Causality and Consistency of State Update Schemes in Synchronous Agent-based Simulations

Wen Jun Tan
wjtan@ntu.edu.sg
TUMCREATE and
Nanyang Technological University
Singapore

Philipp Andelfinger
philipp.andelfinger@uni-rostock.de
University of Rostock
Rostock, Germany

David Eckhoff
david.eckhoff@tum-create.edu.sg
TUMCREATE and
Technical University of Munich
Singapore

Wentong Cai
aswtcai@ntu.edu.sg
Nanyang Technological University
Singapore

Alois Knoll
knoll@in.tum.de
Technical University of Munich and
Nanyang Technological University
München, Germany

ABSTRACT

In an agent-based simulation (ABS), a state update scheme carries out the transitions of agents from one state to the next. To produce correct simulation results, the update scheme must respect the cause-and-effect relationships defined by the agent-based model and ensure that the resulting overall simulation state is internally consistent. At the same time, the update scheme should be efficient enough to meet a simulationist's demand for timely results. Considering the common class of synchronous time-driven ABS, a number of update schemes have been employed in the literature and simulation frameworks. In this paper, various implementations of update schemes are analyzed and contrasted with respect to their ability to maintain the simulation correctness as well as their performance characteristics. A semantic model is formulated to define the reference behavior of synchronous time-driven ABS updates and model the dependencies among agent updates using a state access graph. Relying on the formalization, conditions under which different update schemes achieve causality are shown. Further, resolution methods are categorized according to their coordination mechanisms to achieve consistency by resolving conflicts among agent state updates. Through two case studies, the empirical performance of different update schemes and resolution methods are evaluated. For sequential execution, an update scheme based on the agent's dependencies achieves the highest performance, whereas in the parallel case, the choice of update scheme involves a trade-off between execution time and memory usage. If deterministic simulation output is required, decentralized coordination generally outperforms centralized coordination. The results can assist implementers and researchers in their selection of appropriate methods in the design and implementation of agent-based simulators.

SIGSIM-PADS '21, May 31-June 2, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8296-0/21/05…$15.00
https://doi.org/10.1145/3437959.3459262

CCS CONCEPTS

• **Computing methodologies** → **Systems theory**; **Agent / discrete models**; **Simulation environments**.

KEYWORDS

Synchronous Agent-based Simulation, State Update Schemes, Correctness, Causality and Consistency

ACM Reference Format:
Wen Jun Tan, Philipp Andelfinger, David Eckhoff, Wentong Cai, and Alois Knoll. 2021. Causality and Consistency of State Update Schemes in Synchronous Agent-based Simulations. In *Proceedings of the 2021 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (SIGSIM-PADS '21), May 31-June 2, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3437959.3459262

1 INTRODUCTION

Agent-based simulation (ABS) is a method for simulating the emergent behavior of a system by modeling and simulating the interactions of its subparts, called agents [35]. In ABS, the simulated entities are agents that perform actions autonomously and interact with other agents based on certain rules. ABS typically follows a *Sense-Think-Act* cycle [24]: In the *Sense* stage, an agent detects and analyses its neighbors as well as the environment in which it resides. In the *Think* stage, an agent makes a judgment based on the information collected during the Sense stage. The update of states takes place in the *Act* stage. Often, the simulation time is advanced in fixed time-steps at which all agents update their states.

When implementing an agent-based model (ABM), it is important to select a *state update scheme* that reflects the semantics of the model to maintain the correctness of the simulation results [35]. In this paper, *synchronous time-driven* simulations are considered, in which agents are updated all at once by a certain time increment, with all agents' actions being based on the same overall simulation state yielded by the previous update [21]. Incorrect implementation of the model can lead to *causality violations* and *consistency problems* in the simulation states. A causality violation occurs when the incorrect implementation leads to a future state change affecting the past [19]. For instance, in a correct simulation, an observer should see a weapon fire before seeing that the target has been

destroyed [8]. A consistency problem occurs when some agents' actions lead to conflicting states according to the model semantics. An example is given by multiple agents moving to the same location in the simulation space. If such conflicts are not resolved, the simulation enters an inconsistent state. The correct and efficient prevention of causality violations and resolution of consistency problems requires careful consideration of the agents' dependencies when implementing the state update schemes.

The correctness of the simulation is a particularly pressing issue in parallel and distributed ABS, which aims to reduce the frequently substantial running times induced by the detailed interactions of large numbers of agents in a typical ABS [8]. If a certain degree of locality is present in the agent interactions, the simulation can be divided into a number of logical processes (LPs) assigned to different processing elements and executed in parallel. Dependencies among agents across LP boundaries are reflected by communication and synchronization among the processing elements. The opportunity for parallel execution hinges on the independence of some of the agent state updates. Hence, one main challenge of parallel and distributed ABS lies in exploiting the independence of some of the agents' actions while still respecting all dependencies.

In this paper, a systematic approach is proposed to analyze the state update implementations according to the semantics of synchronous time-driven ABS. The state dependencies in ABS are analyzed to determine correctness in terms of simulation results. To clarify the scope and the behavior required for the correctness of a state update scheme, a semantic model is formulated for synchronous time-driven ABS. The dependencies among the states are modeled using a *state access graph* to identify causality violations and consistency problems. Relying on the formalization, different agent state update schemes are analyzed to maintain the causality of the simulation states. A common rule for conflict resolution is proposed that can be applied to a wide range of simulations with critical resources, i.e., resources that cannot be shared among agents. A more specific conflict resolution rule based on agents' priorities is also proposed to maintain consistency of the simulation states and determinism in the simulation results. Resolution methods based on the common rules are categorized according to their coordination mechanisms. Situations under which different update schemes and resolution methods are applicable are pointed out.

Having assessed correctness and applicability aspects, performance measurements are presented and discussed for the update schemes in parallelized implementations on a shared-memory machine, providing indications regarding the relative overheads. Finally, an overview of the state update schemes present in existing agent-based simulators is given.

The main intention of this paper is to enable researchers and implementers to make an appropriate choice of state update scheme, taking into consideration the correctness and performance when designing and implementing agent-based simulators.

2 BACKGROUND

Causality and *consistency* are key properties to which a state update scheme must adhere to generate meaningful simulation results [41]. In the following, the background on the semantics of agent updates is given to sketch the challenges in achieving these properties.

2.1 Semantics of Asynchronous and Synchronous Updates

State updates in cellular automata (CA) can be defined semantically as asynchronous or synchronous updates [2]. *Asynchronous update* was defined by [2] as picking a cell at random and updating it, and *synchronous update* as all cells updating at the same time. As the model semantics differ, the simulation outcomes of these two models exhibit different dynamics. For instance, Schönfisch and de Roos [30] demonstrated through two applications that different update schemes produced different output patterns.

Asynchronous and synchronous updates have also been used in ABS, where Railsback and Grimm [21] discussed the importance of the execution order of agents. In *asynchronous update*, as soon as an agent affects the environment, the environment is updated so that the next agent executed experiences a different environment. Hence, the order in which the agents execute their actions affects the simulation results. *Synchronous update* avoids the effect of the execution order by updating the environment all at once after all agents have executed actions that depend on the environment.

As the key observation from the literature, the simulation results of asynchronous state updates may be dependent on the implementation, e.g., the execution order of agent state updates. This is particularly problematic when implementing asynchronous updates in a parallel and distributed simulation, in which non-determinism in the progress of logical processes may translate to non-determinism of the simulation itself. In contrast, synchronous updates can uncouple the implementations from the results. Hence, this paper focuses on synchronous time-driven ABS.

2.2 State Causality

In synchronous time-driven ABS, for a given point t in logical time, all agents advance by a time increment τ. Their actions should only have causal dependencies on the simulation state observed at t. Consider now two neighboring agents a and b simulated in a sequential simulator implementation, which would update the agents one after the other, in any order. As agents typically base their actions on their neighboring agents' states, the agent updated last would base its action at t on its neighbor's state at $t+\tau$, which is a causality violation [19]. The violation can also occur in a parallel and distributed ABS, where the order of the state update can be non-deterministic. From these examples, it is evident that the dependencies among agent updates must be respected by the update scheme to ensure the correctness of the simulation [15, 41].

One way to prevent the violation is to order the agent updates according to the causal dependencies among the agents. However, if two agents a and b are dependent on each other's state, it cannot be determined whether agent a or b must be processed first.

2.3 State Consistency

For this paper, the simulation state at a given point in logical time is consistent if and only if the agent states are free of conflicts according to the model semantics. Conflicts may occur because some of the agents' actions, while not necessarily logically contradictory, may not be compatible because of the resources required by the action [34]. A conflict is an intention of acquiring at the same time a resource that cannot be shared or is limited in quantity, resulting

in competition for the resource. For example, at a transition from a given point in logical time to the next, multiple agents may attempt to move to the same location in the simulation space [41]. If the location in the simulation space cannot be logically occupied by more than one agent, a conflict arises. There are two classical strategies to address conflicts: *conflict avoidance* and *conflict resolution*.

2.3.1 Conflict Avoidance. The first strategy is to avoid conflicts altogether by integrating suitable rules in the specification of the agent behavior itself [12, 31]. In a traffic simulation, when vehicles from two lanes merge into a single lane, the *right of way* rules dictate that the vehicles on the right-most lane have a higher priority to enter the single lane. In the classical boids simulation [22], the agents avoid conflicts by avoiding spatial collisions. There are two common approaches to collision avoidance: force fields and potential collision detection and avoidance. Force fields avoid collisions by associating a repelling force with every agent, such that each agent is ensured to be at least a given distance from another agent. Similarly, in pedestrian simulation, the Social Force model [10] applies a repelling force to every agent to avoid collisions. However, this approach requires sufficiently small time-steps such that the agents do not move too far and pass through the force field. On the other hand, the agent can detect potential imminent collisions ahead of time and take action to actively avoid them. Reciprocal collision avoidance (RVO2) implements this idea in pedestrian simulations by allowing each agent to independently compute a collision-free trajectory through the simulation space [37].

2.3.2 Conflict Resolution. As a second approach to address the issue of conflicts, modelers can explicitly define the outcome of conflicting agent actions. A coordination mechanism among agents may specify that the resource may only be updated by one agent at a time (mutually exclusive) or updated cumulatively by more than one agent (cumulative) [17]. In this way, a domain-specific simulator can pre-define a common set of conflict resolution rules suitable to the model domain.

If model-specific conflict resolution rules are not available, a generic conflict resolution based on agent priorities or stochastic elements may be applied. In this case, measures must be taken to ensure that the conflict resolution rules do not adversely affect properties such as determinism and fairness [41]. Similar considerations are required for simultaneous events in discrete-event simulations [25], which need to handle events with identical timestamps that may lead to different simulation results.

2.4 Determinism

A simulation is *deterministic* if the same result is obtained from repeated simulation runs using the same pseudo-random number generator seed [25]. Determinism is an important property for ABS, where repeatability of the simulation is often desired [6]. For instance, it is frequently necessary to reproduce scenarios to investigate and understand the results [11]. Further, repeatability simplifies debugging the simulator because errors can be reproduced [8]. In this paper, the determinism of different state update schemes is considered to investigate the correctness of the implementations, such that different implementations of the update schemes obtain the same simulation result.

3 MODELS

In this section, a semantic model is formally described for state updates applied to a synchronous time-driven ABM. Subsequently, patterns of the causal dependencies among agents are analyzed by constructing state access graphs.

3.1 Semantic Model for Synchronous Time-driven Agent-based Simulation

A semantic model is proposed for synchronous time-driven agent-based simulation loosely based on the formalizations by [7, 28]. Scheutz and Schermerhorn [28] defined an update function that depends on the states of the resources and neighboring agents to produce an output. However, the update function does not model any changes to the resources. Ferber and Miiller [7] proposed an *influence reaction model* to model the influence produced by agents' behavior and the reactions of resources. By combining these two formalizations, this semantic model can represent the agent interactions with the resources and the interactions among the neighboring agents. The model considers the state updates in each logical time-step.

Let A be the set of agents, R be the set of resources and τ be the time-step size of the simulation. An ABM consists of a set of agent states and resource states, $\{S_a^t : a \in A\}$ and $\{S_r^t : r \in R\}$ respectively. The transition of the agent states from logical time-step t to $t + \tau$ is modeled as a two-phase update process:

Agent Update Phase: $\forall a \in A$,

$$\boldsymbol{R}_a^t = \{S_r^t : r \in {R_i}_a^t\} \tag{1a}$$

$$\boldsymbol{N}_a^t = \{S_b^t : b \in N_a^t\} \tag{1b}$$

$$< I_a^t, \boldsymbol{I}_a^t > = f_a(S_a^t, \boldsymbol{R}_a^t, \boldsymbol{N}_a^t), \text{ where } \boldsymbol{I}_a^t = \{I_{a,r}^t : r \in {R_o}_a^t\} \tag{1c}$$

$$S_a^{t+\tau} = g_a(S_a^t, I_a^t) \tag{1d}$$

Resource Update Phase: $\forall r \in R$,

$$I_r^t = \prod_{a \in A_r^t} I_{a,r}^t \tag{1e}$$

$$S_r^{t+\tau} = g_r(S_r^t, I_r^t) \tag{1f}$$

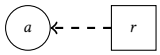
where ${R_i}_a^t$ and ${R_o}_a^t$ are the sets of resources sensed and updated respectively by agent a at time t. N_a^t is the set of neighboring agents of agent a at time t. A_r^t is the set of agents updating a resource r at time t. In the *agent update phase*, each agent a senses the states of the resources, $\boldsymbol{R}_a^t$, (Eq. 1a) and its neighboring agents $\boldsymbol{N}_a^t$ (Eq. 1b). Based on the sensed states, the agent produces a set of influences for the resources, $\boldsymbol{I}_a^t$, and influence for itself, I_a^t, using the function f_a (Eq. 1c). $I_{a,r}^t$ is an influence in $\boldsymbol{I}_a^t$ from agent a to a resource r in ${R_o}_a^t$. The agent updates its next state, $S_a^{t+\tau}$, using the function g_a based on its current state and the agent's influence (Eq. 1d). In the *resource update phase*, all the influences for a resource r are combined using the function $\prod$ (Eq. 1e) to obtain the resulting influence I_r^t. The resource r updates its next state using the function g_r in Eq. 1f according to the influence I_r^t.

3.2 State Access Graph

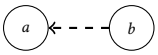
Based on the semantic model described earlier, the state updates within a logical time-step are modeled as a *state access graph*, defined as follows:

Definition 3.1 (State Access Graph)**.** A state access graph is a directed graph $G = (A, R, E)$, where A is the set of agent states, R is the set of resource states, E is the set of directed edges representing the state accesses among the agents and resources.

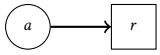
First, consider an agent $a \in A$ sensing nearby resources ${R_i}_a^t$ and its neighboring agents N_a^t. When agent a senses the state of a nearby resource $r \in {R_i}_a^t$, there is an edge (r, a) from the resource r to agent a (see Fig. 1a). When agent a senses the state of a neighboring agent b, there is an edge (b, a) from agent b to agent a (see Fig. 1b). Next, the agent a updates itself and the resources, ${R_o}_a^t$. There is an edge (a, a) when the agent a updates itself (see Fig. 1d) and an edge (a, r) when agent a updates the resource $r \in {R_o}_a^t$ (see Fig. 1c). At each logical time-step, this graph evolves as the environment changes affect the neighborhood of an agent and the surrounding resources that an agent can sense.



(a) Edge (r, a) – Agent a senses a resource r.



(b) Edge (b, a) – Agent a senses agent b.



(c) Edge (a, r) – Agent a updates resource r.

(d) Edge (a, a) – Agent a updates itself.

Figure 1: State accesses among agents and resources. Dashed arrows represent sensing, while solid arrows represent updates.

Two types of dependencies are defined in the state access graph:

Lemma 3.1 (Agent Dependency). *An agent dependency is a direct dependency between two agents.*

$$(a, b) \vee (b, a) \text{ where } a, b \in A \tag{2}$$

Lemma 3.2 (Resource Dependency). *A resource dependency is an indirect dependency between two agents via accesses to a common resource.*

$$(a, r) \wedge (r, b) \text{ where } a, b \in A \text{ and } r \in R \tag{3}$$

There are two special cases of cyclic dependencies, i.e., the dependencies form a closed chain: *self-dependency* and *local dependency*. Self-dependency is a special case of agent dependency where an agent only updates itself (see Fig. 1d). The local dependency is a special case of resource dependency where a resource is only read and updated by a single agent.

Lemma 3.3 (Self-dependency). *A self-dependency is a cyclic agent dependency involving a single agent.*

$$(a, b) \vee (b, a) \text{ where } a, b \in A, a = b \tag{4}$$

Lemma 3.4 (Local dependency). *A local dependency is a cyclic resource dependency involving only a single agent and a resource.*

$$(a, r) \wedge (r, b) \text{ where } a, b \in A \text{ and } r \in R, a = b \tag{5}$$

4 IMPLEMENTATIONS

In this section, the state update schemes implemented according to the semantic model are described in detail, including various agent state update schemes to prevent causality violation and resolution methods to resolve state conflicts.

4.1 Agent State Update Schemes

In the *agent update phase*, there are causal dependencies among the agents and between agents and resources. To maintain causality, the update scheme must resolve these dependencies, i.e., execute the corresponding read or write operations in the correct order. This constraint is defined as follows:

Definition 4.1 (State Causality Constraint)**.** $\forall (u, v), (w, u) \in E$, where $u, v, w \in A \cup R$ and $u \neq v$, (u, v) must precede (w, u).

To prevent causality violations from occurring, the dependencies between agents are categorized to identify suitable agent state update schemes: (i) *independent update*, (ii) *ordered update*, (iii) *two-state update*, and (iv) *temporary state update*.

4.1.1 Independent Agents.

First, considering independent agents that have no dependencies with each other in a logical time, as defined in the following:

Definition 4.2 (Independent Agents)**.** Agents in a state access graph are *independent agents* if and only if all dependencies are only self-dependencies (Lemma 3.3) or local dependencies (Lemma 3.4).

Proposition 4.1 (Independent Update). *For independent agents, any agent update order satisfies the causality constraint.*

Agents in local and self-dependencies can be processed in any order since these dependencies obey the causality constraint. Hence, the above proposition is true as there are no other dependencies that could violate the constraint.

The independence of the state enables an arbitrary agent update ordering, enabling for instance a parallel and distributed execution without synchronization within a single time-step. While it seems unlikely for most simulations to satisfy the requirements for independent update throughout their entire running time, there may be periods in logical time during which agent updates are independent. For example, in a spatial agent-based simulation, the agents may be sufficiently far apart so they do not sense each other [1, 9].

4.1.2 Acyclic Dependent Agents.

Dependent agents are agents connected by dependencies. First, considering dependent agents without any cyclic dependencies among two or more agents:

Definition 4.3 (Acyclic Dependent Agents)**.** Agents in a state access graph are acyclic dependent agents if they have no cyclic dependencies outside of self-dependencies (Lemma 3.3) or local dependencies (Lemma 3.4).

Proposition 4.2 (Ordered Update). *For acyclic dependent agents, there is an agent update order that satisfies the causality constraint.*

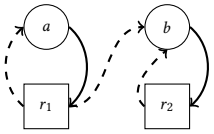


Figure 2: Causal dependency between agent a and b – Agent a senses and updates resource r_1. Agent b senses resource r_1 and r_2, and updates resource r_2.

First, local and self-dependencies in the state access graph can be excluded as they satisfy the causality constraint. After that, it is possible to find a linear ordering for updating the agents by sorting according to the causality constraint. Hence, the above proposition is true.

A method is proposed to execute the agent update according to the causality constraint. First, the state access graph is reduced to an agent graph, which represents the causal dependencies among the agents. The local and self-dependencies are removed from consideration by merging resource vertices into the agent vertices for local dependencies and removing edges for self-dependencies. For example, in Figure 2, agent a senses resource r_1 and updates it, while agent b also senses r_1. Hence, agent b needs to be processed before agent a. After merging r_1 to a and r_2 to b, the state access graph is reduced to an agent graph containing the dependency (a, b). This implies that agent b has a causal dependency on agent a.

Algorithm 1 Ordered Update Scheme

```
1: while A ≠ ∅ do                                  ▷ Agent Update Phase
2:     for all a ∈ A parallel do
3:         if outDegree(a) = 0 then
4:             AgentUpdate(a)
5:             A ← A \ a
6:     end for
7: ...                                             ▷ Resource Update Phase
```

Next, Algorithm 1 executes the agents in topological order. *outDegree(a)* returns the number of outgoing edges of agent a. Agents with no outgoing edge are processed first. This ensures that the agent which is being updated has no other agents that are causally dependent on it. After updating the agent, the agent is removed from the agent graph A (line 5). This is processed iteratively until all agents have been updated. Following the example in Figure 2 where there is only a dependency (a, b) after reducing the agent graph, agent b is updated first as the out-degree is zero, then agent a is updated. If the dependencies among the agents do not change throughout the simulation, an ordered list of agents can be precomputed before the simulation and the agents can be updated according to the list during the simulation.

4.1.3 Cyclic Dependent Agents.

Next, considering dependent agents with cyclic dependencies among two or more agents:

Definition 4.4 (Cyclic Dependent Agents)**.** Agents in a state access graph with cyclic dependencies that are not self-dependencies (Lemma 3.3) and local dependencies (Lemma 3.4).

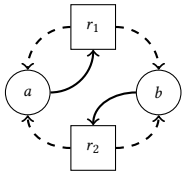


Figure 3: Cyclic dependencies between agent a and b – Agent a senses resource r_1 and r_2, and updates resource r_1. Agent b senses resource r_1 and r_2, and updates resource r_2.

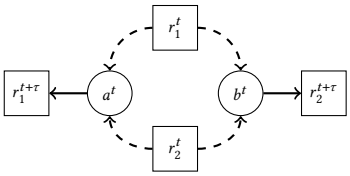


Figure 4: Converting the *state access graph* from Figure 3 into a *two-state access graph* separates the resource states into the current state at t and the future state at $t + \tau$.

For example, the state access graph in Figure 3 is reduced to an agent graph with two edges (a, b) and (b, a). Since there is a cyclic dependency between a and b, it is impossible to find an ordering that would satisfy the causality constraint. To still perform the agent updates in a causality-preserving manner, a state access graph is transformed into a *two-state access graph* by separating the states for the agents and resources into the current states and new states:

Definition 4.5 (Two-state Access Graph)**.** A two-state access graph is a directed graph, $G_2 = (A^t, A^{t+\tau}, R^t, R^{t+\tau}, E^t, E^{t+\tau})$, where A^t and $A^{t+\tau}$ are the current and new states of agents respectively, R^t and $R^{t+\tau}$ are the current and new states of resources respectively, E^t is a set of directed edges representing state accesses to the current states of agents and resources, and $E^{t+\tau}$ is a set of directed edges representing state updates from the current states to the new states.

The two sets of edges in the two-state access graph represent the following relations: An edge in E^t represents an agent a reading a resource r: (r^t, b^t), or an agent b reading an agent a: (a^t, b^t). An edge in $E^{t+\tau}$ represents an agent a updating a resource r: $(a^t, r^{t+\tau})$, or an agent a updating itself: $(a^t, a^{t+\tau})$. Figure 4 shows the state access graph from Figure 3 transformed into a corresponding two-state access graph.

Proposition 4.3 (Two-state Update). *Processing cyclic dependent agents using two separate states in any order satisfies the causality constraint.*

When separating the current and new states, the target vertex of any edge is a new state, and new states have no outgoing edges, i.e., $\forall (u, v) \in E^{t+\tau} : (u \in A \vee u \in R) \wedge (v \in A^{t+\tau} \vee v \in R^{t+\tau})$. Hence, the resulting two-state access graph is acyclic. Since in an acyclic state access graph, the state causality constraints are trivially satisfied, the above proposition is true.

Algorithm 2 shows the pseudocode for two-state update. To implement the two-state access graph, two sets of state variables are used, S^1 and S^2. In contrast, the independent and ordered updates

Algorithm 2 Two-state Update Scheme

1: **for all** $a \in A$ **parallel do** ▷ *Agent Update Phase*
2: $\boldsymbol{R}_a \leftarrow \{\mathcal{R}(S_r^1) : r \in R_i{}_a^t\}$
3: $\boldsymbol{N}_a \leftarrow \{\mathcal{R}(S_b^1) : b \in N_a^t\}$
4: $< I_a, \boldsymbol{I}_a > \leftarrow f_a(S_a^1, \boldsymbol{R}_a, \boldsymbol{N}_a)$
5: $S_a^2 \leftarrow g_a(S_a^1, I_a)$
6: **end for**
7: ... ▷ *Resource Update Phase*
8: $swap(S^1, S^2)$

require only a single set of state variables. S^1 represents the current state, which agents use as a basis of their behaviors; and S^2 represents the new state, which agents update. At the end of a time-step, the current state is swapped with the new state. As the update steps across agents are independent, all agents can be processed in parallel. For example, in Figure 4, agent a or b can be processed at the same time, e.g., making the update scheme suitable for execution on parallel computing platforms, e.g., GPUs or FPGAs [39, 40]. The key disadvantage of this method is the doubling of memory utilization by storing the current and new states.

Algorithm 3 Temporary State Update Scheme.

1: **for all** $a \in A$ **parallel do** ▷ *Agent Update Phase*
2: $\boldsymbol{R}_a \leftarrow \{\mathcal{R}(S_r) : r \in R_i{}_a^t\}$
3: $\boldsymbol{N}_a \leftarrow \{\mathcal{R}(S_b) : b \in N_a^t\}$
4: $< I_a, \boldsymbol{I}_a > \leftarrow f_a(S_a, \boldsymbol{R}_a, \boldsymbol{N}_a)$
5: $Synchronize()$
6: $S_a \leftarrow g_a(S_a, I_a)$
7: **end for**
8: ... ▷ *Resource Update Phase*

Another update scheme, **temporary state update**, that requires only a single set of state variables is shown in Algorithm 3. This scheme can reduce the memory utilization if the temporary variables use less memory space compared to the additional simulation states. However, the need for synchronization introduces additional overhead. First, all the agents compute their influences in line 4 into a set of temporary variables I_a and $\boldsymbol{I}_a$. A global synchronization (line 5) is required before performing the update (line 6).

4.2 State Conflict Resolution Methods

Agent state update schemes only address the issue of causality. In the *resource update phase*, the resources do not sense any agents, so they do not violate the causality constraint. The $\prod$ function in Eq. 1e combines the influences from different agents to obtain the resulting influence used to update the resource. Conflicts may arise when multiple agents attempt contradictory actions. In the following, assume that contradictory actions are reflected by simultaneous resource updates by multiple agents, for instance, expressing the agents' intention to move to the same location in the simulation space. Accordingly, a conflict during the resource update phase is defined as follows:

Definition 4.6 (State conflict)**.** There is more than one agent updating a resource.

$$\exists(a, r) \wedge (b, r), \text{ where } a, b \in A, r \in R \text{ and } a \neq b \tag{6}$$

To resolve state conflicts, a common rule is proposed for updating resources in simulations, the *mutually exclusive update rule*, based on which only one agent is able to update a resource. The other agents can either perform no update or seek another resource.

Definition 4.7 (Mutually Exclusive Update Rule)**.** Given a non-empty set A_r^t of agents generating influences to update resource r at logical time-step t, exactly one agent $a_r \in A_r^t$ is selected by a tie-breaking function Φ to update resource r.

Algorithm 4 Mutually Exclusive Update Rule

1: ... ▷ *Agent Update Phase*
2: **for all** $r \in R$ **parallel do** ▷ *Resource Update Phase*
3: $a_r \leftarrow \Phi(A_r^t)$
4: $S_r^{t+\tau} \leftarrow g_r(S_r^t, I_{a_r,r}^t)$
5: **end for**

The application of this rule is shown as pseudo-code in Algorithm 4. Instead of combining the influences using the $\prod$ function, the tie-breaking function Φ determines the winner a_r (line 3) and allows the influence from a_r to be used to update the resource (line 4). There are different tie-breaking mechanisms to determine the winning agent: (i) *no coordination*, (ii) *centralized coordination*, and (ii) *decentralized coordination*.

The simulation model may specify the exact rule for breaking ties. Alternatively, unique priorities can be assigned so that each conflict is won by the agent with the highest priority.

$$\Phi(A) = \{a | \max_{a \in A}(a.priority)\} \tag{7}$$

This tie-breaking function is referred to as the *priority update rule*. The priorities may either be statically assigned and used throughout the entire simulation, re-determined after each update, or chosen on a per-conflict basis [41]. As the priorities ensure a deterministic conflict resolution, resolution methods that implement the priority update rule produce the same simulation output.

In the following subsections, five resolution methods are discussed to apply the tie-breaking mechanisms with coordination using the priority update rule.

4.2.1 No Coordination. A simple tie-breaking method is the *execution-order* based approach used in the MatSim traffic simulator [32]. This approach only requires a single update phase, in which the first agent successfully updates the resource, whereas each of the other agents seeks another resource. Agent and resource update phases are not required to be separated. In a parallel implementation, the agents atomically update the resource. Although this method identifies a winner for each conflict, the results depend on the execution order of the agent updates. Thus, determinism cannot be guaranteed.

4.2.2 Centralized Coordination. In centralized coordination, resources make a centralized decision to select a winner out of the competing agents based on the priority update rule [38]. During the agent update phase, the agent priorities are collected together with the agent influences for tie-breaking in the resource update phase. There are two methods for collecting the priorities: A *push* method, where agents indicate their priority to the resources; and a *pull* method, where resources query the interested agents for their priority. In both methods, multiple iterations of updates in a time-step may be required for all agents to update the resources.

In the push method, each agent appends its priority to a per-resource list. Then each resource selects the agent with the highest priority from the list. The agents that have not been selected choose another resource in the next iteration. If the newly selected resource has selected the winning agent in the previous iteration, the resource needs to perform the tie-breaking again. In a parallel implementation, synchronized access to the per-resource list is required as multiple agents may append to the list concurrently.

In the pull method, each agent selects a resource and stores the decision temporarily as part of the agent state. Subsequently, each resource scans for interested agents and selects the winning agent based on the agents' priorities. This process repeats until all agents have obtained resources. A parallel implementation of this method does not require atomic operations [23]. However, as the number of agents potentially interested in each resource may be large, the overhead for scanning can be substantial [41].

4.2.3 Decentralized Coordination. In decentralized coordination, agents are responsible for coordination on their behalf, i.e., there is no moderator or controller that centrally determines a winner based on the priority update rule for each conflict [38]. As the conflict resolution and resource update are performed in the agent update phase, the resource update phase is no longer required.

In *iterative* decentralized coordination, each resource is associated with a variable storing the highest priority of any agent that has attempted to update the resource. An agent intending to update a resource sets the priority variable to its priority if the variable is unset or the currently stored priority is lower. If the agent was able to store its priority, it also updates the resource. In this fashion, updates by higher-priority agents displace any previous updates. The displaced agents may select another resource in the next iteration. A parallel implementation of this method utilizes atomic maximum operations to ensure race-free updates by the agents with the highest priorities [16].

There is a variant of decentralized coordination using only a *single iteration* for processing each update. If an agent observes that a higher-priority agent has already updated a resource, the current agent immediately attempts to obtain another resource. Otherwise, the displaced agent is determined and another resource selection and an update attempt is performed for this displaced agent immediately. In a parallel implementation, some LPs may need to process many displaced agents, which may result in workload imbalance.

The properties of different resolution methods are summarized in Table 1. Only coordination mechanisms that utilize the priority update rule can guarantee deterministic simulation results. Intuitively, higher performance would be expected for the approaches requiring only a single update phase. Similarly, non-iterative approaches would be expected to perform better, as the repeated global synchronization required in iterative approaches is avoided.

Table 1: Properties of the Resolution Methods.

Resolution Method	Update Phases	Iterative	Deterministic
None (Execution-order)	1	✗	✗
Centralized (Pull)	2	✓	✓
Centralized (Push)	2	✓	✓
Decentralized (Iterative)	1	✓	✓
Decentralized (Non-Iterative)	1	✗	✓

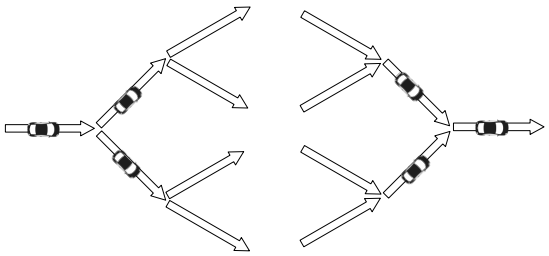
5 PERFORMANCE EVALUATION

In this section, the performance overhead of different agent state update schemes and resolution methods are evaluated through two case studies. The experiments were run on a shared-memory machine with the following hardware configurations: two Intel Xeon E5-2690v3 (2.6GHz, 12 cores) processors (i.e., 24 physical cores), and 128 GB RAM. The processors operate in cluster-on-die mode, which splits each processor into two clusters of six cores, dividing the processor into two NUMA nodes. There are cache conflicts if threads allocated to different NUMA nodes access the same cache line [18]. GCC 4.9.3 is used with OpenMP support. To obtain insights into the performance of the evaluated approaches, various configurations for the number of OpenMP threads are considered.

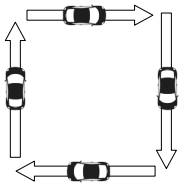
5.1 Traffic Simulation

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 agents accelerate and decelerate according to a car-following model. Often, the intelligent driver model (IDM) is used [36], in which the acceleration is a function of the agent's velocity and the net distance gap and velocity difference to the leading agent. IDM ensures that the agents maintain a safe distance from the leading agent and guarantees collision-free driving. This simulator is built on an existing microscopic traffic simulator [33]. Only single-lane roads are considered for clarity in the analysis. Since each agent on a lane only depends on the leading agent in front of it, the dependencies among the lanes can be considered instead of dependencies among the agents on a lane-by-lane basis.

There are two state variables to be maintained for each agent: (i) the position of the agent on the road and (ii) the velocity of the agent. During the *agent update phase*, the velocity and position of each agent are computed according to IDM. Then in the *resource update phase*, each agent is updated to its new position. When the ordered update is used, the simulator only stores a single copy of the road states (agent positions) and agent states (agent velocities). Two-state update stores the road states and agent states in two sets of state variables. Temporary state update only stores a single set of state variables. The update computes the agents' accelerations (influences) and stores them in temporary variables during the



(a) Out-tree network. (b) In-tree network.



(c) Cyclic network.

Figure 5: Three road network scenarios in traffic simulation. (a) The out-tree network has agents diverging into different roads, and hence no state conflicts. (b) The in-tree network has agents merging into a single road, resulting in state conflicts. (c) The cyclic network has cyclic dependencies among the agents.

agent update phase. Then in the *resource update phase*, the velocities and positions are updated based on the accelerations.

In a collision-free simulation, it is unrealistic for vehicles to overlap spatially. Thus, when agents compute their new positions independently and attempt to move into the same space, state conflicts arise. Given the collision-free movement defined by IDM, such conflicts may only occur when moving across lanes. To resolve conflicts, a lane-merging model merges the traffic from different roads, ensuring there is no overlapping of agents. Based on the different road networks, suitable agent state update schemes are applied and the user-defined conflict resolution is utilized in the resource update phase when necessary.

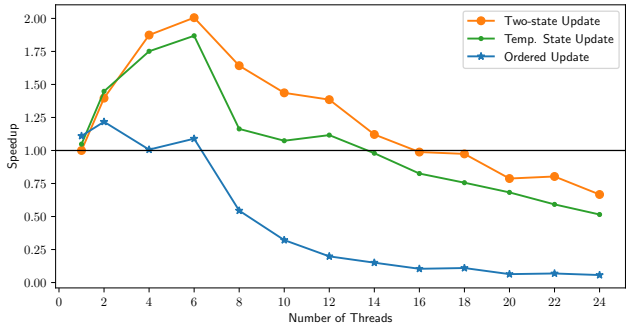
Three different road networks based on real-world scenarios are evaluated: out-tree, in-tree, and cyclic network, as shown in Figure 5. In the out-tree network (Fig. 5a), agents diverge into different lanes. This represents after-work traffic where the drivers are traveling from central business districts back home. Due to the traffic divergence, there are no state conflicts. Hence, it is possible to apply the ordered update using a linear ordering of the lanes based on the road network dependencies. Two-state and temporary state update can also be applied to the out-tree network. In the in-tree network (Fig. 5b), agents merge from multiple lanes into a single lane. This represents before-work traffic during which drivers travel from their homes to the central business districts. Conflict resolution is required to maintain the state consistency of the agents when they enter the same lane. The cyclic network (Fig. 5c) introduces cyclic dependencies among the lanes, which often occurs in real-world road networks. Since there is no linear ordering of lanes, only two-state and temporary state updates can be used.

For the experiments, the networks were configured as follows: the out-tree and in-tree networks are tree networks comprised of five levels, where lanes either diverge or merge. Each network consists of 30 road segments. The cyclic network is a square connected in a cycle, with five road segments in a straight line on each side. Each road segment is 2000 meters long. The time-step size τ for agent updates was set to 0.6 seconds. A total of 10,000 agents were simulated, terminating once agents have exited the network. For each road network, all update schemes are verified to produce the same simulation results. Performance evaluations are done for both serial and parallel execution of ordered, two-state and temporary state updates. As the dependencies among the agents do not change during the simulation, the serial execution order was determined offline.

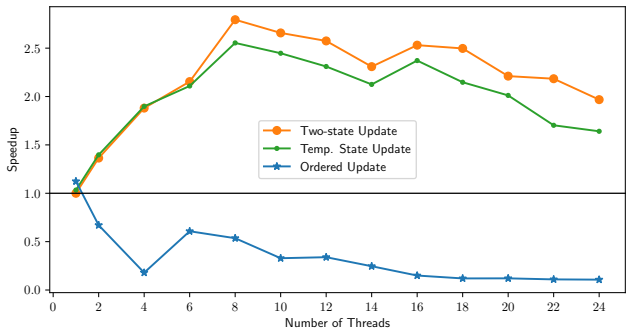
Performance results are evaluated by the speedup relative to single thread execution of two-state update. Figure 6 shows the speedup on different road networks. The performance results of the out-tree network are shown in Figure 6a. For serial execution, ordered update achieves the best performance (1.25×) as the execution order can be computed offline. Temporary state update exhibits better serial performance (1.2×) than two-state update (1.12×) as two-state update requires additional memory transfers.

For parallel execution, ordered update achieves a speedup of only 1.22× for two threads. There is a slowdown for ordered update using more than two threads mainly due to the limited parallelism in parts of the out-tree network. For example, in Figure 5a, the first lane can only be executed by one thread. As the lanes branch out, more lanes can be processed in parallel. On the other hand, two-state update outperforms temporary state update, as temporary state update requires an additional global synchronization among the threads. Parallel execution of two-state and temporary state updates achieves a speedup using two or more threads, with the best speedup of 2.01× and 1.87× respectively using six threads. The speedup decreases beyond six threads due to the cache conflicts among the NUMA cores. A slowdown is observed beyond 14 threads for two-state update and beyond 12 threads for temporary state update.

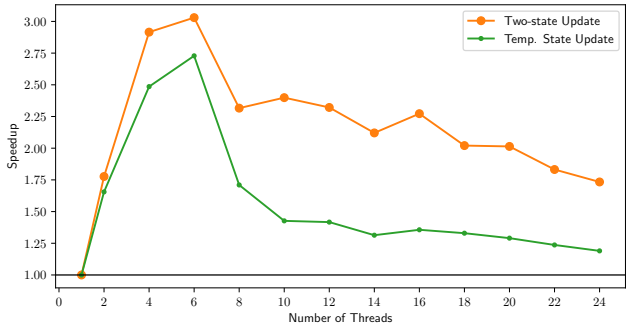
The performance results of the in-tree network are shown in Figure 6b. The in-tree network shows similar serial performance characteristics as the out-tree network (ordered update at 1.16×, two-state update at 1.03×, temporary state update at 1.07×). Parallel execution of ordered update shows a slowdown when more than two threads are used. Both two-state and temporary state update exhibit a speedup with more than two threads, the maximum being 2.79× and 2.55× respectively. The speedup decreases beyond 8 threads. As the agents enter the in-tree network through the leaf edges, there are more agents in the network which increases the workload per road compared to the out-tree network. Ordered update faces limited parallelism due to tree structure. However, better parallel performance can be observed for two-state and temporary state updates as the agents can be executed in parallel (see Algorithms 2 and 3).



(a) Speedup on the out-tree road network.



(b) Speedup on the in-tree road network.



(c) Speedup on the cyclic road network.

Figure 6: Speedup on the road networks relative to single thread execution of *two-state update*.

The performance results of the cyclic network are shown in Figure 6c. Temporary state update achieves a maximum parallel speedup of 2.73× using six threads, but the speedup decreases beyond six threads. Overall, two-state update outperforms temporary state update, with the best speedup over serial execution at 3.03× using six threads. As with temporary state update, the speedup decreases when more than six threads are used, which again is in accordance with the processor's NUMA design.

In summary, ordered update exhibited the highest serial performance compared to other state update schemes. Generally, two-state update achieves the highest parallel performance compared

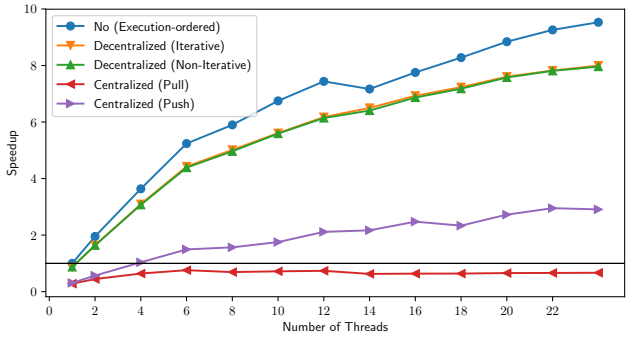


Figure 7: Performance evaluation of conflict resolution methods in Schelling's segregation model.

to other state update schemes in this scenario. Compared to temporary state update, two-state update is simpler to implement but requires twice the memory to store the states, resulting in more memory transfers. In contrast, temporary state update uses only a single set of states but requires global synchronization.

5.2 Segregation Model

Schelling's Segregation model simulates the self-segregation of two populations of agents on a grid over time [27]. For each time-step, the agents may observe and move within a limited neighborhood, e.g., of 3×3 cells. This simulator is implemented based on [41], but it does not consider fairness in this evaluation. During the *agent update phase*, agents determine a happiness value based on the number of agents from the same population group in their neighborhood. If the happiness is below a threshold, the agent decides to move to a new location. During the *resource update phase*, the agents compete with other agents moving to the same location.

Each grid cell stores either the identifier of the agent in the cell or an empty value. Since an agent's population group does not change throughout the simulation, it is not stored as part of the agent's state. Hence, agents do not contain any state. As each agent needs to scan the agents in its neighborhood, there are dependencies between an agent and its neighboring agents. Since there can be cyclic dependencies between the agents, two-state update is applied. As this case focuses on the resolution methods, temporary state update is not evaluated. The five resolution methods from Table 1 are compared to address conflicts when agents compete for cells: no coordination (execution order), decentralized coordination (iterative and non-iterative), and centralized coordination (pull and push). A fixed number of agents is maintained while varying the number of threads.

The segregation is configured with the following parameters: The percentage of populated cells is 50%. The simulation space is a grid of about 4 million ($2048 \times 2048 = 2^{22}$) cells. The agents' happiness threshold is set to 5. The simulation terminated after 100 time-steps. The simulations with decentralized and centralized coordination are verified to be deterministic and produce the same output.

Figure 7 shows the speedup of the conflict resolution methods relative to single thread execution-ordered update. For execution-ordered update, the speedup increases with the number of threads. There is a dip in speedup at 14 threads as threads are assigned to different processors. Decentralized coordination methods have a lower speedup compared to execution-ordered update due to the additional processing of displaced agents. Non-iterative decentralized coordination is associated with less synchronization overhead, but there may be workload imbalance among the threads. Overall, there is no significant difference in the performance between the iterative and non-iterative approaches. With centralized pull-based coordination, the overhead of scanning for interested agents is too large to achieve a speedup over the single-thread execution-ordered update. Although centralized push-based coordination exhibits a speedup, it is still significantly slower than the decentralized coordination methods. There is an overhead of maintaining a list of agent priorities, and multiple iterations are required if the agent is unable to acquire its resource.

In conclusion, as is to be expected, the best performance is achieved without coordination, which cannot guarantee determinism. If determinism is required, decentralized coordination is recommended.

6 STATE UPDATE SCHEMES IN EXISTING AGENT-BASED SIMULATORS

The state update schemes of existing agent-based simulators are shown in Table 2. The approaches used to prevent causality violations and resolve conflicts are summarized.

Independent update has been implemented in *SWAGES* as asynchronous update [29], which utilizes spatial information to identify regions where the agents do not have any influence on other agents, such that these agents can be updated concurrently.

Two-state update has been implemented in two simulators. In *FLAME GPU*, agents communicate through so-called message boards representing the states, where the agents read from one board and write to another board to avoid synchronization for every write [3]. A *continuous space ABS* (CS-ABS) for GPU stores the agent states in two arrays, in which agents read data from the first array and write their updated state to the second array [14].

The other simulators implement temporary state updates. *HLA_RePast* update the agents in the local LPs at every time-step before updating the shared states across the LPs [4]. *D-Mason* divides each simulation step into two phases: *Sense* and *Think-Act* [5]. *SWAGES* can also be executed synchronously which collects the current state first before updating the agents [29]. *Pandora* separates the *Sense-Think* stage from the *Act* stage [26]. In *MASS CUDA*, each agent first reads the neighboring agent states into an array, then updates all the agents [13]. *eVolutus* processes the agents synchronously in an arbitrary order, which executes the *Sense-Think* stage first, then executes the *Act* stage [20].

Only three of the simulators (HLA_RePast, FLAME GPU, and MASS CUDA) implement methods to resolve agent state conflicts. *HLA_RePast* supports conflict resolution using two mechanisms: *mutual exclusion* (execution-ordered) only allows one agent to perform the update, while *cumulative update* combines the updates from multiple agents [17]. *FLAME GPU* uses a parallel centralized pull-based coordination where the resources (i.e., unoccupied cells) read all requests to determine the interested agents [23]. *MASS CUDA* supports three conflict resolution approaches: (1) execution-order, (2) smallest priority, and (3) user-provided.

7 CONCLUSION

Causality and consistency are key properties to achieve correctness in synchronous time-driven ABS. To reason about the ability of different state update schemes to satisfy these properties, we proposed a semantic model for synchronous time-driven ABS. Based on the semantic model, different implementations of the state update schemes were analyzed to identify necessary conditions to achieve correctness in the simulation results.

We constructed state access graphs to analyze the dependencies of the state updates within a logical time-step. The dependencies between the agents were categorized into independent, acyclic, and cyclic dependencies. The different agent state update schemes were evaluated on a shared-memory machine using a traffic simulation where the dependencies between the agents can be explicitly defined through the road network structure.

The choice of agent state update scheme depends on the simulation scenario. For simulations where the dependencies do not change through the simulation, ordered update was shown to achieve the best serial performance. However, in the general case, when dependencies are dynamic, e.g., the agents move around in the environment and interact with their neighbors, two-state or temporary state update achieved better parallel performance.

Both two-state and temporary state updates are easily parallelizable on shared memory processors, with different strengths and weaknesses. Two-state update is straightforward to implement while requiring more memory and performing more memory transfers. Temporary state update is more complex to implement and requires global synchronization. Evaluations show that the relative performance of these methods depends on the simulation scenario.

Agent state update schemes only ensure causality. State conflicts resulting from the agents' competition for limited resources must be solved separately. A common rule is proposed for state conflict resolution that can be applied to a wide range of simulations, such as spatial simulations. Based on this rule, three coordination mechanisms are described with five conflict resolution methods. The performance of these resolution methods was evaluated using Schelling's segregation.

While the best performance was achieved when resolving conflicts without any coordination, the resulting simulation output cannot be guaranteed to be deterministic. Determinism is required to reproduce a given simulation output, e.g., for verification and validation. Hence, decentralized coordination is recommended, which generally outperforms centralized coordination.

In future work, a detailed memory analysis of the case studies can provide a closer understanding of the memory demands of the update schemes. The analysis of conflict resolution schemes can also be expanded to different update rules, e.g., simulations in which multiple agents may update a resource at the same time. The analysis of ABS implemented in optimistic discrete-event simulators can also be considered. This will require extending the state access graph to a temporal graph spanning a period of logical time.

Table 2: Causality violation prevention and state conflict resolution in existing ABS

Simulator	Causality Violation Prevention	State Conflict Resolution
CS-ABS [14]	Two-state update	-
D-Mason [5]	Temporary state update	-
eVolutus [20]	Temporary state update	-
FLAME GPU [3]	Two-state update	Centralized coordination (Pull)
MASS CUDA [13]	Temporary state update	Execution-ordered, agent priority, and user-provided
Pandora [26]	Temporary state update	-
HLA_RePast [4]	Temporary state update	Execution-ordered, or cumulative update
SWAGES [29]	Synchronous – Temporary state update Asynchronous – Independent update	-

ACKNOWLEDGMENT

Financial support was provided by the Deutsche Forschungsgemeinschaft (DFG) research grant UH-66/15-1 (MoSiLLDe).

REFERENCES

[1] Philipp Andelfinger, Yadong Xu, Wentong Cai, David Eckhoff, and Alois Knoll. 2018. Fast-Forwarding Agent States to Accelerate Microscopic Traffic Simulations. In *Proceedings of the 2018 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation - SIGSIM-PADS '18*. ACM Press, Rome, Italy, 113–124. https://doi.org/10.1145/3200921.3200923

[2] Hugues Bersini and Vincent Detours. 1994. Asynchrony Induces Stability in Cellular Automata Based Models. In *Artificial Life IV: Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems*. The MIT Press, Cambridge, Massachusetts, USA. https://doi.org/10.7551/mitpress/1428.001.0001

[3] Simon Coakley, Marian Gheorghe, Mike Holcombe, Shawn Chin, David Worth, and Chris Greenough. 2012. Exploitation of High Performance Computing in the FLAME Agent-Based Simulation Framework. In *2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems*. IEEE, Liverpool, 538–545. https://doi.org/10.1109/HPCC.2012.79

[4] Nicholson Collier and Michael North. 2013. Parallel Agent-Based Simulation with Repast for High Performance Computing. *SIMULATION* 89, 10 (Oct. 2013), 1215–1235. https://doi.org/10.1177/0037549712462620

[5] Gennaro Cordasco, Francesco Milone, Carmine Spagnuolo, and Luca Vicidomini. 2014. Exploiting D-Mason on Parallel Platforms: A Novel Communication Strategy. In *Euro-Par 2014: Parallel Processing Workshops*. Springer, Porto, Portugal, 407–417.

[6] Nuno Fachada, Vitor V. Lopes, Rui C. Martins, and Agostinho C. Rosa. 2017. Parallelization Strategies for Spatial Agent-Based Models. *International Journal of Parallel Programming* 45, 3 (June 2017), 449–481. https://doi.org/10.1007/s10766-015-0399-9

[7] Jacques Ferber and Jean-Pierre Miiller. 1996. Influences and Reaction : A Model of Situated Multiagent Systems. In *Proceedings of Second International Conference on Multi-Agent Systems (ICMAS-96)*. The AAAI Press, Kyoto, Japan, 72–79.

[8] Richard M. Fujimoto. 2000. *Parallel and Distributed Simulation Systems*. Vol. 300. Wiley New York, New York, NY, USA.

[9] Jack Harris and Matthias Scheutz. 2012. New Advances in Asynchronous Agent-Based Scheduling. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*. World Comp, Athens, 7.

[10] Dirk Helbing and Peter Molnar. 1995. Social force model for pedestrian dynamics. *Physical review E* 51, 5 (1995), 4282.

[11] David R. C. Hill, Claude Mazel, Jonathan Passerat-Palmbach, and Mamadou K. Traore. 2013. Distribution of Random Streams for Simulation Practitioners. *Concurrency and Computation: Practice and Experience* 25, 10 (2013), 1427–1442. https://doi.org/10.1002/cpe.2942

[12] Nick Jennings. 1994. *Cooperation in industrial multi-agent systems*. Series in Computer Science, Vol. 43. World Scientific, New Jersey, USA. https://doi.org/10.1142/2257

[13] Lisa Kosiachenko, Nathaniel Hart, and Munehiro Fukuda. 2019. MASS CUDA: A General GPU Parallelization Framework for Agent-Based Models. In *Advances in Practical Applications of Survivable Agents and Multi-Agent Systems: The PAAMS Collection (Lecture Notes in Computer Science)*. Springer International Publishing, Cham, 139–152. https://doi.org/10.1007/978-3-030-24209-1_12

[14] Xiaosong Li, Wentong Cai, and Stephen John Turner. 2016. Supporting Efficient Execution of Continuous Space Agent-Based Simulation on GPU. *Concurrency and Computation: Practice and Experience* 28, 12 (2016), 3313–3332. https://doi.org/10.1002/cpe.3808

[15] Henry X. Liu, Wenteng Ma, R. Jayakrishnan, and Will Recker. 2005. Distributed Large-Scale Network Modeling with Paramics Implementation. In *Proceedings. 2005 IEEE Intelligent Transportation Systems, 2005*. IEEE, Vienna, Austria, 232–238. https://doi.org/10.1109/ITSC.2005.1520053

[16] Mikola Lysenko and Roshan M. D'Souza. 2008. A Framework for Megascale Agent Based Model Simulations on Graphics Processing Units. *Journal of Artificial Societies and Social Simulation* 11, 4 (2008), 10.

[17] Rob Minson and Georgios K. Theodoropoulos. 2008. Distributing RePast Agent-Based Simulations with HLA. *Concurrency and Computation: Practice and Experience* 20, 10 (2008), 1225–1256. https://doi.org/10.1002/cpe.1280

[18] Daniel Molka, Daniel Hackenberg, Robert Schöne, and Wolfgang E. Nagel. 2015. Cache Coherence Protocol and Memory Performance of the Intel Haswell-EP Architecture. In *2015 44th International Conference on Parallel Processing*. IEEE, Beijing, China, 739–748. https://doi.org/10.1109/ICPP.2015.83

[19] James J Nutaro and Hessam S Sarjoughian. 2003. A unified view of time and causality and its application to distributed simulation. In *Summer Computer Simulation Conference*. Citeseer, Montreal, Quebec, Canada, 419–425.

[20] Kamil Piętak and Paweł Topa. 2018. Towards Multi-Agent Simulations Accelerated by GPU. In *Parallel Processing and Applied Mathematics (Lecture Notes in Computer Science)*. Springer International Publishing, Cham, 456–465. https://doi.org/10.1007/978-3-319-78054-2_43

[21] Steven F. Railsback and Volker Grimm. 2019. *Agent-Based and Individual-Based Modeling: A Practical Introduction*. Princeton university press, Princeton, New Jersey, USA.

[22] Craig W. Reynolds. 1987. Flocks, Herds and Schools: A Distributed Behavioral Model. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '87)*. Association for Computing Machinery, New York, NY, USA, 25–34. https://doi.org/10.1145/37401.37406

[23] Paul Richmond. 2014. Resolving Conflicts between Multiple Competing Agents in Parallel Simulations. In *Euro-Par 2014: Parallel Processing Workshops*, Vol. 8805. Springer International Publishing, Cham, 383–394. https://doi.org/10.1007/978-3-319-14325-5_33

[24] Patrick F. Riley and George F. Riley. 2003. SPADES - a Distributed Agent Simulation Environment with Software-in-the-Loop Execution. In *Proceedings of the 2003 Winter Simulation Conference*, Vol. 1. IEEE, New Orleans, LA, USA, 817–825 Vol.1. https://doi.org/10.1109/WSC.2003.1261500

[25] Robert Ronngren and Michael Liljenstam. 1999. On Event Ordering in Parallel Discrete Event Simulation. In *Proceedings of 13th Workshop on Parallel and Distributed Simulation (PADS 99)*. IEEE, Atlanta, Georgia, USA, 38–45. https://doi.org/10.1109/PADS.1999.766159

[26] Xavier Rubio-Campillo. 2014. Pandora: A Versatile Agent-Based Modelling Platform for Social Simulation. In *Proceedings of SIMUL 2014*. IARIA, Nice, France, 6.

[27] Thomas C. Schelling. 1971. Dynamic Models of Segregation. *The Journal of Mathematical Sociology* 1, 2 (July 1971), 143–186. https://doi.org/10.1080/0022250X.1971.9989794

[28] Matthias Scheutz and Paul Schermerhorn. 2006. Adaptive Algorithms for the Dynamic Distribution and Parallel Execution of Agent-Based Models. *J. Parallel and Distrib. Comput.* 66, 8 (Aug. 2006), 1037–1051. https://doi.org/10.1016/j.jpdc.2005.09.004

[29] Matthias Scheutz, P. Schermerhorn, R. Connaughton, and Aaron Dingler. 2006. SWAGES - An Extendable Distributed Experimentation System for Large-Scale Agent-Based Alife Simulations. In *Proceedings of Artificial Life X*. MIT Press, Bloomington, USA, 412–419.

[30] Birgitt Schönfisch and André de Roos. 1999. Synchronous and Asynchronous Updating in Cellular Automata. *Biosystems* 51, 3 (Sept. 1999), 123–143. https://doi.org/10.1016/S0303-2647(99)00025-8

[31] Yoav Shoham and Moshe Tennenholtz. 1995. On social laws for artificial agent societies: off-line design. *Artificial intelligence* 73, 1-2 (1995), 231–252.

[32] David Strippgen and Kai Nagel. 2009. Using Common Graphics Hardware for Multi-Agent Traffic Simulation with CUDA. In *Proceedings of the 2nd International Conference on Simulation Tools and Techniques (Simutools '09)*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), Rome, Italy, 1–8. https://doi.org/10.4108/ICST.SIMUTOOLS2009.5666

[33] Wen Jun Tan, Philipp Andelfinger, Wentong Cai, Alois Knoll, Yadong Xu, and David Eckhoff. 2020. Multi-thread State Update Schemes for Microscopic Traffic Simulation. In *2020 Winter Simulation Conference (WSC)*. IEEE, USA, 1–12.

[34] Catherine Tessier, Laurent Chaudron, and Heinz-Jürgen Müller. 2006. *Conflicting Agents: Conflict Management in Multi-Agent Systems*. Vol. 1. Springer Science & Business Media, Berlin, Germany.

[35] Jonathan Thaler and Peer-Olaf Siebers. 2019. The Art of Iterating: Update-Strategies in Agent-Based Simulation. In *Social Simulation for a Digital Society: Applications and Innovations in Computational Social Science (Springer Proceedings in Complexity)*. Springer International Publishing, Cham, 21–36. https://doi.org/10.1007/978-3-030-30298-6_3

[36] Martin Treiber, Ansgar Hennecke, and Dirk Helbing. 2000. Congested Traffic States in Empirical Observations and Microscopic Simulations. *Physical Review E* 62, 2 (Aug. 2000), 1805–1824. https://doi.org/10.1103/PhysRevE.62.1805

[37] Jur van den Berg, Stephen J Guy, Jamie Snape, Ming C Lin, and Dinesh Manocha. 2011. Rvo2 library: Reciprocal collision avoidance for real-time multi-agent simulation.

[38] Tom Wagner, John Phelps, and Valerie Guralnik. 2004. Centralized VS. Decentralized Coordination: Two Application Case Studies. In *An Application Science for Multi-Agent Systems*, Thomas A. Wagner (Ed.). Vol. 10. Kluwer Academic Publishers, Boston, 41–75. https://doi.org/10.1007/1-4020-7868-4_4

[39] Jiajian Xiao, Philipp Andelfinger, David Eckhoff, Wentong Cai, and Alois Knoll. 2018. Exploring Execution Schemes for Agent-Based Traffic Simulation on Heterogeneous Hardware. In *2018 IEEE/ACM 22nd International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*. ACM, Madrid, Spain, 1–10. https://doi.org/10.1109/DISTRA.2018.8601016

[40] Jiajian Xiao, Philipp Andelfinger, David Eckhoff, Wentong Cai, and Alois Knoll. 2019. A Survey on Agent-Based Simulation Using Hardware Accelerators. *Comput. Surveys* 51, 6 (Jan. 2019), 131:1–131:35. https://doi.org/10.1145/3291048

[41] Mingyu Yang, Philipp Andelfinger, Wentong Cai, and Alois Knoll. 2018. Evaluation of Conflict Resolution Methods for Agent-Based Simulations on the GPU. In *Proceedings of the 2018 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (SIGSIM-PADS '18)*. ACM, New York, NY, USA, 129–132. https://doi.org/10.1145/3200921.3200940