Proceedings of the 2022 Winter Simulation Conference
B. Feng, G. Pedrielli, Y. Peng, S. Shashaani, E. Song, C.G. Corlu, L.H. Lee, E.P. Chew, T. Roeder, and P. Lendermann, eds.

ADVANCED TUTORIAL: PARALLEL AND DISTRIBUTED METHODS FOR SCALABLE DISCRETE SIMULATION

Philipp Andelfinger

Institute for Visual and Analytic Computing
University of Rostock
Albert-Einstein-Straße 22
Rostock 18059, GERMANY

Wentong Cai

School of Computer Science and Engineering
Nanyang Technological University
50 Nanyang Avenue
Singapore 639798, SINGAPORE

ABSTRACT

Discrete simulation is an indispensable approach to investigate systems whose complexity prohibits analytical modeling and for which real-world experimentation is costly or dangerous. To keep pace with system models of increasing levels of detail and scope, a simulator's ability to make full use of the available hardware, typically through parallel and distributed simulation, is a vital concern. Building on well-studied synchronization algorithms, the field's focus has shifted towards aspects such as the avoidance of redundant computations in ensemble studies and the exploitation of heterogeneous hardware platforms. In this tutorial, we describe the fundamental notions of parallel and distributed simulation and summarize the main classes of synchronization algorithms as well their use when applied under the constraints of the domains of transportation and spiking neural networks. Current research directions and challenges are discussed in light of the tension between efficiency through specialization and wide applicability through generalization.

1 INTRODUCTION

The behavior of many systems studied in science and engineering can be described in terms of sequences of instantaneous state changes over time. A discrete simulation imitates such a system computationally by iteratively updating state variables stored in memory along a virtual timeline. In *time-driven* simulations, virtual time advances by a global time step at which all state variables can be updated. In contrast, *event-driven* simulations allow updates, referred to as events, to be scheduled with arbitrary *timestamps* and targeting only specific simulated entities.

The steady increase in computational power available to researchers and engineers has allowed simulations to support the design, evaluation, and optimization of systems at ever-increasing scales and levels of detail. Parallel and distributed simulation (Fujimoto et al. 2017) emerged as a family of methods to divide the execution of an individual simulation run across processors interconnected via shared memory or across a network. This approach contrasts with the straightforward parallelization of ensemble studies, where full simulation runs can be assigned to different processors without the need for inter-processor communication and synchronization. Within the field of parallel and distributed computing, the parallelization of a discrete simulation run poses unique challenges, since simulation is frequently applied to model systems whose runtime behavior is highly dynamic and difficult to predict. Hence, the computational workload and the data dependencies defined by the state updates can only be observed in full while the simulation is executed. A wealth of literature on synchronization algorithms guarantees adherence to the dynamic data dependencies, typically relying on the temporal ordering of updates as defined by event timestamps.

Since the breakdown of Dennard scaling from around 2005 onwards, the largest performance gains in CPU architectures are achieved by increasing the number of parallel cores rather than by increasing the clock frequency of individual cores. Around the same time, graphics processing units (GPUs) evolved into

general-purpose compute devices capable of massively parallel computations across thousands of processing elements. In light of these developments, parallelization has become an inevitable requirement to keep pace with the computational demands posed by increasingly large and complex simulation models.

Both the choice of suitable parallelization techniques and the achievable performance benefits depend strongly on properties of the simulation model and the hardware platform. The complex interaction between aspects such as the workload induced by the model behavior, the synchronization algorithm, and the hardware-specific cost for inter-processor communication has motivated a wealth of literature on performance analysis and prediction for parallel simulations (Nicol 1993; Ferscha et al. 1997; Liu et al. 1999; Perumalla et al. 2005). However, domain-agnostic frameworks that achieve high-performance parallel execution without the need for manual model-specific optimizations remain elusive. Instead, domain-specific simulation frameworks provide optimized implementations tailored to the efficient parallel execution of models in specific areas such as multi-agent systems (Richmond and Chimeh 2017; Piccione et al. 2019; Cosenza et al. 2021) or spiking neural networks (Tikidji-Hamburyan et al. 2017).

In what follows, we give an overview of the state of the art in methods for scalable simulation. Compared to previous tutorials and surveys in this context (Fujimoto 2015; Fujimoto 2016), we more directly contrast classical algorithms using a shared notation and put particular emphasis on recent application domains. Section 2 introduces the main categories of fundamental parallel and distributed simulation algorithms. Section 3 illustrates considerations required for efficient parallelization under model-specific constraints on the examples of applications in transportation and spiking neural networks. In Section 4, we discuss current research directions on the use of parallel simulation techniques to support overall simulation studies and the utilization of heterogeneous hardware environments. Section 5 summarizes and concludes the paper.

2 FUNDAMENTAL ALGORITHMS

In a discrete-event simulation, dependencies pertain to *events*, i.e., changes of the state variables. Naturally, the result of a sequence of events operating on a state variable can depend on the order in which the events are executed. Hence, to generate the same results as a sequential reference simulation, a parallel simulation must maintain this order. Suppose we have two events e_0, e_1 operating on the same state variable, as well their timestamps $T(e_0), T(e_1)$ with $T(e_0) < T(e_1)$. Then, e_0 must be executed prior to e_1. This is trivially guaranteed in a sequential simulation, which iteratively executes the event with the earliest timestamp. However, in a parallel or distributed simulation, events are dynamically exchanged among processors, creating the hazard of receiving an event with an earlier timestamp than a previously executed event. Simply executing such a *straggler event* may invalidate the simulation results. In parallel and distributed simulations, the state variables are partitioned to form *logical processes* (LPs), each of which may be handled by a separate processor. In this setting, a correct ordering is achieved by satisfying the *local causality constraint*: if events pertaining to each individual LP are executed in non-decreasing timestamp order, causality is satisfied globally.

There are two main approaches to satisfy the local causality constraint. *Conservative* synchronization algorithms rule out straggler events entirely based on the processors' current progress in virtual time and on minimum delays in virtual time between events and their effects on remote LPs. *Optimistic* (also referred to as *speculative*) algorithms allow straggler events to occur. When a straggler event is received, any invalidated computations are rolled back. Rollbacks are enabled either by storing previous simulation states in memory or by reversing events computationally.

A second categorization of synchronization algorithms distinguishes the coordination among processors. In *synchronous* algorithms, all processors jointly alternate between the execution of events and synchronization, largely coupling the progress in virtual time among the processors. In contrast, *asynchronous* algorithms afford processors a larger amount of freedom to independently advance in virtual time.

In the following, we detail the key ideas and basic principles of the fundamental synchronization algorithms for parallel and distributed simulation. Table 1 lists well-known algorithms from each of the categories discussed above.

Table 1: Fundamental synchronization algorithms for parallel and distributed simulation.

	Conservative	**Optimistic**
Synchronous	Bounded Lag (Lubachevsky 1989)	Breathing Time Buckets (Steinman 1991)
	YAWNS (Nicol 1993)	Breathing Time Warp (Steinman 1993)
Asynchronous	CMB (Bryant 1977; Chandy and Misra 1981)	Time Warp (Jefferson et al. 1987)
	Time-of-Next-Event (Grošelj and Tropper 1991)	Moving Time Window (Sokol 1988)

2.1 Conservative Synchronization

In conservative synchronization, violations of the local causality constraint are ruled out in advance by executing an event only if it can be guaranteed that no events with smaller timestamp can be received. Hence, the essence of conservative synchronization is to determine periods in virtual time for which the arrival of events from other LPs can be ruled out. An LP may execute events in this period, referred to as *safe* events. The basis for identifying such periods is the concept of *lookahead*, which is an LP's capability to determine the timestamps of future events. The availability of lookahead and the amount of virtual time covered depend on the simulation model and may fluctuate over the course of a simulation. A conservative synchronization algorithm orchestrates the computation of lookahead information, its dissemination among the LPs, and the execution of safe events.

2.1.1 Extracting Lookahead

As conservative synchronization restricts LPs to executing only safe events, the effective parallelism and thus the speedup over a sequential execution depends strongly on the lookahead. Quantitatively, lookahead can be defined as follows: if at virtual time t, an LP can guarantee that the earliest event targeting a remote LP has a timestamp of $t+l$, then the LP's lookahead is l. Importantly, this consideration includes any local events that may be scheduled as a consequence of events received from other LPs. To extract sufficiently large amounts of lookahead, model properties are assessed, with the required degree of sophistication required depending on the model class. An example is given by the model of a packet-switched computer network, where an LP may represent the network nodes within a certain geographical region and an event typically represents a packet arrival. Here, a fundamental lower bound on the delay between the creation of a packet at one LP and its arrival at another LP can be computed by making the conservative assumption of speed-of-light propagation of network packets. Depending on the specific simulation model, additional delays may be introduced depending on the sending node's state such as its current load or protocol state (Peschlow et al. 2009). By taking the node state into account, the amount of lookahead may be increased at the cost of additional modeling effort and complexity. Further improvements may be achieved by taking a graph-driven view of the possible interactions among the simulated entities (Lubachevsky 1989; Meyer and Bagrodia 1999; Deelman et al. 2001). Following the possible paths and delays of future events, the minimum timestamp of any future event that may affect another LP can be computed.

Stochastic model elements may decide on the creation of new events and their timestamps using a pseudo-random number generator. In the absence of other model-specific bounds, predicting future event timestamps requires presampling of the pseudo-random number stream (Loper and Fujimoto 2000), which can be viewed as a partial execution of projected future events in advance solely for lookahead extraction.

Since sophisticated lookahead extraction procedures may involve substantial amounts of precomputation, a suitable approach should be chosen in light of the achieved increase in parallel simulation performance.

2.1.2 Synchronous Algorithms

Synchronous algorithms proceed in a series of rounds within each of which the LPs execute events up to a bound in virtual time before synchronizing at a global barrier. The time window covered within a round is determined based on the LPs' lookahead and may be the same across all LPs or determined for each LP specifically. The Bounded Lag algorithm (Lubachevsky 1989) and YAWNS (yet another windowing

Algorithm 1: Synchronous conservative simulation using the YAWNS algorithm.

```
do
    min_ts ← compute_global_min_timestamp()
    foreach lp_i in parallel do
        curr_event_i ← q_i.peek()    // consider earliest event in lp_i's event queue
        while curr_event_i.ts < min(min_ts + lookahead, end_time) do
            execute curr_event_i
            q_i.pop()    // remove earliest event
            curr_event_i ← q_i.peek()
        send generated events
while min_ts < end_time
```

network simulator) (Nicol 1993) are quintessential representatives of this category of algorithms. A sketch of the YAWNS algorithm assuming identical lookahead for all LPs is given as pseudo code in Algorithm 1.

2.1.3 Asynchronous Algorithms

In asynchronous algorithms, LPs alternate between event processing and idle periods individually. Rather than communicating lookahead information at global synchronization points, asynchronous algorithms are based on message passing via first-in-first-out (FIFO) queues among LP pairs.

We briefly sketch the Chandy-Misra-Bryant (CMB) algorithm (Bryant 1977; Chandy and Misra 1981), arguably the most widely known asynchronous conservative algorithm. In the CMB algorithm, events sent across a queue are assumed to follow timestamp order. Now, if all of an LP's incoming FIFO queues contain at least one message, the minimum timestamp of all messages in the queues is a lower bound on the timestamp of any event received by a remote LP. This information is sufficient to safely execute events up to the lower bound, but allows for cycles of LPs waiting for messages to occur, leading to deadlocks.

The CMB algorithm avoids deadlocks using so-called *null messages* through which LPs exchange lookahead information as lower bounds on the timestamp of any future event sent to a given remote LP. CMB variants differ in the policy according to which null messages are sent, either actively using some local policy or only when requested by another LP (Su and Seitz 1988). Pseudo code for the basic execution scheme of an LP using the CMB algorithm is given in Algorithm 2, based on Cai and Turner (1995).

The Time-of-Next-Event algorithm (Grošelj and Tropper 1991) aims to avoid the potential of CMB for generating large numbers of null messages by an alternative approach to deadlock avoidance. In their algorithm, LPs are unblocked through shortest path computations across sequences of empty FIFO queues in order to determine lower bounds on events that may be sent across each empty queue.

Algorithm 2: Asynchronous conservative simulation using the CMB algorithm.

```
foreach lp_i in parallel do
    EIT_i ← ∞    // initialize earliest possible timestamp of an incoming event
    do
        extract messages with timestamps < EIT_i from input queues
        EIT_i ← minimum timestamp in any input queue
        EOT_i ← min(EIT_i, curr_event_i.ts) + lookahead    // earliest possible timestamp of an outgoing event
        execute extracted events, store generated events in event pool
        send events from pool with timestamps < EOT_i
        send null message with timestamp EOT_i on any output queue without new events
        now_i ← min(EIT_i, EOT_i)
    while now_i < end_time
```

2.2 Optimistic Synchronization

In optimistic synchronization, temporary violations of the local causality constraint are permitted, but subsequently detected and corrected using a rollback mechanism. Much of the difficulty of resolving detected causality violations is caused by the potential for transitive errors, since an event that has been executed in error may have created new events targeting other LPs, which must be canceled when the original event is rolled back. Thus, optimistic synchronization algorithms must either rule out the potential for transitive errors entirely or provide a mechanism to eliminate them as part of the rollback procedure.

To reduce the frequency of causality violations, the degree of optimism may be restricted statically or dynamically (Reiher et al. 1989; Turner and Xu 1991; Palaniswamy and Wilsey 1994; Wang and Trropper 2007), enabling tradeoffs between parallelism and rollback costs.

2.2.1 Enabling Rollbacks

A rollback is required when a straggler event is encountered, i.e., an LP that is currently at virtual time t_1 receives an event with virtual time $t_0 < t_1$. To remedy this situation, the LP rolls back *at least* to t_0, after which regular event processing may resume starting with the newly received event. The previous LP state may be restored either purely based on snapshots stored in memory, referred to as state saving, or using reverse computation (Carothers et al. 1999). In the former case, each LP regularly creates a snapshot of its local simulation state. A rollback then involves resetting the simulation state to the latest snapshot prior to the timestamp of the received straggler event. State saving mechanisms differ in the frequency of snapshot creation and may store snapshots either in full or as deltas over previous snapshots (Ronngren et al. 1996). Rollbacks based on reverse computation employ *reverse event handlers*, which implement the inverse of the event handlers called during regular event processing. By executing the reverse handlers of all events executed at or after t_0 in reversed order, the LP state prior to the straggler event's timestamp is restored. However, many mathematical and logical operations are not bijections and thus not uniquely reversible without further information. Thus, in practice, forward event handlers are extended with facilities for storing the disambiguation information required for a subsequent reversal (Ronngren et al. 1996; West and Panesar 1996; Carothers et al. 1999).

2.2.2 Synchronous Algorithms

Given a mechanism for local rollbacks at an LP, the key differentiating characteristic among optimistic synchronization algorithms is the mechanism used to prevent or correct transitive causality violations.

The synchronous algorithm Breathing Time Buckets (Steinman 1991) (cf. Alg. 3) prevents the occurrence of transitive violations altogether. In each round, the LPs first execute local events up to an agreed-upon

Algorithm 3: Synchronous optimistic simulation using the Breathing Time Buckets algorithm.

```
do
    min_ts ← compute_global_min_timestamp()
    foreach lp_i in parallel do
        curr_event_i ← q_i.peek()    // consider earliest event in lp_i's event queue
        while curr_event_i.ts < min(min_ts + τ_0, end_time) do
            execute curr_event_i
            q_i.pop()    // remove earliest event
            curr_event_i ← q_i.peek()
    t_x ← global minimum timestamp of a newly generated event targeting a remote LP
    foreach lp_i in parallel do
        roll back events with timestamps ≥ t_x
        send remaining newly generated events
while min_ts < end_time
```

point in virtual time at which they enter a global barrier. Since the choice of the bound is not reliant on lookahead, any of the executed events may later be invalidated by a straggler event. Newly generated events targeting remote LPs are held at the generating LP. Once all LPs have reached the barrier, the earliest timestamp t_x of any event to be exchanged among LPs is determined using a global reduction. Earlier events are guaranteed not to be invalidated later, whereas executed events with timestamps beyond t_x are rolled back. This algorithm guarantees that once an event e is received from a remote LP, the event that generated e will never be rolled back, i.e., transitive errors are ruled out. The Breathing Time Warp algorithm (Steinman 1993) combines the principles of Breathing Time Buckets with the treatment of causality violations using a synchronous variant of the Time Warp algorithm described next.

2.2.3 Asynchronous Algorithms

Time Warp (Jefferson et al. 1987) (cf. Alg. 4) is an asynchronous algorithm that does not avoid transitive errors but instead provides a mechanism for cascading rollbacks.

When an LP receives a straggler event, the LP rolls back previously executed events to restore its state at a time prior to the straggler's timestamp. If executed events past the straggler's timestamp have generated new events, they are eliminated by sending so-called antimesssages. Now, the LP resumes regular event execution starting with the straggler event.

On reception of an antimessage, there are two cases. If the receiving LP has not processed the event corresponding to the antimessage, the original event is discarded. If the LP has already processed the event, the LP rolls back to a state prior to the timestamp of the antimessage.

In Time Warp, events, states, and antimessages remain in memory until it is guaranteed that their timestamps cannot be reached by any upcoming rollback. In the absence of lookahead, this property is satisfied for all events with timestamps less than the earliest unexecuted event in the simulation, the *global virtual time* (GVT). The LPs periodically perform GVT computations (Samadi 1985; Mattern 1993) to limit the memory consumption of the lists. Importantly, in distributed simulations, the GVT computation must account for *transient* messages, i.e., events that have been sent but not yet received.

The Moving Time Window algorithm (Sokol 1988) operates asynchronously as does Time Warp, but restricts the LPs' progress earlier than $GVT + w$, with w being a tuning parameter whose optimal value depends on the model. By tuning w, the aggressiveness of the optimistic execution can be varied to balance opportunities for parallel execution with the frequency of rollbacks.

Algorithm 4: Asynchronous optimistic simulation using the Time Warp algorithm.

```
foreach lp_i in parallel do
    now_i ← −∞    // initialize lp_i's simulation time
    foreach event from event list with now_i < event.ts < end_time do
        now_i ← event.ts
        execute event, send generated events
    foreach received message do
        if message.is_antimessage then
            if corresponding event has been executed already then
                roll back to state before message.ts, send antimessages
            remove event from event list
        else if message.is_event then
            if message.ts < now_i then
                roll back to state before message.ts, send antimessages
            insert event in event list
```

3 APPLICATIONS

We illustrate the use of parallel and distributed methods on the example of transportation and spiking neural networks, where problem scales beyond the capabilities of sequential simulations are routinely encountered.

3.1 Transportation

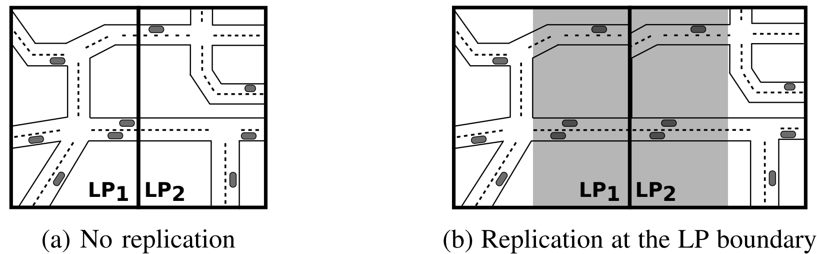
Road traffic simulations (Barceló 2010) are commonly used in the design, evaluation, and optimization of transportation systems. We focus on *microscopic* models, in which vehicles and other entities such as traffic signals are represented individually, enabling detailed simulations of heterogeneous behaviors and interactions.

Microscopic traffic models typically consider traffic participants in the form of so-called driver-vehicle-units (DVUs), which jointly model a driver's intention and its realization through the vehicle. Given an origin-destination pair and a route choice, a DVU's short-term movement is usually defined by two models. The longitudinal movement is defined by a *car following model*, which determines the DVU's new acceleration by the distance and velocity of the DVU in front. The lateral movement is defined by a *lane changing model*, which considers the positions and velocities of nearby DVUs to determine whether a lane change is carried out. Most car following models are formulated as ordinary differential equations with respect to continuous time and space. Coupling among the equations is introduced by the leader-follower relationships among the vehicles in a road network. Traffic simulators usually solve car-following models by step-wise numerical integration with a fixed step size (Treiber and Kanagaraj 2015). Most lane changing models, on the other hand, are inherently discontinuous in nature: when invoked, a lane change is triggered and completed either instantaneously or not at all. Further discontinuous portions of microscopic traffic models include the instantaneous changes of traffic light phases and special events such as road closures.

The detailed representation of individual traffic entities makes microscopic traffic simulations computationally demanding. Simulation studies often restrict their scope to small numbers of junctions with stochastically modeled inflows. Parallel and distributed simulation holds the promise of enabling the consideration of scenarios spanning large urban areas at full detail. The literature has also proposed the use of traffic simulation for real-time decision making (Pell et al. 2017). Based on dynamic data collected from sensors, *what-if* simulations predict the effects of interventions such as changes to traffic light controls to reduce congestion. To permit timely decisions, such simulations must proceed faster than wall-clock time.

At first appearance, microscopic traffic simulation seems to readily lend itself to parallelization. Firstly, interactions are highly localized since DVUs react only to nearby entities, suggesting a simple partitioning according to the topology of the road network. When representing the road network as a graph with vertices representing junctions and weighted edges representing roads and their expected traffic load, graph partitioning tools such as METIS (Karypis and Kumar 1997) are able to determine a partitioning with low edge cut and low workload imbalance. However, it has been shown that to achieve high performance, it is far more important to reduce the synchronization overhead by minimizing each LP's number of neighbors (Xu, Cai, Eckhoff, Nair, and Knoll 2017). In addition, traffic patterns can vary severely over the course of a simulation. For instance, commuter traffic in the mornings and evenings may be reflected by high workload in residential areas not observed throughout the day. Dynamic load balancing schemes have been proposed to adapt to the changing traffic patterns based on vehicle counts per LP (Zhang et al. 2007; Sun et al. 2012) or using the LPs' hardware utilization (Igbe 2010). Of course, the benefit of load balancing must be weighed against the overhead induced by frequent migration of vehicles among LPs (Igbe 2010).

A second desirable property for parallel execution is given by the physical constraints on the DVUs' velocities, which permit the computation of lower bounds on the time until a vehicle reaches an LP boundary (Andelfinger et al. 2020), facilitating lookahead extraction. Unfortunately, if roads that cross LP boundaries experience high traffic loads, the amount of available lookahead may be small, which tightly couples the progress of adjacent LPs. Several approaches have been proposed to tackle this problem. Firstly, additional lookahead can be introduced by computation replication (Xu, Viswanathan, and Cai



(a) No replication (b) Replication at the LP boundary

Figure 1: Computation replication to reduce the frequency of synchronization.

2017). As shown in Figure 1, road network regions at LP boundaries are simulated by both neighboring LPs, which delays the need for synchronization. Given an overlapping region covering n time steps, it suffices to exchange state information among LPs every n steps to guarantee correctness of the results. Secondly, the synchronization may be relaxed to produce approximate results at lower overhead. By locally predicting the behavior of DVUs of remote LPs, the frequency of synchronizations can be reduced. Finally, in discrete-event formulations of traffic simulation models, the need for lookahead can be avoided altogether by employing optimistic synchronization algorithms (Yoginath and Perumalla 2009; Hanai et al. 2015).

3.2 Spiking Neural Networks

Spiking neural networks (SNNs) (Ghosh-Dastidar and Adeli 2009) are artificial neural networks that follow a modeling approach closer to biological principles compared to the dense networks most commonly used in machine learning. The defining characteristic of SNNs is the inter-neuron communication in the form of action potentials referred to as spikes, which are modeled as instantaneous occurrences in continuous time. Integration across the spikes observed at incoming synapses of a neuron determines the times at which new spikes are generated. Apart from their uses in machine learning (Tavanaei et al. 2019), SNNs serve a key role in studies aiming to simulate and understand the functioning of mammalian brains. Due to the sparse nature of the spikes transmitted throughout SNNs and their suitability to be implemented using neuromorphic hardware platforms, SNNs hold the promise to substantially reduce the energy costs of many machine learning tasks (Nunes et al. 2022). Computationally, SNNs differ from other types of artificial neural networks in the temporal sparsity of the inter-neuron communication along irregular topologies, allowing their execution to be naturally described in terms of discrete-event simulation.

The parallel execution of SNN simulators such as NEURON (Hines and Carnevale 1997), NEST (Diesmann and Gewaltig 2001), and Brian (Goodman and Brette 2009) as well as GPU-based variants (e.g., (Fidjeland et al. 2009; Chou et al. 2018) typically follow the window-based conservative algorithm YAWNS (cf. Section 2.1.2). Within a window, each LP handles the spikes received by a subset of all neurons. Newly generated spikes are executed at the receiving neurons in subsequent windows. This approach is exact if the window size is smaller than any synaptic delay, which is equivalent to a globally applicable lookahead. These strictly synchronous approaches have been successful in achieving near-linear scaling in benchmark problems (Migliore et al. 2006; Jordan et al. 2018). Recently, some work has considered the optimistically synchronized simulation of SNNs (Plagge et al. 2018; Pimpini et al. 2022).

4 RESEARCH DIRECTIONS

The focus in parallel and distributed simulation reearch has moved from the acceleration of individual simulation runs on homogeneous processors towards considerations of how the methods can benefit overall simulation studies in modern hardware environments. Many further research directions have been outlined in a recent roundtable report of the US Department of Energy (Perumalla et al. 2022).

4.1 Avoiding Redundant Computations

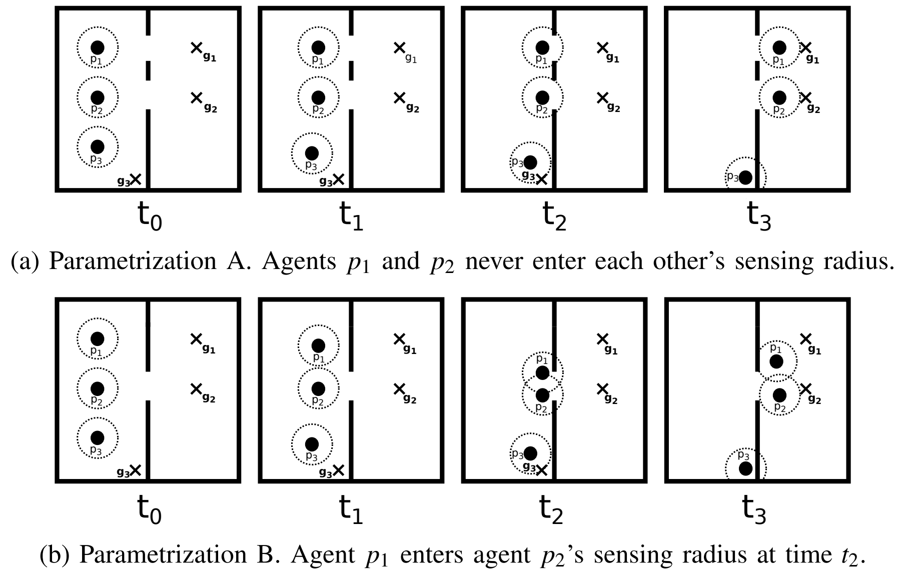
Simulation studies often require many runs of the same or similar scenarios, either to achieve sufficient statistical significance under stochasticity or to cover a range of parameters, e.g., for sensitivity analysis or optimization. The focus is then on the computational and memory demands of an *ensemble* of simulation runs. Since many parameters are often shared among the simulation runs, a significant portion of the computations and states encountered in an ensemble study may be identical across multiple runs.

Figure 2 shows two time-driven agent-based pedestrian simulations. Each pedestrian p_i moves towards a goal g_i in the simulation space and reacts to other pedestrians in direct vicinity as defined by a sensing radius. Initially, the two parametrizations differ only in the doors available to reach the goal, which affects the paths chosen of the agents. We see that at time t_1, the path taken by p_1 starts to diverge between the two parametrizations. However, p_1 has not entered another pedestrian's sensing radius. At time t_2, pedestrian p_1 enters p_2's sensing range in parametrization B, affecting p_2's trajectory. The subsequent behavior of pedestrians p_1 and p_2 now differs between the two runs, whereas p_3 still behaves the same.

Memoization is a generic approach used to avoid repeated identical computations by maintaining a cache of previous computational results in memory (Michie 1968). If a specific code segment, e.g., a function call, with the same input has been encountered before, the output can be retrieved from the cache. By automatically applying memoization in the context of ensemble studies, repeated computations can be avoided across simulation runs (Stoffers et al. 2018).

In *updateable simulation* (Ferenci et al. 2002), similarities among repeated simulations are exploited by storing a timestamp-ordered list of events observed in an initial execution. Subsequent simulations can then reexecute unchanged sequences of events in aggregate. While updateable simulation relies on a user-specified function to determine which events can be reused, *exact-differential simulation* (Hanai et al. 2015) employs the rollback mechanism from optimistic synchronization to achieve transparency.

Simulation cloning (Hybinette and Fujimoto 1997; Hybinette and Fujimoto 2001) is an approach that aims to eliminate redundant computations and reduce the memory consumption of ensemble studies by sharing partial simulation states among multiple runs. Instead of caching previous results, simulation cloning aims to avoid revisiting identical computations at the chosen granularity altogether. This is achieved by computing simulation states that are identical across multiple runs only once. All runs within the ensemble proceed concurrently, creating copies of partial states only when a unique trajectory is entered.



(a) Parametrization A. Agents p_1 and p_2 never enter each other's sensing radius.

(b) Parametrization B. Agent p_1 enters agent p_2's sensing radius at time t_2.

Figure 2: Two parametrizations of a pedestrian simulation.

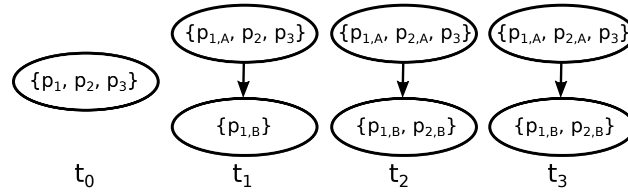


Figure 3: Cloning tree corresponding to the simulations shown in Figure 2.

At the heart of simulation cloning is the concept of the *cloning tree*, which represents the mapping of simulated entities to sets of simulation runs. Figure 3 shows the evolution of the cloning tree over the course of the two pedestrian simulations from our example. Initially, all pedestrians are in the same positions in both parametrizations. Thus, only one copy of their states exists. Since p_2 and p_3 are unaffected by the absence of the top door in parametrization B, their state updates from t_0 to t_1 are computed only once. In contrast, p_1 is cloned, resulting in two versions of its state, representing the paths taken in the two parametrizations. For the two versions, denoted as $p_{1,A}$ and $p_{1,B}$, separate updates are now computed at each step. At time t_2, pedestrian $p_{1,B}$ has entered p_2's sensing range, whereas $p_{1,A}$ has not. The influence of $p_{1,B}$ on p_2 in parametrization B requires p_2 to be cloned, creating the two versions $p_{2,A}$ and $p_{2,B}$. Since p_3 never interacts with the other pedestrians, only one copy exists throughout the simulation.

The benefit of cloning in this simple example can be quantified by counting the number of unique states. Including the initial states, a total of 24 states are visited without cloning. Simulation cloning considers only unique states, resulting in a total of only 17 pedestrian states.

Simulation cloning has been explored at several levels of granularity from the generic logical process level of the original proposals (Hybinette and Fujimoto 1997; Hybinette and Fujimoto 2001), to the level of individual agents in agent-based simulations (Li et al. 2017), to cells in grid-based simulation spaces (Yoginath and Perumalla 2018), to individual state variables (Pecher et al. 2018).

4.2 Exploiting Heterogeneous Hardware Platforms

In the past decades, general-purpose computing hardware has diversified into heterogeneous platforms in which multi-core CPUs are augmented by accelerators such as graphics processing units (GPUs), CPU-based manycore devices, and field-programmable gate arrays (FPGAs), creating a need to reconsider simulation algorithms and data structures. Hardware acceleration of discrete simulations can take several forms. One approach is to execute only specific model portions on an accelerator, whereas the simulator core remains on a CPU. For instance, in a network simulation, detailed models of wireless signal transmissions may be offloaded to a GPU (Bai and Nicol 2010; Andelfinger et al. 2011). This approach limits the porting efforts, but requires frequent data transfers between a host CPU and an accelerator. In contrast, executing the entirety of a simulation on an accelerator requires an implementation of the core simulator functionalities tailored to the target hardware. In both cases, the use of parallel computing resources can be increased by executing multiple simulation runs in parallel (Kunz et al. 2012; Komarov and D'Souza 2012).

Among today's accelerators, GPUs are the most widely used for discrete simulations (Xiao et al. 2019). Compared to CPU architectures, modern GPU architectures dedicate a larger proportion of transistors to data processing rather than caching and flow control (NVIDIA Corporation 2022). The unit of execution on a GPU is a group of 32 or 64 threads operating in lockstep. During execution of a GPU program, a hardware scheduler dynamically assigns thread groups to the available resources in order to hide memory access latencies. Control flow branches taken only by some of the threads in a group are serialized, resulting in reduced performance. In addition, the performance of GPU is heavily dependent on the memory access patterns. If memory accesses by consecutive threads target consecutive memory locations, groups memory accesses are *coalesced*, i.e., served in aggregate in a single memory transaction. Hence, GPUs perform particularly well for workloads with high degrees of regularity in the control flow and memory accesses.

The difficulty when transforming a simulation into a largely regular workload suitable for efficient GPU-based execution depends on the considered modeling paradigm. Let us first consider time-driven simulations as frequently used to execute agent-based models. In the simplest case, each time step of such a simulation involves updating the state of each agent present in the simulation. As the updates are dense in time, a simple one-to-one mapping of GPU threads to agents is possible. On the other hand, both the control flow and the memory accesses within the agent updates may vary. By assigning agents to threads according to their states, the probability of divergence in the control flow is reduced (Kunz et al. 2012).

The memory accesses during agent updates may also be highly irregular. For instance, an agent may interact with others within a certain radius. If the neighbors' variables are scattered in memory, opportunities for coalescing are rare. Suitable data layouts can increase these opportunities (Pham et al. 2018).

In event-driven simulations, state updates occur only at specific points in time, allowing events to be sparsely distributed both in time and across the simulated entities. The main challenge in GPU-based execution thus lies in efficiently identifying sufficient numbers of independent events to exploit the GPU's massively parallel hardware resources. The GPU's underlying single-instruction multiple-data processing approach suggests the use of synchronous parallel simulation algorithms. In Park and Fishwick (2010), synchronization follows the conservative window-based procedure of the YAWNS algorithm (cf. Section 2.1.2). In discrete-event simulators, priority queues are employed to iteratively select and execute the earliest pending event. In CPU-based simulators, efficient implementations typically rely on pointer-based data structures, e.g., calendar queues (Brown 1988). On GPUs, however, the mapping between simulated entities and threads is fine-grained, leading to much smaller numbers of pending events handled by a GPU thread. In combination with the substantial cost of dynamic memory allocations and cache misses, this suggests the use of dense array-based priority queue implementations (Baudis et al. 2017). The probability for a thread to remain idle can be reduced by assigning multiple entities to the same thread, as in the generic approach of thread coarsening (Zoppetti et al. 2000; Magni et al. 2013). However, the cost of operations required to insert new events and/or extract the earliest event increases with the number of events in the queue. By dynamically aggregating and deaggregating queues based on runtime measurements, parallelism and overhead can be balanced (Andelfinger and Hartenstein 2014; Liu and Andelfinger 2017).

Many-core CPUs such as Intel's Xeon Phi combine dozens of general-purpose cores on a chip. As commodity multi-core CPUs are now reaching similar core counts, parallel simulation algorithms must be able to efficiently support large numbers of parallel LPs in shared memory. Targeting Intel Xeon Phi using the Time Warp algorithm, the literature has identified the execution of communication and synchronization tasks on separate threads and efficient means of interaction between the threads as key considerations (Williams et al. 2021).

Making full use of the capabilities of accelerators often requires detailed knowledge of the target hardware. To alleviate the need for time-intensive development and debugging, several domain-specific simulation frameworks automatically translate model specifications to accelerator code. These frameworks provide optimized implementations of common simulator functionalities, whereas the model code is generated from a representation in a domain-specific language (Cosenza et al. 2021; Chou et al. 2018).

Since CPUs and different accelerators vary in their performance on a given type of computation, new challenges arise when aiming to optimally use the available devices in combination. In purely CPU-based parallel and distributed simulations, low synchronization overhead and an evenly balanced workload is often achieved by domain decomposition, i.e., by dividing the simulation space along the geographical or logical topology defined by the model. In a heterogeneous hardware environment, however, different models may vary severely in their execution times on different devices. Hence, it may be more efficient to choose a functional decomposition across different models used throughout the simulation, either statically or based on runtime measurements and performance models (Bai and Nicol 2010; Xiao et al. 2020).

5 CONCLUSION

We provided an overview of established methods and research directions in methods for high-performance discrete simulations to meet the computational demands of large-scale and detailed simulation studies. Algorithms for parallel and distributed simulation to accelerate and scale up the execution of individual simulation runs are well-studied, with recent refinements focusing on specific application domains or emerging hardware platforms. Current research aims at the fruitful use of parallel simulation techniques in overall simulation studies to avoid redundant computations and to exploit heterogeneous hardware platforms.

Efforts towards making the parallelization of models largely transparent to the developer operate between the two poles of maximizing performance by specialization to model properties on one hand, and generalization on the other hand. A balance is provided by domain-specific frameworks offering optimized parallel simulation facilities tailored to common properties shared by entire classes of models.

ACKNOWLEDGMENT

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

REFERENCES

Andelfinger, P., and H. Hartenstein. 2014. "Exploiting the Parallelism of Large-Scale Application-Layer Networks by Adaptive GPU-based Simulation". In *Proceedings of the 2014 Winter Simulation Conference*, edited by A. Tolk, S. Y. Diallo, I. O. Ryzhov, L. Yilmaz, S. Buckley, and J. A. Miller, 3471–3482. Piscataway, NJ: Institute of Electrical and Electronics Engineers.

Andelfinger, P., J. Mittag, and H. Hartenstein. 2011. "GPU-based Architectures and their Benefit for Accurate and Efficient Wireless Network Simulations". In *International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*. July 25th–27th, Singapore, 421–424.

Andelfinger, P., Y. Xu, D. Eckhoff, W. Cai, and A. Knoll. 2020. "Fidelity and Performance of State Fast-Forwarding in Microscopic Traffic Simulations". *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 30(2):1–26.

Bai, S., and D. M. Nicol. 2010. "Acceleration of Wireless Channel Simulation using GPUs". In *European Wireless Conference*. April 12th–15th, Lucca, Italy, 841–848.

Barceló, J. 2010. "Models, Traffic Models, Simulation, and Traffic Simulation". In *Fundamentals of Traffic Simulation*, 1–62. New York, NY: Springer.

Baudis, N., F. Jacob, and P. Andelfinger. 2017. "Performance Evaluation of Priority Queues for Fine-Grained Parallel Tasks on GPUs". In *International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. September 20th–22th, Banff, Canada, 1–11.

Brown, R. 1988. "Calendar Queues: a Fast O(1) Priority Queue Implementation for the Simulation Event Set Problem". *Communications of the ACM* 31(10):1220–1227.

Bryant, R. 1977. "Simulation of Packet Communications Architecture Computer Systems". Master's thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge. https://www.cs.cmu.edu/~bryant/pubdir/MIT-LCS-TR-188.pdf, accessed 25th September 2022.

Cai, W., and S. J. Turner. 1995. "An Algorithm for Reducing Null-Messages of CMB Approach in Parallel Discrete Event Simulation". Technical report, University of Exeter.

Carothers, C. D., K. S. Perumalla, and R. M. Fujimoto. 1999. "Efficient Optimistic Parallel Simulations using Reverse Computation". *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 9(3):224–253.

Chandy, K. M., and J. Misra. 1981. "Asynchronous Distributed Simulation via a Sequence of Parallel Computations". *Communications of the ACM* 24(4):198–206.

Chou, T.-S., H. J. Kashyap, J. Xing, S. Listopad, E. L. Rounds, M. Beyeler, N. Dutt, and J. L. Krichmar. 2018. "CARLsim 4: An Open Source Library for Large Scale, Biologically Detailed Spiking Neural Network Simulation using Heterogeneous Clusters". In *International Joint Conference on Neural Networks*. July 8th–13th, Rio de Janeiro, Brazil, 1–8.

Cosenza, B., N. Popov, B. Juurlink, P. Richmond, M. K. Chimeh, C. Spagnuolo, G. Cordasco, and V. Scarano. 2021. "Easy and Efficient Agent-Based Simulations with the OpenABL Language and Compiler". *Future Generation Computer Systems* 116:61–75.

Deelman, E., R. Bargodia, R. Sakellariou, and V. Adve. 2001. "Improving Lookahead in Parallel Discrete Event Simulations of Large-Scale Applications using Compiler Analysis". In *Workshop on Parallel and Distributed Simulation*. May 15th–18th, Lake Arrowhead, California, USA, 5–13.

Diesmann, M., and M.-O. Gewaltig. 2001. "NEST: An Environment for Neural Systems Simulations". *Forschung und wissenschaftliches Rechnen, Beiträge zum Heinz-Billing-Preis* 58:43–70.

Ferenci, S. L., R. M. Fujimoto, M. H. Ammar, K. Perumalla, and G. F. Riley. 2002. "Updateable Simulation of Communication Networks". In *Proceedings of the Workshop on Parallel and Distributed simulation*. May 12th–15th, Washington DC, USA, 107–114.

Ferscha, A., J. Johnson, and S. J. Turner. 1997. "Early Performance Prediction of Parallel Simulation Protocols". In *Proceedings of the World Congress on System Simulation*. September 1st–4th, Singapore, 282–287.

Fidjeland, A. K., E. B. Roesch, M. P. Shanahan, and W. Luk. 2009. "NeMo: a Platform for Neural Modelling of Spiking Neurons using GPUs". In *International Conference on Application-Specific Systems, Architectures and Processors*. July 7th–9th, Boston, Massachusetts, USA, 137–144.

Fujimoto, R. 2015. "Parallel and Distributed Simulation". In *Proceedings of the 2015 Winter Simulation Conference*, edited by L. Yilmaz, W. K. V. Chan, I. Moon, T. M. K. Roeder, C. Macal, and M. D. Rossetti, 45–59. Piscataway, NJ: Institute of Electrical and Electronics Engineers.

Fujimoto, R. M. 2016. "Research Challenges in Parallel and Distributed Simulation". *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 26(4):1–29.

Fujimoto, R. M., R. Bagrodia, R. E. Bryant, K. M. Chandy, D. Jefferson, J. Misra, D. Nicol, and B. Unger. 2017. "Parallel Discrete Event Simulation: The Making of a Field". In *Proceedings of the 2017 Winter Simulation Conference*, edited by W. K. V. Chan, A. D'Ambrogio, G. Zacharewicz, N. Mustafee, G. Wainer, and E. Page, 262–291. Piscataway, NJ: Institute of Electrical and Electronics Engineers.

Ghosh-Dastidar, S., and H. Adeli. 2009. "Spiking Neural Networks". *International Journal of Neural Systems* 19(04):295–308.

Goodman, D. F., and R. Brette. 2009. "The Brian Simulator". *Frontiers in Neuroscience* 3:26.

Grošelj, B., and C. Tropper. 1991. "The Distributed Simulation of Clustered Processes". *Distributed Computing* 4(3):111–121.

Hanai, M., T. Suzumura, G. Theodoropoulos, and K. S. Perumalla. 2015. "Exact-Differential Large-Scale Traffic Simulation". In *Proceedings of the Conference on Principles of Advanced Discrete Simulation*. June 15th–17th, London, UK, 271–280.

Hines, M. L., and N. T. Carnevale. 1997. "The NEURON Simulation Environment". *Neural Computation* 9(6):1179–1209.

Hybinette, M., and R. Fujimoto. 1997. "Cloning: A Novel Method for Interactive Parallel Simulation". In *Proceedings of the 1997 Winter Simulation Conference*, edited by S. Andraddottir, K. J. Healy, D. H. Withers, and B. L. Nelson, 444–451. Piscataway, NJ: Institute of Electrical and Electronics Engineers.

Hybinette, M., and R. M. Fujimoto. 2001. "Cloning Parallel Simulations". *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 11(4):378–407.

Igbe, D. 2010. *Dynamic Load Balancing of Parallel Road Traffic Simulation*. Ph. D. thesis, School of Electronics and Computer Science, University of Westminster. https://westminsterresearch.westminster.ac.uk/item/90644/dynamic-load-balancing-of-parallel-road-traffic-simulation, accessed 25th September 2022.

Jefferson, D., B. Beckman, F. Wieland, L. Blume, and M. DiLoreto. 1987. "Time Warp Operating System". In *Proceedings of the Symposium on Operating Systems Principles*. November 8th–11th, Austin, Texas, USA, 77–93.

Jordan, J., T. Ippen, M. Helias, I. Kitayama, M. Sato, J. Igarashi, M. Diesmann, and S. Kunkel. 2018. "Extremely Scalable Spiking Neuronal Network Simulation Code: from Laptops to Exascale Computers". *Frontiers in Neuroinformatics* 12:2.

Karypis, G., and V. Kumar. 1997. "METIS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices". Technical report, University of Minnesota.

Komarov, I., and R. M. D'Souza. 2012. "Accelerating the Gillespie Exact Stochastic Simulation Algorithm using Hybrid Parallel Execution on Graphics Processing Units". *PLoS One* 7(11):e46693.

Kunz, G., D. Schemmel, J. Gross, and K. Wehrle. 2012. "Multi-level Parallelism for Time- and Cost-Efficient Parallel Discrete Event Simulation on GPUs". In *Workshop on Principles of Advanced and Distributed Simulation*. July 15th–19th, Zhangjiajie, China, 23–32.

Li, X., W. Cai, and S. J. Turner. 2017. "Cloning Agent-Based Simulation". *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 27(2):1–24.

Liu, J., D. Nicol, B. Premore, and A. Poplawski. 1999. "Performance Prediction of a Parallel Simulator". In *Proceedings of the Workshop on Parallel and Distributed Simulation*, 156–164. ACM/IEEE/SCS.

Liu, X., and P. Andelfinger. 2017. "Time Warp on the GPU: Design and Assessment". In *Proceedings of the Conference on Principles of Advanced Discrete Simulation*. May 24th–26th, Singapore, 109–120.

Loper, M. L., and R. M. Fujimoto. 2000. "Pre-Sampling as an Approach for Exploiting Temporal Uncertainty". In *Proceedings of the Workshop on Parallel and Distributed Simulation*. May 28th–31st, Bologna, Italy, 157–164.

Lubachevsky, B. D. 1989. "Efficient Distributed Event-Driven Simulations of Multiple-Loop Networks". *Communications of the ACM* 32(1):111–123.

Magni, A., C. Dubach, and M. F. O'Boyle. 2013. "A Large-Scale Cross-Architecture Evaluation of Thread-Coarsening". In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. November 17th–22nd, Denver, Colorado, USA, 1–11.

Mattern, F. 1993. "Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation". *Journal of Parallel and Distributed Computing* 18(4):423–434.

Meyer, R. A., and R. L. Bagrodia. 1999. "Path Lookahead: a Data Flow View of PDES Models". In *Proceedings of the Workshop on Parallel and Distributed Simulation*. May 1st–4th, Atlanta, Georgia, USA, 12–19.

Michie, D. 1968. ""Memo" Functions and Machine Learning". *Nature* 218(5136):19–22.

Migliore, M., C. Cannia, W. W. Lytton, H. Markram, and M. L. Hines. 2006. "Parallel Network Simulations with NEURON". *Journal of Computational Neuroscience* 21(2):119–129.

Nicol, D. M. 1993. "The Cost of Conservative Synchronization in Parallel Discrete Event Simulations". *Journal of the ACM (JACM)* 40(2):304–333.

Nunes, J. D., M. Carvalho, D. Carneiro, and J. S. Cardoso. 2022. "Spiking Neural Networks: A Survey". *IEEE Access* 10:60738–60764.

NVIDIA Corporation 2022. "NVIDIA CUDA C Programming Guide". https://docs.nvidia.com/cuda/cuda-c-programming-guide, accessed 25th September 2022.

Palaniswamy, A. C., and P. A. Wilsey. 1994. "Scheduling Time Warp Processes using Adaptive Control Techniques". In *Proceedings of the 1994 Winter Simulation Conference*, edited by J. D. Tew, S. Manivannan, D. A. Sadowski, and A. F. Seile, 731–738. Piscataway, NJ: Institute of Electrical and Electronics Engineers.

Park, H., and P. A. Fishwick. 2010. "A GPU-based Application Framework Supporting Fast Discrete-Event Simulation". *Simulation* 86(10):613–628.

Pecher, P., J. Crittenden, Z. Lu, and R. Fujimoto. 2018. "Granular Cloning: Intra-Object Parallelism in Ensemble Studies". In *Proceedings of the Conference on Principles of Advanced Discrete Simulation*. May 23rd–25th, Rome, Italy, 165–176.

Pell, A., A. Meingast, and O. Schauer. 2017. "Trends in Real-Time Traffic Simulation". *Transportation Research Procedia* 25:1477–1484.

Perumalla, K., M. Bremer, K. Brown, C. Chan, S. Eidenbenz, K. S. Hemmert, A. Hoisie, B. Newton, J. Nutaro, T. Oppelstrup, R. Ross, M. Schordan, and N. Urban. 2022, 5. "Computer Science Research Needs for Parallel Discrete Event Simulation (PDES)". Technical report, United States Department of Energy Office of Science.

Perumalla, K. S., R. M. Fujimoto, P. J. Thakare, S. Pande, H. Karimabadi, Y. Omelchenko, and J. Driscoll. 2005. "Performance Prediction of Large-Scale Parallel Discrete Event Models of Physical Systems". In *Proceedings of the 2005 Winter Simulation Conference*, edited by M. E. Kuhl, N. M. Steiger, F. B. Armstrong, and J. A. Joines, 356–364. Piscataway, NJ: Institute of Electrical and Electronics Engineers.

Peschlow, P., A. Voss, and P. Martini. 2009. "Good News for Parallel Wireless Network Simulations". In *Proceedings of the International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems*. October 26th–30th, Tenerife, Canary Islands, Spain, 134–142.

Pham, N. Q. A., R. Fan, and W. Cai. 2018. "Optimizing Agent-Based Simulations for the GPU". In *International Conference on High Performance Computing & Simulation*. July 16th–20th, Orleans, France, 171–179.

Piccione, A., M. Principe, A. Pellegrini, and F. Quaglia. 2019. "An Agent-Based Simulation API for Speculative PDES Runtime Environments". In *Proceedings of the Conference on Principles of Advanced Discrete Simulation*. June 3rd–5th, Chicago, Illinois, USA, 83–94.

Pimpini, A., A. Piccione, B. Ciciani, and A. Pellegrini. 2022. "Speculative Distributed Simulation of Very Large Spiking Neural Networks". In *Proceedings of the Conference on Principles of Advanced Discrete Simulation*. June 8th–10th, Atlanta, Georgia, USA, 93–104.

Plagge, M., C. D. Carothers, E. Gonsiorowski, and N. Mcglohon. 2018. "NeMo: A Massively Parallel Discrete-Event Simulation Model for Neuromorphic Architectures". *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 28(4):1–25.

Reiher, P. L., F. Wieland, and D. Jefferson. 1989. "Limitation of Optimism in the Time Warp Operating System". In *Proceedings of the 1989 Winter Simulation Conference*, edited by E. A. MacNair, K. J. Musselman, and P. Heidelberger, 765–770. Piscataway, NJ: Institute of Electrical and Electronics Engineers.

Richmond, P., and M. K. Chimeh. 2017. "Flame GPU: Complex System Simulation Framework". In *International Conference on High Performance Computing & Simulation*. July 17th–21st, Genoa, Italy, 11–17.

Ronngren, R., M. Liljenstram, R. Ayani, and J. Montagnat. 1996. "A Comparative Study of State Saving Mechanisms for Time Warp Synchronized Parallel Discrete Event Simulation". In *Proceedings of the Annual Simulation Symposium*. April 8th–11th, New Orleans, Louisiana, USA, 5–14.

Samadi, B. 1985. *Distributed Simulation, Algorithms and Perfomance Analysis*. Ph. D. thesis, University of California, Los Angeles. https://www.proquest.com/docview/303368540, accessed 25th September 2022.

Sokol, L. 1988. "MTW: A Strategy for Scheduling Discrete Simulation Events for Concurrent Execution". In *Multiconference on Distributed Simulation*. February 3rd–5th, San Diego, California, USA, 34–42.

Steinman, J. 1991. "SPEEDES: Synchronous Parallel Environment for Emulation and Discrete Event Simulation". In *Multi-Conference on Advances in Parallel and Discrete Simulation*, Volume 23. January 23rd–25th, Anaheim, California, USA, 1111-1115.

Steinman, J. S. 1993. “Breathing Time Warp”. In *Proceedings of the Workshop on Parallel and Distributed Simulation*. Mayth–19th, San Diego, California, USA, 109–118.

Stoffers, M., D. Schemmel, O. S. Dustmann, and K. Wehrle. 2018. “On Automated Memoization in the Field of Simulation Parameter Studies”. *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 28(4):1–25.

Su, W.-K., and C. L. Seitz. 1988. “Variants of the Chandy-Misra-Bryant Distributed Discrete-Event Simulation Algorithm”. Technical report, Department of Computer Science, California Institute of Technology, Pasadena.

Sun, X., F. Chen, X. Li, and X. Wang. 2012. “An Improved Dynamic Load Balancing Algorithm for Parallel Microscopic Traffic Simulation”. In *International Conference on Measurement, Information and Control*, Volume 2. May 18th–20th, Harbin, China, 600–604.

Tavanaei, A., M. Ghodrati, S. R. Kheradpisheh, T. Masquelier, and A. Maida. 2019. “Deep Learning in Spiking Neural Networks”. *Neural Networks* 111:47–63.

Tikidji-Hamburyan, R. A., V. Narayana, Z. Bozkus, and T. A. El-Ghazawi. 2017. “Software for Brain Network Simulations: a Comparative Study”. *Frontiers in Neuroinformatics* 11:46.

Treiber, M., and V. Kanagaraj. 2015. “Comparing Numerical Integration Schemes for Time-Continuous Car-Following Models”. *Physica A: Statistical Mechanics and its Applications* 419:183–195.

Turner, S. J., and M. Q. Xu. 1991. “Performance Evaluation of the Bounded Time Warp Algorithm”. In *Proceedings of the Workshop on Parallel and Distributed Simulation*. January 23th–26th, Anaheim, California, USA, 117–128.

Wang, J., and C. Trropper. 2007. “Optimizing Time Warp Simulation with Reinforcement Learning Techniques”. In *Proceedings of the 2007 Winter Simulation Conference*, edited by S. G. Henderson, B. Biller, M.-H. Hsieh, J. Shortle, J. D. Tew, and R. R. Barton, 577–584. Piscataway, NJ: Institute of Electrical and Electronics Engineers.

West, D., and K. Panesar. 1996. “Automatic Incremental State Saving”. In *Proceedings of the Workshop on Parallel and Distributed Simulation*. May 22nd–24th, Philadelphia, Pennsylvania, USA, 78–85.

Williams, B., A. Eker, K. Chiu, and D. Ponomarev. 2021. “High-performance PDES on Manycore Clusters”. In *Proceedings of the Conference on Principles of Advanced Discrete Simulation*. May 31st–June 2nd, Virtual event, 153–164.

Xiao, J., P. Andelfinger, W. Cai, P. Richmond, A. Knoll, and D. Eckhoff. 2020. “OpenABLext: An Automatic Code Generation Framework for Agent-Based Simulations on CPU-GPU-FPGA Heterogeneous Platforms”. *Concurrency and Computation: Practice and Experience* 32(21):e5807.

Xiao, J., P. Andelfinger, D. Eckhoff, W. Cai, and A. Knoll. 2019. “A Survey on Agent-Based Simulation using Hardware Accelerators”. *ACM Computing Surveys (CSUR)* 51(6):1–35.

Xu, Y., W. Cai, D. Eckhoff, S. Nair, and A. Knoll. 2017. “A Graph Partitioning Algorithm for Parallel Agent-Based Road Traffic Simulation”. In *Proceedings of the Conference on Principles of Advanced Discrete Simulation*. May 24th–26th, Singapore, 109–120.

Xu, Y., V. Viswanathan, and W. Cai. 2017. “Reducing Synchronization Overhead with Computation Replication in Parallel Agent-Based Road Traffic Simulation”. *IEEE Transactions on Parallel and Distributed Systems* 28(11):3286–3297.

Yoginath, S. B., and K. S. Perumalla. 2009. “Reversible Discrete Event Formulation and Optimistic Parallel Execution of Vehicular Traffic Models”. *International Journal of Simulation and Process Modelling* 5(2):104–119.

Yoginath, S. B., and K. S. Perumalla. 2018. “Scalable Cloning on Large-Scale GPU Platforms with Application to Time-Stepped Simulations on Grids”. *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 28(1):11–26.

Zhang, D., C. Jiang, and S. Li. 2007. “Research on Dynamic Load Balancing Algorithms for Parallel Transportation Simulations”. In *International Workshop on Advanced Parallel Processing Technologies*. November 22nd–23th, Guangzhou, China, 560–568.

Zoppetti, G. M., G. Agrawal, L. Pollock, J. N. Amaral, X. Tang, and G. Gao. 2000. “Automatic Compiler Techniques for Thread Coarsening for Multithreaded Architectures”. In *International Conference on Supercomputing*. May 8th–11th, Santa Fe, New Mexico, USA, 306–315.

AUTHOR BIOGRAPHIES

PHILIPP ANDELFINGER is a postdoctoral researcher at University of Rostock. His research focuses on parallel simulation using heterogeneous hardware and simulation-based optimization. His e-mail address is philipp.andelfinger@uni-rostock.de.

WENTONG CAI is a professor in the School of Compute Science and Engineering (SCSE) at Nanyang Technological University (NTU), Singapore. He is a member of the IEEE and the ACM. He is an Associate Editor of the ACM Transactions on Modelling and Computer Simulation (TOMACS), an editor of the Future Generation Computer Systems (FGCS), and was an editorial board member of the Journal of Simulation (JOS). His research interests are in the areas of Modelling and Simulation, and Parallel and Distributed Computing. His email address is aswtcai@ntu.edu.sg.