A Multi-threaded Execution Model for the Agent-Based SEMSim Traffic Simulation*

Heiko Aydt[1], Yadong Xu[1,2], Michael Lees[3], and Alois Knoll[4]

[1] TUM CREATE Ltd., Singapore
[2] Nanyang Technological University, Singapore
[3] University of Amsterdam, The Netherlands
[4] Technical University of Munich, Germany

Abstract. An efficient simulation execution engine is crucial for agent-based traffic simulation. Depending on the size of the simulation scenario the execution engine would have to update several thousand agents during a single time step. This update may also include route calculations which are computationally expensive. The ability to dynamically re-calculate the route of agents is a feature often not required in classical microscopic traffic simulations. However, for the agent-based traffic simulation which is part of the Scalable Electro-Mobility Simulation (SEMSim) platform, the routing ability of agents is an important feature. In this paper, we describe a multi-threaded simulation engine that explicitly supports routing capabilities for every agent. In addition, we analyse the efficiency and performance of our execution model in the context of a Singapore-based simulation scenario.

1 Introduction

In the context of electro-mobility research we are currently developing the SEMSim (Scalable Electro-Mobility Simulation) platform. This platform will provide the capability to simulate the various aspects related to electro-mobility for an entire city. In particular, this includes microscopic simulation of the traffic system (SEMSim Traffic) and microscopic simulation of the power system (SEMSim Power). SEMSim Traffic is more precisely an agent-based traffic simulation where each agent represents a driver-vehicle unit. This agent not only integrates typical driver behaviour models (e.g., car following, gap acceptance and lane changing models) but also explicit vehicle component models (e.g., drive train, battery, air conditioning).

Integrating these models and the ability to couple SEMSim Traffic with SEMSim Power allows us to study the impact of the disruptive technology 'electric vehicle' on the traffic and power infrastructure of an entire city. For example, the SEMSim platform will allow us to study the impact of the electric vehicle

* This work was financially supported by the Singapore National Research Foundation under its Campus for Research Excellence And Technological Enterprise (CREATE) programme.

G. Tan et al. (Eds.): AsiaSim 2013, CCIS 402, pp. 1–12, 2013.

population on the power system under various electro-mobility scenarios (e.g., placement of charging infrastructure, vehicle design, regulations and operating policies). The goal of this research is to develop the capability to analyse an entire city, such as Singapore, from a complex systems perspective.

Coupling of the different simulation entities (e.g., SEMSim Traffic and SEMSim Power) will be done by means of the High Level Architecture (HLA)[1]. An electro-mobility simulation will thus be distributed across multiple nodes. In addition, SEMSim Traffic will also be parallelised to deal with the large number of agents and the variety of models they incorporate. Parallelisation can be done in multiple ways which we will further discuss in Section 2. In this paper we focus on parallelisation by means of using a multi-threaded execution engine.

We follow a bottom-up modelling approach for developing the SEMSim platform. As already mentioned, for SEMSim Traffic this means that individual agents will be equipped with driver behaviour and vehicle component models. We therefore also refer to this kind of simulation as nanoscopic traffic simulation, as opposed to microscopic traffic simulation which does not consider vehicle component models for individual agents. Large-scale microscopic traffic simulations are known to be very compute intensive. The additional vehicle models make a nanoscopic traffic simulation even more computationally demanding. An efficient simulation execution engine is thus very important.

In general, the agents in an agent-based simulation have the ability to react upon perceived changes in the environment. In the context of agent-based traffic simulation, for example, this refers to the ability of an agent to plan routes and drive from its current location to the destination. Driving requires the agent to accelerate/decelerate to achieve the desired velocity. In addition, it may have to change lanes at appropriate locations, avoiding collisions with other vehicles at any time. While dynamic re-routing may not be needed in many microscopic traffic simulations, this is an important feature in SEMSim Traffic. For example, it enables agents to change their routes towards the nearest charging station if necessary. In this paper, we describe a multi-threaded simulation execution engine that explicitly supports routing capabilities for every agent.

The remainder of this paper is structured as follows. Section 2 gives an overview of the different parallelisation techniques used for microscopic traffic simulations. Section 3 and Section 4 explain the underlying SEMSim Traffic simulation model and execution engine, respectively. We evaluate the proposed execution engine and discuss the experimental results in Section 5. We present our conclusions and discuss future work in Section 6.

2 Related Work

Parallel computing methods are used by a variety of existing microscopic traffic simulators. In general, parallelisation can be achieved in two ways: (1) by distributing the simulation on multiple nodes by using some form of partitioning and (2) by using multi-threading to exploit the advantages of shared-memory systems. Most work published in the literature is concerned with distributing

a simulation on multiple nodes. For this purpose, it is generally distinguished between functional decompositioning and domain decompositioning. Functional decompositioning can be found in DYNAMIT/MITSIM [3,4]. However, according to [9] functional decompositioning may be easier to implement but poses limitations on the achievable speed-up.

Using domain compositioning for parallel traffic simulation has been described in various works. Nagel and Rickert describe how domain decompositioning has been used in a parallel implementation of TRANSIMS [9] which is based on cellular automata for representing driving dynamics. Different geographical regions are processed by multiple CPUs. Various key challenges for domain decompositioning, concerned with dividing the network, is explained by Klefstad et al. [8]. Yet another parallel traffic simulation that uses domain decompositioning is FastTrans. Thulasidasan et al. describe different strategies to perform domain decompositioning and show how different strategies affect the simulation performance [13].

Distributing a simulation on multiple nodes has several advantages. For example, large-scale simulations may be too large to fit into the memory of a single compute node. If memory limitations are not an issue, it is in principle also possible to distribute the simulation on multiple cores on the same compute node. However, in this case, the advantages of a shared memory system may not be fully utilised. In principle, a distributed simulation is not limited to the number of processors (or cores). Of course, scalability limits the amount of cores that can effectively be used.

In contrast to distributed simulation, one of the main disadvantages of the multi-threading approach is the total number of available processors (or cores) which is limited to the number of processors available on a single compute node. As of 2013, high-end processors (e.g., Intel Xeon E7 family) provide up to 10 cores with support for hyper-threading which effectively doubles the number of threads that be executed concurrently. Another limitation is the memory available. Large-scale simulations that require large amounts of memory may be too big to fit into the memory of one compute node. Nevertheless, one important advantage of multi-threading solutions is the significantly reduced communication overhead needed for synchronisation. Given the current trend towards many-core processors, multi-threading solutions deserve serious consideration.

The work by Barceló et al. [2] is very relevant to the work presented in this paper. They describe the multi-threaded execution engine used for AIMSUN2 and analyse its performance. Their execution engine is based on the idea of grouping parts of the road network (i.e., lanes) together in such a way as to minimise the need for synchronisation. This is the same approach taken by us in this paper. A key difference between AIMSUN2 and SEMSim Traffic is the way how traffic is generated. According to [2], vehicles in AIMSUN2 do not have knowledge about their complete path along the network. In contrast, the agents in our traffic simulation have complete routes to follow in order to reach the destination. This is an important difference because route calculation is computationally expensive and the execution engine presented here takes this explicitly into consideration.

A noteworthy and very recent development is the IBM Mega Traffic Simulator (Megaffic), a microscopic traffic simulation that aims for being used in large-scale traffic simulation of mega cities [10]. Megaffic is based on a platform for massive agent-based simulation: X10-based Agent eXecutive Infrastructure for Simulation (XAXIS) [11]. Suzumura et al. evaluate the performance in terms of scalability, including an analysis of the scalability on a single node with a different number of threads [12], showing that Megaffic achieves a five-fold speed-up when using 12 threads.

3 Simulation Model

The road network is a directed graph in which edges represent lanes and vertices represent the start and end points of a lane in terms of longitude and latitude coordinates. The length L of a lane follows from the geographical distance between the start and end point of this lane. A lane is modelled by a spatial queue, i.e., a queue of agents in which each agent has also a spatial location relative to the start point in addition to its logical position in the queue. Figure 1 illustrates the concept of spatial queues. Agents are not directly placed in the queue. Instead, place holder objects that have the same size as the agent are placed in the queue. These place holder objects have a reference to the corresponding agent.

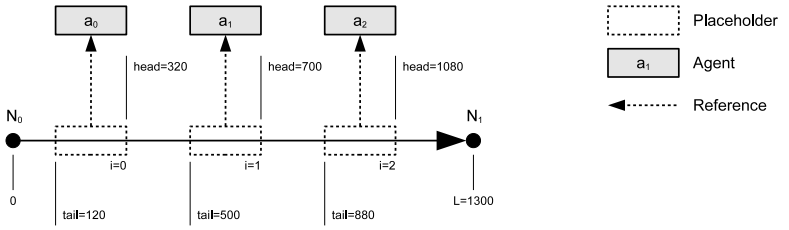


Fig. 1. A lane with an origin N_0 and destination N_1 is represented by a spatial queue with length L. This queue can contain an arbitrary number of agents. The index i indicates the logical position of an agent within the queue. In addition, the spatial position of the agent is indicated as distance from the origin of the lane N_0 to the head and tail of the vehicle.

When moving forward, agents can crossover from one lane to the next. During that transition period, an agent can be partially located on one lane and partially on the next (see Figure 2(a)). Technically this is done by having two place holder objects in either queue, both of which are referring to the same agent (see Figure 2(b)). Each time step, the place holder objects of the various agents in the simulation are moved forward by some distance Δs which depends on the agents current speed. Once an agent is leaving a lane, i.e., when its tail reaches the end of the lane, the place holder object is removed. Similarly, a new place holder object is created when an agent is entering a lane when its head reaches the beginning of the lane.

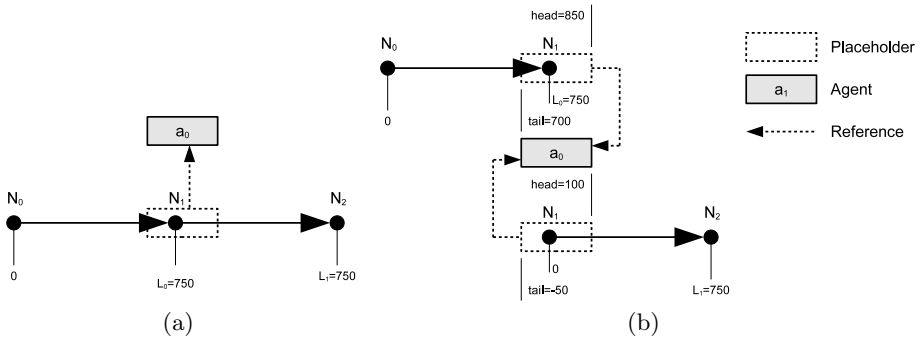


Fig. 2. An agent a_0 which is overlapping on two lanes (a) has place holder objects in each lane that reserve the required space for the agent (b). When the agent is moved forward, all place holder objects associated with this agent need to be moved. If a place holder object leaves the lane entirely, it is deleted and the agent is entirely part of the next lane.

This kind of crossing over is not to be confused with lane changes to adjacent lanes to the left or right for which there is no transition period.

Agent-based traffic simulation requires an agent to perceive its environment and executing certain actions which include route calculation, lane changes (if necessary or desired) and moving forward with a certain acceleration and speed. There are a number of standard models for lane changing and acceleration which can be found in the literature. For example, car following models (e.g., [7] and [14]) are used to determine the acceleration of the agent at every time step. This typically depends on the current speed and the distance to the next car in front as well as the speed of the car in front. In this paper, we focus on the following three operations an agent can perform: (1) routing, (2) lane changing and (3) moving forward (by accelerating and decelerating).

4 Execution Model

Microscopic traffic simulators typically generate fixed traffic before executing the simulation. Two common practices are using origin-destination matrices (e.g., PARAMICS [5]) and stochastic turning ratios at road junctions (e.g., AIMSUN [2] and VisSim [6]). For example, in case of AIMSUN, agents do not have knowledge about the complete route. Instead their driving directions are based on some probabilistic decision model. While this execution model avoids calculating entire routes, it would be too restrictive in the context of nanoscopic traffic simulation where energy consumption of individual agents is being considered. An important feature of the execution model presented in this section is the routing feature which is also the key difference to the multi-threaded execution engine used in AIMSUN/MT [2]

In principal, a time-stepped execution models is used as the underlying models involved in moving agents (such as the car following model) are inherently time-stepped. However, for any other behavioural models (e.g., decision models) or vehicle component models this may not be case. Therefore, in SEMSim Traffic we use a discrete-event engine that allows to schedule events at arbitrary moments of time. The event triggering of the execution model will be scheduled recurringly with a certain time interval (typically less or equal to 1 second), thus effectively emulating time-stepped behaviour. This approach enables to use time-stepped models alongside with model components that schedule events infrequently.

Since the scope of this paper is the execution model only, we consider time-stepped execution of the simulation where agents are updated once every time step. During each of these updates, an agent may re-calculate its route, change lanes if necessary and move forward. In a multi-threaded by-agent parallel execution model, all agents are distributed to multiple threads each of which is executing the various operations. This naive form of parallel execution may cause significant synchronisation overhead: initialising an agent (i.e., placing an agent on a lane), changing lanes or moving forward requires information of the state of multiple lanes. In order to maintain the integrity of a spatial queue, it is important that only one thread is operating on a queue at a time.

There are five synchronisation cases that requires threads to have varying mutually exclusive access to lanes: (1) inserting an agent into a queue (i.e., placing an agent onto a lane), (2) changing lanes, (3) moving forward without overlaps, (4) moving forward with overlap to the next queue and (5) moving forward with overlap to the previous queue. These cases are illustrated in Figure 3. A thread that operates on the granularity of agents needs mutually exclusive access to various spatial queues. During this operation, none of the other agents that are contained by the affected lanes can be processed as the exclusive access concerns the entire spatial queue.

Efficient parallelisation requires to minimise the need for synchronisation. This can be achieved by forming blocks of queues that are direct or indirect neighbours to each other. Neighbouring lanes are those lanes that can be reached by changing lanes. For example, if a road splits into two different directions, then the lanes of those two alternatives are not considered neighbours since lane changing between them is impossible. Figure 4 shows an example how several spatial queues are grouped together to form blocks.

Based on the concept of blocks, we can now introduce an alternative execution engine which performs block processing rather than processing by agents. The execution model is thus referred to as by-block execution model. The by-block parallel execution model has the advantage that threads operate on the granularity of blocks rather than agents. As a consequence, the above-mentioned synchronisation cases (1), (2) and (3) are completely eliminated as agent placement on lanes, lane changing and moving forward without overlaps is done within the boundaries of a single block. Since only one thread can process a block, there are only potential needs for synchronisation in cases (4) and (5).

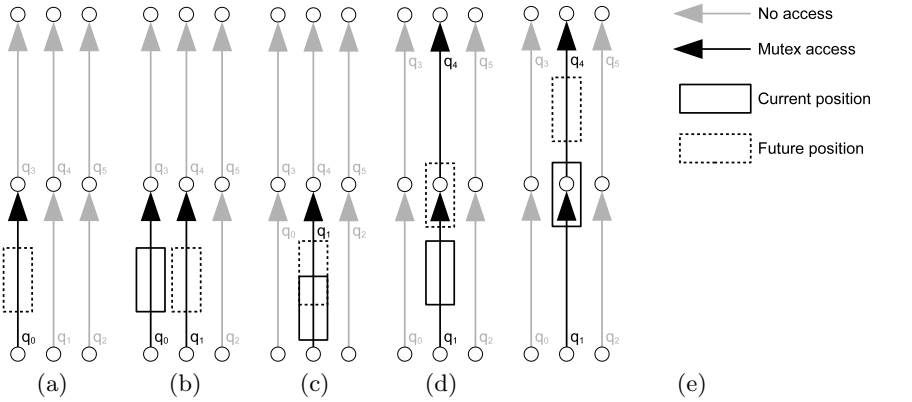


Fig. 3. Overview of the various synchronisation cases where threads need mutually exclusive access to spatial queues. Inserting an agent into a queue requires mutually exclusive access of this queue (see 3(a)). Changing lanes requires mutually exclusive access to the current lane and the target lane (see 3(b)). Moving forward requires mutually exclusive access at least to the current lane (see 3(c)). In case the agent is moving from one queue to another, mutually exclusive access to the next or previous queue is also needed (see 3(d) and 3(e), respectively).

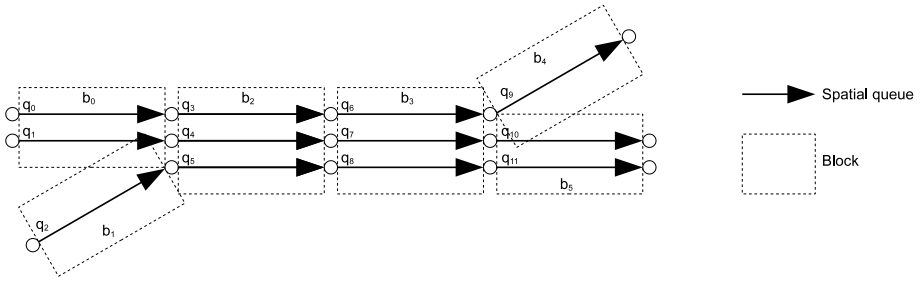


Fig. 4. Overview of grouping spatial queues to blocks. Spatial queues that belong to different branches are not grouped into one block because lane changing between them is impossible. For example, spatial queues q_0 and q_1 belong to a different block than spatial queue q_2.

We further reduce the probability of synchronisation by declaring blocks either as 'even' blocks or 'odd' blocks and processing them in batches. Which block is declared 'even' or 'odd' is arbitrary and not important so as long it is guaranteed that adjacent blocks are never of the same group. For example, consider Figure 4. If block b_0 is, say, 'even' then it follows that blocks b_1, b_2, b_3, b_4 and b_5 are 'even', 'odd', 'even', 'odd' and 'odd', respectively. Even and odd blocks are also kept separately in different sets B_0 and B_1. The first batch will process even blocks and the second batch will process odd blocks.

The by-block execution model, illustrated in Algorithm 1, performs two different steps. Step 1 is concerned with updating the routes of newly created agents or existing agents that need to change their route. Step 2 processes the blocks and updates the position of the agents in the various blocks. Step 2 is executed twice, first for the even blocks in B_0 and then for the odd blocks in B_1. A multi-threading parallel execution engine can be realised by having n threads performing the necessary operations during Step 1 and Step 2. For this purpose, each Step 1 worker thread removes one agent a from A and processes it. Similarly, Step 2 worker threads remove a block b from B and process it. Worker threads will continue until agents in A or all blocks in B have been processed. The processed agents and blocks are then consolidated and represent the set of agents and blocks that are being processed in the next cycle.

Algorithm 1. By-block execution model

$A \leftarrow \emptyset$;
$B_0 \leftarrow \emptyset$;
$B_1 \leftarrow \emptyset$;
repeat
  $t \leftarrow t + \delta t$;
  $A \leftarrow A \cup$ generate_new_agents(t);
  perform_step1(A);
  perform_step2(B_0);
  perform_step2(B_1);
until $A = \emptyset$;

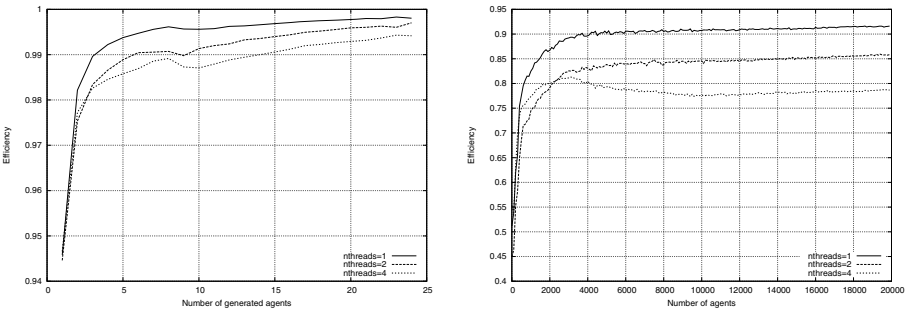
5 Experimental Evaluation

We evaluate the proposed parallel by-block execution model in two ways. First, we analyse the efficiency during the execution of Step 1 and Step 2 depending on the number of agents for an increasing number of threads. Second, we analyse the achieved speed-up depending on the number of agents for an increasing number of threads. For evaluation purposes, a Java implementation of the execution engine has been used. For the simulation scenario, we use road network of Singapore and generate traffic up to 20,000 agents in a 2-stage experiments. In Stage 1, the simulation generates agents up to a limit of 20,000 agents at which point Stage 2 begins. In Stage 2 no new agents are generated and no route calculations take place. The 2-stage approach is used in order to see the impact of Step 1 on the overall performance. The experiments have been performed on a system equipped with one Intel i5-2520M CPU with 4 cores, running at 2.5 GHz, and 8 GB memory.

We compare the efficiency E_1 and E_2 of Step 1 and Step 2, respectively, by measuring the time spent on useful computation t_u in comparison to the total time t_s spent to perform a step:

$$E = \frac{t_u}{t_s} \tag{1}$$

Overhead is caused by thread synchronisation and the time required to initialise the worker threads and the idling time by worker threads that need to wait for other threads to finish. In particular the latter can cause significant overhead if all threads except one have already finished their execution and need to wait for the last thread to finish. The overhead can be expected to increase with the number of threads. This is also supported by our experimental results, shown in Figure 5.



(a) (b)

Fig. 5. Efficiency of Step 1 (5(a)) and Step 2 (5(b)) execution depending on the number of threads

Figure 5 illustrates the efficiency of Step 1 and Step 2 depending on the number of threads used. The efficiency for Step 1 is significantly better than the efficiency for Step 2. This indicates that Step 2 is of greater importance in order to achieve good scalability. Figure 6 shows the average number of thread synchronisations during a single Step 2 execution. These results indicate that synchronisation is not a significant obstacle in order to achieve good scalability. Inefficiencies in Step 2 are thus due to thread initialisation and idling.

Regardless the issue of efficiency, the actual time spent on Step 1 and Step 2 is very different. Figure 7 illustrates the execution speed of the simulation during Stage 1 (with routing and thus with Step 1) and Stage 2 (without routing and thus without Step 1). Simulation execution during Stage 2 is faster than during Stage 1 by the order of roughly one magnitude. This somewhat reduces the impact of lower efficiency for Step 2 as Step 1 is more important in terms of execution time. However, as we explained in Section 1, SEMSim Traffic will

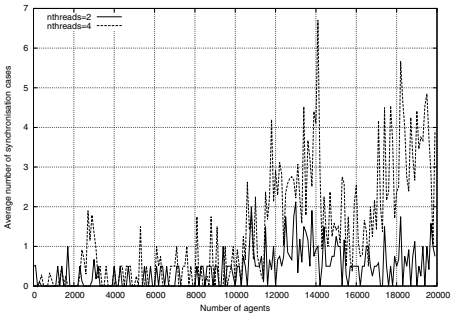


Fig. 6. Average number of synchronisation cases during Step 2 depending on the number of threads and agents

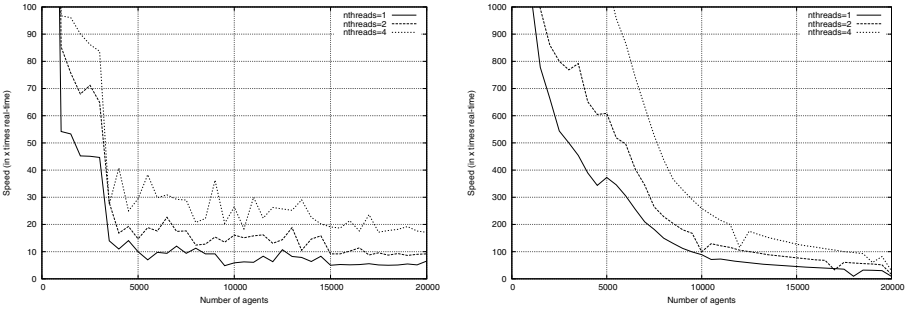


Fig. 7. Execution speed of the simulation in terms multiple to real-time for Stage 1 (7(a)) and Stage 2 (7(b))

incorporate vehicle component models that need updating which may be triggered during Step 2. This may change the proportions of Step 1 and 2 to the overall execution time significantly.

6 Conclusions and Future Work

An efficient execution model is crucial for large-scale agent-based traffic simulation of an entire city. An important feature of any agent-based simulation is the ability of agents to react to perceived changes in the environment. In the context of agent-based traffic simulation, for example, this refers to the ability of the agent to change its route. This feature is often not supported by many microscopic traffic simulations that use a more or less static form of traffic assignment. The AIMSUN2 engine, to which our approach is most closely related, does not consider explicit routing. In contrast, SEMSim Traffic requires agents to be able to change their routes. Our execution engine thus explicitly includes the case of routing in a two-step execution model.

Our results indicate that the simulation time is dominated by Step 1 (routing) for which a high degree of efficiency can be achieved. Future work will focus on improvements for the efficiency of Step 2 as well as the evaluation of our multi-threading execution engine on more cores. Furthermore, we will investigate possible hybrid approaches that combine the advantages of distributing the simulation on multiple multi-core shared memory nodes.

A by-agent execution model, in which all agents are updated concurrently by different threads, may be the more natural execution model for an agent-based simulation. However, it will also inevitably lead to more synchronisation cases. In contrast, the by-block execution model minimises the need for synchronisation and does not alter the agent-based model itself, i.e., agents are updated in exactly the same way in both execution models. The only difference is the actual sequence in which agents are being updated. This may affect the behaviour of the model. Future work will thus investigate the qualitative differences between the two execution models.

References

1. IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)– Framework and Rules. IEEE Std 1516-2010 (Revision of IEEE Std 1516-2000), pp. 1–38 (2010)
2. Barceló, J., Ferrer, J.L., Garcia, D.: Microscopic traffic simulation for ATT systems analysis: a parallel computing version. In: 25th Anniversary of CRT, pp. 1–16 (1998)
3. Ben-Akiva, M., Koutsopoulos, H.N., Antoniou, C., Balakrishna, R.: Traffic Simulation with DynaMIT. In: Barceló, J. (ed.) Fundamentals of Traffic Simulation. International Series in Operations Research & Management Science, vol. 145, pp. 363–398. Springer, New York (2010)
4. Ben-Akiva, M., Koutsopoulos, H.N., Toledo, T., Yang, Q., Choudhury, C.F., Antoniou, C., Balakrishna, R., Barceló, J.: Traffic Simulation with MITSIMLab. In: Barceló, J. (ed.) Fundamentals of Traffic Simulation. International Series in Operations Research & Management Science, vol. 145, pp. 233–268. Springer, New York (2010)
5. Cameron, G.D.B.: PARAMICS–Parallel Microscopic Simulation of Road Traffic. The Journal of Supercomputing 53, 25–53 (1996)
6. Fellendorf, M., Vortisch, P., Barceló, J.: Microscopic Traffic Flow Simulator VISSIM. In: Barceló, J. (ed.) Fundamentals of Traffic Simulation. International Series in Operations Research & Management Science, vol. 145, pp. 63–93. Springer, New York (2010)
7. Gipps, P.G.: A behavioural car-following model for computer simulation. Transportation Research Part B: Methodological 15(2), 105–111 (1981)
8. Klefstad, R., Zhang, Y.: A Distributed, Scalable, and Synchronized Framework for Large-Scale Microscopic Traffic Simulation. In: IEEE Conference on Intelligent Transportation Systems, pp. 813–818 (2005)
9. Nagel, K., Rickert, M.: Parallel implementation of the TRANSIMS. Parallel Computing 27, 1611–1639 (2001)
10. Osogami, T., Imamichi, T., Mizuta, H., Morimura, T., Raymond, R., Suzumura, T., Takahashi, R., Id, T.: Research Report IBM Mega Traffic Simulator. Technical report (2012)

11. Saraswat, V.A., Sarkar, V., von Praun, C.: X10: concurrent programming for modern architectures. In: Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 271–271. ACM (2007)
12. Suzumura, T., Kanezashi, H.: Highly Scalable X10-Based Agent Simulation Platform and Its Application to Large-Scale Traffic Simulation. In: 2012 IEEE/ACM 16th International Symposium on Distributed Simulation and Real Time Applications, pp. 243–250. IEEE (October 2012)
13. Thulasidasan, S., Kasiviswanathan, S., Eidenbenz, S., Galli, E., Mniszewski, S., Romero, P.: Designing systems for large-scale, discrete-event simulations: Experiences with the FastTrans parallel microsimulator. In: 2009 International Conference on High Performance Computing (HiPC), pp. 428–437. IEEE (December 2009)
14. Treiber, M., Hennecke, A., Helbing, D.: Congested traffic states in empirical observations and microscopic simulations. Physical Review E 62(2), 1805 (2000)