps in parallel onto their own subset of the *Places* array.

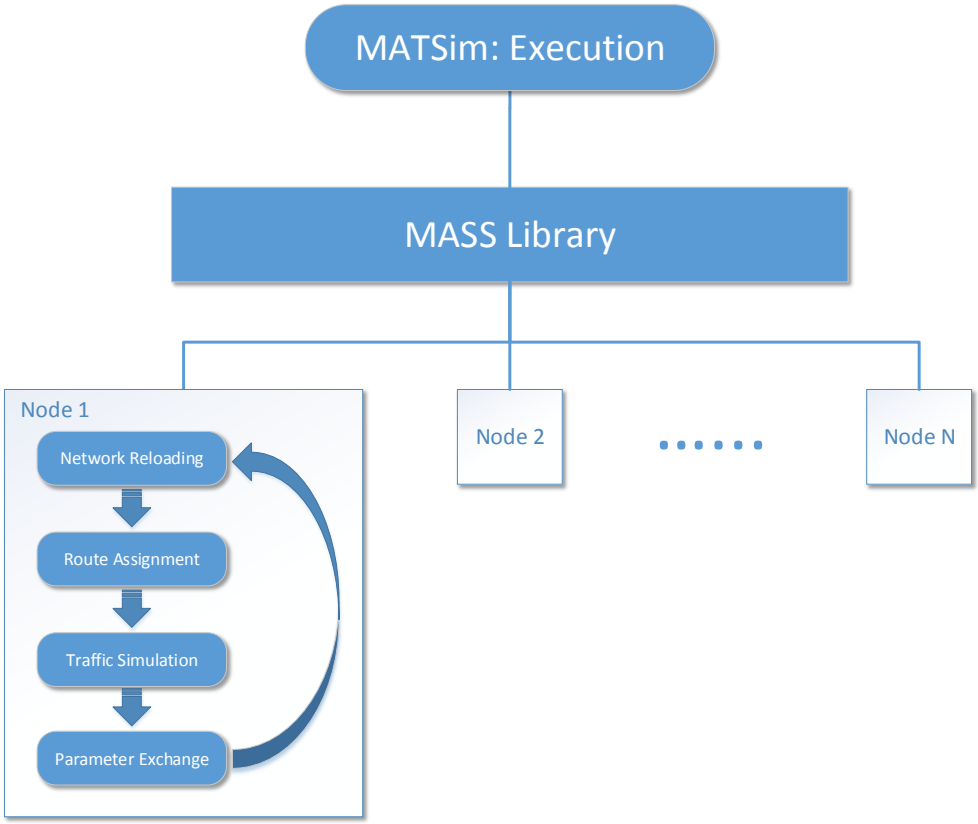

Figure 3: MASS-parallelized MATSim's system overview

3.3 Parallelization Techniques

In MATSim's traffic network, a node represents an intersection, and a link models a street segment from one to another intersection. To read such a traffic network into memory from MATSim's XML file, we used a collection of adjacency lists, where a node has a pair of single lists, each with incoming and outgoing links respectively. For instance, given a traffic network in Figure 4, node N_B has an incoming list of L_1 and an outgoing list of L_2 and L_4. The entire adjacency lists are then maintained as a *LinkedHashMap* object that distinguishes each node with a key, (i.e., a node ID) and retrieves this node's incoming and outgoing links as the value.

Upon a start of the MASS library, the main program instantiates a two-dimensional array of *Places* over a cluster of computing nodes and invokes a *callAll* function for initializing each *Place* element:

```
LinkedHashMap adjacencyLists;
Places network = new Places( "TrafficNetwork", NULL, size, size );
network.callAll( setNeighbor, adjacencyLists );
```

where $size = \sqrt{\#nodes + \#links}$. All *Places* receive the above-mentioned adjacency lists, (i.e., *LinkedHashMap*), based on which each *Place* is set to represent either a different node or link and to store its connectivity to all outgoing links or the next destination node in one of the private data members, named *neighbors*.

However, this mapping scheme from a MATSim traffic network to a MASS *Places* array encounters a restriction of the MASS library's *exchangeAll()* specification. As shown in lines 14-19 of Figure 2 (left), *exchangeAll()* assumes that all *Place* elements have the same indices for their adjacent neighbors to communicate with each other. These indices must be defined in a vector of *int* array before a cyclic simulation begins.

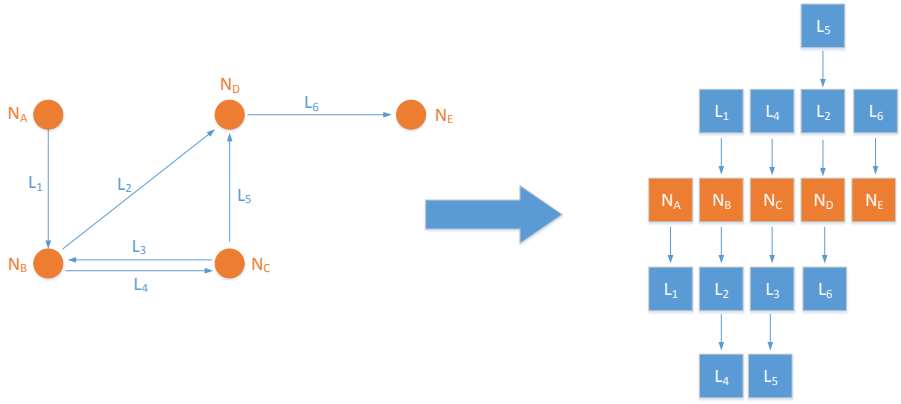

Figure 4: Network mapping using adjacent list

In our MATSim parallelization, each *Place* element must maintain its own adjacent neighbors in the *neighbors* variable that may not necessarily point to only north, east, south, and west neighbors but also some farther neighbors as illustrated in Figure 5. Moreover, each *Place* has a different number of adjacent neighbors, depending on the number of street segments emanating from the corresponding intersection. Therefore, we cannot provide *exchangeAll* with a uniform collection of *destinations*. For this reason, we added to *exchangeAll* a new feature that allows each *Place* element to customize its neighbors for exchanging data with them.

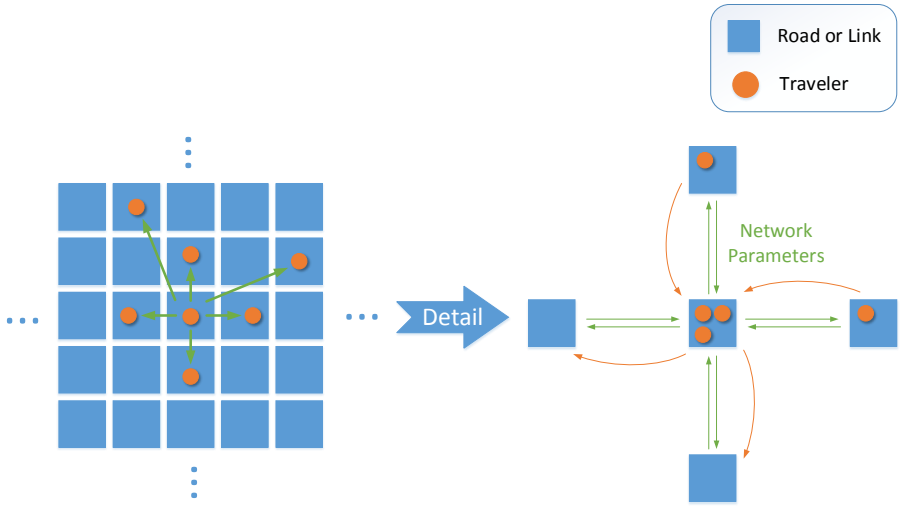

Figure 5: Data flow for network parameters

Once the MASS library maps a MATSim traffic network over a cluster of computing nodes, the MATSim simulation engine iterates data exchange and traffic simulation as shown in Figure 6 (left). The data exchange is performed in two steps: (1) sending traffic data to neighboring nodes and links through *exchangeAll* (line 5) and (2) receiving the data from neighbors through *callAll* (line 7). In step 1, *exchangeAll* has each *Place*, namely each node or link invoke the *exchangeParameter()* function lines 5-14) as shown in Figure 6 (right), which returns the speed of vehicles running on it, its capacity to receive new vehicles, its traffic flow capacity if it is a link, and the link's end-to-end distance. These parameters are then passed to all the neighboring *Places*' *inMessages* buffer. In step 2, *callAll* has each *Place* invoke the *collectParameter()* function (lines 16-24) in Figure 6 (right), which retrieves parameters

from the *inMessages* buffer. Of importance is that each node or link can repeat such parameter exchange without being aware of the location and adjacency of their neighbors and furthermore without controlling the underlying inter-processor communication and synchronization.

After traffic data exchange, the MATSim simulation engine updates the traffic status of all nodes and links through two *callAll* invocations, each calling *doSimStepNode* and *doSimStepLink* of all *Place* elements in parallel (lines 9 and 10 respectively) in Figure 6 (left). We emphasize that these two functions are MATSim-original and thus no modification was made to them for our parallelization work.

```
1  public class MatSimSimulationEngine {
2   private Places network; a collection of TrafficNetwork
3    public void run() {
4      // send traffic info to neighboring nodes/links
5     network.exchaneAll(1,exchangeParameter);
6      // retrieve traffic info from neighboring nodes/links
7      network.callAll(collectParameter);
8      // simulate new traffic flow of all nodes/links
9      network.callAll(doSimStepNode, time);
10     network.callAll(doSimStepLink, time);
11 } } }
```

```
1  public class TrafficNetwork exnteds Place {
2    private String type;// Node or Link
3    private double speed, capacity, . . ;//my data
4    // called from exchangeAll to send my data to neighbors
5    public Object exchangeParameter() {
6      Object[] parameters=new Object[4];
7      parameters[1]=this.speed;
8      parameters[2]=this.capacity;
9      if (type.equlas("Link")) {
10       parameters[3]=this.flowCapacity;
11       parameters[4]=this.distance;
12     }
13     return parameters;// sent to all neighbors
14   }
15   // called from callAll to retrieve neighbors' data
16   public void collectParameter() {
17     Object[] parameters=getInMessages();
18     if (parameters.length > 0) {
19       this.speed=(double)parameters[0]
20       this.capacity=(double)parameters[1]
21       if (type.equlas("Link")) {
22         this.flowCapacity=(double)parameters[2]
23         this.distance=(double)paramater[3]
24   } } }
25 }
```

Figure 6: MASS-parallelized MATSim simulation engine (left) and traffic network (right)

4 PORTABILITY AND PERFORMANCE EVALUATION

This section demonstrates the MASS library's efficient portability and usability as well as its moderate level of performance improvement when applying it to parallelization of transport simulations. We also discuss future performance improvements to make MASS more competitive.

4.1 Portability and Usability

Our MATSim parallelization techniques described in Section 3.3 imply the following four programmability merits: (1) automatic logical-to-physical network mapping, (2) unawareness of the underlying inter-processor communication and synchronization, (3) reuse of the original simulation logics, and (4) maximized use of the underlying computing resources.

All traffic planners and simulation modelers are not computing specialists. In particular, parallel computing requires them to obtain special programming skills and debugging experiences in multithreading, inter-processor communication, and synchronization among processes and threads. The MASS library's merits 1 and 2 relieve such users from these programming burdens. Therefore, they can put more focus on their model design.

The percentage of modified or added code portion in the original source also gives a considerable impact to parallelization of traffic simulations. To parallelize MATSim with the MASS library, we added 583 lines of code in eight new files to the original core source whose total size is 5,144 lines in 46 files,

which corresponds to only 11.3% and 17.4% changes respectively in terms of code lines and the number of files. In fact, we haven't touched the actual simulation logic such as *doSimStepNode* and *doSimStepLink* at all. Therefore, we feel that the MASS library contributes to merit 3: reuse of the original MATSim logics.

The MASS library executes a given simulation with communicating multi-threaded processes, each running on a different cluster node and utilizing all CPU cores by spawning the same number of threads. So far, MATSim made available its multithreaded version to the public. In case if users hope to parallelize MATSim over a cluster system for both CPU and memory scale-up purposes, the most conventional approaches are to use JavaSpaces (JS - JavaSpaces Servie Specification) or mpiJava (mpiJava Home Page): the former is based on the concept of shared memory, whereas the latter is the paradigm of message passing. Although JavaSpaces can automate the distribution of a MATSim traffic network over multiple JavaSpaces servers, an actual simulation must be performed at a client side, which does not contribute to scaling up CPU power. On the other hand, mpiJava allows both a traffic network and its simulation to be distributed over multiple computing nodes. However, developers themselves are responsible for calculating node boundaries and synchronizing all computing nodes when using MPI to exchange traffic data, which drastically increases programming burdens. As a result, the MASS library supports merit 4: effectively utilizing the underlying computing resources.

We have also conducted a survey on the MASS library's general programmability and usability in our graduate course: CSS534 Parallel Programming (CSS534 Final Project: Coding a Parallel Application with MASS). The survey elucidates the following pros and cons of the MASS library, which can be applied to parallelization of traffic simulations.

Pros:

1. MASS users are relieved from details of parallelization, which drastically reduces parallelized code and actual coding efforts (as discussed in the above merits 2 and 3).
2. The MASS library gives graphical visualization and debugging tools, which contributes to mitigation of users' debugging efforts.

Cons:

1. Users who get used to popular parallelization techniques such as Java threads, mpiJava, and hybrid MPI/OpenMP in C++ need a paradigm shift to agent-based modeling, which makes users feel that the MASS library's learning curve is very steep.
2. Since the MASS library is still on its pre-release stage, the users feel that they need to contact the library developers for receiving more information on performance tuning.

4.2 Execution Performance

For performance analysis, we have chosen a sample simulation scenario provided by the MATSim code repository. This scenario declares 4,700 nodes and 13,000 links in an XML file. We use a Giga Ethernet cluster of 16 DELL computing nodes, each with 1.6GHz 4-core i7 CPU and 16GB memory, which is connected to a dual-processor NFS server.

Figure 7 compares the original MATSim's sequential execution (in orange), the publicly available multithreaded MATSim's single execution (in green), and MASS-parallelized MATSim execution (in blue) using 1, 2, 4, and 8 computing nodes, all single threaded. The results show that MASS performed slightly better than the original MATSim did when running with 4 or 8 nodes.

Figure 8 compares the multithreaded MATSim (in green), MASS with two computing nodes (in blue), and MASS with four computing nodes (in orange) in terms of performance improvements using 1, 2, and 4 threads. Note that the multithreaded MATSim runs only on a single computing node. As the results turned out, MASS with four computing nodes with four threads performed twice better than the original MATSim sequential execution. However, MASS with two computing nodes slows down when increasing the number of threads from two to four. This performance degradation resulted from unbalanced logical-to-physical

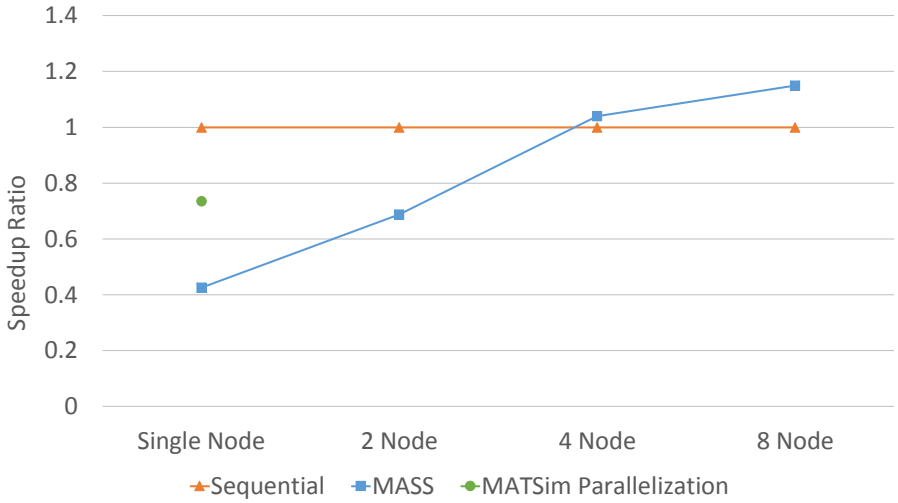

Figure 7: Execution comparison with sequential version

network mapping in the MASS-parallelized MATSim. Its network data is read into MASS Places in the exact order of the input XML file that declares all nodes, (i.e., intersections) first and thereafter all links, (i.e., road segments). When MASS uses only two computers, the first computer reads all node elements and some link elements in MASS Places, whereas the second computer reads the rest of link elements. Since simulation updates the status of vehicles in links, the second computer is overloaded with many links. To be worse, since vehicle agents migrate from one to another link through their incident node, they tend to move from the second to the first computer and immediately back to the second computer. Therefore, the two-computer configuration does not bring a big performance improvement at all, in which situation a higher degree of multithreading even increases thread management overheads and cache-line thrashing among CPU cores.

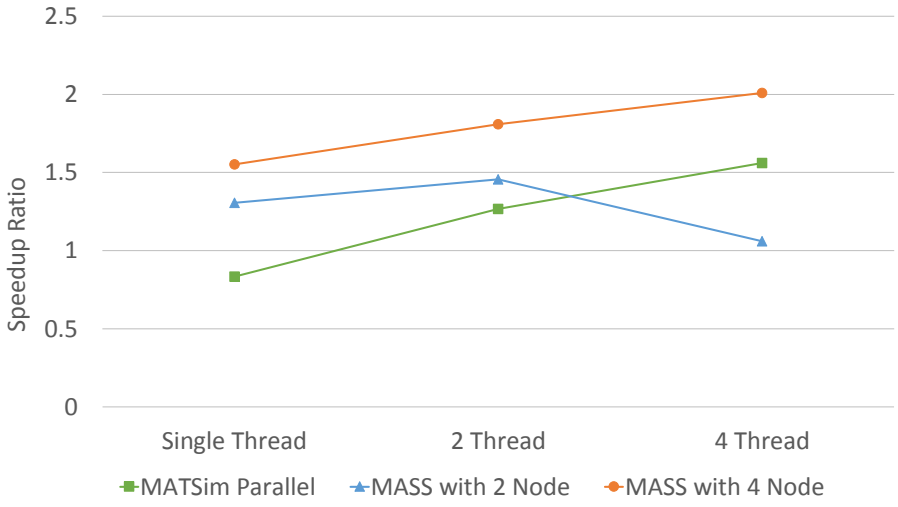

Figure 8: Execution comparison with parallelized MATSim

There are two reasons for obstructing the MASS performance improvements. One is the current *exchangeAll* implementation that copies all the neighbors' traffic data to each *Place*'s *inMessage* buffer. To mitigate these data-copying overheads, if these neighboring *Place*s reside on the same local machine, only

a reference to their traffic data should be exchanged. Another reason is the current scheme to map nodes and links to MASS *Place* elements as mentioned above. These nodes and links are assigned to *Place[0][0], [0][1], [0][2], ..., [size-1][size-1]* respectively in the order of reading them from a given XML scenario file. This causes a load imbalance and increases the number of messages exchanged among different computing nodes whenever invoking *exchangeAll*. To address these problems, node and link declarations in XML should be sorted for their logical proximity.

4.3 Further Discussions

To make MASS contribute more to parallelization of traffic simulation not only from its efficient portability and usability but also its scalable performance, we are considering the following performance improvements:

1. **Asynchronous agent migration:** MASS agents need a synchronous invocation of *Agents.manageAll()* for each cycle of their actual migration. This is the constraint that leaves MASS-parallelized simulation to run in conservative scheduling. Therefore, we are currently revising MASS to support asynchronous migration without invoking *Agents.manageAll()*.
2. **Pool of idle agents:** A numerous repetition of memory allocation and de-allocation kills multi-threaded parallelization. Pooling idle agents is expected to mitigate this repetition so as to use heap space more effectively.
3. **Various time management:** Once asynchronous agent migration is made available, we will facilitate optimistic, breathing-bucket, and conservative time management, so that the MASS can adapt the best scheduling strategy to a given traffic simulation. For instance, we feel that optimistic scheduling would work better to MATSim parallelization. On the other hand, if MASS C++ is used to parallelize TRANSIMS, conservative scheduling would work best due to the nature of TRANSIMS' implementation with cellular automata.

5 CONCLUSIONS

This paper presented a parallelization of the MATSim transport simulator, using the MASS library. We discussed the library's portability, usability, and execution performance when parallelizing transport simulations. The MASS library's efficient portability and usability were demonstrated for transport simulation in both quantitative and qualitative analyses. On the other hand, its execution performance still needs to be improved although our results showed some parallelization benefits as compared to the sequential execution. By tuning up the current implementation of the MASS *exchangeAll()* function and improving our logical-to-physical network mapping scheme, we believe that MASS has potential to serve as an efficient parallelization library for MATSim and other transport simulation systems. Our next plan is to apply MASS C++ to TRANSIMS' micro-simulator whose traffic network is maintained in cellular automata and thus can be smoothly ported to MASS Places.

ACKNOWLEDGMENTS

We are very grateful to Mr. Matthew Sell, a software engineer at Fluke Corporation and a UWB undergraduate student for all his technical support in building the MASS library development environment.

REFERENCES

Balmer, M., K. Meister, K. Nagel, and K.W.Axhausen. 2008, July. "Agent-based simulation of travel demand: Structure and computational performance of MATSim-T". Work report no.504, Institute for Transport Planning and Systems, ETH Zurich, Zurich, Switzerland.

Barcelo, J., J. Ferrer, D. Garcia, M. Florian, and E. Saux. 1992, December. "Parallelization of Microscopic Traffic Simulation for ATT Systems Analysis". *Journal de Physique* Vol.2 (No.12): 2221–2229.

Barrett, C. L. et al. 1999, May 28. "TRANSSIMS(TRansportation ANalysis SIMulation System) Volume 0 - Overview". La-ur-99-1658, Los Alamos National Laboratory.

Cameron, G., and G. Duncan. 1996, January. "PARAMICSParallel Microscopic simulation of road traffic". *Journal of Supercomputing* Vol.10 (No.1): 25–53.

Chuang, T., and M. Fukuda. 2013, December. "A Parallel Multi-Agent Spatial Simulation Environment for Cluster Systems". In *Proc. 16th IEEE International Conference on Computational Science and Engineering - CSE2013*, to appear. Sydney, Australia: IEEE CS.

CSS534 Final Project: Coding a Parallel Application with MASS. "http://courses.washington.edu/css534/prog/prog4.pdf".

Fukuda, M. 2010, May. "MASS: Parallel-Computing Library for Multi-Agent Spatial Simulation". http://depts.washington.edu/dslab/sensorgrid/doc/massspec.pdf, Distributed Systems Laboratory, Computing & Software Systems, University of Washington Bothell, Bothell, WA.

JS - JavaSpaces Servie Specification. "https://river.apache.org/doc/specs/html/js-spec.html".

MATSim Homepage 2012. "http://www.matsim.org".

mpiJava Home Page. "http://www.hpjava.org/mpiJava.html".

Nagel, K., and F. Marchal. 2003, August. "Computational methods for multi-agent simulations of travel behavior". In *Proc. of International Association for Travel Behavior Research (IATBR)*. Lucerne, Switzerland: ETH Zurich.

Rickert, M., and K. Nagel. 2001, March. "Dynamic traffic assignment on parallel computers in TRANSIMS". *Future Generation Computer Systems* Vol.17 (No.5): 637–648.

AUTHOR BIOGRAPHIES

ZHIYUAN MA is a graduate student in Master of Science in Computer Science and Software Engineering at the University of Washington Bothell. His research interests include mobile agents and parallel computing. His email address is zachma@uw.edu.

MUNEHIRO FUKUDA is a professor of the Division of Computing and Software Systems, the School of Science, Technology, Engineering, and Mathematics at the University of Washington Bothell. He holds a Ph.D. in Information and Computer Science from University of California, Irvine. His research interests include multi-agent systems, mobile agents, agent-based simulation, and parallel computing. His email address is mfukuda@uw.edu.