Special Issue Article

Simulation

Simulation: Transactions of the Society for Modeling and Simulation International
1–17
© The Author(s) 2023
DOI: 10.1177/00375497231158930
journals.sagepub.com/home/sim

Synchronous speculative simulation of tightly coupled agents in continuous time on CPUs and GPUs

Philipp Andelfinger and Adelinde M Uhrmacher

Abstract
Traditionally, parallel discrete-event simulations of agent-based models in continuous time are organized around logical processes exchanging time-stamped events, which clashes with the properties of models in which tightly coupled agents frequently and instantaneously access each other's states. To illustrate the challenges of such models and to derive a solution, we consider the domain-specific modeling language ML3, which allows modelers to succinctly express transitions and interactions of linked agents based on a continuous-time Markov chain (CTMC) semantics. We propose synchronous optimistic synchronization algorithms tailored toward simulations of fine-grained interactions among tightly coupled agents in highly dynamic topologies and present implementations targeting multicore central processing units (CPUs) as well as many-core graphics processing units (GPUs). By dynamically restricting the temporal progress per round to ensure that at most one transition or state access per agent, the synchronization algorithms enable efficient direct agent interaction and limit the required agent state history to only a single current and projected state. To maintain concurrency given actions that depend on dynamically updated macro-level properties, we introduce a simple relaxation scheme with guaranteed error bounds. Using an extended variant of the classical susceptible-infected-recovered network model, we benchmark and profile the performance of the different algorithms running on CPUs and on a data center GPU.

Keywords
Parallel simulation, agent-based simulation, multiagent simulation, optimistic synchronization, GPU

1. Introduction

Agent-based simulation is an established method to study systems of interacting entities in domains such as sociology,[1] traffic engineering,[2] or systems biology.[3] A simple approach to the time advancement in agent-based simulations is taken by synchronous time-driven simulations, in which all agents' state transitions occur concurrently at discrete points in simulation time. This approach provides ample opportunities for parallel execution: since the agents' transitions at a given time step are logically concurrent, the transitions can be assigned to different processing elements to reduce execution times. However, in many real-world systems, transitions may occur at arbitrary points in time.[4] If agent-based models are thus specified with respect to continuous time, an execution using the limited granularity given by a fixed time step size may cause deviations from an execution according to the strict model semantics. Furthermore, conflicts may occur among the actions taken by agents within a time step.[5] To resolve conflicts, a time-driven simulation cannot rely on the precedence relation defined by the fine-grained transition times available in a continuous-time simulation. Furthermore, the time step size defines a lower bound on the propagation delay of effects throughout the simulation,[6] whereas in a continuous-time formulation, no such bound must be imposed.

On the contrary, while a continuous-time execution can represent the model semantics to machine precision, when considering parallelization to reduce execution times, a synchronization algorithm is required to satisfy the resulting stricter ordering constraints. A variety of parallel

Institute for Visual and Analytic Computing, University of Rostock, Germany

Corresponding author:
Philipp Andelfinger, Institute for Visual and Analytic Computing, University of Rostock, Albert-Einstein-Straße 22, 18059 Rostock, Germany.
Email: philipp.andelfinger@uni-rostock.de

discrete-event simulation (PDES) algorithms have been developed to ensure the efficient and correct execution of simulation models, typically oriented around logical processes that exchange events specified in continuous simulation time.[7,8]

A challenge to an efficient execution using PDES is given by the tightly interlinked life courses of communities of agents encountered in many agent-based models.[9] To cater to this ''tight coupling,'' multiagent modeling and simulation environments such as Repast,[10] NetLogo,[11] or Mesa[12] allow agents to directly carry out read and write accesses to the attributes of other agents. These direct write accesses to an agent's state variables by other agents appear to be at odds with the autonomy of agents, which describes an agent's ability to make decisions and effect state changes based on its own perception of the environment.[13] However, social systems tend to be rife with social structures in which not all participants act with full autonomy.[14] For instance, when a family migrates to another country, children may not take an active part of the decision-making process, but will be directly impacted by the migration. Similarly, employees at a company may have their employment terminated without their active involvement. Although it is possible to model such processes as actions taken as a consequence of autonomous observations and deliberations based on the environment state, as formally supported by the Influence-Reaction model,[15,16] direct accesses to the agents' state reflect in these cases mechanisms of real-world social systems and allow for more succinct modeling, and are thus supported by the majority of agent-based modeling simulation tools.

To illustrate the challenges of parallel simulation of tightly coupled agent-based models, we consider the domain-specific modeling language ML3,[17,18] which allows modelers to succinctly express transitions and interactions of linked agents based on a continuous-time Markov chain (CTMC) semantics. We highlight key properties of ML3 models to determine the requirements for an efficient parallel execution and present a novel optimistic synchronization algorithm. In optimistic PDES, some computations are carried out speculatively without regard for potential violations of ordering constraints. If a violation occurs, the affected simulation state is rolled back to a previous correct state. To cope with the tight coupling among separate agents' states, the algorithm follows two main ideas. First, agents may directly read and write other agents' states, without the exchange of events or messages as used in traditional PDES approaches. Second, a synchronous mode of time advancement restricts both the temporal deviation among processors and avoids cascading rollbacks and the overhead required for their handling entirely. In addition, we propose a variant of the algorithm that avoids the locking of agent states and exposes fine-grained parallelism, making this variant well-suited for the execution on graphics processing units (GPUs). The lock-free variant of our algorithm inherits most properties of the base algorithm, but employs a light-weight messaging mechanism to represent direct interagent accesses. Implementations of the proposed algorithms for execution on multicore central processing units (CPUs) and GPUs are evaluated using a variant of a classical epidemic network model,[19] which we extend to stress the performance-critical aspects of ML3 models.

We summarize our contributions as follows:

- We describe performance-critical aspects of ML3 models and their implications for parallelization.
- We propose and detail a synchronous optimistic synchronization algorithm for models of tightly coupled agents.
- We present the additions and modifications to the base algorithm required to achieve lock-free execution.
- Performance measurements under challenging model configurations using scenarios populated by up to 2^{26} agents demonstrate speedups of up to 5.1 using 32 CPU cores and 10.9 on a data center GPU compared with an efficient sequential baseline.

This article is an extended version of our previous conference publication.[20] The new material over the conference version includes the design of the lock-free variant of the algorithm, the description of its implementation targeting many-core GPUs, and new profiling results aiming to further elucidate the causes for the observed performance of the two algorithms depending on the model configuration.

The paper is structured as follows: In Section 2, we describe key properties of the considered class of models, their implications for parallel execution, and existing methods for parallel simulation of tightly coupled systems. In Section 3, we propose optimistic synchronization algorithms tailored to the identified requirements. In Section 4, we describe our implementation of the two optimistic algorithms and our sequential baseline. In Section 5, we evaluate the performance of the algorithm and present profiling results on the different hardware platforms. Section 6 summarizes our results and concludes the paper.

2. Fundamentals and related work

ML3 is a modeling language offering constructs to succinctly model complex interactions among agents linked within dynamically evolving topologies. In the present paper, the language serves as a representative of a model class in which agent states are updated in a tightly linked manner using rate-driven transitions. In the following, we introduce the key ideas of ML3 and their implications for the parallelized execution of ML3 models. A comprehensive discussion and a formal semantics of the language are provided in Reinhardt et al.[18] Furthermore, we discuss existing work on the parallelization of the underlying Stochastic Simulation Algorithm and on the consideration of macro properties in parallel and distributed simulations. Our proposed synchronization algorithms are closely related to Steinman's[21] classical *breathing time buckets* algorithm. Hence, we describe how our algorithms differ from Steinman's before discussing a number of synchronization approaches that deviate from the assumption of a strict separation of the simulated entities' state or the strict reliance on events to represent interactions.

2.1. ML3

The design of ML3 is oriented around the requirements of agent-based modeling in demography.[17,18] The simulation state is held by the attributes of individual agents, which are comparable to classes and objects in object-oriented programming languages. Connections and relationships among agents are represented by links, which are bidirectional relations with per-direction role names such as "parent" and "child." The topology reflected by the agents' links can be dynamically updated throughout a model's execution. The model behavior is specified in terms of agent-specific rules that define their possible state transitions in terms of their conditions, timings, and effects.

Transitions in ML3 are specified using rules comprised of three elements:

- A *guard expression* defines a predicate that must hold for the rule to apply, i.e., for the transition to occur.
- A *waiting time expression* defines the time or the rate at which the transition occurs. Rates are defined either as a rate constant or as a function of current or other agents' attributes, and may also be time-dependent.
- An *effect* specifies the state changes to occur when the transition is actuated, which may affect the current or other agents' attributes and links.

As an example, the following rule is an excerpt from the susceptible-infected-recovered model commonly used in epidemiology[22] formulated in ML3.

```
Person
  | ego.status = ''susceptible''      // guard
  @ a * ego.network.filter(           // waiting time
    alter.status = ''infected'').size()
  -> ego.status := ''infected''       // effect
```

The rule applies only to agents in the "susceptible" state. The transition occurs at a rate dependent on the constant a and the current number of neighbors in the infected state. This implies an exponentially distributed sojourn time with a mean of $1/an$, with n being the number of infected neighbors. When the transition is actuated, the agent enters the "infected" state.

Importantly, the variables affected by a transition's effect may instantaneously alter dependent transition rates of the current or other agents.

Algorithm 1: Pseudo code of simulating an ML3 model using the Next Reaction Method.[18]

```
1  foreach agent in agents do
2      foreach rule in agent.rules do
3          if rule.guard_expression() then
4              rate ← rule.rate_expression()
5              schedule_transition(rule, random_exponential(rate))
6  While !termination_criterion do
7      transition ← select_earliest_scheduled_transition()
8      affected_rules ← transition.rule.effect()
9      foreach rule in affected_rules do
10         retract_transition(rule)
11         if rule.guard_expression() then
12             rate ← rule.rate_expression()
13             schedule_transition(rule, transition.time +
14                                  random_exponential(rate))
```

2.1.1. CTMC and stochastic race.

ML3 allows waiting times to be defined either in the form of rates of an underlying CTMC,[23] in the form of rates of distributions dependent on the global simulation time, or by specifying fixed transition times directly. In the present article, we focus on transitions following the CTMC semantics, i.e., the simulation is a memoryless stochastic process with exponentially distributed inter-transition times. In this setting, the simulation proceeds as a sequence of *stochastic races*: given several imminent transitions with associated transition rates, the transition to occur next is selected stochastically. As a consequence of each transition, other transition's rates may be updated, after which the next stochastic race commences.

This mode of simulation is inspired by stochastic simulation algorithms, key variants of which were originally proposed by Gillespie[24] to simulate biochemical reaction networks. Two specific algorithms in this category include the *Direct Method*[24] and the *Next Reaction Method*.[25] The Direct Method and its variants select the next transition directly based on the current transition rates: similarly to the generation of pseudo-random numbers adhering to an empirical distribution, a uniform random variate determines the index of the next transition depending on their relative rates. A second uniform random variate is transformed according to the sum of all transition rates to generate the transition time. As each selection of a transition and its time requires the transition rates to be current, the Direct Method suggests a serialized mode of execution. In contrast, the Next Reaction Method draws tentative timestamps for all possible transitions. The transition with the earliest timestamp is carried out, which may affect the rates of other transitions. Such dependent transitions are then rescheduled according to the updated rates. Algorithm 1 shows pseudo code for the simulation of an ML3 model according to the Next Reaction Method.

2.1.2. Parallelization challenges. The choice of simulation algorithm has important implications for the parallel execution of ML3 models. The Next Reaction Method may discard many tentatively scheduled transitions if the degree of coupling among agents is high. However, the scheduled events provide valuable information of the time and agent assignment of future transitions assuming independence of the transition rates. Since this information permits a speculative parallel execution of transitions, we rely on the Next Reaction Method throughout the paper, which also does not require transition delays to be distributed exponentially. The execution of ML3 models using the Next Reaction Method can be viewed as a special case of discrete-event simulation, which may suggest the use of well-established methods for parallelization. However, we identify three properties of ML3 models and, more generally, tightly coupled agent-based models that pose challenges to traditional PDES approaches:

1. Direct read and write accesses across agent boundaries to support tightly coupled life courses of agents,
2. State-dependent creation and removal of links and agents to account for the dynamic structure of systems,
3. Global access to the state of a population of agents to calculate macro-level properties which may influence the agent behavior at the micro level.[26]

We briefly illustrate the occurrence and implications of each of these properties in turn, relying on an ML3 formulation of a migration model.[17] The model, which represents decision-making processes involved in migrations from Senegal to France, was originally developed in Netlogo.[27] The model excerpts shown are slightly simplified for brevity.

2.1.2.1. State access across agent boundaries. As do other agent-based modeling and simulation environments, ML3 permits read and write accesses to arbitrary agents as part of the guard expressions, waiting time expressions, and effects that make up a rule. The following excerpt is part of the "effect" component of a rule in the migration model:

```
(ego.friends + ego.friends.collect(alter.friends)−
 ego.familyMembers()−ego).filter(ego.canMarry(alter))
```

The expression filters an agent's friends and its second-degree friends according to the predicate function `canMarry()`, which in turn accesses the agents' attributes. The accesses occur instantaneously during the transition. In a scenario in which an agent has exactly 10 direct friends, each evaluation of the expression requires accesses to up to 100 friends and friends-of-friends. Considering the execution of such a transition in a parallel simulation, we note that the agent must be able to access the other agents' states *at the simulation time of the transition*. This can be achieved by maintaining a history of previous states, rolling back agents to previous states when required by write accesses, e.g., using the classical Time Warp algorithm for optimistic PDES.[28] However, the Time Warp algorithm assumes that the interactions among simulation objects occur through the exchange of events in a message-passing style, which would require a "read request" and "read response" event to carry out a single read access between two agents. Furthermore, the two events involved in a read access would carry the same timestamp as the current transition. The resulting tight temporal coupling among the involved logical processes runs counter to the asynchronous mode of execution defined by the Time Warp algorithm.

2.1.2.2. State-dependent creation and removal of links. At a transition, an agent may create and remove links based on its own state, the state of other agents, or randomly. In the following excerpt from the migration model, an agent moving to a new address creates links to the current inhabitants of the address and the inhabitants of the neighboring addresses:

```
ego.friends+=?address.inhabitants +
            ?address.neighbors.collect(
                        alter.inhabitants) - [ego]
```

The resulting topology evolves dynamically over the course of the simulation, which has two key consequences for parallelization: first, if the simulation state is partitioned into logical processes, frequent repartitioning is required to minimize attribute accesses across process boundaries. In the presence of randomized link creations, the proportion of accesses across logical processes may be large even with frequent repartitioning. Second, even if a specific rule restricts its accesses to direct neighbors of an agent a_0, topology information cannot easily be exploited to determine independent transitions, as a concurrent transition of an agent a_1 may create or remove a link to a_1, which would invalidate the topology information.

2.1.2.3. Globally accessing the state of a population of agents. State variables of entire populations of agents may be accessed by an agent. As an example, in the migration model, a migrant randomly selects a new address from a global set of all addresses, filtering by country and current inhabitation:

```
Address.all.filter(alter.location = ''host country''&&
                   !alter.hasInhabitants()).random()
```

Each of the addresses is represented as an agent. For this expression to be evaluated correctly, all previous updates to the set of addresses must be visible. More generally, by imposing ordering constraints among transitions at separate agents, the presence of accessing globally entire populations of agents can severely limit the concurrency among transitions. When strictly adhering to the model semantics, the extreme case of this type of global access at every transition implies a complete serialization of the simulation.

Overall, while the rate-based transition times in ML3 models necessitate the use of an optimistic approach to synchronization, the tight and difficult-to-predict coupling among agents suggests a scheme that emphasizes the efficiency of attribute accesses and limits the frequency and cost of rollbacks.

2.2. Simulations of tightly coupled systems using time warp

The parallel and distributed execution of the stochastic simulation algorithm in its traditional domain of biochemical systems has been explored by several authors. The Next Subvolume Method, which is a spatial extension of the Next Reaction Method based on a regular grid, has been executed using variants of the Time Warp algorithm.[29–31] As in our work, constraining the optimistic execution proved beneficial for performance.[31] Similarly, Goldberg et al. employ Time Warp to execute the Next Reaction method. In contrast to our synchronization algorithm, Time Warp requires interactions among simulation objects to be mediated through events and allows logical processes to progress asynchronously.

Several works have evaluated and optimized agent-based simulations running in a Time Warp kernel. Pawlaszczyk and Timm[32] proposed method for avoiding excessive optimism based on model-specific properties. Based on the interaction protocol followed by the agents, structural knowledge about the interaction is exploited to reduce the frequency of rollbacks. Using model similar to our experiments, Rao[33] highlighted the challenges of executing susceptible-infected-recovered models using Time Warp. A key focus was the reduction in message exchanges among logical processes to ameliorate the increasing overhead with larger numbers of processor cores. A direct comparison of our synchronization algorithms' performance to the use of a Time Warp kernel when executing ML3 models is provided in Andelfinger et al.[34] The relative performance of the algorithms is determined largely by the amount of locality in the agent interactions: If interactions are confined to large numbers of small regions, Time Warp vastly outperforms the synchronous approach. On the contrary, if the scenario contains sufficiently large numbers of agents, the synchronous algorithm achieves speedup regardless of the amount of locality.

2.3. Breathing time buckets

Breathing time buckets is a synchronous algorithm in which the processing within each round is bounded by the timestamp of the earliest newly created event. Any computation beyond this point in simulation time, referred to as the *event horizon*, is rolled back. Newly scheduled events are only sent among logical processes at the end of a round, leading to an execution scheme that alternates between the processing and exchange of events. Our base algorithm differs in its deviation from a traditional event-oriented representation of state changes and in their definition of the event horizon: as detailed above, the considered agent-based models combine individual state transitions at an agent with read and/or write accesses as well as potential rescheduling operations at neighboring agents, all occurring at the same point in simulation time. Since the breathing time buckets algorithm delays event exchanges to the end of a round, any such combined operation would require multiple rounds to complete. Instead, our base algorithm permits direct access to the neighboring agents' states. Such interagent accesses may be invalidated throughout the processing of a round, requiring rollbacks. To minimize the required state history per agent and to avoid costly interactions among threads to signal access invalidations, the algorithm maintains the invariant that at the end of a round, only at most one access per agent is committed. As an agent may only schedule events targeting itself, the creation and subsequent execution of an event within the same round would constitute two accesses to the agent, which is not permitted in our algorithm. Thus, our definition of event horizon is stricter than the one used in the breathing time buckets algorithm. The lock-free variant of our algorithm represents interagent accesses explicitly in the form of temporary messages, but maintains our deviating definition of the event horizon.

2.4. Alternative synchronization approaches

Several mechanisms for synchronization and state accesses have been proposed that depart from the classical parallel and distributed simulation paradigm wherein simulation objects are assigned to logical processes and all interactions among objects are represented in the form of event exchanges. Ghosh and Fujimoto[35] proposed the concept of space-time memory, which introduces a versioning of variables in Time Warp simulations to correctly handle concurrent accesses in shared memory settings. Chen et al.[36] aggregate logical processes to groups within which variables can be accessed directly. Marziale et al.[37]

propose the dynamic combination of simulation objects based on runtime information regarding the frequency of mutual accesses. Pellegrini et al.[38] described a scheme that achieves a transparent versioning of global variables. Substantial performance gains are achieved by avoiding some of the event exchanges required in traditional implementations. Ianni et al.[39] proposed an optimistic parallel simulation system in which all threads obtain events from a single shared list. Events are processed without a fixed mapping between threads and logical processes or simulation objects, while still avoiding conflicting state accesses. Pellegrini and Quaglia[40] presented a mechanism for transparent access to the state of arbitrary simulation objects in optimistic simulations. This is achieved by an operating system-level redirection of memory accesses. To guarantee correctness, the current event is suspended and a new event is scheduled that handles the actual state access. As in our synchronization algorithm, these approaches loosen the mapping between threads and simulation objects encountered in traditional PDES approaches. Our work differs in avoiding the use of events to represent state accesses across objects altogether, and our synchronous approach, which reflects the tightly coupled nature of the agent-based models.

In 2020, Chen et al.[41] presented a mechanism for agent interaction in optimistically synchronized distributed simulations in which interactions are mediated by mailboxes holding time-stamped messages. Since our work assumes a shared memory execution environment, agent interactions take place without mediation. By permitting at most one access to each agent in a round, we avoid maintaining a state history similar to a mailbox beyond a "current" and "projected" agent state. We leave an exploration of variants of our synchronization algorithm with per-agent state histories within each window to future work.

A variety of methods have been proposed to control the degree of optimism in speculative synchronization algorithms by limiting the relative progress of logical processes to balance the exploitation of parallelism and the overhead for rollbacks.[42–45] In our synchronous approach, the progress within each window is limited by an upper bound in simulation time calculated based on the effective size of a previous window. During each round, the effective window size is gradually reduced. Since threads promptly terminate the current round once the current window size prevents any further progress, the sensitivity to the initial window size is low (cf. Section 5), avoiding the need for sophisticated mechanisms to determine a suitable initial window size.

While outside the scope of our present work, some agent-based models may benefit from efficient mechanisms to support global range queries, e.g., to select all agents within a certain age group. Several efficient algorithms for this purpose targeting distributed environments have been proposed and evaluated in the literature.[46,47]

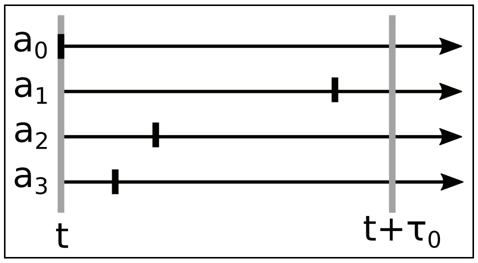

Figure 1. Example of the transitions (vertical lines) of four agents a_i scheduled after a global synchronization point at simulation time t. Initially, the event horizon limiting the speculative execution of transitions is set to $t + \tau_0$.

3. Synchronous speculative synchronization algorithm

Based on the properties of ML3 models detailed in the previous section, our central idea is to acknowledge that the considered models represent tightly coupled systems in which a typical transition directly and instantaneously affects several agents scattered throughout the topology, and in which the propagation of effects over time can be swift and difficult to predict. As a consequence, we propose a synchronous algorithm that, instead of emphasizing the maximization of parallelism, aims to avoid some of the overheads for state accesses and rollbacks that may be incurred by more aggressive speculative algorithms. Our synchronization algorithm is based on the simple observation that starting from a global synchronization point, the earliest access in simulation time at an agent can never be invalidated by another access to the same agent.

3.1. Overview

The simulation takes a round-based approach, wherein we ensure that at the end of each round, only the earliest access at each agent is committed. Hence, the delta in simulation time by which the simulation can advance throughout a round depends on the interactions among the agents. We illustrate the desired behavior of the round-based execution on the example of simulation populated by four agents (cf. Figure 1). The current simulation time t is the timestamp of the earliest scheduled transition. During the current round, we consider all transitions up to $t + \tau_0$ for execution, where τ_0 is a tunable initial window size in simulation time. Figure 2 shows the operations associated with the agents' transitions. For simplicity, we

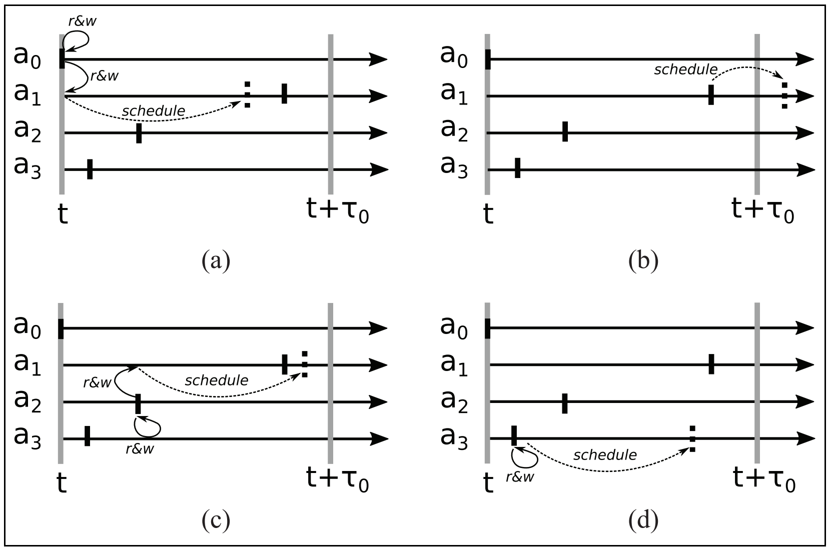

Figure 2. Scheduled transitions and their effects. When executed in parallel, agents may carry out their transitions in any order. Our synchronization algorithm guarantees that of the accesses to a_1 caused by the transitions of a_0, a_1, and a_2, only the earliest access takes effect and is committed. (a) Agent a_0, (b) Agent a_1, (c) Agent a_2, and (d) Agent a_3.

assume that all attribute accesses take place in read and write mode. While the operations of agents assigned to a single thread occur in order of ascending timestamps, the order of operations across threads is arbitrary. Importantly, multiple transitions executed speculatively within the round may access the same agent. Finally, Figure 3 shows the desired outcome of the round: the final window size τ guarantees that we commit exactly those transitions whose accesses occurred earliest in simulation time for all accessed agents, along with any involved write accesses. Furthermore, the scheduling of any events speculatively scheduled by the newly committed transitions takes effect.

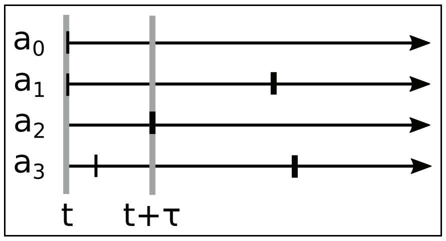

Figure 3. Final scheduled transitions (wide lines) and committed agent accesses (thin lines) at the end of the round. Throughout the processing of transitions, the effective event horizon $t + \tau$ has been gradually reduced to guarantee that at most one access per agent is committed. In the example, the transitions at agents a_0 and a_3 and the associated write accesses are committed, whereas the transitions of agents a_1 and a_2 are deferred to future rounds.

3.2. Mechanism

Having illustrated the intended outcome of a synchronization round, we now describe the mechanisms by which this outcome is achieved. Algorithm 2 shows the main loop of our synchronization algorithm as pseudo code. We assume that agents are assigned to threads by a static partitioning scheme, e.g., randomly. Each round of simulation begins with a global reduction to determine the minimum timestamp among all threads' scheduled transitions. The initial upper bound on the timestamp of transitions considered for execution, the *event horizon* is initialized based on the minimum timestamp and the tunable initial window size τ_0. Our choice of τ_0 is discussed in Section 5. Each thread populates a list `window_transitions` holding all local transitions scheduled for execution before the initial event horizon (line 4). Subsequently, each thread executes the transitions in the list in order of ascending timestamp. The timestamp order of execution is exploited in line 6 to terminate the execution of transitions early if the event horizon does not permit any further progress in the current round.

As we will detail below, the agent accesses carried out during transitions reduce the event horizon to guarantee that only at most one access at each agent is committed. While executing transitions, each thread populates a list of agents accessed by the local agents, which may include agents assigned to other threads. By storing the agents at the thread the access originated from, the need for locking the lists is avoided.

A global barrier ensures that the final event horizon of the current round has been determined (line 9). Now, all transactions with timestamps lower than the event horizon

Algorithm 2: Main speculative parallel simulation loop.

```
1  while !termination_criterion do
2      ev_horizon ← get_global_min_timestamp() + $\tau_0$
3      foreach thread in parallel do
4          window_transitions ←"local events before
           ev_horizon"
5          foreach event in window_transitions do
6              if event.timestamp $\geqslant$ ev_horizon then
7                  break
8              event.execute()
9          barrier()
10         foreach event in window_transitions do
11             if event.agent.earliest_access < ev_horizon then
12                 commit transition, enqueue new events
13             else
14                 roll agent back to previous state
15         foreach list in accessed_agents_lists do
16             foreach agent in list do
17                 if agent.is_local() and
18                 agent.earliest_change < ev_horizon then
19                     commit transition, enqueue new events
20                 else
21                     roll agent back to previous state
22         barrier()
```

Algorithm 3: Wrapper for agent accesses.

```
1  procedure Agent::try access (now):
2      lock(mutex)
3      if now < earliest_access then
           //access is earliest in round so far
4          if earliest_access $\neq \infty$ then
5              self ← old_self roll back prev. access
               // defer transition associated with
               // previous access to future round
6              atomic_min(&ev_horizon, earliest_access)
7          perform_access()
8          earliest_access ← now
9      else
           // access not earliest in round,
           // defer associated transition
10         atomic_min(&ev_horizon, now)
11     unlock(mutex)
```

are committed, and new transitions scheduled in the process are enqueued. Agents whose transitions occurred at or after the event horizon are rolled back. Similarly, agent accesses that occurred earlier than the event horizon are committed, and all other accesses are rolled back. To identify accessed agents, each thread iterates through all the threads' lists of accessed agents. As the lists are not written to, no locking is required. Once all threads have reached the final barrier, the simulation has advanced to the event horizon and may enter a new round.

In the pseudo code of Algorithm 2, the updates of the event horizon within a round were implicit. Algorithm 3 shows this aspect in detail: on an agent access, a per-agent mutex is obtained. If the new access arrives at the earliest observed simulation time, the access is carried out, displacing a prior access with larger timestamp, if any. Otherwise, the new access fails and the event horizon is updated to defer the new access to a future round, which globally restricts the processing of transitions at all threads.

We point out the close relationship of our synchronization algorithm to the conflict resolution mechanisms used in time-driven agent-based simulations, in which all agents concurrently advance their states from simulation time t to $t + \tau$. A number of approaches have been proposed and evaluated to resolve the potential conflicts that may emerge when accessing limited resources, e.g., if multiple agents move to the same location in the simulation space.[5,48,49] In a "push" approach, agents register their desired state accesses at the simulation objects, potentially displacing previously registered agents based on static or dynamic priorities. At the end of a round, the highest-priority registered agent gains access to the object. This process repeats until all agents have gained access to an object or given up, at which point simulation time is increased to $t + \tau$. Both in this procedure and in our optimistic synchronization algorithm, agents concurrently attempt to access other simulation objects, potentially displacing each other. In both cases, a round concludes once at most one access per entity can be committed, deferring displaced agent accesses to a future round. The key difference between conflict resolution in time-driven simulations and our synchronization algorithm lies in the definition of the access priorities: in the considered class of agent models, the timestamps in continuous time associated with transitions define priorities prescribing an access order. In contrast, given a model that assumes concurrent transitions of all agents at fixed time steps, priorities within each time step must be defined separately based either on model properties or according to some other criterion, e.g., based on pseudo-randomness.[5]

3.3. Fine-grained lock-free variant

The synchronization algorithm described above employs locks to immediately carry out remote agent accesses triggered by a transition. A new access immediately displaces any access with a higher timestamp applied previously. In the following, we describe a variant that avoids the need for locking by carrying out accesses only after the final event horizon of a given round has been determined. By explicitly storing messages we refer to as *access requests*, of which at most one can subsequently be actuated by a given agent, this lock-free variant surrenders the direct interagent accesses of the lock-based scheme. However, the synchronous execution still rules out cascading

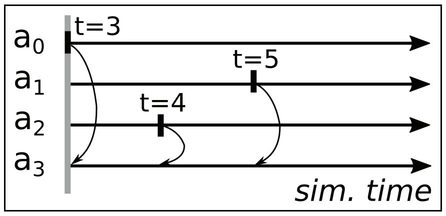

Figure 4. Transitions within a synchronization round to illustrate the lock-free scheme. Agents 0, 1, and 2 attempt to execute transitions, each accessing agent 3. Only the access at time 3 triggered by the transition at agent 0 is actuated in this round.

rollbacks and hence avoids the need for antimessages. The lock-free variant differs from the original synchronization algorithm in several regards described in the following.

3.3.1. Interagent accesses. We illustrate the handling of accesses using the example shown in Figure 4. Within the current round, agents 0, 1, and 2 execute transitions at simulation times 3, 5, and 4. Each of the transitions involves a write access to agent 3. As previously, the accesses update the event horizon (cf. Algorithm 3). However, no locking occurs (lines 2 and 11), the agent is not accessed immediately (line 7), and thus, an immediate rollback (line 5) is never required. Instead of carrying out the access immediately (line 7), an access *request* is appended atomically to agent 3's *access request list*. Since our synchronization algorithm only allows the earliest of any accesses to an individual agent to be committed, only the access at simulation time 3 by agent 0 can be actuated, whereas the accesses triggered by the transitions at agents 1 and 2 are deferred to a later round. The final content of agent 3's access request list depends on the order at which the access requests arrive, as accesses with larger timestamp than any previously recorded access request are discarded immediately. In contrast to write accesses, pure read accesses to an agent are carried out immediately as in the locking-based scheme, without first being appended to the access request list. Since the updating of the event horizon rules out situations where multiple transitions read from the same agent within a round, the result of a read access prior to the final event horizon is always correct.

Figure 5 illustrates the resulting the possible final content of agent 3's access request list for all possible arrival orders of the access requests.

3.3.2. Presampling of transition times. An interagent access may trigger the scheduling or rescheduling of a transition at the accessed agent. The timestamp of such newly scheduled transitions must be considered in the calculation of the event horizon. In the base algorithm described in Section 3.2, accesses are carried out immediately, allowing us to obtain the exact timestamp of any new transition immediately. In contrast, since the lock-free scheme defers accesses to the end of a round, the timestamp of newly scheduled transitions is not be known. This is solved based on the observation that in the considered model class, transition delays are drawn from exponential distributions and thus depend only on the transition rate and a pseudo-random number. Hence, to ensure that no newly scheduled transition interferes with accesses to be committed in the present round, we require the user to supply an upper bound on the rate of a newly scheduled transition. The upper bound can be computed dynamically based on the agent state at the beginning of the current round and the guarantee that at most one access may occur before scheduling the new transition. Using this upper bound on the transition rate, the earliest possible transition time can be computed by presampling the random number stream of the current agent. The earliest transition time for a given agent is computed on the first arrival of an access request within a round. In Section 5.1, we give an example of computing such an upper bound for the simulation model used in our experiments.

3.3.3. Deferred actuation of accesses. Once the final event horizon has been computed, agents either commit their executed transitions or roll back to their previous state as shown in Algorithm 2. Agents with nonempty access request lists now traverse their lists and actuate any accesses with timestamps prior to the event horizon, scheduling any new transitions created in doing so.

By employing fine-grained unsorted lists to store access requests, the lock-free algorithm is well-suited for the execution on GPUs. An implementation targeting GPUs will be detailed and evaluated in Sections 4.2 and 5.

3.4. Global state access

We briefly describe our simple mechanism to support globally accessing state variables of all agents to calculate a macro property of the system, e.g., the number of agents being in a specific state. Let us consider an extreme case in which all agents' transition rates depend on a macro property that is updated during every transition. In addition to requiring a rescheduling at all agents after every transition, every transition depends on the transition prior to it, resulting in a complete serialization of the simulation. Thus, as our intention is to execute transitions in parallel, a strict adherence to these global updates is infeasible. Instead, we approximate such global dependencies by notifying dependent agents only once the variable has changed by a configurable amount.

In the sequential case, this type of approximation is trivially implemented by storing the value of the global variable during the last notification and triggering new

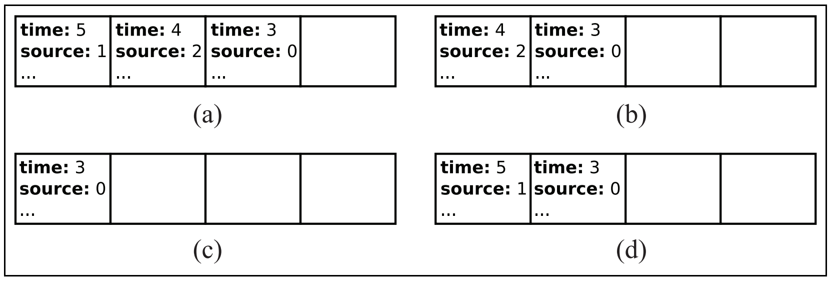

Figure 5. Possible contents of agent 3's access request list at the end of the example round, depending on the requests' arrival order as identified by the access timestamps. (a) Order: (5, 4, 3), (b) Order: (4, 3, 5) or (4, 5, 3), (c) Order: (3, 4, 5) or (3, 5, 4), and (d) Order: (5, 3, 4).

notifications once the change exceeds the threshold. However, during speculative parallel execution, the threads alter the global variable in an unpredictable order and may roll back prior changes. To identify the exact point in time at which the threshold is crossed, we maintain a variable `abs_change` holding the sum of the *absolute* changes to the global variable, independently of their sign. This variable provides an upper bound on the amount of change of the global variable in either positive or negative direction. During a parallel simulation round as described previously, `abs_change` is updated atomically by all threads. We terminate the round at the point $t_{\text{threshold}}$ in simulation time when `abs_change` crosses the configured change threshold, which is a potential point in time at which the actual change threshold may have been crossed. After committing and rolling back transitions, the value of the global variable is consistent with the sequential simulation at $t_{\text{threshold}}$. We can now check whether the configured threshold has in fact been crossed by the transition that terminated the round, and trigger notifications if needed. Finally, `abs_change` is set to the actual change and the next round of simulation commences. This simple scheme guarantees notifications at exactly the same points in simulation time as in the sequential simulation. In Section 5, we explore the sequential and parallel simulation performance in the presence of an approximately updated global variable that represents a macro property that the micro behavior of the agent-based model depends on.

4. Implementation

In the following, we describe implementations of three simulation algorithms: a sequential reference as a baseline for the subsequent experiments, our base algorithm targeting CPUs, and the lock-free algorithm targeting GPUs.

4.1. CPU implementation

Our starting point is a sequential C++ simulator implementation created from scratch. The simulator employs Gillespie's *Next Reaction Method*: an event is scheduled for each rule whose conditions are satisfied, according to the current transition rate. The simulation advances by executing the transition associated with the earliest scheduled event, which may entail updates to zero or more conditions or rates and subsequent (re-)scheduling of events.

We implemented two mechanisms to avoid executing events that have been retracted to reschedule transitions: (1) removal from the pending event list and (2) skipping based on an agent attribute. The first mechanism requires a data structure that permits efficient element removal. We employed the `set` container from the C++ standard template library to represent the pending event list, which is implemented as a red-black-tree and allows for logarithmic-time insertion and removal. In the second mechanism, events are stored in an STL `priority_queue` container, which internally relies on an implicit heap and in our experiments achieved vastly faster element insertion, but does not support efficient element removal. Each agent possesses one attribute per rule that holds the timestamp of the next transition. Whenever an event is scheduled or retracted, the corresponding attribute is updated. Thus, when considering an event for execution, the simulator can now compare the timestamp of the earliest event with the timestamp stored at the corresponding agent to decide whether the event has been retracted and should thus be skipped. With this second approach, an explicit removal of events is avoided, at the cost of retaining many of the retracted events in the list. Our experiments rely on the second mechanism, which during initial tests consistently outperformed explicit event removal.

The agent states are stored in two arrays, one holding the "current" states at simulation time t, i.e., the lower bound of the current window, and one holding the projected states to be determined during the current round. At the end of a round, the projected states are either rolled back to time t, i.e., discarded, committed by being copied to form the new "current" states at the lower bound of the next window. As described in Section 3, during an agent access, the target agent may be rolled back. By restricting the state history to only the state at *t*, rolling back an agent identified by `agent_id` involves only the trivial and inexpensive assignment:

```
*this=previous_states[agent_id];
```

To achieve deterministic results, each agent draws pseudo-random numbers from a separate random number stream generated by the Xoroshiro128** generator,[50] which passes the BigCrush test suite from the TestU01 library.[51] Since rollbacks require copying the previous random number generator state as part of the overall agent state, both the memory consumption and execution time of the simulation benefit from the generator's comparatively small state size of 128 bits.

For multithreading, we employ POSIX threads and the associated facilities for barriers and mutual exclusion. Threads are pinned to physical CPU cores to improve cache usage and to avoid unnecessary nonuniform memory accesses when using fewer than the available number of cores. For atomic access to the shared upper bound on the current window (`ev_horizon`), we employ the atomic operations library from the C++ standard template library.

4.2. GPU implementation

We also implemented our synchronous algorithm for execution on GPUs. The hardware architecture of modern GPUs is characterized by dedicating more transistors to the processing of arithmetic and logical operations rather than control flow and caching.[52] Small groups of threads, frequently referred to as warps, execute instructions in lockstep. Support for divergent control flow within a warp is achieved by discarding results computed by threads which do not follow a given control flow branch. Typically, an efficient GPU program schedules many more threads than there are physical processing elements, allowing the GPU hardware scheduler to dynamically switch among warps to hide memory access latencies. In addition, memory accesses of several adjacent threads to adjacent locations in memory are *coalesced*, i.e., served using a single memory transaction. Due to these architectural choices, GPUs excel at programs that approximate a single-instruction multiple-data (SIMD) type of processing, which runs counter to the highly divergent control flow inherent to the locking-based synchronization algorithm presented in Section 3.2. However, the lock-free algorithm variant described in Section 3.3 enables an efficient GPU-based implementation. In the following, we describe the key considerations when implementing the algorithm using NVIDIA CUDA.

A key computational step in the considered class of simulations is the selection of imminent transitions. As described in Section 2.1, each type of transition carried out

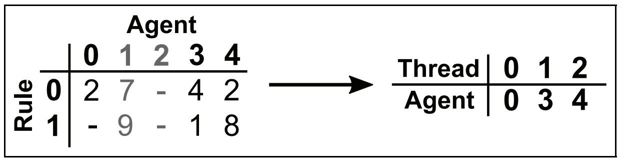

Figure 6. Transition selection in the GPU implementation assuming an event horizon of 3. Transition timestamps are stored using one array per rule (left). The transition selection step assigns each GPU thread the identifier of one agent with any transition prior to simulation time 3 (right).

by an agent is bound to a rule and occurs at a time determined according to a rate, which may depend on the simulation state. Given an initial event horizon at the start of a round, our SIMD approach to identifying imminent transitions within the event horizon employs a one-to-one mapping between GPU threads and agents. To maximize the opportunities for memory coalescing, the agents' transition timestamps are stored as arrays on a per-rule basis, allowing adjacent threads to access adjacent timestamps while iterating across the rules. In doing so, we collect the identifiers of *active* agents, i.e., those with imminent transitions, in a dense array, for which transitions are again executed using a one-to-one mapping between GPU threads and agents. Figure 6 illustrates this simple approach, which is efficient if the proportion of active agents per round is high. Otherwise, substantial overhead is incurred by redundant checks for inactive agents. To avoid redundant checks, we employ the concept of *epochs*, which are periods in simulation time spanning multiple rounds. We maintain a boolean array referred to as a *transition mask* that stores for each agent whether a transition *may* occur within the current epoch. This idea is akin to Steinman's[53] event list management, in which the temporal ordering of events occurring in the far future is postponed. The transition mask allows us to temporarily exclude agents from consideration for which transitions can be ruled out. At epoch boundaries, the transition mask is initialized by setting an agent's boolean value to `true` if there is a scheduled transition with a timestamp within the next epoch, and to `false` otherwise. Over the course of the epoch, agents whose mask is false are not checked for imminent transitions. When an interagent access triggers the scheduling of a new transition within the current epoch, the accessed agent's boolean value is set to `true`. We note that due to rescheduling, the agent may ultimately still remain inactive throughout the epoch. Nevertheless, as will be evaluated in Section 5, the transition mask severely reduces the overhead for identifying imminent transitions, without altering the simulation results.

Each agent draws pseudo-random numbers using CUDA's default XORWOW[54] generator, which uses 192 bits of mutable state. When a agent appends an access request to another agent's list, a unique offset is determined based on hardware-supported atomic instructions.

5. Experiments

5.1. Model

To evaluate the benefits and limitations of the proposed synchronization algorithm, we constructed a simulation model that emphasizes the hard-to-parallelize properties of tightly coupled agent-based models as detailed in Section 2.1. The model is based on an agent-based formulation of the classical susceptible-infected-recovered model as described by Macal,[55] to which we introduce rate-based transition probabilities in continuous time: in place of the per-step transition probabilities of the original time-driven model, transition times are drawn from an exponential distribution according to rates dynamically updated based on the neighboring agents' states. An agent is infected by its neighbors at a rate equal to its number of infected neighbors. Recovery from an infection and the return to the susceptible state take place at a rate of 1.

Initially, each agent creates mutual links with eight unique neighbors chosen uniformly at random. To exercise the capability to dynamically track interagent dependencies, the agents randomly move within the topology by cutting the ties to their current neighbors and selecting a new set of eight random neighbors. The movement rate can be configured to depend on the overall number of infected agents, which represents the challenging case of a macro property changing when agents are newly infected or recovered. Given the proportion p of infected agents, an agent moves at a rate of $0.1 \times (1 - p)$. If the macro property is not taken into account, the movement rate is constant at 0.1.

5.2. Setup

The experiments using our CPU implementations were conducted on a system equipped with two 16-core Intel Xeon E5-2683v4 CPUs and 256GiB of RAM, running CentOS Linux 7.9.2009. Hyperthreading was disabled. All speedup values are given in relation to sequential runs on this system. The GPU implementation was evaluated on a system equipped with an AMD EPYC 7413 and an NVIDIA RTX A6000, running Debian 10 within a virtual machine restricted to two CPU cores and 32 GiB of RAM. We plot averages of three runs for each data point with 95% confidence intervals. All experiments from our previous conference publication[20] were carried out once again to account for changes in the window size adaptation and to extend the simulation duration to 1 unit of simulation time independently of the scenario size. Sequential simulation runs showed that the events in the simulation are fine-grained, with processing times per transition of 2 to $10 \mu s$ including all overheads. In the CPU-based parallel simulations, agents are assigned to threads by an ascending identifier. Since links among agents are created and changed uniformly at random, this is equivalent to a static random partitioning.

Table 1. Difference in overall number of transitions and final ratio of infected agents among the entire agent population with the CPU and GPU implementations. All 99% confidence intervals across 100 runs include 0, supporting the equivalence of the results.

#Agents	Diff. #Transitions	Diff. Inf. Ratio [10^{-4}]
2^{18}	-73.1 ± 214.2	-1.62 ± 4.12
2^{20}	235.6 ± 446.5	1.55 ± 1.87
2^{22}	79.5 ± 908.5	0.03 ± 1.07

5.3. Results

5.3.1. Verification. The correctness of the parallelized CPU implementation was verified by direct comparison of agent states and timestamps of scheduled transitions at the end of the simulation, which were observed to be identical between the sequential and parallel simulations. Since the CPU and GPU implementations rely on different random number generators, the GPU implementation is verified statistically. Table 1 compares the overall number of transitions and the percentage of agents in the infected state at termination for three model configurations. The table shows the average differences between runs of the CPU and GPU implementations across 100 runs per configuration with 99% confidence intervals. We observe that at all scenario sizes, the confidence interval includes 0, suggesting that aside from the random number generation, the implementations behave equivalently.

5.3.2. Simulator configuration. An essential parameter when executing GPU-based programs is the thread blocks size, which determines the grouping of GPU threads into blocks that are assigned non-preemptively to the GPU hardware. The optimal choice of thread block size depends non-trivially on the overall number of threads to be executed, the register usage of the code executed by each thread, the degree of divergence across code paths taken by the individual threads, as well as hardware properties.[56] As a basis for all further experiments, we thus determined optimal thread block sizes experimentally. In doing so, we differentiate between those GPU kernels that operate on *all* agents, and those that only operate on *active* agents, i.e., agents with an imminent transition (cf. Section 4.2). Figure 7 shows the number of committed events per second wall-clock time with different thread block sizes for the two types of kernels when executing simulations with

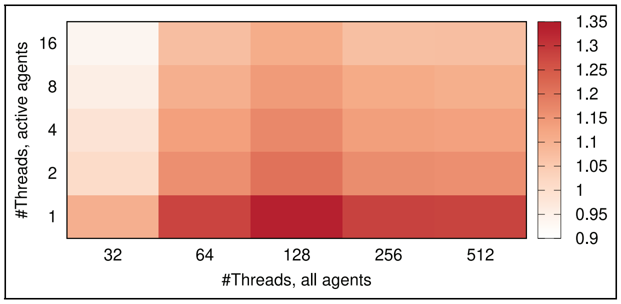

Figure 7. Heatmap of the GPU simulation performance as (10^6) committed transitions per second wall-clock time depending on the configured thread block size for kernels operating on all agents and only the active agents. Due to the divergent per-thread computations in the kernels handling active agents, best performance is observed using only one thread per block.

2^{20} agents. We observe that highest performance is achieved when using a thread block size of 128 for the kernels operating on all agents. The kernels operating only on the previously identified active agents performed best using the smallest possible thread block size of only 1. We ascribe this somewhat surprising result to the relatively complex code involved in an agent's transition. Separate threads will thus rarely follow identical code paths, eliminating most of the benefit of grouping threads within thread blocks. This is in contrast to the kernels operating on all agents such as the identification of imminent transitions, which follow largely the same code path for all agents. In accordance with these results, we set the thread block sizes to 128 and 1 in all subsequent experiments.

In our synchronization algorithm, the maximum progress per round depends on the initial window size τ_0. The choice of τ_0 involves a model-specific tradeoff between the opportunity for parallel processing on one hand, and the overhead of executing events that are subsequently rolled back on the other hand. To account for the simulation conditions at runtime, we periodically set τ_0 to a multiple w of the average effective window size τ observed at the end of the rounds during the previous period. As the window size can only decrease during a round, w should be set to a value above 1 to avoid a gradual decay of τ_0 toward 0. We observed only little dependence of the simulation performance on the adaptation period and thus configured a fixed period of 100 rounds. Figure 8 shows the event rate achieved by parallel execution of the modified susceptible-infected-recovered model using 32 threads, varying the window size factor w. The global counter of infected agents was disabled. First, we observe that the performance increases with higher agent count. Simulations using values for w of 2.5 and 5.0 typically achieve somewhat higher performance than those using

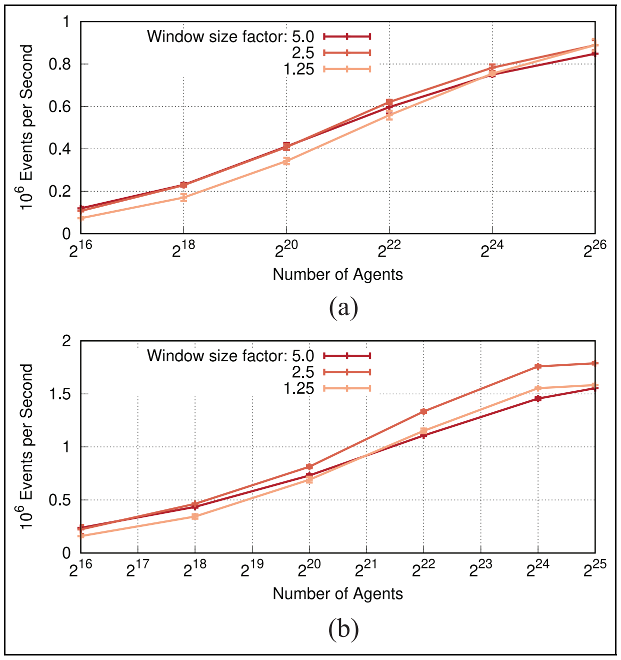

Figure 8. Committed events per second wall-clock time depending on the window size factor *w*, which determines the amount of optimism. We note that on both platforms, the simulation performance is quite robust to changes in *w*. (a) CPU implementation, 32 threads and (b) GPU implementation.

1.25. The difference diminishes with increasing agent counts. The choice of w strongly affects the rollback ratio, i.e., the number of rollbacks per committed transition. In a CPU-based parallel simulation of 2^{20} agents with $w = 1.25$, the ratio is only 0.28. With $w = 2.5$ and $w = 5.0$, the ratio increases to 1.57 and 4.27, i.e., the number of transitions that are rolled back exceeds the number of committed transitions. On the contrary, larger values of w still allow for more transitions to be committed in each round, decreasing the number of rounds to complete the simulation. When setting w to 1.25, 2.5, and 5.0, a total of 1,555,168 transitions were committed within 11,694, 9210, and 9190 rounds. In light of these results, we conclude that the event rate is reasonably robust to the window size factor, even though the number of rollbacks may differ substantially.

5.3.3. Scaling. Figure 9 shows speedup results for the CPU implementation when varying the number of threads, with the global counter of infected agents disabled. At 2^{20} agents, most parallel configurations start to outpace the sequential execution, with the exception of the configurations using two threads. With two threads, the overhead involved in executing events speculatively is only barely amortized at the largest agent count of 2^{26} agents. In

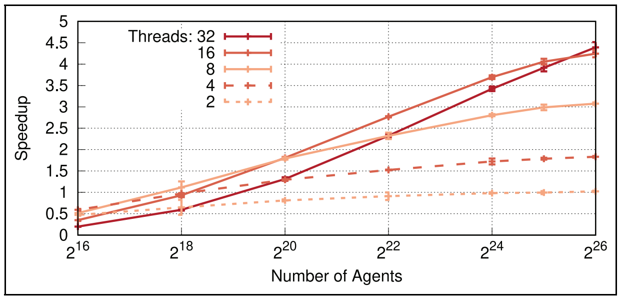

Figure 9. Speedup of the CPU implementation over sequential execution depending on the number of agents and threads. Performance increases are observed at 2^{18} agents and beyond. An increase in the number of cores is consistently only beneficial at sufficiently large numbers of agents.

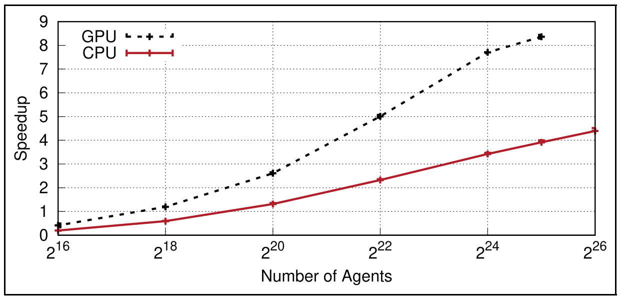

Figure 11. Speedup comparison between the CPU implementation using 32 threads and the GPU implementation over the sequential CPU-based reference. On the GPU, the agent count was limited to 2^{25} by the graphics memory capacity.

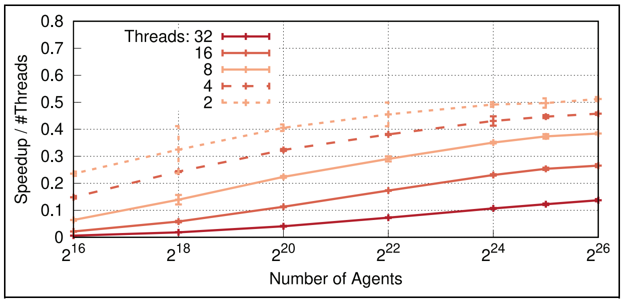

Figure 10. Efficiency of the CPU implementation. As expected, each doubling of the thread count decreases the efficiency, even though the speedup of sufficiently large scenarios increases with larger numbers of 9 threads.

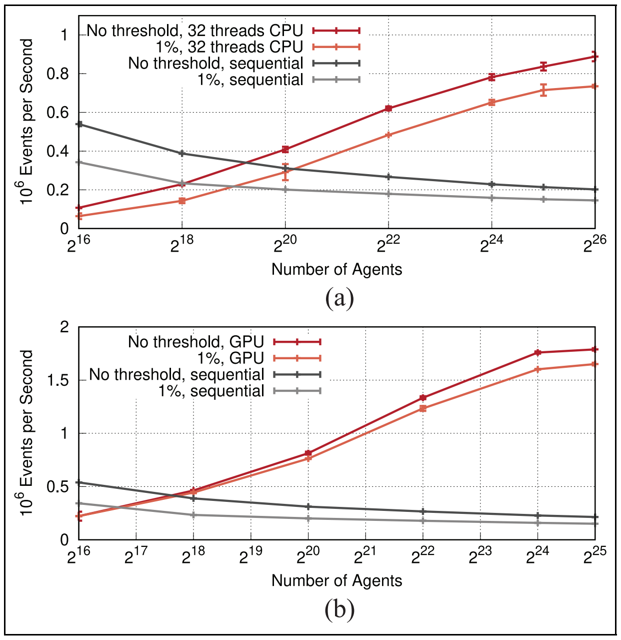

Figure 12. Committed transitions per second wall-clock time with and without a global counter updated whenever the number of infected agents has changed by 1% of the total agents. While the notification overhead reduces the absolute processing rates, the speedup over the sequential case is largely maintained. (a) CPU implementation and (b) GPU implementation.

contrast, large amount of parallelism of the runs with 32 threads enables substantial performance gains at large agent counts, with a maximum speedup of 4.4 at 2^{26} agents. Between agent counts of 2^{22} and 2^{25}, best performance was achieved using 16 threads, which coincides with the number of physical cores on a single CPU socket of our test system. Figure 10 shows the resulting efficiency, i.e., the speedup divided by the number of threads. In accordance with expectations, an increase in the number of threads decreases the efficiency, whereas an increase in the number of agents leads to higher efficiency. Figure 11 compares the performance of the CPU and GPU implementations. Given the GPU memory capacity of 48 GiB, the scenario size was limited to a maximum of 2^{25} agents. We observe that the GPU implementation outperforms the CPU variant in all cases, even at the smallest considered agent counts. At the largest scenario size of 2^{25} agents, the speedup was 8.4 over the sequential CPU-based execution, and 2.1 over the CPU-based parallel variant. Overall, we conclude that our synchronization algorithm substantially accelerates simulations of large populations.

5.3.4. Macro property accesses. Figure 12 compares the number of committed transitions per second wall-clock time with the global counter disabled and enabled, i.e., with or without taking into account the macro property of the number of infected agents an individual's movement rate. When enabled, all the agents' movement rates are newly calculated every time the number of infected agents has increased or decreased by 1% of the overall agent count. As expected, the induced recalculation of rates and rescheduling of events decrease performance both in the

Table 2. Profiling results showing percentages of execution time spent on the different simulation steps. In spite of the substantial differences in the approaches and the resulting performance, similar percentages of the execution time are ultimately spent on the execution of transitions.

	Seq. CPU	Parallel CPU	GPU
Collect Trans.	65.2%	34.2%	49.1%
Execute Trans.	33.6%	43.1%	31.4%
Other	1.2%	22.7%	19.5%

GPU: graphics processing unit; CPU: central processing unit.

sequential, and parallel CPU-based and GPU-based execution. At 2^{25} agents, the rate of the CPU implementation is reduced from 214,000 to 151,000 committed transactions per second in the sequential case, and from 837,000 to 715,000 in the parallel case using 32 threads. Although the absolute performance is lower when considering the macro property, the speedup increases from 3.9 to 4.7. At 2^{26} agents, the speedup increases to the largest overall value observed on the CPU of 5.1. The results on the GPU with 2^{25} agents exhibit the same trends, with a reduction in the committed rate from 1,790,000 to 1,650,000 transitions per second and an increase in speedup from 8.4 to 10.9.

5.3.5. Profiling. We carried out profiling runs for simulations of 2^{25} agents, with the macro property disabled. The results were generated using Google's Gperftools (https://gperftools.github.io/) for the CPU variants, and NVIDIA NSight Systems (https://gperftools.github.io/gperftools/cpuprofile.html) for the GPU variant. The parallel CPU variant was executed using 16 threads.

Table 2 shows our profiling results. We observe that between 31.5% and 43.1% of the execution time is spent on the actual execution of transitions. Hence, although the algorithms and their implementations differ to a large degree, the relative overhead remains roughly the same.

In the sequential CPU variant, the execution time is dominated by the collection of imminent transitions, with only little overhead apart from this step. The main costs during execution of the transitions are incurred by the notification of neighbors and the scheduling of new transitions (22.7% and 14.3% of the overall execution time, respectively).

The parallel CPU variant reduces the cost for collecting imminent transitions to only 34.2%. However, as summarized as ''other,'' another 22.7% of the execution time are spent on the additional steps required as part of the algorithm presented in Section 3.2. The execution of transitions itself is dominated by the agent access wrapper (27.5% of the overall execution time), which acquires and releases mutexes and updates the event horizon.

Finally, in the GPU implementation of the lock-free algorithm, which performs best out of the three variants, nearly half of the execution time is spent collecting imminent transitions. The overhead of this step could be further reduced by considering multiple agents in aggregate and adjusting the group size dynamically, as shown in our previous work.[57] The remaining 19.5% of execution time summarized as ''other'' is expended on the initialization of the simulation rounds (9.5%), the actuation of accesses (7.6%), and on rollbacks (2.5%).

6. Conclusion

We presented synchronous speculative synchronization algorithms targeting models of tightly coupled agents in continuous time. The algorithms restrict the state history required for rollbacks to only a single entry per agent and avoid cascading rollbacks entirely. Our base algorithm enables direct attribute accesses across agents without mediation through events or messages. A second algorithm relies on messages to avoid the need for locking and exposes fine-grained parallelism for GPU-based execution. On the example of an extended susceptible-infected-recovered agent-based model that emphasizes the challenging characteristics of the considered model class, we showed that our synchronization algorithms can accelerate simulations of 6.7×10^7 agents by a factor of up to 5.1 using 32 CPU cores, and simulations of 3.6×10^7 agents by a factor of 10.9 using a data center GPU.

Our experiments focused on agent-based models in which event times are determined based on dynamically updated transition rates, and in which state updates at one agent entail instantaneous accesses to other agents' attributes. However, the mechanisms for efficient direct access among simulation objects are general. Thus, a promising direction for future work lies in exploring the benefits of our synchronous speculative algorithms when executing discrete-event models of other systems of tightly coupled entities.

Funding

The author(s) disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: Financial support was provided by the Deutsche Forschungsgemeinschaft (DFG) research grant (grant no. 25856074), UH-66/15-2 (MoSiLLDe).

ORCID iD

Philipp Andelfinger https://orcid.org/0000-0002-0211-7136

References

1. Gilbert N and Troitzsch KG. *Simulation for the social scientist*. Maidenhead: McGraw Hill Education, 2005.
2. Bazzan AL and Klügl F. A review on agent-based technology for traffic and transportation. *Knowl Eng Rev* 2014; 29: 375–403.

3. An G, Mi Q, Dutta-Moscato J, et al. Agent-based models in translational systems biology. *WIRES Syst Biol Med* 2009; 1: 159–171.
4. Willekens F. Continuous-time microsimulation in longitudinal analysis. In: Zaidi A, Harding A and Williamson P (eds) *New frontiers in microsimulation modelling*. Farnham: Ashgate Publishing, 2009, pp. 413–436.
5. Yang M, Andelfinger P, Cai W, et al. 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*, Rome, 23–25 May 2018, pp. 129–132. New York: ACM.
6. Köster T, Giabbanelli PJ and Uhrmacher A. Performance and soundness of simulation: a case study based on a cellular automaton for in-body spread of HIV. In: *Proceedings of the 2020 winter simulation conference (WSC)*, Orlando, FL, 14–18 December 2020, pp. 2281–2292. New York: IEEE.
7. Fujimoto RM. Parallel and distributed simulation systems. In: *Proceedings of the 2001 winter simulation conference*, Arlington, VA, 9–12 December 2001, vol. 1, pp. 147–157. New York: IEEE.
8. Jafer S, Liu Q and Wainer G. Synchronization methods in parallel and distributed discrete-event simulation. *Simul Model Pract Th* 2013; 30: 54–73.
9. Noble J, Silverman E, Bijak J, et al. Linked lives: the utility of an agent-based approach to modeling partnership and household formation in the context of social care. In: *Proceedings of the 2012 winter simulation conference (WSC)*, Berlin, 9–1 December 2012, pp. 1–12. New York: IEEE.
10. North MJ, Collier NT, Ozik J, et al. Complex adaptive systems modeling with Repast Simphony. *Complex Adapt Syst Model* 2013; 1: 3.
11. Tisue S and Wilensky U. NetLogo: design and implementation of a multi-agent modeling environment. In: *Proceedings of the agent conference on social dynamics: interaction, reflexivity and emergence (updated)*, Chicago, IL, 7–9 October 2004, pp. 16–21. Cham, Switzerland: Springer.
12. Masad D and Kazil JL. Mesa: an agent-based modeling framework. In: *Proceedings of the 14th Python in science conference (SciPy'2015)*, Austin, TX, 6–12 July 2015, pp. 53–60. https://portal.issn.org/resource/issn/2575-9752
13. Railsback SF and Grimm V. *Agent-based and individual-based modeling: a practical introduction*. Princeton, NJ: Princeton University Press, 2019.
14. O'Sullivan D and Haklay M. Agent-based models and individualism: is the world agent-based? *Environ Plann A* 2000; 32: 1409–1425.
15. Ferber J and Müller JP. Influences and reaction: a model of situated multiagent systems. In: *Proceedings of the 2nd international conference on multi-agent systems (ICMAS'96)*, Kyoto, Japan, 10–13 December 1996, pp. 72–79. Palo Alto, CA: AAAI.
16. Michel F. The IRM4S model: the influence/reaction principle for multiagent based simulation. In: *Proceedings of the 6th international joint conference on autonomous agents and multiagent systems*, Honolulu, HI, 14–18 May 2007, pp. 1–3. New York: ACM.
17. Warnke T, Reinhardt O, Klabunde A, et al. Modelling and simulating decision processes of linked lives: an approach based on concurrent processes and stochastic race. *Popul Stud* 2017; 71: 69–83.
18. Reinhardt O, Warnke T and Uhrmacher AM. A language for agent-based discrete-event modeling and simulation of linked lives. *ACM T Model Comput S* 2022; 32: 1–26.
19. Keeling MJ and Eames KTD. Networks and epidemic models. *J R Soc Interface* 2005; 2: 295–307.
20. Andelfinger P and Uhrmacher A. Optimistic parallel simulation of tightly coupled agents in continuous time. In: *Proceedings of the 2021 IEEE/ACM 25th international symposium on distributed simulation and real time applications (DS-RT)*, Valencia, 27–29 September 2021, pp. 1–9. New York: IEEE.
21. Steinman J. SPEEDES: synchronous parallel environment for emulation and discrete-event simulation. In: *Proceedings of the SCS western multi-conference on advances in parallel and discrete simulation*, San Diego, CA, 16–19 May 1993, vol. 23, pp. 1111–1115. New York: ACM.
22. Allen LJ and Lahodny GE Jr. Extinction thresholds in deterministic and stochastic epidemic models. *J Biol Dynam* 2012; 6: 590–611.
23. Ross SM, Kelly JJ, Sullivan RJ, et al. *Stochastic processes*, vol. 2. New York: Wiley, 1996.
24. Gillespie DT. A general method for numerically simulating the stochastic time evolution of coupled chemical reactions. *J Comput Phys* 1976; 22: 403–434.
25. Gibson MA and Bruck J. Efficient exact stochastic simulation of chemical systems with many species and many channels. *J Phys Chem A* 2000; 104: 1876–1889.
26. Squazzoni F. The micro-macro link in social simulation. *Sociologica* 2008; 2: 1–26.
27. Klabunde A, Zinn S, Willekens F, et al. Multistate modelling extended by behavioural rules: an application to migration. *Popul Stud* 2017; 71: 51–67.
28. Jefferson D and Sowizral H. Fast concurrent simulation using the time warp mechanism. Part I: local control. Technical report, RAND Corporation, Santa Monica, CA, December 1982.
29. Jeschke M, Park A, Ewald R, et al. Parallel and distributed spatial simulation of chemical reactions. In: *Proceedings of the 2008 22nd workshop on principles of advanced and distributed simulation*, Roma, 3–6 June 2008, pp. 51–59. New York: IEEE.
30. Dematté L and Mazza T. On parallel stochastic simulation of diffusive systems. In: *Proceedings of the international conference on computational methods in systems biology (CMSB'2008)*, Rostock, 12–15 October 2008, pp. 191–210. Berlin; Heidelberg: Springer.
31. Wang B, Hou B, Xing F, et al. Abstract Next Subvolume Method: a logical process-based approach for spatial stochastic simulation of chemical reactions. *Comput Biol Chem* 2011; 35: 193–198.
32. Pawlaszczyk D and Timm IJ. A hybrid time management approach to agent-based simulation. In: *Proceedings of the annual conference on artificial intelligence*, Bremen, 14–17 June 2006, pp. 374–388. Berlin; Heidelberg: Springer.

33. Rao DM. Efficient parallel simulation of spatially-explicit agent-based epidemiological models. *J Parallel Distr Com* 2016; 93: 102–119.
34. Andelfinger P, Piccione A, Pellegrini A, et al. Comparing speculative synchronization algorithms for continuous-time agent-based simulations. In: *Proceedings of the 2022 IEEE/ACM 26th international symposium on distributed simulation and real time applications (DS-RT)*, Alès, 26–28 September 2022, pp. 57–66. New York: IEEE.
35. Ghosh K and Fujimoto RM. Parallel discrete event simulation using space-time memory. Technical report, Georgia Institute of Technology, Atlanta, GA, 1994.
36. Chen L-L, Lu Y-S, Yao Y-P, et al. A well-balanced Time Warp system on multi-core environments. In: *Proceedings of the 2011 IEEE workshop on principles of advanced and distributed simulation*, Nice, 14–17 June 2011, pp. 1–9. New York: IEEE.
37. Marziale N, Nobilia F, Pellegrini A, et al. Granular time warp objects. In: *Proceedings of the 2016 ACM SIGSIM conference on principles of advanced discrete simulation*, Banff, AB, Canada, 15–18 May 2016, pp. 57–68. New York: ACM.
38. Pellegrini A, Vitali R, Peluso S, et al. Transparent and efficient shared-state management for optimistic simulations on multi-core machines. In: *Proceedings of the 2012 IEEE 20th international symposium on modeling, analysis and simulation of computer and telecommunication systems*, Washington, DC, 7–9 August 2012, pp. 134–141. New York: IEEE.
39. Ianni M, Marotta R, Cingolani D, et al. The ultimate share-everything PDES system. In: *Proceedings of the 2018 ACM SIGSIM conference on principles of advanced discrete simulation*, Rome, 23–25 May 2018, pp. 73–84. New York: ACM.
40. Pellegrini A and Quaglia F. Cross-state events: a new approach to parallel discrete event simulation and its speculative runtime support. *J Parallel Distr Com* 2019; 132: 48–68.
41. Chen S, Hanai M, Hua Z, et al. Efficient direct agent interaction in optimistic distributed multi-agent-system simulations. In: *Proceedings of the 2020 ACM SIGSIM conference on principles of advanced discrete simulation*, Miami, FL, 15–17 June 2020, pp. 123–128. New York: ACM.
42. Dickens PM, Nicol DM, Reynolds PF Jr, et al. Analysis of bounded time warp and comparison with YAWNS. *ACM T Model Comput S* 1996; 6: 297–320.
43. Ferscha A. Probabilistic adaptive direct optimism control in time warp. In: *Proceedings of the 9th workshop on parallel and distributed simulation*, Lake Placid, NY, 14–16 June 1995, pp. 120–129. New York: IEEE.
44. Rajaei H, Ayani R and Thorelli LE. The local Time Warp approach to parallel simulation. In: *Proceedings of the 7th workshop on parallel and distributed simulation*, San Diego, CA, 16–19 May 1993, pp. 119–126. New York: ACM.
45. Wang J and Tropper C. Optimizing time warp simulation with reinforcement learning techniques. In: *Proceedings of the 2007 winter simulation conference*, Washington, DC, 9–12 December 2007, pp. 577–584. New York: IEEE.
46. Ewald R, Chen D, Theodoropoulos GK, et al. Performance analysis of shared data access algorithms for distributed simulation of multi-agent systems. In: *Proceedings of the 20th workshop on principles of advanced and distributed simulation (PADS'06)*, Singapore, 24–26 May 2006, pp. 29–36. New York: IEEE.
47. Chen D, Ewald R, Theodoropoulos GK, et al. Data access in distributed simulations of multi-agent systems. *J Syst Software* 2008; 81: 2345–2360.
48. Tan WJ, Andelfinger P, Cai W, et al. Multi-thread state update schemes for microscopic traffic simulation. In: *Proceedings of the 2020 winter simulation conference (WSC)*, Orlando, FL, 14–18 December 2020, pp. 182–193. New York: IEEE.
49. Tan WJ, Andelfinger P, Eckhoff D, et al. 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*, Suffolk, VA, 31 May–2 June 2021, pp. 57–68. New York: ACM.
50. Blackman D and Vigna S. Scrambled linear pseudorandom number generators, 2018, https://arxiv.org/abs/1805.01407
51. L'Ecuyer P and Simard R. TestU01: A C library for empirical testing of random number generators. *ACM T Math Software* 2007; 33: 22.
52. NVIDIA Corporation. NVIDIA CUDA C programming guide (version 11.6.0), 2022, https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
53. Steinman JS. Discrete-event simulation and the event horizon. *ACM SIGSIM Simulat Digest* 1994; 24: 39–49.
54. Marsaglia G. Xorshift RNGs. *J Stat Softw* 2003; 8: 1–6.
55. Macal CM. To agent-based simulation from system dynamics. In: *Proceedings of the 2010 winter simulation conference*, Baltimore, MD, 5–8 December 2010, pp. 371–382. New York: IEEE.
56. Torres Y, Gonzalez-Escribano A and Llanos DR. UBench: exposing the impact of CUDA block geometry in terms of performance. *J Supercomput* 2013; 65: 1150–1163.
57. Andelfinger P and Hartenstein H. Exploiting the parallelism of large-scale application-layer networks by adaptive GPU-based simulation. In: *Proceedings of the winter simulation conference 2014*, Savannah, GA, 7–10 December 2014, pp. 3471–3482. New York: IEEE.

Author biographies

Philipp Andelfinger is a postdoctoral researcher at the Institute for Visual and Analytic Computing of the University of Rostock. He received his PhD in Computer Science from the Karlsruhe Institute of Technology in 2016.

Adelinde M Uhrmacher is professor at the Institute for Visual and Analytic Computing of the University of Rostock and head of the Modeling and Simulation Group. She holds a PhD in Computer Science from the University of Koblenz and a Habilitation in Computer Science from the University of Ulm.