Zero Lookahead? Zero Problem. The Window Racer Algorithm

Philipp Andelfinger
Till Köster
Adelinde Uhrmacher
philipp.andelfinger@uni-rostock.de
till.koester@uni-rostock.de
adelinde.uhrmacher@uni-rostock.de
Institute for Visual and Analytic Computing, University of Rostock
Rostock, Germany

ABSTRACT

Synchronization algorithms for parallel simulation struggle to attain speedup if the simulation entities are tightly coupled and their interactions are difficult to predict. Window Racer is a novel parallel synchronization algorithm for shared-memory architectures specifically targeted toward attaining speedup in these challenging cases. The key idea is to allow the processing elements to speculatively execute sequences of dependent events even across partition boundaries through fine-grained locking and low-overhead rollbacks, while at the same time negotiating a global synchronization window that rules out transitive rollbacks. In performance measurements using a variant of the PHold benchmark model, Window Racer outperforms an established implementation of the Time Warp algorithm in model configurations where events are often scheduled with near-zero delay. In an ablation study, we pinpoint the performance impact of our algorithm's individual features by reducing Window Racer to two existing algorithms. We further study the algorithm's ability to attain speedup in simulations of bio-chemical reaction networks, a particularly challenging class of simulations with tightly coupled state transitions.

CCS CONCEPTS

• **Computing methodologies** → **Massively parallel and high-performance simulations**; **Discrete-event simulation**.

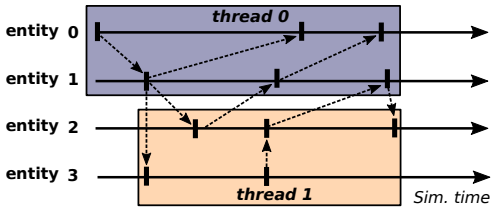
KEYWORDS

PDES, Optimistic Synchronization, Time Warp

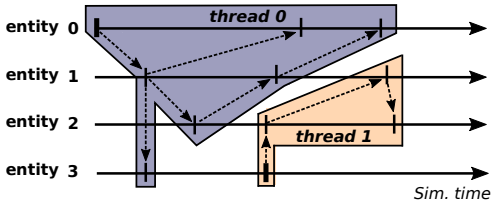
ACM Reference Format:
Philipp Andelfinger, Till Köster, and Adelinde Uhrmacher. 2023. *Zero Lookahead? Zero Problem.* The Window Racer Algorithm. In *ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (SIGSIM-PADS '23), June 21–23, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3573900.3591115

SIGSIM-PADS '23, June 21–23, 2023, Orlando, FL, USA

ACM ISBN 979-8-4007-0030-9/23/06…$15.00
https://doi.org/10.1145/3573900.3591115



(a) Classical synchronization is based on a strict entity-based partitioning in shared memory. Each thread is responsible for events executed at a subset of the simulated entities (highlighted in shades of gray). Thread interactions are required when event scheduling crosses thread boundaries.



(b) Window Racer applies a less restrictive assignment of entities to threads. Each thread attempts to execute event chains caused by local entities regardless of the dependent events' correspondence to entities, reducing the need for thread interactions.

Figure 1: Contrasting classical synchronization approaches with the proposed Window Racer algorithm.

1 INTRODUCTION

Optimistic synchronization algorithms for parallel and distributed simulation [17] can be applied to models where lookahead information, which would provide lower bounds on the delays between events and their creation, is unavailable. These algorithms execute events speculatively and carry out rollbacks when an erroneous event ordering is detected. Much of the research on these algorithms has focused on the Time Warp algorithm [25], whose asynchronous mode of execution allows processing elements (PEs) to advance in simulation time with a considerable degree of independence. However, some discrete-event simulation models, such as those representing bio-chemical reaction networks [19, 20], are comprised of simulation objects that frequently interact across PE boundaries with short delays or even instantaneously. When executing models with these properties using Time Warp, frequent rollbacks can limit

the performance [26]. Synchronous algorithms such as Breathing Time Buckets [46] restrict the progress of the PEs more severely by computing a globally applicable window in simulation time. Hence, the amount of simulation time covered in a round can be very short, leading to large amounts of overhead due to frequent synchronization among PEs.

Window Racer is a synchronous algorithm aimed toward discrete-event models of tightly coupled systems involving large numbers of atomic simulation objects, which we refer to as entities. The Window Racer algorithm is tailored to shared-memory architectures as encountered in modern compute nodes, which routinely offer between dozens and hundreds of cores. The algorithm's name is inspired by viewing the PEs as engaged in a "race" to fit as many events as possible into a gradually closing synchronization window. Within each synchronization round, the PEs negotiate the final size of the synchronization window as part of the event execution through fine-grained locking and atomic operations. Compared to the Breathing Time Buckets algorithm, substantially larger window sizes are obtained by allowing PEs to execute newly scheduled events even if they target remote entities (cf. Figure 1).

Beyond introducing Window Racer, the present work aims to shed light on the types of models for which the algorithm is suited and the impact of its algorithmic ideas on its performance. To this end, we provide measurement results against the established simulator ROOT-Sim[38] for various parametrizations of the PHold model [16] extended by events with near-zero delay. In our implementation of Window Racer, which we make available to the community[1], the algorithm's main novel features can be toggled individually, allowing us to reduce the implementation to two existing algorithms. We use this capability to carry out an ablation study showing the influence of the main algorithmic ideas using the same underlying code. Having obtained an understanding of the performance properties of the algorithm, experiments using models of bio-chemical systems showcase the real-world applicability of the algorithm.

Our main contributions are as follows:

- Window Racer, a speculative synchronization algorithm targeting simulation models with zero lookahead.
- A performance comparison against ROOT-Sim using a synthetic benchmark model.
- An ablation study to pinpoints the performance impact of Window Racer's algorithmic features.
- Measurements for simulations of bio-chemical reaction networks against a pool of state-of-the-art sequential simulators.

The remainder of the paper is structured as follows. In Section 2, we discuss existing work on optimistic synchronization algorithms and simulations of bio-chemical reaction networks. Section 3 describes our proposed algorithm. Section 4 presents measurements using a synthetic benchmark and models of bio-chemical systems. Section 5 summarizes our findings and concludes the paper.

2 RELATED WORK

Much of the literature on optimistic algorithms for parallel and distributed simulation focuses on the Time Warp algorithm [25], in which the processing elements (PEs) operate asynchronously and communicate in a message-passing style. In the following, we use Time Warp as our reference point to describe existing works on efficient simulations of tightly interacting entities by exploiting shared-memory architectures or by restricting the degree of optimism. An overview of simulation methods for bio-chemical reaction networks provides background for our case study in Section 4.3.

2.1 Exploiting Shared Memory

Among the main overhead sources in the Time Warp algorithm is the handling of transitive errors, which may occur when a straggler event invalidates the creation of events that have already been sent to remote PEs. To cancel an event created in error, traditional Time Warp implementations send a so-called antimessage to the remote PE. Fujimoto showed the potential for efficiency improvements in shared-memory environments by avoiding the explicit sending of antimessages [15]. Instead, a tree-based data structure is maintained to track chains of event creations and is traversed to roll back an event and all event creations it has caused. In Window Racer, the need for explicit cancellation of individual event creations is avoided entirely by negotiating a global synchronization window that precludes transitive errors.

More extensive changes to the execution flow of Time Warp-based simulations allow PEs to access shared simulation state. Space-time memory and related approaches [18, 32, 36] introduce a temporal versioning of shared state variables, enabling efficient read and write accesses at each PE's local point in time. Pellegrini et al. propose an extension to Time Warp and operating system-level mechanisms to transparently support events that carry out read and write accesses to multiple entities [37]. In Window Racer, entity accesses across entities are managed through entity-level locking, state saving, and rollbacks, reducing the need for explicit event exchanges among PEs.

Ianni et al. presented a share-everything approach [24] that abandons the binding of simulation entities to PEs altogether. Instead, the PEs retrieve events from a shared event list pertaining to all entities. Compared to the traditional Time Warp, their approach focuses the computation on the globally earliest events, whose swift execution is decisive for the overall performance. Similarly to this work, Window Racer loosens the binding of the entities to PEs during event execution. However, in Window Racer, the threads immediately and largely independently execute causal chains of events across arbitrary entities without the potential for congestion on a shared event list.

Finally, the cost of entity interactions can be reduced by aggregating entities both in traditional Time Warp [8] and in shared memory-specific variants [30]. In effect, aggregation combines sets of entities into joint sequential processes. As a consequence, event exchanges within each process can be handled by a single PE. On the other hand, larger degrees of aggregation reduce the concurrency in the simulated system and increase the cost of rollbacks due to the increased state size. Window Racer's synchronous execution scheme benefits from the concurrency offered by models involving large numbers of entities. To offer a fair performance comparison to Time Warp implementations, we carried out our measurements presented in Section 4.1 at various aggregation levels, reporting on the best-performing Time Warp configuration.

[1] URL omitted for anonymous review

2.2 Bounded Optimism and Synchronous Algorithms

Models in which the entities frequently generate events with short or zero-valued temporal delays offer fewer opportunities for asynchronous processing. A variety of methods have been proposed to bound the degree of optimism in Time Warp using a tuneable window size [42, 48] or adaptive approaches based on estimations of mis-speculation probabilities or overheads [13, 35, 51].

Breathing Time Buckets [44] is a synchronous optimistic algorithm. In every synchronization round, each PE executes events up to a tuneable upper bound in simulation time. Once all PEs have finished execution, the *event horizon* is determined as the timestamp of the earliest newly created event crossing a PE boundary. All PEs roll back events with timestamps later than the event horizon and discard events generated by the rolled-back events. Events generated before the event horizon are delivered to their target PEs. This synchronization scheme is risk-free, i.e., it avoids transitive errors in the sense that it is guaranteed that any event delivered to a remote PE will eventually be committed. As a consequence, the need for antimessages is avoided. The algorithm as described above is suited both for distributed and shared-memory architectures. The Minimum Time Buckets algorithm [52] is a specialization of Breathing Time Buckets for shared-memory environments in which updates to the event horizon are communicated to all PEs. Compared to the original algorithm, the resulting event horizon in each round remains unaltered. However, PEs can terminate the execution phase earlier, reducing the probability of executing events guaranteed to be rolled back. Breathing Time Warp [45] combines Breathing Time Buckets with Time Warp by splitting each synchronization round into a Time Warp phase allowing for transitive errors and a risk-free Breathing Time Buckets phase.

S^3A [2] is a domain-specific algorithm targeting simulations of multi-agent systems. As in Breathing Time Buckets, each synchronization round in S^3A computes a globally valid time window that permits each agent to carry out at most one transition. While being tolerant to immediate inter-PE accesses handled by entity-level locking, S^3A's restrictive synchronization scheme incurs the need for large amounts of concurrency to achieve significant speedup.

Window Racer shares its risk-free synchronous execution mode with Breathing Time Buckets and Minimum Time Buckets, as well as the fine-grained locking with the S^3A algorithm. However, it differs from these approaches in its execution of chains of dependent events independently of the events' correspondence to simulation entities. This feature, in conjunction with identifying the window bound based on entity-level timestamps, allows Window Racer to extract larger window sizes and thus performance in models with limited concurrency. In Section 4.2, we disable these defining features of Window Racer to lay bare their individual impacts.

2.3 Bio-Chemical Reaction Networks

Bio-chemical systems are frequently simulated using a family of algorithm subsumed under the term Stochastic Simulation Algorithm (SSA) [21]. Models used in this context abstract from the spatial distribution of individual model entities referred to as species. Instead, the simulation state is comprised of sum densities or populations of species of certain categories, which are altered by discrete events in continuous time. Computationally, this mode of simulation leads to a tightly coupled simulation state updated by fine-grained transitions often associated with short or zero-valued delays.

Concrete algorithms from the SSA family fall in two main classes, the Direct Method and the First Reaction Method. Most popular approaches follow the Direct Method, which determines the next reaction firing time by a mathematical expression based on all possible reactions. The tight dependence of the next reaction on the overall system state makes parallelization challenging.

In contrast, the First Reaction Method iteratively draws tentative firing times, executes the earliest firing, and updates the system state accordingly. A parallelization of this approach has shown to attain high speedup on graphics processing units [10]. A more sophisticated algorithm originating from the First Reaction Method is the Next Reaction method (NRM). In this approach, tentative firing times are scheduled in a priority queue. When a firing has occurred, all dependent reactions are rescheduled via a static dependency graph. Thanh et al. exploited the independence among the reaction updates in a shared-memory environment [47]. Since the NRM naturally aligns with the discrete-event paradigm, established algorithms from optimistic parallel and distributed simulation can also be applied, as outlined by Goldberg et al. [22]. Nevertheless, the gainful parallelization of SSA models remains challenging due to their fundamental underlying approximation: by abstracting from spatial properties of the system, all communication is instant. Furthermore, tight dependencies among reactions allow only for little inherent parallelism.

When modeling large systems, the assumption of spatial homogeneity at the core of SSA is no longer appropriate. One intermediate approach between purely non-spatial SSA and fully spatial multi-particle simulation [27] is the Next Subvolume Method (NSM) [11]. Here, the system is divided into subvolumes, each of which contains its own population. Diffusion reactions occur among the species in adjacent volumes, corresponding to physical movement and balancing among subvolumes. An implementation of the NSM in the Aurora simulation environment [26] demonstrated the difficulties of achieving speedup in a distributed-memory setting, where an efficient parallelization across computationally fine-grained events is particularly challenging. The Abstract Next Subvolume Method represents a parallelization of the NSM by combining the Direct Method with a parallel execution of the NRM [49]. Different implementations of the Abstract Next Subvolume Method using Breathing Time Warp have been shown to outperform both Time Warp and Breathing Time Buckets [50]. Another approach to parallelizing the NSM is to design dedicated parallel algorithms tailored to the NSM's known characteristics. One example is to estimate the communication among simulation entities based on model-specific knowledge and to minimize the overhead by performing rollbacks only selectively. This approach has been applied to the All Events Method, in which transitions are more costly than in the NSM, making speedup more achievable [4], and later to the NSM [29].

The Window Racer algorithm presented in this paper is tailored for efficient execution in the presence of events with near-zero or zero delay. In Section 4.3, we apply Window Racer to a series of NSM problems and study its performance with reference to a pool of state-of-the-art sequential simulation tools.

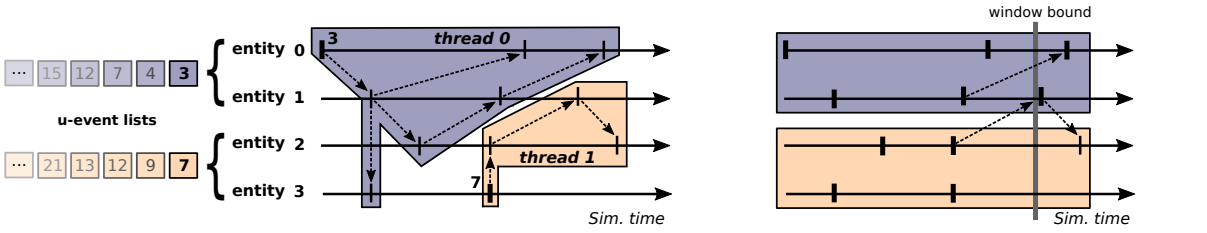


Figure 2: The Window Racer Algorithm. Each thread maintains a u-event list holding unconditional events for a subset of entities. In the *execute* phase, threads process event chains generated by local entities' unconditional events regardless for the thread assignment of newly generated events, up to a dynamically computed window bound. In the *commit* phase, each thread identifies local events that have become unconditional, which are inserted in the thread's u-event list for the next iteration.

3 THE WINDOW RACER ALGORITHM

When entity interactions are largely local according to some notion of proximity or are subject to sufficiently large temporal delays, classical asynchronous synchronization algorithms such as the null-message protocol or Time Warp can extract parallelism from periods in simulated time during which identifiable model segments operate independently. If, on the other hand, the entities' state changes are tightly coupled, these algorithms may struggle to advance efficiently in simulated time due to stalled PEs (conservative algorithms) or frequent rollbacks (optimistic algorithms).

The underlying design rationale of the Window Racer algorithm is to operate under the expectation of global entity interactions with little or no temporal delay. This assumption leads to an algorithm that operates optimistically to make progress without the need for guarantees on the delays between events, yet restricts the independent progress of PEs to limit the frequency and cost of rollbacks. In Window Racer, the simulation advances as a series of dynamically sized time windows negotiated among threads in a shared-memory setting. Transitive errors across PEs and the resulting need for antimessages are avoided entirely, which puts Window Racer in the category of risk-free algorithms [40].

Window Racer's design takes inspiration from Steinman's Breathing Time Buckets [46] and the S^{3}A algorithm [2], from which it differs in two features crucial to its performance.

- **Execution of newly generated events regardless of thread assignment.** Through fine-grained locking, threads directly and safely access the state of arbitrary entities. Hence, while threads are executing events, new events are never exchanged as messages or inserted in remote threads' event lists.
- **Rollbacks and identification of window bound based on entity-level straggler events.** The bound in simulation time up to which events can be committed is negotiated as part of the event execution. Although each thread holds event lists pertaining to an entire set of simulation entities, stragglers are identified on the level of individual entities, decreasing their frequency and cost.

The impact of these features over the existing algorithms is assessed in the ablation study presented in Section 4.2.

In the following, we describe Window Racer's round-based synchronous mode as shown in Figure 2. Pseudo code is given in Algorithms 1 and 2. In line with the concept of conditional knowledge [7], we distinguish *unconditional* and *conditional* events, i.e., events that are certain to occur at some point throughout the simulation, and tentative events whose creation may be an effect of mis-speculation. In the following, we refer to unconditional and conditional events as *u-events* and *c-events*. Each thread maintains its own u-event list (`uel`) holding unconditional events pertaining to the thread's assigned model segment according to the chosen partitioning. In addition, each thread holds a temporary c-event list (`cel`), which is cleared before each iteration of the algorithm. In contrast to the u-event list, a thread's c-event list can hold conditional events targeting local and remote entities.

Algorithm 1 Main loop of the Window Racer algorithm.

```
while not termination_criterion_reached():
  lower_bound = compute_global_minimum_ts()
  upper_bound = lower_bound + τ0

  # Execute phase
  parallel for thread in threads:
    while (thread.uel ∪ thread.cel).earliest_event().ts < upper_bound:
      ev = (thread.uel ∪ thread.cel).pop_earliest_event()
      ev.entity.try_execute(ev)

      # Local insertion regardless of entity's thread assignment
      thread.cel.insert(newly_created_events)

    cel.clear()

  barrier_synchronize()

  # Commit phase
  parallel for thread in threads:
    for entity in threads.entities:
      if len(entity.event_list) == 0 or
         entity.latest_state.timestamp ≥ upper_bound:
        entity.clear_event_list_and_state_list()
        continue

      entity.roll_back(latest_state_before_upper_bound)

      for event in entity.event_list:
        if event.creation_timestamp < upper_bound and
          event.timestamp ≥ upper_bound:
          thread.uel.insert(event)
          # All other events have already been committed
          # or can be discarded

      entity.clear_event_list_and_state_list()

  barrier_synchronize()
```

The algorithm proceeds in a series of rounds, each encompassing the two phases *execute* and *commit*. At the beginning of a round, the threads determine the initial upper bound of a global synchronization window in simulation time as the sum of the earliest timestamp among all events and an initial window size τ_0, which is a tuning parameter. In the *execute* phase, each thread iteratively executes the earliest event from the u-event or c-event list. When the current event schedules a new event, the new event is inserted into the thread's local c-event list. Importantly, while unconditional events always pertain to thread-local entities, conditional events may pertain to entities assigned to remote threads. It is particular to Window Racer that even remote conditional events are executed by the current thread. In order to avoid race conditions, the execution of remote entities' events requires mutual exclusion, which we attain by fine-grained locking on the entity level. Before a thread attempts to execute an event at a thread-local entity (unconditional events or conditional events) or a remote entity (conditional events only), the thread acquires a per-entity lock and appends the event to an unordered per-entity event list. Now, the thread determines whether the event can be executed in the current synchronization window. One of three possible situations is encountered:

(1) The event's timestamp lies within the current window and past the last state change of the destination entity. In this case, the current entity state is saved, and the event is executed.
(2) The event lies within the current window, but the entity state has already been changed by at least one event with a larger timestamp. To avoid the need for antimessages, any such previously executed events and any of their effects must be excluded from the current window. Hence, the window's upper bound is reduced to exclude all events with larger timestamps than the current event. Subsequently, the entity is rolled back to the earliest state prior to the current event's timestamp, and the event is executed.
(3) The event's timestamp lies beyond the current window. Then, the event is not executed.

Algorithm 2 Entity locking and update of the window bound when attempting to execute an event.

```
function entity::try_execute(event):
  entity.(lock)
  entity.event_list.append(event)

  # Upper window bound has decreased enough to exclude the event
  if event.ts ≥ upper_bound:
    entity.unlock()
    return

  # The event is a straggler
  if event.ts < entity.latest_change_timestamp:
    displaced_state = entity.earliest_state_after(event.ts)
    atomically_set(upper_bound = min(upper_bound, displaced_state.ts))
    entity.state = entity.latest_state_before(event.ts) # Rollback
    entity.execute(ev)
    entity.unlock()
    return

  entity.save_state()
  entity.execute(ev)
  entity.unlock()
```

In all cases, the entity is subsequently unlocked. The execute phase terminates once none of the threads hold any unconditional or conditional events earlier than the window bound.

In the *commit* phase, each thread visits each local entity with a non-empty event list, i.e., each entity subject to tentative state changes during the execute phase. If necessary, the entity is rolled back to the latest state earlier than the final window bound. Given the final window bound, the thread identifies those of the entity's events that have become unconditional. These events are certain to occur in the future but cannot be committed during the current window. This holds true for an event if it has been created earlier than the window bound yet holds a timestamp at or beyond the window bound. All other events in the entities' lists can be discarded as they have either already been committed as part of the entity state, will never occur, or will later be newly created as a consequence of an event from the u-event lists. Since the commit phase operates only on local entities and the thread-local u-event list, each thread carries out this phase independently.

4 EVALUATION

The evaluation of our implementation of Window Racer aims to answer three research questions:

- *How does Window Racer's performance compare to an established optimistic simulator based on Time Warp?*
- *How do Window Racer's main algorithmic features contribute to its performance?*
- *Can Window Racer provide substantial speedup for simulations of bio-chemical reaction networks?*

Our implementation of Window Racer was created from scratch using C++ and Linux pthreads. For random number generation, we employ the Xoroshiro128** generator [5]. The speedup measurements using PHold rely on our own C++ implementation as a baseline, which we found to deliver higher performance for this model than the sequential execution modes of ROOT-Sim and ROSS [6].

The measurements were carried out on a single machine equipped with a 16-core AMD Ryzen 9 7950X processor running Debian GNU/Linux 11. Where not otherwise specified, the parallel simulation runs were executed using the full 32 logical threads offered by the processor, which we observed to typically generate the highest performance in large model configurations. Frequency scaling on the operating system level was disabled, i.e., the CPU ran at least at its regular base frequency of 4.5 GHz, with a maximum boost frequency of 5.7 GHz.

The experiments with bio-chemical reaction networks were executed on a dual-socket system equipped with two 16-core Intel Xeon E5-2683 v4 CPUs running Ubuntu 22.04.1 LTS.

In Window Racer, each round's initial window size τ_0 (cf. Section 3) is set based on the final window size observed in the previous synchronization round, multiplied by a constant factor of 100 chosen arbitrarily. We found that the performance is affected only marginally even when decreasing or increasing this tuning parameter by an order of magnitude.

In the performance plots, each measurement represents an average of three runs.

4.1 Synthetic Benchmarks

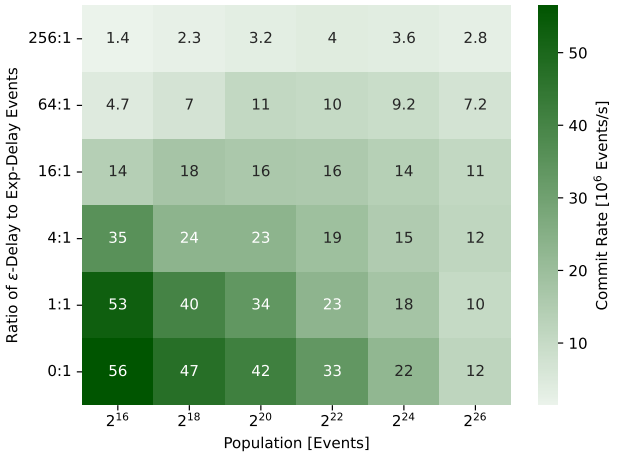
A basic experiment setup using the synthetic benchmark model PHold [16] would configure the desired number of entities and initialize each entity with a fixed number of initial events. However, the simulators may achieve varying performance when aggregating entities to different degrees by forming groups of entities considered jointly, which reduces the overhead compared to maintaining per-entity event lists. On the other hand, excessive aggregation can limit the exploitable concurrency and increase the probability and cost of rollbacks. Hence, we vary the aggregation level for ROOT-Sim, reporting the best-performing configuration. In PHold, an aggregation of entities by a factor of two is equivalent to halving the number of entities while keeping the overall number of events in the system constant. We account for the increase in state size through aggregation by allocating memory corresponding to one pseudo-random number generator state of 16 bytes per entity.

In our experiments, the destination entities for events are drawn uniformly at random while excluding the current entity, i.e., the proportion of remote events is set to 1. In addition to events delayed randomly according to a exponential distribution as in traditional PHold, we configure the ratio of events scheduled with near-zero delay. Given the current event's timestamp t, these *ϵ-delay* events are generated with the closest future timestamp $t + \epsilon$ representable as a double-precision floating point number. The choice to include ϵ-delay events rather than zero-delay events was made to focus our evaluation on the synchronization performance in the presence of tightly coupled transitions rather than the handling of simultaneous events, e.g., via forms of superdense time [33].

Figure 3 shows the measurement results when varying the overall number of events and the ratio of ϵ-delay events to exponential-delay events. For ROOT-Sim, we varied the aggregation level for each model configuration to obtain $2^8, 2^{10}, 2^{12}, \ldots, 2^{24}$ entities and show results for the best-performing aggregation level. Since we found Window Racer's event rate to almost always be highest without aggregation, we only show results with aggregation disabled.

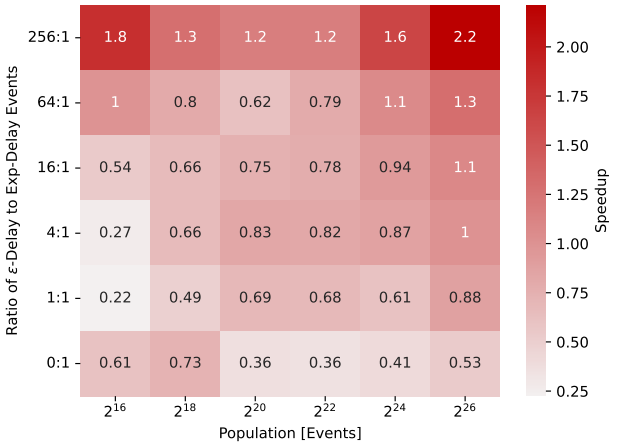
In the speedup results of Figure 3c, we observe that in many configurations, ROOT-Sim's best aggregation level outperformed Window Racer without aggregation. However, in model configurations with large populations and large ratios of ϵ-delay events, Window Racer exhibits better performance. This is consistent with an intuition of the algorithmic differences between the two simulators: with frequent ϵ-delay events, PEs in Time Warp can rarely advance without subsequent rollbacks, which may cascade. Window Racer's synchronous and fine-grained mode of execution results in less frequent and less costly rollbacks.

We now turn to the dependence of ROOT-Sim's performance on the aggregation level. Figure 4 shows Window Racer's speedup over ROOT-Sim depending on the aggregation level for four model configurations. We observe that ROOT-Sim's performance depends strongly on the aggregation level, with optimal sizes of entity groups ranging from 1 to 256 in these examples. These results show that an unfavorable choice can degrade the performance immensely, suggesting a need for model-specific configuration by the user or runtime adaptations. Window Racer achieves its best performance without aggregation and thus does not require this type of configuration.



(a) ROOT-Sim using the best-performing aggregation level.

(b) Window Racer without aggregation.



(c) Speedup of Window Racer over ROOT-Sim. Window Racer outperforms ROOT-Sim with high ϵ-delay ratios and large populations.

Figure 3: Comparing Window Racer and ROOT-Sim in the adapted PHold benchmark, varying the overall number of events and the ratio of ϵ-delay to exponential-delay events.

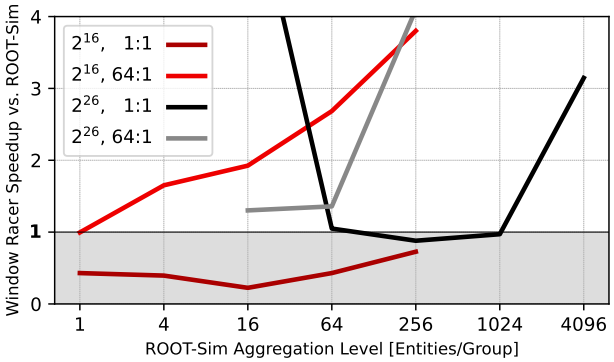
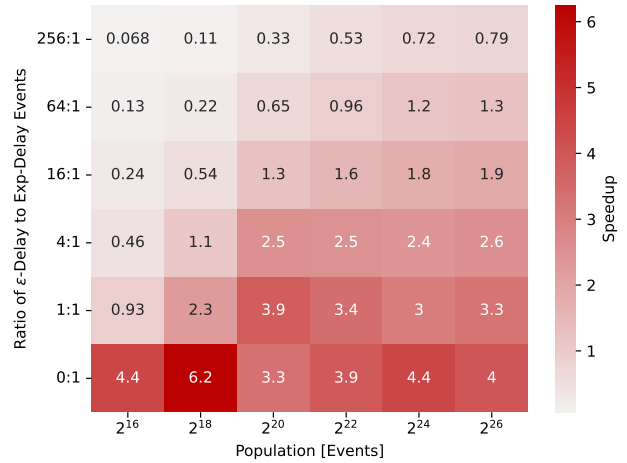


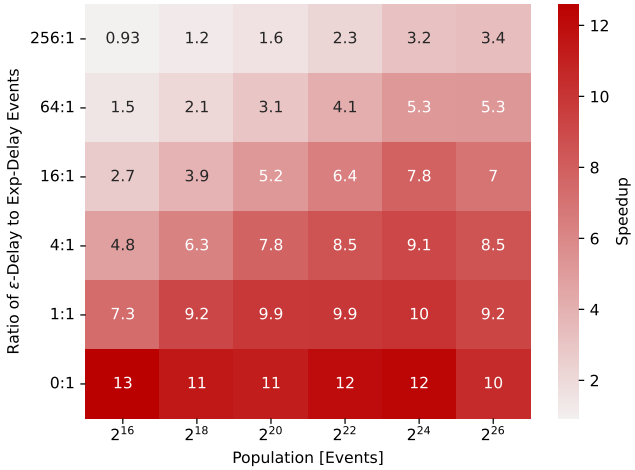
Figure 4: Speedup attained by Window Racer over ROOT-Sim with different aggregation levels. Each curve shows one PHold configuration defined by the population and the ratio of ϵ-delay to exponential-delay events. ROOT-Sim's performance depends severely on the choice of a suitable aggregation level, which varies with the model configuration.

Figure 5 shows the speedup achieved by Window Racer over sequential simulations using the same code base. We note that the sequential implementation employs an important optimization: any newly generated ϵ-delay event is executed immediately, i.e., without ever being inserted in an event list. For this reason, the sequential implementation performs particularly well with high ratios of ϵ-delay events. The results show that in a traditional PHold configuration, in which events are not associated with any computations apart from random number generation to determine the destination entity and time, the sequential simulations outperform Window Racer for model configurations with small populations or high ϵ-delay event ratios. In other cases, Window Racer achieved a speedup of up to 6.2. Since models used in real-world studies carry out non-zero computational work within events, we also show results with an artificial compute time of 1 μs per event. This low amount of computational load is sufficient for Window Racer to outperform the sequential simulator in almost all cases, with a maximum speedup of about 13.

To study the differences in the performance between the different simulators more closely, we profiled PHold configuration with a population of 2^{24} and a ratio of ϵ-to-exponential events of 64:1. For this model configuration, all simulators achieved similar commit rates between 8 and 9 million events per second. We consider three metrics: the Time Warp efficiency is the number of committed events divided by the number of events executed in total. The CPU utilization is computed as the process' user and kernel CPU time divided by the wall-clock execution time. Finally, parallel and distributed simulation algorithms can also differ in their energy consumption [14]. We report the processor's self-reported estimates in Joules obtained via the RAPL interface [9]. Table 1 shows the profiling results. The results follow the intuition on the different simulators: the sequential baseline utilizes only one of the CPU's hardware threads and completes the simulation using the lowest amount of energy. Comparing the parallel simulators, we see that Window Racer's more cautious form of speculation leads to higher



(a) No compute time per event.



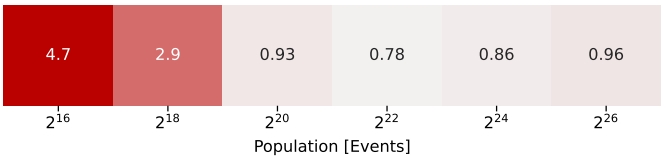
(b) Compute time 1 μs.

Figure 5: Speedup attained by Window Racer over a sequential execution with compute times of 0 and 1 μs per event. 1 μs of compute time per event is sufficient for substantial speedup in almost all model configurations.

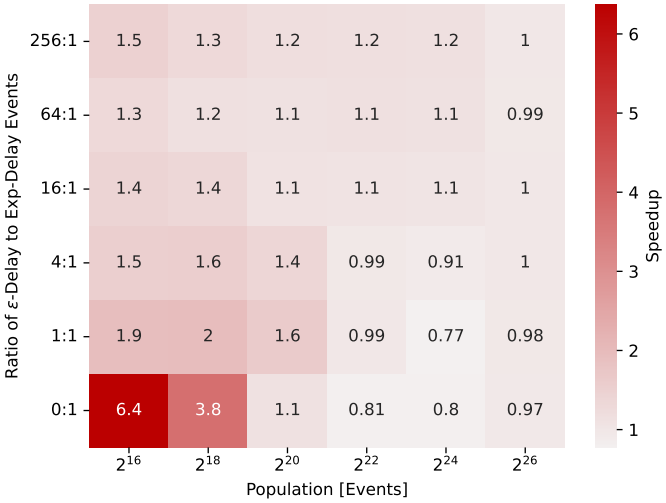
	Efficiency	CPU Utilization	Energy [J]
Sequential	100.0%	3.1%	501
ROOT-Sim	45.2%	95.8%	4194
Window Racer	74.4%	53.0%	3541

Table 1: Efficiency, utilization, and energy measurements with a population of 2^{24} and an ϵ-delay event ratio of 64:1.

efficiency compared to ROOT-Sim. Since threads can be stalled at barriers, Window Racer runs at a lower CPU utilization than ROOT-Sim's near full utilization and achieves a moderately lower energy consumption.



(a) Window Racer's speedup over Minimum Time Buckets (MTB). Model configurations with ϵ-delay events are omitted since MTB's performance degraded to miniscule event rates.



(b) Window Racer's speedup over S^{3}A. Window Racer is faster for small and medium populations and with frequent ϵ-delay events.

Figure 6: Ablation study: Window Racer's speedup over other algorithms executed using the same code base.

4.2 Ablation Study

We now investigate the determinants of Window Racer's performance. This is achieved by disabling or restricting its key algorithmic features so that Window Racer is reduced to two existing algorithms, while still relying on the same implementation.

We revisit the Window Racer's key features as previously discussed in Section 3:

(1) Execution of newly generated events regardless of thread assignment.
(2) Rollbacks and identification of window bound based on entity-level straggler events.

Disabling (1) yields the Minimum Time Buckets (MTB) algorithm, a shared-memory variant of the classical Breathing Time Buckets algorithm. In this configuration, any newly scheduled event targeting a remote thread is excluded from the current synchronization window, postponing its execution to the next round.

A behavior in line with the S^{3}A algorithm [2] is achieved as follows: firstly, we restrict (1) so that only new ϵ-delay events are immediately executed across thread boundaries, i.e., newly scheduled remote events with larger delay are postponed to the next window. Secondly, the state saving history for each entity used in (2) is limited to a single entry. This configuration follows S^{3}A's approach of determining synchronization windows that include at most one state update for each entity.

Figure 6 shows the results of the ablation study. For MTB, scheduling an ϵ-delay event targeting a remote thread truncates the synchronization window at the current event's timestamp. We found that in these model configurations, an average of little more than one event was committed per synchronization round, making MTB unsuitable for models with ϵ-delay events. Since the commit rate degraded to only about 30 000 events per second independently of the population size, we exclude these results from the plot. However, without ϵ-delay events, MTB was able to somewhat outperform Window Racer for populations of 2^{20} and beyond. In these model configurations, the simulation offers sufficient parallelism so that Window Racer's fine-grained thread interactions provide no further benefit. On the other hand, for smaller populations, Window Racer achieved a speedup of up to 4.7 over MTB.

S^{3}A was competitive across a wider range of model configurations, somewhat outperforming Window Racer for combinations of large populations with low ϵ-delay event ratios. The trend of the results is similar to MTB in that with sufficiently large populations, Window Racer's immediate execution of event chains across thread boundaries is not necessary to extract sufficient parallelism. Window Racer again provides performance improvements particularly for smaller populations, with the highest speedup of 6.4 with a population of 2^{16} and without ϵ-delay events.

Overall, we see that Window Racer is more efficient in its extraction of parallelism than the other synchronous algorithms, extending the range of gainfully parallelizable simulations towards smaller scenarios.

4.3 Bio-Chemical Reaction Networks

In the following, we study Window Racer's performance when simulating models of spatial bio-chemical reaction networks using the Next Reaction Method (cf. Section 2.3). In our implementation, each simulation entity represents one type of reaction. We distinguish two types of events: the first type corresponds to a reaction firing, representing a state transition. The second type instantaneously notifies other entities of a firing, signaling the need to update their tentative firing times. This explicit signaling is used to avoid state sharing among entities. Instead, each entity maintains a copy of the relevant state, typically comprising the integer-valued population of one or two species.

We employ a specialized u-event list (cf. Section 3) that permits rescheduling, i.e., when a firing is scheduled and one has already been scheduled for the same reaction type, the original firing is replaced in the u-event list. Furthermore, we reduce the number of rescheduling operations by immediately discarding firings known to be outdated according to a per-reaction attribute that holds the next firing time. Using this approach, the commit phase (re)schedules at most one operation per reaction type.

As performance baselines, we employ a pool of state-of-the-art and custom sequential simulators. The BioSimulators interface [41] provides a standardized means to execute models using various different simulators. We carried out our measurements using the established simulators BioNetGen [12], libroadrunner [43], pSSAlib [34],

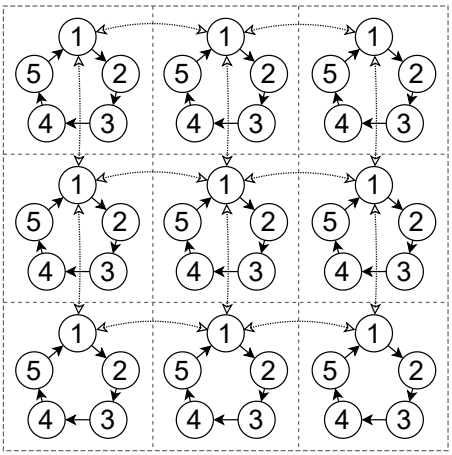


Figure 7: Illustration of the bio-chemical reaction network model. The dashed grid represents the subvolumes, circles and arrows indicate species and reactions. Dotted arrows signify diffusion transitions among subvolumes.

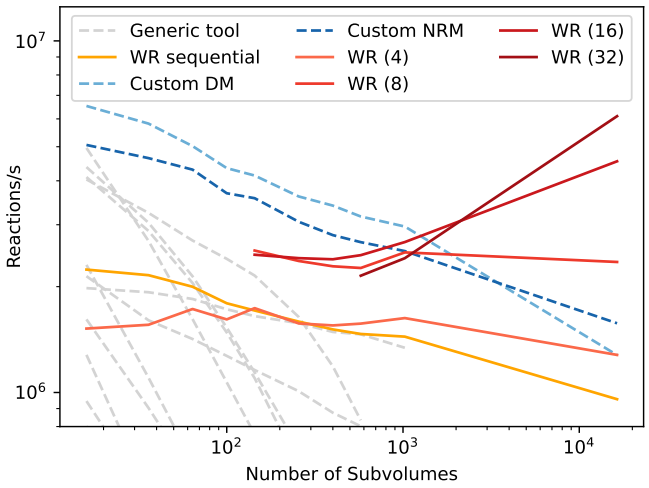


Figure 8: Performance results for the reaction network with weak coupling. Dashed lines: sequential simulators. Blue dashed lines: custom sequential implementations of NSM and DM. Red: Window Racer, varying the number of threads.

ML-Rules [31], COPASI [23] and GillesPy [1]. Furthermore, we implemented two custom minimalist sequential simulators using the Logarithmic Direct Method [28] and the Next Reaction Method with data structures optimized for large systems. The latter two were found to perform best among the sequential simulators, likely due to their restricted functionality and scope as well as highly optimized data structures.

Our performance measurements were carried out for the synthetic spatial reaction network illustrated in Figure 7. In the model, each subvolume contains five types of species that cyclically transform into one another ($S1 \rightarrow S2$, $S2 \rightarrow S3, \ldots$). The first species type in each subvolume can diffuse to the neighboring rectangular

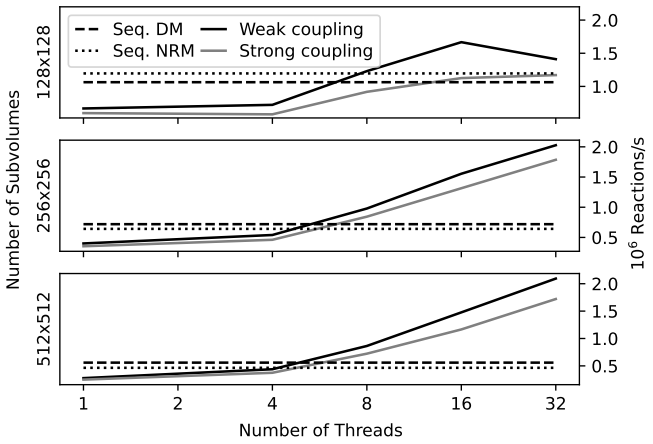


Figure 9: Window Racer's performance for large reaction networks with two different coupling levels among subvolumes. The horizontal lines indicate the two fastest sequential simulators, whose performance is independent of the coupling.

areas. In our performance results, we report reaction firings per second wall-clock time. Each firing entails about 8.5 events on average, comprised of the firing event itself and the subsequent notifications for the dependent reaction types.

The first experiment uses a reaction network with a low degree of coupling among subvolumes, resulting in only one in five hundred events crossing the subvolume boundaries. The results are shown in Figure 8. We observe that Window Racer outperforms the sequential simulators for larger numbers of subvolumes. Beyond 2 000 subvolumes, speedup is attained over all sequential simulators with 16 and 32 threads.

In the second experiment, the model is configured with a much stronger degree of coupling, diffusion occurring at either the same or double the rate of the regular reactions, which is more representative of typical spatial bio-chemical network models [29]. The results are shown in Figure 9. In these configurations, significantly larger networks are required to achieve speedup over the sequential case. Furthermore, the degree of coupling has a pronounced effect on the performance. Increasing the number of threads steadily increases performance, indicating that Window Racer could scale beyond the available 32 threads. For the largest reaction network, we observed a maximum speedup of 8 over the fastest sequential simulator.

Overall, we observe that Window Racer is able to substantially outperform sequential simulators for spatial bio-chemical reaction networks given sufficiently large network sizes. The benefits seen in practical studies will depend on the need for such extensive scenarios and on the degree of coupling in the modeled system.

5 CONCLUSIONS AND FUTURE WORK

We presented Window Racer, a synchronous risk-free optimistic synchronization algorithm for discrete-event simulations on shared-memory systems. Window Racer is able to extract speedup from models of tightly coupled systems with global and unpredictable entity interactions. Its defining characteristic and main driver of its performance benefits is the immediate execution of event chains without regard for the events' thread assignment, which is achieved

through fine-grained locking and atomic operations. In measurements using the PHold benchmark model, Window Racer demonstrated particular benefits compared to an established simulator based on Time Warp in model configurations with large numbers of entities and with frequent near-instantaneous interactions. Compared to other synchronous optimistic algorithms, Window Racer is able to extract speedup from smaller scenarios. We also showed that Window Racer accelerates spatial simulations of bio-chemical reaction networks, a challenging class of models for parallelization.

In future work, models with uneven load distribution could be better supported by exploiting Window Racer's loose binding of entities to threads. This could take the form of a gradual repartitioning in which already scheduled events are still executed according to the previous entity assignment, or of work stealing whenever threads become idle. To combine the benefits of Window Racer's cautious form of speculative execution with the more aggressive Time Warp, we are currently exploring a joint synchronous/asynchronous simulation approach [39]. Finally, similarly to the S^3A algorithm [3], Window Racer's synchronous execution scheme may lend itself to further acceleration via many-core processors such as graphics processing units.

ACKNOWLEDGMENTS

We extend our thanks to Andrea Piccione for his thoughtful suggestions on an early version of Window Racer and to Leon Herrmann for providing the synthetic bio-chemical reaction network models. Financial support was provided by the Deutsche Forschungsgemeinschaft (DFG) research grant ESCeMMo (UH-66/13, 225222086).

REFERENCES

[1] John H. Abel, Brian Drawert, Andreas Hellander, and Linda R. Petzold. 2016. GillesPy: A Python Package for Stochastic Model Building and Simulation. *IEEE Life Sciences Letters* 2, 3 (Sept. 2016), 35–38. Conference Name: IEEE Life Sciences Letters.

[2] Philipp Andelfinger and Adelinde Uhrmacher. 2021. Optimistic parallel simulation of tightly coupled agents in continuous time. In *2021 IEEE/ACM 25th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*. IEEE, 1–9.

[3] Philipp Andelfinger and Adelinde M Uhrmacher. 2023. Synchronous speculative simulation of tightly coupled agents in continuous time on CPUs and GPUs. *SIMULATION* (2023), 00375497231158930.

[4] Pavol Bauer, Jonatan Lindén, Stefan Engblom, and Bengt Jonsson. 2015. Efficient inter-process synchronization for parallel discrete event simulation on multicores. In *Proceedings of the 3rd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*. 183–194.

[5] David Blackman and Sebastiano Vigna. 2021. Scrambled linear pseudorandom number generators. *ACM Transactions on Mathematical Software (TOMS)* 47, 4 (2021), 1–32.

[6] Christopher D Carothers, David Bauer, and Shawn Pearce. 2002. ROSS: A high-performance, low-memory, modular Time Warp system. *J. Parallel and Distrib. Comput.* 62, 11 (2002), 1648–1669.

[7] K Mani Chandy and Jay Misra. 2006. *Conditional Knowledge as a basis for Distributed Simulation*. Technical Report. California Institute of Technology, Pasadena. Department of Computer Science.

[8] Li-li Chen, Ya-shuai Lu, Yi-ping Yao, Shao-liang Peng, et al. 2011. A well-balanced Time Warp system on multi-core environments. In *2011 IEEE Workshop on Principles of Advanced and Distributed Simulation*. IEEE, 1–9.

[9] Howard David, Eugene Gorbatov, Ulf R Hanebutte, Rahul Khanna, and Christian Le. 2010. RAPL: Memory power estimation and capping. In *ACM/IEEE International Symposium on Low power Electronics and Design*. 189–194.

[10] Cristian Dittamo and Davide Cangelosi. 2009. Optimized parallel implementation of gillespie's first reaction method on graphics processing units. In *2009 International Conference on Computer Modeling and Simulation*. IEEE, 156–161.

[11] Johan Elf and Måns Ehrenberg. 2004. Spontaneous separation of bi-stable biochemical systems into spatial domains of opposite phases. *Systems biology* 1, 2 (2004), 230–236.

[12] James R. Faeder, Michael L. Blinov, and William S. Hlavacek. 2009. Rule-Based Modeling of Biochemical Systems with BioNetGen. In *Systems Biology*, Ivan V. Maly (Ed.). Vol. 500. Humana Press, Totowa, NJ, 113–167. Series Title: Methods in Molecular Biology.

[13] Alois Ferscha. 1995. Probabilistic adaptive direct optimism control in Time Warp. In *Proceedings of the Workshop on Parallel and Distributed Simulation*. 120–129.

[14] Richard Fujimoto and Aradhya Biswas. 2015. An empirical study of energy consumption in distributed simulations. In *International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*. IEEE, 163–170.

[15] Richard M Fujimoto. 1989. *Time Warp on a shared memory multiprocessor*. Technical Report. Utah University, Salt Lake City. School of Computing.

[16] Richard M Fujimoto. 1990. Performance of Time Warp under synthetic workloads. In *Proceedings of the SCS Multiconference on Distributed Simulations, 1990*, Vol. 22. 23–28.

[17] Richard M Fujimoto. 2001. Parallel and distributed simulation systems. In *Proceeding of the 2001 Winter Simulation Conference*, Vol. 1. IEEE, 147–157.

[18] Kaushik Ghosh and Richard M Fujimoto. 1994. *Parallel discrete event simulation using space-time memory*. Technical Report. Georgia Institute of Technology.

[19] Michael A Gibson and Jehoshua Bruck. 2000. Efficient exact stochastic simulation of chemical systems with many species and many channels. *The Journal of Physical Chemistry A* 104, 9 (2000), 1876–1889.

[20] Daniel T Gillespie. 1976. A general method for numerically simulating the stochastic time evolution of coupled chemical reactions. *Journal of computational physics* 22, 4 (1976), 403–434.

[21] Daniel T Gillespie. 2007. Stochastic simulation of chemical kinetics. *Annu. Rev. Phys. Chem.* 58 (2007), 35–55.

[22] Arthur P. Goldberg, David R. Jefferson, John A. P. Sekar, and Jonathan R. Karr. 2020. Exact Parallelization of the Stochastic Simulation Algorithm for Scalable Simulation of Large Biochemical Networks.

[23] Stefan Hoops, Sven Sahle, Ralph Gauges, Christine Lee, Jürgen Pahle, Natalia Simus, Mudita Singhal, Liang Xu, Pedro Mendes, and Ursula Kummer. 2006. COPASI—a COmplex PAthway SImulator. *Bioinformatics* 22, 24 (Dec. 2006), 3067–3074.

[24] Mauro Ianni, Romolo Marotta, Davide Cingolani, Alessandro Pellegrini, and Francesco Quaglia. 2018. The ultimate share-everything PDES system. In *ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*. 73–84.

[25] David Jefferson and Henry Sowizral. 1982. *Fast concurrent simulation using the Time Warp mechanism. Part I. Local control*. Technical Report. Rand Corporation, Santa Monica, CA.

[26] Matthias Jeschke, Roland Ewald, Alfred Park, Richard Fujimoto, and Adelinde M Uhrmacher. 2008. A parallel and distributed discrete event approach for spatial cell-biological simulations. *ACM SIGMETRICS Performance Evaluation Review* 35, 4 (2008), 22–31.

[27] Margaret E Johnson, Athena Chen, James R Faeder, Philipp Henning, Ion I Moraru, Martin Meier-Schellersheim, Robert F Murphy, Thorsten Prüstel, Julie A Theriot, and Adelinde M Uhrmacher. 2021. Quantifying the roles of space and stochasticity in computer simulations for cell biology and cellular biochemistry. *Molecular Biology of the Cell* 32, 2 (2021), 186–210.

[28] Hong Li and Linda Petzold. 2006. Logarithmic direct method for discrete stochastic simulation of chemically reacting systems. *Journal of Chemical Physics* 16 (2006), 1–11.

[29] Jonatan Linden, Pavol Bauer, Stefan Engblom, and Bengt Jonsson. 2019. Exposing inter-process information for efficient pdes of spatial stochastic systems on multicores. *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 29, 2 (2019), 1–25.

[30] Nazzareno Marziale, Francesco Nobilia, Alessandro Pellegrini, and Francesco Quaglia. 2016. Granular Time Warp objects. In *Proceedings of the 2016 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*. 57–68.

[31] Carsten Maus, Stefan Rybacki, and Adelinde M. Uhrmacher. 2011. Rule-based multi-level modeling of cell biological systems. *BMC Systems Biology* 5, 1 (Oct. 2011), 166.

[32] Horst Mehl and Stefan Hammes. 1995. How to integrate shared variables in distributed simulation. *ACM SIGSIM Simulation Digest* 25, 2 (1995), 14–41.

[33] James Nutaro. 2020. Toward a theory of superdense time in simulation models. *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 30, 3 (2020), 1–13.

[34] Oleksandr Ostrenko, Pietro Incardona, Rajesh Ramaswamy, Lutz Brusch, and Ivo F. Sbalzarini. 2017. pSSAlib: The partial-propensity stochastic chemical network simulator. *PLOS Computational Biology* 13, 12 (April 2017), e1005865. Publisher: Public Library of Science.

[35] Avinash C Palaniswamy and Philip A Wilsey. 1994. Scheduling Time Warp processes using adaptive control techniques. In *Proceedings of Winter Simulation Conference*. IEEE, 731–738.

[36] Alessandro Pellegrini, Sebastiano Peluso, Francesco Quaglia, and Roberto Vitali. 2016. Transparent speculative parallelization of discrete event simulation applications using global variables. *International journal of parallel programming* 44 (2016), 1200–1247.

[37] Alessandro Pellegrini and Francesco Quaglia. 2019. Cross-state events: A new approach to parallel discrete event simulation and its speculative runtime support. *Journal of parallel and distributed computing* 132 (2019), 48–68.

[38] Alessandro Pellegrini, Roberto Vitali, and Francesco Quaglia. 2012. The rome optimistic simulator: Core internals and programming model. In *4th International ICST Conference on Simulation Tools and Techniques*.

[39] Andrea Piccione, Philipp Andelfinger, and Alessandro Pellegrini. 2023. Hybrid Speculative Synchronisation for Parallel Discrete Event Simulation. In *Proceedings of the 2023 SIGSIM Conference on Principles of Advanced Discrete Simulation (SIGSIM-PADS '23)*. ACM, New York, NY, USA, 12 pages.

[40] Paul F Reynolds Jr. 1988. A spectrum of options for parallel simulation. In *Proceedings of the 20th conference on Winter simulation*. 325–332.

[41] Bilal Shaikh, Lucian P Smith, Dan Vasilescu, Gnaneswara Marupilla, Michael Wilson, Eran Agmon, Henry Agnew, Steven S Andrews, Azraf Anwar, Moritz E Beber, et al. 2022. BioSimulators: a central registry of simulation engines and services for recommending specific tools. *Nucleic acids research* 50, W1 (2022), W108–W114.

[42] L Sokol. 1988. MTW: A strategy for scheduling discrete simulation events for concurrent execution. *Distributed Simulation'88* (1988), 34–42.

[43] E. T. Somogyi, M. T. Karlsson, M. Swat, M. Galdzicki, and H. M. Sauro. 2013. libRoadRunner: A High Performance SBML Compliant Simulator. Pages: 001230 Section: New Results.

[44] Jeff S Steinman. 1992. SPEEDES-A multiple-synchronization environment for parallel discrete-event simulation. *International Journal in Computer Simulation;(United States)* 2 (1992).

[45] Jeff S Steinman. 1993. Breathing Time Warp. In *Proceedings of the seventh workshop on Parallel and distributed simulation*. 109–118.

[46] Jeff S Steinman. 1994. Discrete-event simulation and the event horizon. *ACM SIGSIM Simulation Digest* 24, 1 (1994), 39–49.

[47] Vo Hong Thanh and Roberto Zunino. 2011. Parallel stochastic simulation of biochemical reaction systems on multi-core processors. *Proceedings of CSSIM* (2011), 162–170.

[48] Stephen J Turner and Ming Q Xu. 1990. *Performance evaluation of the bounded Time Warp algorithm*. University of Exeter, Department of Computer Science.

[49] Bing Wang, Bonan Hou, Fei Xing, and Yiping Yao. 2011. Abstract next subvolume method: A logical process-based approach for spatial stochastic simulation of chemical reactions. *Computational Biology and Chemistry* 35, 3 (2011), 193–198.

[50] Bing Wang, Yiping Yao, Yuliang Zhao, Bonan Hou, and Shaoliang Peng. 2009. Experimental analysis of optimistic synchronization algorithms for parallel simulation of reaction-diffusion systems. In *2009 International Workshop on High Performance Computational Systems Biology*. IEEE, 91–100.

[51] Jun Wang and Carl Tropper. 2007. Optimizing Time Warp simulation with reinforcement learning techniques. In *Winter Simulation Conference*. IEEE, 577–584.

[52] Xuehui Wang and Lei Zhang. 2017. Design and Analysis of a Minimum Time Buckets Synchronization Algorithm for Parallel and Distributed Simulation. In *International Symposium on Autonomous Decentralized System (ISADS)*. IEEE, 96–103.