Proceedings of the 2020 Winter Simulation Conference
K.-H. Bae, B. Feng, S. Kim, S. Lazarova-Molnar, Z. Zheng, T. Roeder, and R. Thiesing, eds.

MULTI-THREAD STATE UPDATE SCHEMES FOR MICROSCOPIC TRAFFIC SIMULATION

Wen Jun Tan
Philipp Andelfinger
Wentong Cai
Alois Knoll

Nanyang Technological University
50 Nanyang Ave
639798, SINGAPORE

Yadong Xu
David Eckhoff

TUMCREATE Ltd.
1 Create Way
138602, SINGAPORE

ABSTRACT

Microscopic traffic simulation is an essential tool for the evaluation of intelligent transportation systems (ITS). With the increasing complexity of ITS applications, higher-detail simulation models, and the need to analyze large-scale scenarios, simulation run-times can grow exceedingly large. One way to counter this problem is the use of parallel computing techniques, such as shared-memory multi-thread parallelism. While the foundations of parallel traffic simulation are well-known, the effects of different synchronization and agent-update mechanisms on simulation performance have not been explored systematically. In this paper, we first analyze the common properties of models used in microscopic traffic simulation to understand the impact of their data dependencies. We discuss synchronous and asynchronous agent update schemes and compare them in terms of performance and requirements. We conclude that although it requires more memory and additional conflict handling, the synchronous agent-state updating approach is favourable in terms of scalability.

1 INTRODUCTION

In microscopic agent-based road traffic simulation, each agent is usually modeled as a driver-vehicle unit (DVU) that makes autonomous decisions based on behavioral models and its environment. The emerging behavior resulting from the interactions of single agents enables the studying of system parameters such as traffic flow or traffic safety metrics. These simulations can support the solving of some of the severe problems of modern large cities, such as congestion, emissions, or traffic light optimization. However, when applied to a large-scale, e.g., city-wide or even nation-wide, microscopic traffic simulation still faces modeling and computational challenges. Modeling challenges include the realistic representation of human behavior in conceptual or mathematical models, which often requires large amounts of empirical data for calibration and validation. With the increasing availability of high-detail traffic data enabled by ubiquitous sensing systems, we expect novel and more realistic behavioral models in the near future.

The second class of challenges, i.e., the computational challenges, are given by the time and hardware constraints under which traffic simulations are conducted, especially when considering the exploration of large parameter spaces and the collection of statistical results where a large number of simulation runs are required. Beyond a certain scale, simulation at the microscopic level of detail is impractical due to too long computation times. One common way to approach this problem is parallel computing, where the simulation is executed in parallel by a number of logical processors (LPs). However, parallel traffic simulations require careful design of the parallel algorithms to efficiently utilize the available hardware.

Parallelization is commonly associated with overheads and tends to complicate the implementation of the simulator. These overheads, in addition to imbalances in the distribution of computational tasks, lead to the fact that parallel simulations usually do not scale linearly as more processors are introduced (Fujimoto 2016). An additional domain-specific challenge for the simulation of intelligent transport systems (ITS) is that typically, a combination of models with fundamentally different computational requirements is used, further complicating efficient parallel execution.

Shared-memory multi-thread parallelism is one of the common parallel programming models due to the increasing number of CPU cores on a single machine. In shared-memory programs, a number of threads are executed in the same address space, typically orchestrated by a multi-threading library such as the Open Multi-Processing (OpenMP) framework. Threads interact by accessing locations in the shared memory address space, achieving mutual execution based on atomic operations or locks. To execute a parallel simulation using multi-threads, each LP can be mapped to a thread.

In this work, we aim to explore the challenges and benefits of different parallel execution approaches using a multi-thread architecture for microscopic traffic simulation. To achieve efficient parallel execution, we first need to understand the impact of data dependencies of the models used in microscopic traffic simulation and select the agent state update required for these models. The state update also determines which multi-thread execution scheme lends itself better to parallel execution.

There are existing parallel and distributed traffic simulators, such as PARAMICS-MP (Liu et al. 2004), AIMSUN (Barceló et al. 1998), TRANSIMS (Nagel and Rickert 2001), dSUMO (Bragard et al. 2016), and CityMos (Zehe et al. 2017). The road network is typically partitioned across LPs, and requires synchronizations between the LPs to maintain the consistency of simulation. To the best of our knowledge, the present paper is the first to explore the performance implications of different state update mechanisms for multi-threaded traffic simulations.

The contributions of this paper can be summarized as follows:

- We analyze the properties of common microscopic traffic simulation models and their implications on parallel execution.
- We discuss and compare the two approaches of agent state updates, *asynchronous* and *synchronous* updates, and propose multi-thread parallel execution schemes based on their different characteristics.
- We present a comparison of different state update schemes by conducting an extensive performance evaluation on the impact and overhead of multi-thread parallel execution.

The remainder of this paper is organized as follows: We first present a model taxonomy for microscopic traffic simulations (Section 2). In Section 3, we discuss two approaches to agent state updating and propose the multi-thread execution of agent state updates. With the help of an extensive simulation study, we give insights into various performance aspects of state updates in parallel traffic simulation in Section 4. Before we conclude the paper, we discuss the advantages and disadvantages of the state update schemes in Section 5.

2 DEPENDENCIES IN TRAFFIC SIMULATION MODELS

Vehicular traffic can be modeled as a multi-agent system (Kesting et al. 2008; Chen and Cheng 2010), where agents are individual, autonomous, collaborative, and reactive entities. Agents interact with each other as well as their environment, the latter consisting of the road infrastructure, e.g., the road network or traffic light signals.

During the simulation, the simulated world evolves through a virtual simulation time. At any point in simulation time, agents and the environment have *states* represented by state variables. For example, the state variables of an agent, that is, a DVU, consist of acceleration, velocity, and position; the state of a traffic light is its current signal phase. These states change as the simulation progresses by executing the state update functions of the employed models. For example, a DVU's state can be updated by driver behavior models such as car-following (Brackstone and McDonald 1999), lane-changing (Hidas 2002), and route-choice models (Dell'Orco and Marinelli 2017).

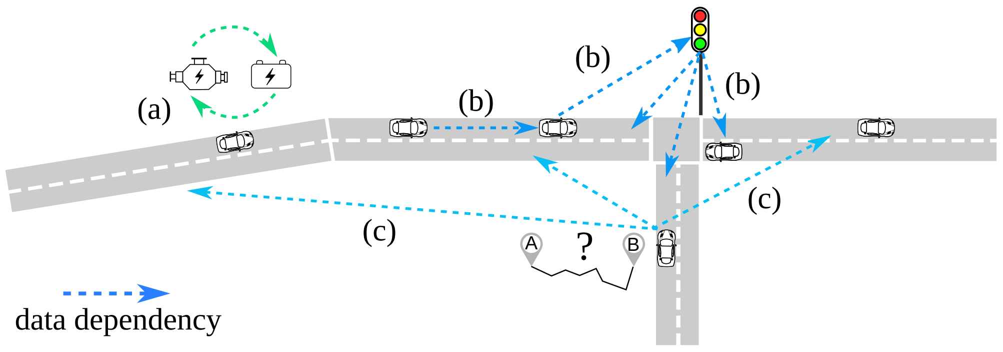


Figure 1: Data dependencies for different types of models. Type (a) are internal models, e.g., models concerned with electric motors and batteries. In contrast, types (b) and (c) are external models. Type (b) models only extend to limited local space, e.g., car-following models and adaptive traffic lights, whereas type (c) can span across the whole road network, e.g., for dynamic route calculations.

Table 1: Taxonomy of commonly used models in microscopic traffic simulation.

Application	**Examples**	**Internal**	**External**	
			Local	**Global**
Emission models	(Rakha and Ahn 2004; Maia et al. 2011)	●		
EV Battery models	(Bi et al. 2017)	●		
General car-following behavior	(Brackstone and McDonald 1999)		●	
Connected autonomous vehicles	(Meignan et al. 2007; Gora and Rüb, I 2016)		●	
Adaptive traffic signal control	(McKenney and White 2013)		●	
Driver information route choice	(Dell'Orco and Marinelli 2017)			●
EV charging	(Bi et al. 2017)			●
Agatz2012 Ride-sharing	(Pelzer et al. 2015)			●

Models can be seen as functions that take as input the current state of the agent and its surroundings, and their outputs are the changes to the state variables. Models in traffic simulation can come in many forms such as differential equations, fuzzy logic, or artificial neural networks. To ensure the correctness of the simulation, *data dependencies* between models and agents must be considered when agents are updated concurrently. These dependencies play a critical role in the design of parallel simulations as they have a significant influence on the induced overheads in terms of locking and state-sharing between the parallel processors.

Different types of models have different data dependencies and thus different synchronization requirements in parallel processing. The models can be classified according to the span of data dependency across the simulation space. We categorize them into three types: *internal*, *external local*, and *external global*. An illustration of the data dependencies of different types is shown in Figure 1.

Internal models only input or output states visible inside the agent. The states they read or modify have no direct effect on other agents, therefore data dependencies do not exceed agent boundaries. Examples are electric motor and battery models for electric vehicles (EVs), as shown as Type (a) in Figure 1. The state of charge of the battery is determined by many factors inside the vehicle such as acceleration and weight, but it is not directly linked to other agents. Therefore, assuming parallel execution on agent-based level (meaning different LPs will not operate on the same agent), these models will not cause conflicts or require synchronization messages between the LPs.

In contrast to internal models, *external* models input or output states that are visible to other agents and the environment. Examples are car-following and lane-changing models that describe the lateral and longitudinal movement of a vehicle based on the road infrastructure as well as speed and distance information of nearby vehicles. The computed acceleration and lane-change decisions are visible to other vehicles nearby. Therefore, during parallel execution, such models generally require synchronization among threads and processes.

Depending on the span of data dependencies across the environment, we further divide external models into *local* and *global*. *Local* means the data dependency does not span across the whole simulation space as illustrated as type (b) in Figure 1. A car-following model only requires the states of other agents and the environment (e.g., traffic lights) within the driver's *perception range*. On the other hand, input and output of *global* models may span over the entire simulation space. For example, dynamic path-finding models may require an agent to know the traffic state of the entire road network, visualized as type (c) in Figure 1. In parallel simulation, global models should therefore be handled with care, as they may lead to expensive global synchronization. In Table 1, we list examples of different types of models commonly used in (sub-)microscopic traffic simulation.

In summary, *internal models* can be updated independently without synchronization, *external local* models require synchronization among a subset of agents, and *external global* models need to be synchronized globally. While global synchronization in external global models is unavoidable, it is a challenge to reduce synchronization in external local models to achieve efficient parallel simulation.

3 MULTI-THREAD STATE UPDATES SCHEMES

In this section, we focus on an agent-based simulation consisting of only external local models, e.g., car-following models. First, the workload division is selected to reduce synchronization. Second, two agent state update schemes are discussed in relation to their synchronization requirements. Third, multi-thread execution schemes are described for parallel execution of the state update schemes.

3.1 Workload Division

In order to parallelize any simulation, the simulation space needs to be partitioned into smaller pieces that can be computed (at least partially) independently. There exist various approaches to do so, for example, traffic simulation can be partitioned on an agent level or spatially, assigning lanes or entire regions to the LPs. This assumes that the number of lanes or regions in the simulation is greater than or equal to the number of LPs. The partitioned simulation is executed in a lock-step manner, where each partitioned workload is executed independently and synchronized at a barrier, usually the time step in traffic simulation, before proceeding to the next step.

When data dependencies necessitate locking to avoid race conditions, a unit of workload may be an area instead of an agent, e.g., a road segment (Aydt et al. 2013). This reduces locking overhead as one LP will be responsible for all agents on this particular segment and will therefore not lock agent by agent. Other considerations include the memory layout: in traffic simulation, it is common to store agents on the same lane next to each other in memory, allowing for faster access times. It therefore makes sense to treat a *lane* as an atomic unit of workload.

3.2 State Update Schemes

Agents inside agent-based simulation commonly follow a standard *Sense-Think-Act* paradigm. They sense the environment which may include other agents, then, in the think stage, they make decisions based on what they have sensed, and lastly, act on the environment as a result.

In parallel simulation, there are several challenges here: first, sensing the environment comes with reading state variables of other agents, who at the same time might be updated by other LPs. This requires locking and negatively affects speed-up. In the *Act* stage, agents may reserve *resources* of the environment, for example, some space on a lane. A *conflict* happens if two agents who are processed simultaneously aim to reserve the same resource. These conflicts need to be resolved before the simulation can progress.

A general consideration in time-stepped agent-based simulation is the order according to which agent states are updated at a given time step. Railsback and Grimm (2011) differentiate two ways of scheduling

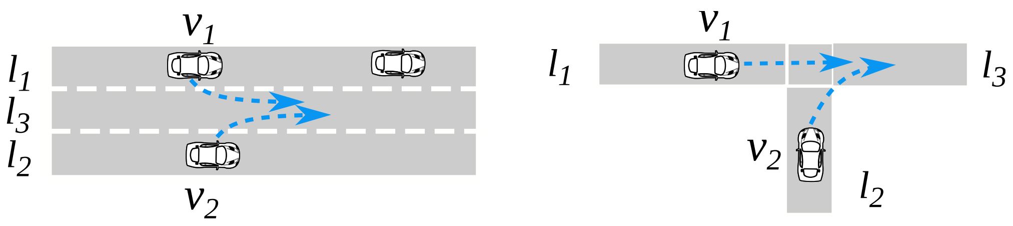


Figure 2: Examples for conflicts in traffic simulation. Left: Vehicles v_1 and v_2 both want to change lane at the same time. Right: Both vehicles want to advance to lane l_3.

agent updates: *asynchronous* and *synchronous*. In this section, we discuss two types of agent state update schemes that deal with the two problems described earlier in different ways.

3.2.1 Asynchronous Update

Simply updating agents one after the other in a random or predefined order leads to *asynchronous* agent updates. Given a time step size Δt, the actions of an agent when updating its state from time t to $t+\Delta t$ typically depend on the states of nearby agents, some of which may be at t, while others may already have been updated to $t+\Delta t$. When executing updates in parallel, an additional concern is the locking of agent and environment states to avoid race conditions.

Often, agents compete for simulated resources such as locations in the simulation space. In traffic simulations, vehicles driving on different lanes may intend to advance to the same new lane in the same time step (cf. Figure 2). With the asynchronous update, the update order determines which of the vehicles will acquire the resource, while all other vehicles will find the new lane occupied. In such scenarios, care must be taken not to introduce a systematic bias that is not part of the simulation model specification (Yang et al. 2018). For instance, if vehicles on lane l_1 in Figure 2 are consistently updated first, vehicles advancing from l_1 to l_3 may frequently block vehicles on l_2 from entering l_3. A common way to address this issue is to shuffle the list of agents before updating.

3.2.2 Synchronous Update

To avoid locking during the *Sense* and *Think* stages, agents can instead be updated in a *synchronous* manner. All agents perform their actions concurrently, i.e., each agent considers the states of nearby agents at time t. This is achieved by read-only access to both agents' state and environment state at time t, while the new states at $t+\Delta t$ are written to a separate set of variables. While this approach doubles the memory required for visible agent state variables of external models, it will achieve a higher speed-up during the *Sense* and *Think* stages as it does not require locking. However, dependencies may still arise in the *Act* stage.

We revisit the situation depicted in Figure 2 with synchronous updates. Vehicles v_1 and v_2 would both attempt to advance to l_3 at the same time. Since vehicles cannot overlap in the simulation space, this situation constitutes a *conflict*. A conflict resolution mechanism can be introduced to decide which of the agents should be able to advance to the desired lane and to modify the states of any competing agents in order to break the tie.

One possible way to break this tie is to apply a policy such as "the most ahead vehicle acquires the lane". We resolve conflicts on a lane l as follows (cf. Figure 2): If the spaces occupied by two vehicles overlap, the vehicle that is further behind is a) moved back to the original lane if the conflict is caused by lane-changing or b) pushed back to a minimum distance from the other vehicle if the conflict is caused by moving forward. Since this may generate new conflicts, the process repeats until all conflicts have been resolved.

3.3 Multi-thread Parallel Execution Schemes

This subsection describes two multi-thread parallel execution schemes for the efficient execution of agent-based simulation. To maintain data consistency on the concurrent modification of a resource by different threads, the resource has to be locked by the thread accessing it. Two methods of locking can be used: *blocking lock* and *try lock*. Blocking lock ensure that when one thread locks the resource, other threads need to wait for the thread to complete and release the lock. Try lock only acquires the lock when no other thread is locking it, and returns a failure when other thread is currently having the lock. The benefit of try-lock is allowing the thread to process other workloads if the lock fails, thus reducing the waiting time for locking.

3.3.1 Chunk Execution

A straightforward approach to distributing the workload is by grouping it into chunks and then assigning each chunk to a thread. The main overhead for chunk execution is locking. Blocking locks can be used to lock the workload by the thread accessing it. In the worst-case scenario, if all threads attempt to lock the same resource, lock contention will lead to the serialization of execution.

3.3.2 Workload Pool

One of the main goals when partitioning the simulation space is to minimize workload imbalance. The slowest thread determines the maximum achievable speed-up, as all other threads wait at synchronization barriers. That means, that in order to advance to the next time step in the simulation, all agents need to be updated. Using chunk execution, in the worst-case scenario, the simulation can be serialized where all agents residing on one lane are exclusively handled by one thread. Due to workload imbalance between threads, serialized execution of a parallel simulation can perform worse than a sequential one.

Thread pool design aims to minimize this workload imbalance. Workload balance can be maximized by creating a workload pool where threads process one workload after another until all workloads have been processed. Blocking locks can be used to lock all dependencies required by the workload. However, when all threads attempt to acquire the same lock, blocking locks will serialize the execution.

To reduce the time spent waiting to acquire locks, try-locks can be used with two workload pools. When a thread tries to acquire the locks and fails, the workload is placed into another workload pool p_2. After the first workload pool p_1 finishes executing, the workloads in p_2 will be processed. For workloads that fail to acquire locks in p_2, they are placed in p_1. These two pools are executed iteratively until all workloads are processed.

However, there can be a significant overhead if threads frequently fail to acquire locks due to lock contention, and need to place the workload in the next pool. In the best-case scenario, all the workloads can be processed in a single iteration. There is a possibility that only one thread manages to acquire the locks for a workload in each iteration, which will serialize the execution.

3.4 Parallel Execution of State Updates

The locking behavior for asynchronous and synchronous updates are different. Hence, different parallel execution schemes are suitable for each update mechanism.

3.4.1 Asynchronous Update Workload Pool

In the *asynchronous updates*, the input for updating agents can include other agents that might have already been forwarded to time $t+\Delta t$ as well as agents still residing at time t. To avoid race conditions, the lane on which the agent resides and all other lanes within the sensing range of the agent have to be locked. When chunk execution is used for asynchronous updates, processing each lane requires the locking of all

```
$p_1 \leftarrow L_t$
while $|p_1| > 0$ do
    for each thread do
        $l_t \leftarrow p_1.pop()$
        if $tryLock(l_t)$ then
            $Sense(l_t)$
            $Think(l_t)$
            $l_t \leftarrow Act(l_t)$
            $unlock(l_t)$
        else
            $p_2.push(l_t)$
        end if
    end for
    $swap(p_1, p_2)$
end while
```

(a) Asynchronous Update

```
parallel for $l_t \leftarrow L_t$ do
    $Sense(l_t)$
    $Think(l_t)$
end for
parallel for $l_t \leftarrow L_t$ do
    $lock(l_{t+\Delta t})$
    $l_{t+\Delta t} \leftarrow Act(l_t)$
    $unlock(l_{t+\Delta t})$
end for
repeat
    parallel for $l_{t+\Delta t} \leftarrow L_{t+\Delta t}$ do
        $lock(l_{t+\Delta t})$
        $l_{t+\Delta t} \leftarrow ConflictResolution(l_{t+\Delta t})$
        $unlock(l_{t+\Delta t})$
    end for
until #conflicts $= 0$
```

(b) Synchronous Update

Figure 3: Pseudocodes for parallel execution of a time step t.

neighboring lanes. Using a workload pool can increase parallelism, by reducing the waiting time to acquire locks, compared to chunk execution.

Pseudocode for executing a time step t for the asynchronous update using workload pools is shown in Figure 3a. First, the set of all lanes L_t are added to the first workload pool p_1. Each thread will retrieve a lane and attempts to lock the lane and its neighboring lanes. If the locking is successful, the *Sense-Think-Act* stages can be executed, and the lanes are unlocked. If the thread is unable to lock the lane, the lane is placed in the second workload pool p_2. After all lanes in p_1 are completed, the lanes in p_2 will be processed. This is executed iteratively until there is no more workload in the pools.

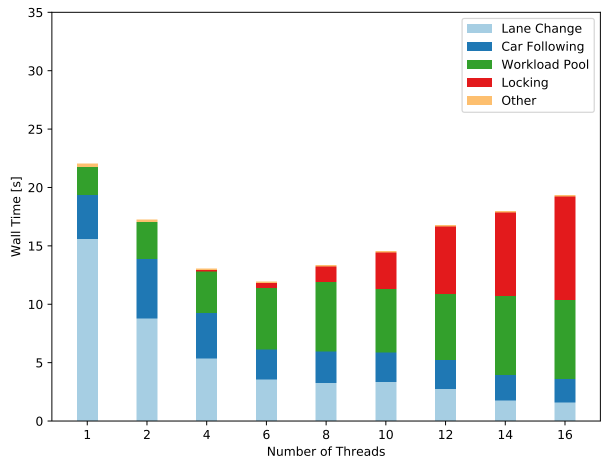
3.4.2 Synchronous Update Chunked Execution

Synchronous updates circumvent the issue of large locking overheads by basing all state updates for $t + \Delta t$ on the states at time t. This means that the new states need to be buffered and not overwritten during the update phase, increasing memory requirements. When chunk execution is used for synchronous updates, locking is not required during the *Sense* and *Think* stages. Hence, the workloads can be executed in parallel during the *Sense-Think* stages without any locks. However, during the *Act* stage, conflicts can occur when agents want to occupy the same resource. Therefore, locking is only required for updating the agents and when resolving conflicts in the *Act* stage. Chunk execution is suitable for synchronous updates, where each of the agent-update stages is executed in chunks.

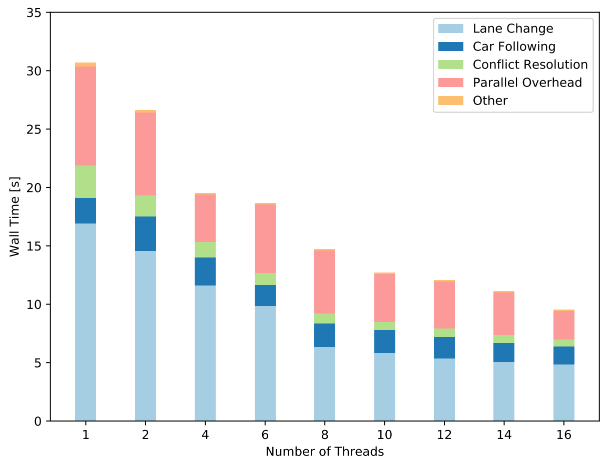
Pseudocode for executing a time step t for the synchronous update is shown in Figure 3b. In each of the *parallel for* sections, the lanes are divided into equal chunks and distributed among the threads. The *Sense* and *Think* stages are performed on the current state t, while the *Act* stage is based on state t and written to state $t + \Delta t$. Based on the next state $t + \Delta t$, conflict resolutions are iteratively executed until there are no more conflicts. However, this also requires splitting the stages into different parallel regions, and increases the overhead of synchronization between the regions.

4 PERFORMANCE EVALUATION

In this section, we evaluate the performance of two parallel state update schemes and compare performance properties (simulation time and speed-up) for microscopic traffic simulation.



(a) Asynchronous update using workload pools.



(b) Synchronous update using chunked execution.

Figure 4: Execution time and its components of each parallel state update scheme across different numbers of threads.

4.1 Simulation Setup

We implemented the parallel design into our microscopic agent-based traffic simulator consisting of only external local models. The agents move according to the intelligent driver car-following model (IDM) and the MOBIL lane-changing model. The time-step Δt length for agent updates was set to 0.6 seconds.

We simulated road traffic on a synthetic grid network of 10×10 intersections, each road being 100 meters long with 3 lanes per direction. Uniform traffic is generated to travel across the grid network. Routing for each agent is precomputed before the start of the simulation.

The experiments were run on a compute node with the following hardware configurations: two Intel Xeon E5-2670 (2.6GHz, 8 cores) CPUs (i.e., 16 physical processors), and 128 GB RAM. We used GCC 7.5 with OpenMP support. To obtain insights into the performance of the evaluated approaches, we use various configurations for the number of OpenMP threads.

4.2 Results

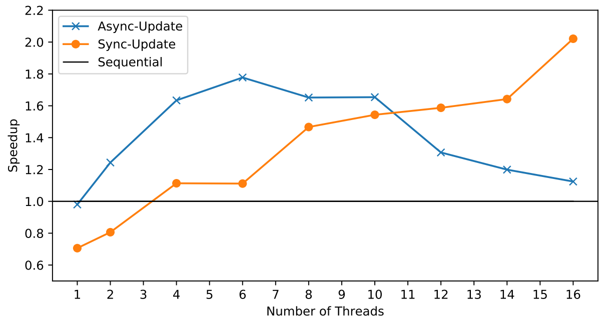
We present the break-down of simulation wall time and speed-up of the two parallel state update schemes. Our results are shown in Figure 4.

4.2.1 Asynchronous Update Workload Pool

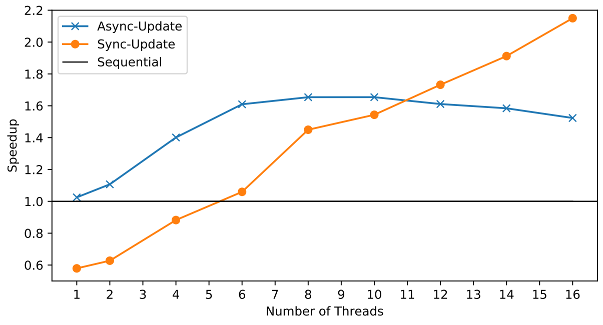
In Figure 4a, we observe the reduction in wall time used by the lane-changing and car-following models as more threads are added. We observed the shortest wall-time when there is a good balance in the time spent between executing the simulation models and processing the workload pool.

However, when more than 6 threads were used, the total simulation time increased, as the probability of different threads attempting to lock non-disjoint sets of lanes increased. When the threads failed to acquire the locks, the workload is placed in the next workload pool. Therefore, the overall time spent in the workload pool increases due to lock failures. This is also directly reflected in the increased time spent in the locking and workload pool. When 16 threads are used, although the time spent in the simulation models are the lowest, there is a considerable overhead from the workload pool and locks.

The overhead in the processing of the workload pool is dependent on the probability of lock failures. First, this is largely affected by the scenario, such as the size of the network and density of traffic. For example in a small road network, the probability of two threads locking non-disjoint set of lanes is higher.



(a) Strong-scaling using a fixed problem size.



(b) Weak-scaling using a varying problem size.

Figure 5: Speed-up of (a) asynchronous update using workload and (b) synchronous update using chunked execution compared to sequential execution.

This will require multiple iterations of the workload pool to process all workloads. In a large road network, the probability of two threads attempting to lock non-disjoint set of lanes is lower. Hence, it is possible to process the workload pool in a single iteration. Second, increasing the number of threads also increases the probability of locking non-disjoint set of lanes.

4.2.2 Synchronous Update Chunked Execution

In Figure 4b, the synchronous update scheme generally shows a decrease in simulation time when more threads are added. The better performance can be attributed to lower locking overhead, since only the *Act* stage and conflict resolution require locking. As each of the *Sense-Think-Act* stages can be parallelized, the synchronous update mechanism offers better scalability compared to the asynchronous approach.

However, there are significant overheads when using synchronous updates, which lead to higher simulation times compared to asynchronous updating when a small number of threads are used. First, there is a workload imbalance when a small number of threads are used as there is no redistribution of lanes among the threads during the simulation. With an increasing number of threads, the workload is distributed more evenly across the threads, reducing the threading overhead as shown in Figure 4b. Second, thread overhead (i.e., forking/joining of parallel regions) is increased for the synchronous update because of the explicit separation of the *Think-Sense-Act* stages, leading to smaller parallel regions. The third overhead is due to additional conflict resolution required by the synchronous update.

4.2.3 Speed-Up

We measured the speedup of parallel asynchronous and synchronous updates compared to a sequential execution, which runs on a single thread without locks or conflict resolution. Our results for strong-scaling and weak-scaling are shown in Figures 5a and 5b respectively.

For the strong-scaling experiment, the total problem size is fixed while the number of threads is varied. In our case, the total number of agents is fixed at $10,000$. When a small number of threads are used ($<= 10$ threads), the asynchronous update scheme has a better speedup compared to the synchronous update one. As there are fewer lock contentions for a small number of threads, the asynchronous update exhibits better performance. Using one thread shows a small slowdown compared to sequential execution, indicating the overhead of executing the workload pool is small. It exhibits the best performance ($1.77\times$ speedup) for six threads. However, the asynchronous update has a reduction in the speedup when more than 6 threads are used. Initially for a small number of threads, the synchronous update is shown to have a slowdown compared to sequential execution. This indicates that the overhead for the threading and two-state update is large. There is only a speedup when four or more threads are used. However, it demonstrates better scaling compared to asynchronous updating, with a maximum speedup of $2.02\times$ using 16 threads.

For the weak-scaling experiment, the problem size per thread is fixed while the number of threads and the total problem size is varied. We scaled the total number of agents by the number of threads, i.e., $1,000$ agents per thread. For an algorithm with good scalability, the parallel part is expected to scale linearly with the number of threads. The asynchronous update exhibits speedup with an increasing number of threads, with the best performance ($1.65\times$ speedup) for 8 threads. However, the speedup decreases for more than eight threads due to the workload pool and locking overheads. Synchronous updates have a slowdown for four threads or fewer, mainly due to workload imbalance. It exhibits approximate linear scaling with $2.15\times$ speedup at 16 threads.

In summary, with a small number of threads, the asynchronous update shows better speedup compared to the synchronous update. However, the synchronous update shows to have better scalability compared to asynchronous updating.

5 DISCUSSION

Asynchronous or synchronous updates have been inspired by related works in cellular automata (Huberman and Glance 1993). While the focus in these works lies on asynchronous or synchronous updates on the modeling side, our focus is on the simulation execution.

There are several issues regarding asynchronous updates. First, when an agent affects the environment, the environment is updated so that the next agent to execute experiences a different environment. Hence, the simulation outcome is dependent on the execution order of the agents, where the order in which workload items are pulled from the pool can affect the simulation outcome. Second, there are lock contentions when multiple threads attempt to update a single resource. The probability of lock-contentions is high when an increasing number of threads are added. This is also dependent on the scenario, e.g., more lock-contentions are expected for congested traffic as many vehicles are close together. In sparse traffic, the likelihood for lock-contention is lower.

There are several advantages and disadvantages of synchronous updating. First, locking is not necessary with synchronous updates in the *Sense* and *Think* stages. Second, the workload division no longer has to be a lane; a finer (per-agent) parallelism granularity can be utilized. This can improve workload balance as the agents can be divided more evenly among the threads, especially for the worst-case scenario where all agents are on a single lane. Third, synchronous updates also introduce determinism in the simulation as the competitions for resources are resolved by the conflict resolution mechanism and no longer depend on the execution order. Fourth, synchronous updating is also more suitable to be offloaded to other compute devices. For example in Xiao et al. (2018), the traffic simulation is partially offloaded to execute on graphics processing units (GPUs) using the synchronous update scheme. The opportunity to avoid extensive locking also fits the synchronous programming style and execution mode of GPUs.

Railsback and Grimm (2011) stated that synchronous updating is useful in situations when the agents do not update their environment or when they deplete a resource at a slower rate compared to the time steps. In traffic simulations, agents compete strongly for a resource, i.e., a position on a lane. Hence, conflict resolution mechanisms are required to maintain the consistency of the simulation. Usually, modelers need to define how such conflicts should be resolved. The problem of maintaining determinism and freedom of bias for simulation models that do not specify a tie-breaking policy is considered in (Yang et al. 2018). A variety of conflict resolution methods for GPUs are also evaluated by the authors.

However, the overhead for conflict resolution can be significant depending on the scenario. For example, on a congested road where vehicles are close together, the agents need to compete with the surrounding agents during lane changing. Since the agents are close together, conflict resolution may also generate new conflicts, which needs to be resolved iteratively until all conflicts have been resolved. These conflict resolutions induce an additional processing overhead during the *Act* stage. Other disadvantages for synchronous updates include overheads from workload imbalance, an increase in parallel regions, poor cache performance, and increased memory usage.

6 CONCLUSION AND FUTURE WORK

In this work, we outlined the challenges of parallel microscopic traffic simulation. We first presented a taxonomy of models with regards to their data dependencies for parallel execution, discussed different design choices for agent update and multi-thread execution schemes. Performance evaluation is conducted on two parallel state update schemes using a synthetic grid scenario. The asynchronous update shows to have a better speedup when a small number of threads are used, while suffering from significant workload pool overhead when more threads are added. Although the synchronous update has a higher thread overhead, it exhibits better scalability across the threads. We conclude that the performance of the asynchronous update scheme largely depends on the scenario, while the advantages of synchronous update (less locking, better scalability) outweigh the disadvantages (more memory consumption, conflict resolution).

There exist a number of research avenues to further improve the performance of traffic simulation. First, we believe that there is a large potential in the field of partitioning and graph-ordering algorithms to reduce the blocking of threads and to achieve better load balance. Second, application-specific synchronization protocols can further improve performance by reducing overhead. Last, automatic resource allocation for an optimal number of threads can be investigated to increase the efficiency of parallel traffic simulation.

ACKNOWLEDGEMENTS

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

REFERENCES

Aydt, H., Y. Xu, M. Lees, and A. Knoll. 2013. “A multi-threaded execution model for the agent-based SEMSim traffic simulation”. In *AsiaSim 2013*, edited by G. Tan, G. K. Yeo, S. J. Turner, and Y. M. Teo, Volume 402, 1–12. Berlin, Heidelberg, Germany: Springer Berlin Heidelberg.

Barceló, J., J. L. Ferrer, and D. Garcia. 1998. “Microscopic traffic simulation for ATT systems analysis. A parallel computing version”. *25th Aniversary of CRT*:1–16.

Bi, R., J. Xiao, D. Pelzer, D. Ciechanowicz, D. Eckhoff, and A. Knoll. 2017. “A Simulation-based Heuristic for City-scale Electric Vehicle Charging Station Placement”. In *2017 IEEE 20th International Conference on Intelligent Transportation Systems (ITSC)*, 2217–2223. Yokohama, Japan: Institute of Electrical and Electronics Engineers, Inc.

Brackstone, M., and M. McDonald. 1999. “Car-following: a historical review”. *Transportation Research Part F: Traffic Psychology and Behaviour* 2(4):181–196.

Bragard, Q., A. Ventresque, and L. Murphy. 2016. “Self-Balancing Decentralized Distributed Platform for Urban Traffic Simulation”. *IEEE Transactions on Intelligent Transportation Systems* 18(5):1190–1197.

Chen, B., and H. H. Cheng. 2010. “A review of the applications of agent technology in traffic and transportation systems”. *IEEE Transactions on Intelligent Transportation Systems* 11(2):485–497.

Dell’Orco, M., and M. Marinelli. 2017. “Modeling the dynamic effect of information on drivers’ choice behavior in the context of an Advanced Traveler Information System”. *Transportation Research Part C: Emerging Technologies* 85(January):168–183.

Fujimoto, R. M. 2016. “Research Challenges in Parallel and Distributed Simulation”. *ACM Transactions on Modeling and Computer Simulation* 26(4):29.

Gora, P., and Rüb, I. 2016. “Traffic Models for Self-driving Connected Cars”. *Transportation Research Procedia* 14:2207–2216.

Hidas, P. 2002. “Modelling lane changing and merging in microscopic traffic simulation”. *Transportation Research Part C: Emerging Technologies* 10(5-6):351–371.

Huberman, B. A., and N. S. Glance. 1993. “Evolutionary games and computer simulations”. *Proceedings of the National Academy of Sciences* 90(16):7716–7718.

Kesting, A., M. Treiber, and D. Helbing. 2008. “Agents for Traffic Simulation”. In *Multi-Agent Systems Simulation and Applications*, edited by A. M. Uhrmacher and D. Weyns, Chapter 11, 325–356. CRC Press.

Liu, H and Ma, W and Jayakrishnan, R and Recker, W 2004. “Large-Scale Traffic Simulation Through Distributed Computing of Paramics”.

Maia, R., M. Silva, R. Araujo, and U. Nunes. 2011. "Electric vehicle simulator for energy consumption studies in electric mobility systems". In *Proceedings of the 2011 IEEE Forum on Integrated and Sustainable Transportation System (FISTS)*, 227–232. Vienna: Institute of Electrical and Electronics Engineers, Inc.

McKenney, D., and T. White. 2013. "Distributed and adaptive traffic signal control within a realistic traffic simulation". *Engineering Applications of Artificial Intelligence* 26(1):574–583.

Meignan, D., O. Simonin, and A. Koukam. 2007. "Simulation and evaluation of urban bus-networks using a multiagent approach". *Simulation Modelling Practice and Theory* 15(6):659–671.

Nagel, K., and M. Rickert. 2001. "Parallel implementation of the TRANSIMS". *Parallel Computing* 27:1611–1639.

Pelzer, D., J. Xiao, D. Zehe, M. H. Lees, A. C. Knoll, and H. Aydt. 2015. "A Partition-Based Match Making Algorithm for Dynamic Ridesharing". *IEEE Transactions on Intelligent Transportation Systems* 16(5):2587–2598.

Railsback, S. F., and V. Grimm. 2011. *Agent-Based and Individual-Based Modeling: a Practical Introduction*. 2nd ed. Princeton University Press.

Rakha, H., and K. Ahn. 2004. "Integration Modeling Framework for Estimating Mobile Source Emissions". *Journal of Transportation Engineering* 130(2):183–193.

Xiao, J., P. Andelfinger, D. Eckhoff, W. Cai, and A. Knoll. 2018. "Exploring Execution Schemes for Agent-Based Traffic Simulation on Heterogeneous Hardware". In *International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*, 243–252. Madrid, Spain: Institute of Electrical and Electronics Engineers, Inc.

Yang, M., P. Andelfinger, W. Cai, and A. Knoll. 2018. "Evaluation of Conflict Resolution Methods for Agent-Based Simulations on the GPU". In *Proceedings of 2018 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (ACM SIGSIM PADS)*, 129–132. New York, USA: Association for Computing Machinery.

Zehe, D., S. Nair, A. Knoll, and D. Eckhoff. 2017. "Towards CityMoS: A Coupled City-Scale Mobility Simulation Framework". In *5th GI/ITG KuVS Fachgespräch Inter-Vehicle Communication (FG-IVC)*. Erlangen, Germany: FAU Erlangen-Nuremberg.

AUTHOR BIOGRAPHIES

WEN JUN TAN is a postdoctoral research fellow at TUMCREATE and Nanyang Technological University (NTU), Singapore. He received his Ph.D. in Computer Science in 2020 from NTU. His research focuses on parallel and distributed agent-based simulation in heterogeneous hardware environments. His e-mail address is wjtan@ntu.edu.sg.

PHILIPP ANDELFINGER was a postdoctoral research fellow at TUMCREATE and NTU. He received his diploma and Ph.D. in Computer Science in 2011 and 2016 from Karlsruhe Institute of Technology (KIT), Germany. His research focuses on parallel and distributed simulation in heterogeneous hardware environments, agent-based simulation, and network simulation. His e-mail address is philipp.andelfinger@gmail.com.

YADONG XU was a postdoctoral research fellow at TUMCREATE. He received his Ph.D. in Computer Science in 2017 from NTU. His research focuses on parallel and distributed simulation in microscopic traffic simulation. His e-mail address is xuya0006@e.ntu.edu.sg.

WENTONG CAI is a Professor in the School of Computer Engineering at NTU and a program principal investigator at TUMCREATE. He received his Ph.D. in Computer Science from University of Exeter (UK) in 1991. His expertise is mainly in the areas of Modeling and Simulation and Parallel and Distributed Computing. He has published extensively in these areas and has received a number of best paper awards at the international conferences for his research in parallel and distributed simulation. He is an associate editor of the ACM Transactions on Modeling and Computer Simulation (TOMACS) and an editor of the Future Generation Computer Systems (FGCS). His email address is aswtcai@ntu.edu.sg.

ALOIS KNOLL is a Professor of Computer Science at Technische Universität München (TUM), a visiting Professor at NTU, and a program principal investigator at TUMCREATE. He received his diploma (M.Sc.) degree in Electrical/Communications Engineering from the University of Stuttgart and his PhD degree in Computer Science from the Technical University of Berlin. He was a full professor at the University of Bielefeld until 2001. Between April 2004 and March 2006 he was Executive Director of the Institute of Computer Science at TUM. His e-mail address is knoll@in.tum.de.

DAVID ECKHOFF is a postdoctoral research fellow and principal investigator at TUMCREATE. He received his Ph.D. and his M.Sc. degree in Computer Science from the University of Erlangen in 2016 and 2009, respectively. His research interests include privacy protection and smart cities with a particular focus on modelling and simulation. His e-mail address is david.eckhoff@tum-create.edu.sg.