UPPSALA
UNIVERSITET

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 1634

Synchronization Techniques in Parallel Discrete Event Simulation

JONATAN LINDÉN

ACTA
UNIVERSITATIS
UPSALIENSIS
UPPSALA
2018

ISSN 1651-6214
ISBN 978-91-513-0241-6
urn:nbn:se:uu:diva-342270

Dissertation presented at Uppsala University to be publicly examined in 2446, ITC, Lägerhyddsvägen 2, Uppsala, Tuesday, 10 April 2018 at 13:15 for the degree of Doctor of Philosophy. The examination will be conducted in English. Faculty examiner: Professor Christopher D. Carothers (Rensselaer Polytechnic Institute, Department of Computer Science).

Abstract

Lindén, J. 2018. Synchronization Techniques in Parallel Discrete Event Simulation. *Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology* 1634. 57 pp. Uppsala: Acta Universitatis Upsaliensis. ISBN 978-91-513-0241-6.

Discrete event simulation is an important tool for evaluating system models in many fields of science and engineering. To improve the performance of large-scale discrete event simulations, several techniques to parallelize discrete event simulation have been developed.

In parallel discrete event simulation, the work of a single discrete event simulation is distributed over multiple processing elements. A key challenge in parallel discrete event simulation is to ensure that causally dependent events are processed in the correct order, so that the same simulation trajectory is produced as in a sequential simulation. To preserve chronology between events processed in parallel, different synchronization protocols have been devised, each carrying a cost in performance.

This thesis presents techniques for reducing synchronization costs in two approaches to parallel discrete event simulation, viz., optimistic space-parallel and share-everything parallel discrete event simulation.

Firstly, we develop a concurrent priority queue, to be used as, e.g., a central event queue in the share-everything approach to parallel discrete event simulation. The priority queue is based on skiplists. It improves over previous queues by incurring fewer global synchronization operations, thereby inducing less contention and improving scalability.

Secondly, we study how to improve the performance of optimistic parallel discrete event simulation by disseminating accurate estimates of timestamps of future events. We present techniques for obtaining the estimates in two different methods for simulation of spatial stochastic models. The estimates allow processing elements to determine when to pause simulation with high precision, thereby reducing the amount of performed useless work.

Finally, we observe that in the applications that we have studied, the phenomena of interest are often non-homogeneous and migrate over time. Due to this, the work distribution tends to become unbalanced among the processing elements. A solution is to rebalance the work dynamically. We propose a fine-grained local dynamic load balancing algorithm for parallel discrete event simulation. The load balancing algorithm reduces the number of events arriving out-of-order, thereby reducing the amount of time spent on corrective actions.

Keywords: Parallel discrete event simulation, Discrete event simulation, PDES, Optimism control

Jonatan Lindén, Department of Information Technology, Computer Systems, Box 337, Uppsala University, SE-75105 Uppsala, Sweden.

ISSN 1651-6214
ISBN 978-91-513-0241-6
urn:nbn:se:uu:diva-342270 (http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-342270)

List of papers

This thesis is based on the following papers, which are referred to in the text by their Roman numerals.

I **A Skiplist-Based Concurrent Priority Queue with Minimal Memory Contention.**
J. Lindén and B. Jonsson. Tech. rep. 2018–003. Dept. of Information Technology, Uppsala University, 2018.

REVISED AND EXTENDED VERSION OF:
A Skiplist-Based Concurrent Priority Queue with Minimal Memory Contention.
J. Lindén and B. Jonsson. *Principles of Distributed Systems*. OPODIS '13, LNCS, vol. 8304. Springer, 2013, pp. 206–220.

II **Efficient Inter-Process Synchronization for Parallel Discrete Event Simulation on Multicores.** P. Bauer, J. Lindén, S. Engblom, B. Jonsson. *Proc. 3rd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation.* SIGSIM-PADS '15, ACM, 2015, pp. 183–194, doi: https://doi.org/10.1145/2769458.2769476.

III **Exposing Inter-Process Information for Efficient PDES of Spatial Stochastic Systems.** J. Lindén, P. Bauer, S. Engblom, B. Jonsson. Under submission.

REVISED AND EXTENDED VERSION OF:
Exposing Inter-Process Information for Efficient Parallel Discrete Event Simulation of Spatial Stochastic Systems. J. Lindén, P. Bauer, S. Engblom, B. Jonsson. *Proc. 2017 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation.* SIGSIM-PADS '17. ACM, 2017, pp. 53–64.

IV **Fine-Grained Local Dynamic Load Balancing in PDES.**
J. Lindén, P. Bauer, S. Engblom, B. Jonsson. Under submission.

Reprints were made with permission from the publishers.

Comments on my Participation

I I am the principal author and the principal investigator.

II I am, together with Pavol Bauer, the principal author and the principal investigator.

III I am, together with Pavol Bauer, the principal author and the principal investigator.

IV I am the principal author and the principal investigator.

Other Publications

Predicting the Cost of Lock Contention in Parallel Applications on Multicores using Analytic Modeling.
X. Pan, J. Lindén, B. Jonsson. 5th Swedish Workshop on Multicore Computing, MCC ’12, 2012.

Contents

1 Introduction

Parallel computing is ubiquitous in today's society, appearing in everything from smartphones to supercomputers. One type of parallel application that has received much attention is computer simulation. Computer simulation is the process of emulating some real or imagined system over time. There are numerous applications of simulation; weather forecasting, traffic and road network design, integrated circuit design, modeling of biological systems, and many more. Simulation can have several goals, such as predicting disease spread in epidemics or verification of integrated circuit design.

One important type of simulation is *discrete event simulation* (DES). DES is typically concerned with the simulation of systems where changes occur instantaneously and in discrete steps. In, e.g., epidemics, where a population can be modeled as consisting of a number of healthy and infected individuals, the spread of infectious diseases can be considered to be such a system: an infection of a healthy individual occur instantaneously, and the infection reduces the number of healthy individuals and increases the number of infected individuals by a discrete step.

A DES evolves by the occurrence of *events*. Each event causes a discrete change in the state of the simulation model and is considered to take place instantaneously. In the above example from epidemics, such an event is the infection of an individual. Each event is associated with a timestamp, indicating the instant in time the event is considered to occur. A discrete event simulator typically maintains a *state*, which represents the state of the simulated system, a collection of scheduled future events, usually stored in an *event queue*, and a *clock* representing the time in the simulation [10]. A discrete event simulation of a model proceeds by repeatedly processing (and removing) the event with the earliest timestamp in the event queue: updating the simulation state according to the type of the event, setting the clock to the time of the event, and scheduling new events that become enabled by the processed event. The newly created events model causal relationships in the system, e.g., an event describing the infection of an individual may induce the creation of a future event describing when the same individual recovers.

The proliferation of increasingly more complex systems that need verification or evaluation, and a demand for more detailed simulations, has driven the development of parallel simulation. Simulating large complex systems by sequential execution would in many contexts take too long time. Instead, the simulation is distributed onto many processing elements (e.g., processors or multiprocessor cores) in parallel to reduce the execution time.

This thesis was motivated by the development of a parallel simulator for applications in systems biology. For many systems being studied in systems biology, e.g., intracellular systems, spatial and stochastic effects that depend on small molecule numbers are important. Therefore, stochastic models are used for modeling such systems, rendering analytic solutions difficult to obtain or even infeasible. Instead, to learn about the behavior of a stochastic (and spatial) system, we can use methods based on simulation. By simulating a stochastic model many times, using computer-generated "random" numbers to mimic the stochasticity of the system, we obtain many different simulation trajectories. We can then ask questions about the average behavior of the system through these trajectories, e.g., what is the mean duration of an oscillation in the system, or, what is the mean time before the system switches state. Parallelization can help speeding up these otherwise time-consuming discrete event simulations. There are several tools for simulation of spatial stochastic models, of which one is the URDME framework [16]. The techniques developed in this thesis aim at providing URDME with an efficient parallel simulator. The models which are studied in this thesis have been taken from systems biology, but can be applied to many other fields as well.

Parallel DES (PDES) is concerned with distributing the work of a sequential DES onto multiple processing elements, which collaborate to perform the simulation faster. Simulation models often contain plenty of potential parallelism, since different parts of a model typically only causally interfere with a few other parts of the model, while the remaining parts are unaffected. However, exploiting the parallelism has proven to be surprisingly hard. The core challenge in PDES is to ensure that causally dependent events are processed in the correct order, even though they are processed on different processing elements. (Un)fortunately, the causality between events is dynamic and hard to predict. PDES has been actively developed in academia during the last three decades, and a rich research literature has been developed [6, 11, 24, 41, 42].

There are multiple approaches to parallelization of DES. The approach that has received the most attention is arguably *space-parallel* simulation. Space-parallel simulation is done by partitioning the model state and distributing the partitions onto so-called *logical processes*. The logical processes are then executed in parallel on multiple processing elements, each one evolving its partition along a local simulation time axis. Events that affect the partition of another logical process are exchanged. Upon receipt of an event at a logical process, it is incorporated at the right time into the local time axis. Since each logical process evolves its time axis independently, such an event may cause a causality error: the received event may have an earlier timestamp than the current time axis. There exist several synchronization protocols to handle local causality errors in space-parallel simulation, but all of them have a cost in performance [24].

Another approach to PDES that has received more attention lately, with the advent of multicore processors, is *share-everything* PDES. Here, all processing elements typically share the complete model state and a single central event queue. Hence an efficient design of the event queue is crucial, to prevent it from becoming a bottleneck. On the other hand, synchronization between the threads is simplified taking place implicitly through the event queue, consequently reducing the risk of causality errors.

This thesis develops techniques for improving the performance of PDES executing on shared-memory computer architectures. More specifically, the techniques developed are intended to improve the parallel efficiency by reducing the synchronization cost in space-parallel and share-everything PDES of spatial stochastic models, to which three different approaches are taken.

In share-everything PDES, the central event queue is frequently accessed by multiple processing elements concurrently. Removing the earliest event from the event queue becomes a hot-spot, as concurrent accesses try to remove the same element and must be synchronized. How can the synchronization cost of concurrent accesses for the event queue in a share-everything PDES be reduced? This question is addressed in Paper I.

In space-parallel PDES, one way of reducing the synchronization cost is to ensure that simulation time progresses at approximately the same rate at each logical process, thereby reducing the risk of costly causality errors. If a logical process could know timestamps of future incoming events, it would be able to optimally advance its local simulation. How can information about future events be extracted and disseminated efficiently in stochastic spatial simulation? This question is addressed in Papers II and III.

In space-parallel PDES, one issue that causes a high synchronization cost uneven distribution of work over the processing elements, especially if the distribution of work varies with time. By dynamically balancing the load between logical processes during the simulation, can the synchronization cost in PDES be reduced? This question is addressed in Paper IV.

Thesis Organization

The remainder of this thesis introduction is organized as follows. First, a brief introduction to DES follows in Chapter 2. Then, in Chapter 3, the main synchronization protocols of PDES are surveyed, and techniques related to the protocols are discussed. At the end of the chapter, challenges in PDES addressed by this thesis are introduced. This thesis concerns PDES of spatial stochastic models with applications in, e.g., systems biology. These systems and related simulation methods are presented in Chapter 4. At the end of the chapter, we present the challenges that are inherent to the application. A concurrent data structure suitable for use as a central event queue in, e.g., share-everything PDES, is described in Chapter 5. At the end of the chapter, challenges related to concurrent event queues are presented. In each of Chapters 3 to 5, related work is discussed throughout the text. Summaries of the included papers follow in Chapter 6, before we conclude in Chapter 7.

2 Discrete Event Simulation

In this chapter, we present some brief preliminaries of simulation and DES.

Simulation can be broadly classified into two categories. *Discrete event simulation* (DES), also known as event-driven simulation, is concerned with the simulation of models, where the state of the system is viewed as a discrete quantity. In such a model, changes to the model state necessarily occur instantaneously and in discrete steps even though such an assumption may not exactly reflect the process being modeled. The time evolution of an example discrete system with state $x(t)$ is shown in Figure 2.1(c). On the other hand, in *discrete time simulation* (also known as continuous simulation, time-driven, or time-step simulation) time progresses in discrete steps, and the state of the simulated system is considered to change continuously, evaluated at the instant of the discrete time-steps. In Figure 2.1, examples of continuous and discrete systems are shown. In Figure 2.1(a) and Figure 2.1(b), a continuous system and its example discrete time simulation are shown. The quantity Δt in Figure 2.1(b) does not have to be constant, and may change during the simulation.

The simulation methods are suitable for different types of problems. Discrete time simulation is suitable when the complete state of the simulated model changes continuously, whereas discrete event simulation is suitable when only a part of the state changes, at irregular intervals. In many cases, event-based approaches can also be used for modeling continuous systems [55], e.g., to reduce computational complexity. In that case, the continuous state is transformed, discretized, into a discrete state space.

Discrete models are most easily described through the possible transitions that may occur in them; e.g., in an epidemics simulation, a person may become infected, which is reflected by a change of infected and healthy individuals

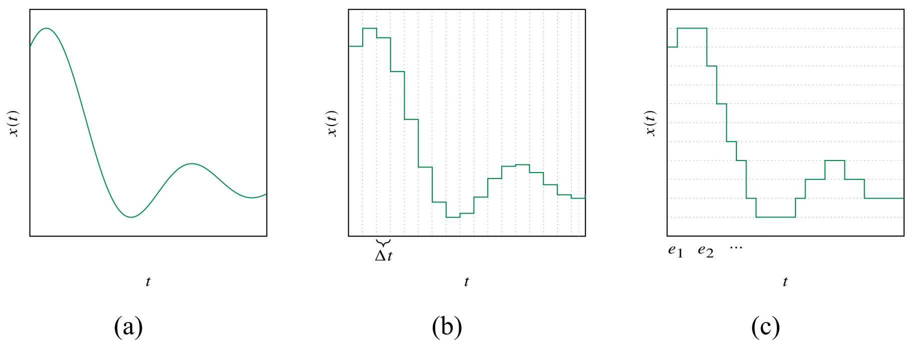

Figure 2.1. A continuous system (a), and its time-step simulation with a fixed step-size of Δt (b). In (c), we see the time evolution of a discrete system, which corresponds to *one* discretization of the state space of the system in (a). $e_1, e_2, \ldots$ denotes the events that take place during the process, and that would be part of a corresponding discrete simulation.

in the state. In the simulated system, each transition is considered to occur instantaneously at a specific instant, which constitutes an *event*. An event may trigger new events with timestamps greater than that of the triggering event. Thus, a simulation can be seen as the processing of a time-ordered sequence of timed transitions.

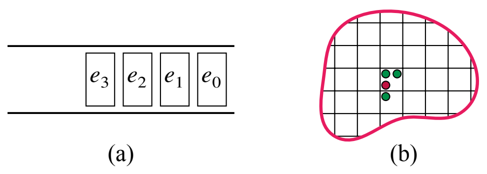

Figure 2.2. Schematic drawing of a DES, consisting of an event queue (a), and a model state (b). There are four events, $e_0, \ldots, e_3$ in the event queue. The model state depicted could, for example, be the discretization of a two-dimensional space, and the colored dots could represent, e.g., healthy and infected individuals in a disease spread simulation.

Now, let us describe the structure of a typical DES more precisely. In DES, we have a countable set of states $s_0, s_1, \ldots \in S$, and a finite number of transition $a_1, \ldots, a_m \in A$. In each state $s \in S$, there is a subset $A(s) \subseteq A$ of feasible transitions, which can occur and trigger a transition to another state. An event e is a pair of a transition $a \in A$ and a timestamp t, $e = \langle a, t \rangle$. Initially, there is a given state s_0. A simulation run is a sequence of states $s_0, s_1, \ldots, s_n$, together with a sequence of events $e_1, \ldots, e_n$, such that s_{i+1} is reached from s_i by the transition of e_i, and for any pair of events e_i, e_{i+1}, the timestamp of e_i is smaller or equal to that of e_{i+1}. A simulation run is generated by a simulator which typically consists of a model state, an event queue, and a clock. An example of an event queue and model state is depicted in Figure 2.2. The event queue contains all the scheduled events, sorted in timestamp order. The clock represents the simulated time, often denoted *virtual time*, and is given by the timestamp of the last processed event. The simulator proceeds by repeatedly

(i) removing the earliest event from the event queue,

(ii) processing the selected event, i.e., update the model state as prescribed by the event, and setting the clock to the timestamp of the event, and

(iii) scheduling new events (if any) and reschedule existing events if affected by the model state changes in (ii).

The simulator stops when the virtual time reaches some predefined end time, or when the event queue is empty. Processing events in timestamp order guarantees that no event with a greater timestamp will affect an event with a smaller timestamp. The ordering of events, also known as the *causality constraint*, corresponds to our intuition about time: the future will not affect the past.

The causality constraint lies at the heart of what makes parallelization of DES. In the next chapter, we will give an overview of some of the main synchronization protocols employed when parallelizing DES.

3 Parallel Discrete Event Simulation

Distributing the work of a sequential DES can be made in several ways; here we focus on parallelization by *data decomposition*, also known as *space-parallel* simulation, in which the model state is partitioned and distributed among processors, each of which is responsible for simulating its part of the state. Parallelization of DES by data decomposition is also the technique having received the most attention. However, before continuing, some other approaches deserve mentioning, following [47].

replicated trials Multiple independent instances of the same sequential DES are run in parallel, which can be used in, e.g., stochastic simulation, either for variance reduction [34] or for parameter space exploration. Each independent instance must be initialized with an independently generated seed for the pseudorandom number generator. Although not truly PDES, it is a useful technique with near optimal weak scaling, i.e., proportionally increasing the problem size and the number of processors does not increase the execution time. Replicated trials as a technique is orthogonal to the other techniques described here (which fall into the category of strong scaling, i.e., shorter execution time with more processors), and can be used in conjunction with them.

functional decomposition Different functions of a sequential DES are located on different processors, e.g., random number generation or (in case of computationally heavy events), different parts of the event processing. The amount of parallelism available using functional decomposition is rather limited.

temporal decomposition In some exceptional cases, it is possible to partition the simulation *time* and let processors simulate different intervals of it in parallel. For each interval, the initial state has to be known, which limits the utility of this method in general. Time-parallel simulation has successfully been used for, e.g., trace-driven cache simulations [35, 54]. There, the time intervals can be simulated without knowing the initial state of the cache, since the simulation model is a program. The beginning of each simulated time interval is then later re-simulated, to correct the result of the simulation. Time partitioning works due to the usually time-limited "memory" effects of caches. The assumption is that most cache lines are replaced within a short enough period, bounding the part of a time interval that needs to be re-simulated.

In this chapter, we outline the structure of PDES by using data decomposition and present three synchronization strategies. In the following, when using the term PDES, we explicitly refer to PDES by data decomposition, if not explicitly stated otherwise.

3.1 Overview of Synchronization Protocols

In this section, we introduce two main synchronization protocols for PDES and one emerging synchronization protocol. First, some general properties of the protocols are discussed.

The first two synchronization protocols described below share the same fundamental building stones. In them, the model state is partitioned onto multiple *logical processes* (LPs), so that each LP independently models its part of the system. Each LP maintains an event queue, containing all scheduled events originating from its part of the model state, and its *local virtual time* (LVT). Each LP advances the simulation and processes events according to its local timeline, using the same DES procedure as described in Chapter 2. Communication between LPs is done via the exchange of *messages*. When an event is processed that also modifies some state controlled by a neighboring LP, a message containing the event is sent to that LP. The receiving LP then schedules the received event in their event queue, and process it when due.

The fundamental principle of DES is the *causality constraint*: the future must not affect the past. In sequential DES, this translates to processing events in timestamp order. In PDES, the causality constraint is central to what makes parallelization hard; when processing multiple events in parallel, it is difficult to maintain a strict ordering on the processed events. Existing approaches to prevent violations of the causality constraint can be divided into two broad classes. *Conservative* methods prevent LPs from processing an event unless it is guaranteed that it will not lead to a violation of the causality constraint. In *optimistic* methods, LPs process events optimistically, assuming that the causality constraint has not been violated, but provide a method to detect and roll back the simulation state when this occurs. With the advent of multicore and manycore processors, a new PDES technique, *share-everything* PDES, has emerged and gained some interest. In *share-everything* PDES, the complete model state is shared by all threads, and synchronization (with respect to time) is handled implicitly by the central event queue.

3.1.1 Conservative PDES

In conservative PDES, an LP may not process an event until it is safe, i.e., until it is guaranteed that processing of the event will not violate the causality principle. The conservative parallel synchronization algorithm was first proposed by Chandy, Misra [11], and Bryant [6] (CMB). In their setting, an LP has an incoming FIFO message channel for each LP that may send it a message. Messages must be inserted in non-decreasing timestamp order into the channels. Each channel has a clock associated with it, whose time is defined to be either the timestamp of the earliest message in the channel or, if it is empty, the time of the last received message from the channel. Simulation proceeds by repeatedly taking the first event from the channel with the earliest clock value. An LP blocks if the queue is empty. A local event is processed if its timestamp is smaller than any clock. Since events are processed in non-decreasing timestamp order, the causality principle is not violated. However, multiple simultaneously empty channels may result in a deadlock: a circular chain of LPs may block on each others' empty channels, unable to proceed. The solution that CMB provides is to let LPs send so-called *null messages*, empty messages which only contain the LVT of the sender. A null message can be seen as a promise of the sender *not* to send any message with a timestamp earlier than that of the null message. At the receipt of a null message, the timestamp of the corresponding channel is updated, allowing the receiving LP to progress. To achieve good performance, conservative approaches are heavily dependent on a lower bound on the minimum time between any two consecutive events, called *lookahead* [24]. The lookahead is usually known beforehand, and can thus be used during simulation to say how far ahead it is safe for an LP to simulate, without synchronizing with neighbors. Hence, a simulation with greater lookahead requires less synchronization, and the lookahead is a sign of more available parallelism. Lookahead can be found in, e.g., simulations of memory-systems, and network simulations, where actions and communication typically have a fixed or minimum duration. Such guarantees, however, do not exist for a wide array of models, e.g., stochastic models with exponentially distributed inter-event times.

3.1.2 Optimistic PDES

The most well known and influential PDES synchronization protocol is Jefferson's *time warp* [42] algorithm, in which the LPs optimistically assume that no causality error will occur, and speculatively simulate along their local time-axis. In Figure 3.1, a possible organization of the LPs in time warp is shown.

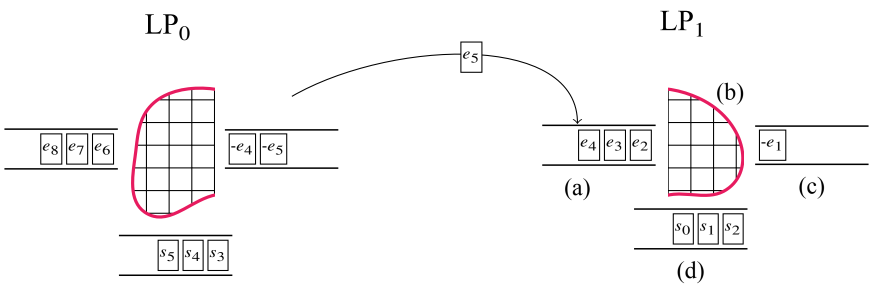

Figure 3.1. A possible organization of the LPs in time warp. An event queue (a) of scheduled events, a model state (b), an output queue (c) with copies of messages sent, and a history (d) of past states or events. LP$_0$ sends a message e_5 to LP$_1$, and puts a negative antimessage -e_5 in its output queue.

The main idea in the time warp protocol is to allow events that arrive too late, i.e., whose timestamp is smaller than the LVT of the receiving LP, so-called *stragglers*. As a consequence, LPs may process events even if it is not safe to do so, and instead, a mechanism to roll back the model state is provided. When a straggler is received, an LP rolls back its state to a point in time preceding the straggler, by undoing the state changes of each succeeding event. Events causally dependent on events that were rolled back have to be canceled. Thus, if an event that is rolled back has caused the transmission of an event to another LP, the effects of that event must be undone as well. This is done via so-called *anti-messages*. For each transmitted event e that has to be undone, a corresponding anti-message a_e is sent. Upon receipt of such an anti-message a_e, an LP must undo (often denoted *annihilate*) the original message e, and consequently roll back all processed events that are causally dependent on e. After the completion of a rollback and the transmission of anti-messages, simulation continues from the point in time just preceding the straggler. The straggler can now be correctly interleaved in the local timeline.

Anti-messages are typically created at the transmission of a message to another LP, and then stored in an LP's *output queue*, sorted in timestamp order. When a straggler is received at an LP, anti-messages with a later timestamp are removed from the output queue and sent to the respective receivers of the original messages.

To restore a previous state of the model, some historical data has to be recorded. Thus, in addition to the event queue and the model state, each LP maintains a *history*, which can be used to reconstruct past model states. The original time warp algorithm proposes saving successive states of the LP in such a history periodically, also called *checkpointing*. To be able to restore a historic state s of an LP using checkpointing, one can restore some checkpointed state s' in the history, such that s' precedes s, and continue forward simulation until s is reached. s is guaranteed to be reached since the simulation is deterministic. Several improvements to restore a previous state have been proposed. *Incremental state-saving* [5] is a straightforward refinement of checkpointing: instead of saving the whole state, incremental deltas of the state are saved instead, saving expensive memory space. As an alternative, the *reverse computation* technique [8] stores the processed events instead. For each event, a backward computation is defined (hence the name reverse computation). Given the last event that was processed, the previous state (before the event was processed) can be reconstructed from the current model state, by calculating the reverse computation of the event. Thus, when doing a rollback, the event history is traversed backward. The reverse computation technique may reduce the amount of space necessary to store the event history, in comparison to the incremental state saving technique. If the event that needs to be stored in the history is smaller than the state delta changed by the event, less memory is consumed by the history. On the other hand, reversing the model state may have a higher computational requirement than incremental state-saving. The choice of state-saving technique is, in this case, a trade-off between memory consumption and computation time. An analysis of the trade-off on a type of shared-memory computer can be found in [9].

An example of the trade-off above is the handling of *pseudorandom number generator*s (PRNGs) in stochastic simulations (a similar example also appear in [8]). PRNGs generate deterministic sequences of numbers that take on the role of random numbers in many scientific applications. In models where sampling of pseudorandom numbers is used, the state of the PRNG has to be reverted during a rollback. If not, the simulation will suffer from a *sampling bias*: certain sampled pseudorandom values tend to cause causality errors more often, and hence they are rolled back more often. If the state of the PRNG is not rolled back together with the model state, these sampled values will actually occur less often in the simulation trajectories. Using an incremental state-saving technique, the seed (state) of the PRNG would have to be saved with each partial state saving. A seed can be anything from 6 bytes (`drand48` in the POSIX standard) to around 2.5kB (the Mersenne twister algorithm [50]). Both the algorithms are reversible, i.e., the sequence of deterministic numbers can be generated forward as well as backward, and the cost of both calculations are approximately the same. Hence, if using the Mersenne twister PRNG, rollback using reverse computation would clearly be more efficient than incremental state saving, while for the `drand48` PRNG, it is less obvious.

As a simulation progresses, the state history grows continuously, even though we hope not to restore arbitrarily old states. Intuitively, it should be possible to discard "old enough" at some point during the simulation. It is desirable to periodically reclaim the memory of these discardable states, from a performance point of view; this is usually called *fossil collection* in the simulation context. A suitable definition for an "old enough" historic state would be to say that it cannot be reached by a rollback. The earliest virtual time to which a simulation can be rolled back is defined by the timestamp of the earliest event in the simulator. This time is denoted *global virtual time* (GVT). Since the simulation never is rolled back to a time before GVT, it is safe to reclaim memory used by the history preceding GVT. When an event's timestamp is smaller than GVT, then it is said to be *committed*, since it no longer can change. Much research has focused on improving GVT calculations since it is often a costly calculation. Two examples of algorithms for GVT calculations are by Mattern [51], in a message-passing programming model, and by Fujimoto and Hybinette [25], for shared-memory systems. The GVT calculation is much harder in a distributed environment compared to in a shared-memory environment, since it amounts to taking a global snapshot, which is a non-trivial problem.

3.1.3 Share-Everything PDES

Multicore processors have opened up for new approaches, thanks to low core-to-core communication latency and high bandwidth. Such approaches include parallelizing a DES workload using a number of worker threads having access to the whole model state and which are allowed to update the whole state. Instead of synchronizing via messages, synchronization is done by temporarily reserving the state which should be updated. Work is distributed to the worker threads using a central timestamp sorted queue [32, 33, 49]. The benefits of such an approach are that it is cheaper in terms of time and complexity to maintain the causality constraint and that the work is always evenly balanced between cores. At least two problems can be identified with a share-everything approach, viz., shared access to a single event queue, and interleaved memory access to the model state.

The first problem is that all threads want to remove the earliest element from the event queue, thereby creating a hot spot. This problem is inherent to having shared access to a single central event queue, regardless of the underlying data structure chosen. To ensure correctness (e.g., only one worker thread succeeds in removing the earliest element), the remove operation has to be protected in some way. Common constructs include *locks*, *lock- and wait-free* methods, and *transactional memory*, which may differ in speed and implementation (some of these constructs are discussed in Chapter 5). However, in all cases, only one thread can succeed in removing the smallest element, and other threads that simultaneously have tried to remove the same element, must fail; the (wasted) time they have spent on trying to complete the remove operation is called *contention*. With an increasing number of worker threads, contention increases. Thus, even if the events in the queue exhibit concurrency and efficiently could be carried out in parallel, the contention of the event queue may decrease the efficiency.

The second problem has to do with memory accesses. Multicore processors and multiprocessor machines are structured so that accesses to memory (and caches) are non-uniform in the time it takes to access a memory location, depending on where it is located in relation to the processor core accessing it. In particular, accessing memory that another processor core modified previously is more expensive than accessing memory that the same processor core previously accessed. Thus, from this point of view, letting all worker threads have interleaved accesses to the whole model state may lead to a performance penalty.

3.2 Adaptive Optimism Protocols

Both conservative and optimistic approaches incur a synchronization cost. Conservative methods may be slow due to excessive synchronization; optimistic methods may suffer from large amounts of rollbacks. To mitigate these problems, many intermediate techniques have been proposed, where the optimistic speculation is regulated by some mechanism; we refer to such techniques as *adaptive* protocols or *optimism control* [12]. The purpose of such intermediate techniques is to be able to do optimistic simulation, while reducing the problems related to it, e.g., rollback explosions. Additionally, there might be an advantage of spending time *not* doing forward simulation (e.g., by blocking local simulation due to too much speculation). There is a double cost associated with a rollback: first, the cost of actually simulating an interval of time, and second, the cost of rolling back the same interval. Figure 3.2 displays the trade-off between the cost of rollbacks and the cost of the optimism control.

The core idea behind the optimism control protocols is to make the LPs advance their local simulation at approximately the same speed, typically leading to a reduced risk of rollbacks and anti-messages. Jafer et al. [41] introduces the term *event temporal locality*, to describe a small maximum distance of timestamps of events being processed in parallel. Due to the strong connotation of the term temporal locality with a small memory location reuse distance, we introduce the term *event synchronicity*, to denote the same situation.

The optimism control protocols have different approaches to improve the event synchronicity. Early methods include the *Moving time window* [65] algorithm, which defines a limit on how far an LP may be ahead of other LPs, by defining a time window w, such that only events with a timestamp smaller than GVT + w are processed. *Risk-free* [53] protocols only allow sending a message between LPs, if it is guaranteed that the message will not be rolled back. *Breathing time buckets* [67] is a risk-free protocol, which avoids rollbacks of messages by splitting the simulation into cycles defined by the *event horizon*: the timestamp of the earliest message generated from processing local events. Events may be processed until the event horizon (if events with timestamps greater than the event horizon have been processed, they are rolled

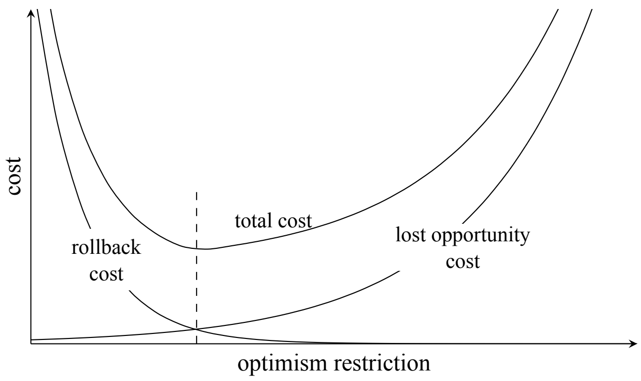

Figure 3.2. The trade-off between optimism and the lost opportunity cost caused by blocking. The x-axis describes the amount of optimism control exerted on the simulation. Initially, when restricting the optimism, the cost of rollbacks reduces. As the restriction on optimism gradually increases, the *lost opportunity* cost, e.g., when an LP blocks even though not strictly necessary, increases, reducing the improvement of the optimism control protocol. Adapted from [66].

back). *Breathing time warp* [68] is a combination of the breathing time buckets protocol and optimistic simulation. Another class of optimism control uses local history to control the optimism. In, e.g., [22], message arrival patterns are observed by LPs, and based on these observations estimates coming arrivals. The local estimates are used to throttle the LP's simulation speed, if necessary. Problems with the techniques mentioned above are due to that they limit optimism either based on predetermined parameters (e.g., the window size), local history, or some aspect of the technique is synchronous. Instead, Srinivasan and Reynolds [66] recognize two key parameters for a successful optimism control protocol: it should be adaptive and use feedback from the simulation, which could be incorporated by techniques disseminating and using (near) perfect state information. They define the *elastic time algorithm* (ETA) that at least partially fulfills these properties. The algorithm is designed with two aspects in mind: which state information should be disseminated, and how can the information be used to control the optimism. In the ETA, the minimum of each LPs current time and the minimum timestamp of sent messages that have not yet been received is disseminated. From this value, the authors define an LP's error potential as its distance (in virtual time) to the slowest LP. The error potential is acted upon so that the simulation speed of an LP gradually decreases with increasing error potential. Their method assumes the global availability of (near) exact information from the LPs state.

3.3 LP Aggregation

Various techniques for sharing structures between LPs, *LP aggregation*, have been proposed to reduce various sources of overhead in optimistic PDES. For example, when events in LPs are computationally small, it may be preferential to share structures, such as event queues, state queues and message buffers between a group of LPs, to reduce, e.g., the memory footprint of the simulation [3, 14, 15, 63]. A typical example of simulations where the computational load of an event is small is integrated circuit simulation. LP aggregation may also be a good choice when executing the simulation on processors which are optimized for consecutive memory accesses, which is the case for most processors are today. On such processors, it is beneficent for performance to keep objects close together in memory that are expected to be accessed at approximately the same time. E.g., it might be significantly more efficient to traverse a (long aggregated) single queue of saved states during a rollback, than to visit multiple (short) queues.

One aggregation technique is clustering of LPs [3, 58]. Avril and Tropper [3] propose that multiple LPs are clustered together. A sequential algorithm is used within a cluster, and between clusters, time warp is used. When an LP sends a message to an LP in the same cluster, the message can be put directly into the receivers input queue. Thus, LPs do not need to maintain individual output queues. Instead, a single cluster output queue is provided. Each LP's event and state queues are still maintained separately.

Schlagenhaft et al. [63] note that in many models, each "basic element" of a process or system is modeled as a single LP. A basic element can, e.g., be a logic gate in an integrated circuit simulation. Instead, they propose that a single LP may represent multiple basic elements. This is the approach taken in [14, 63]. It is similar to the clustering approach taken in [3], but more structures are shared when multiple basic elements are represented by a single LP. In [63], the event queue and the state queue are shared, while in [14], only the event queue is shared.

The drawback of having a single common state queue for a partition is that a straggler causes a rollback of the whole partition located on an LP. Two different solutions are proposed:

Schlagenhaft et al. [63] propose clustering of LPs: they assign multiple smaller LPs controlling many basic elements to each processor, instead of a single LP. When such a smaller LP receives a straggler, only a single partition consisting of many basic elements needs to be rolled back, thereby reducing the rollback overhead.

Deelman and Szymanski [13] propose to only rollback causally dependent events upon the receipt of a straggler. The idea is that the number of causally dependent events are a fraction of all events that otherwise have to be rolled back. Tracing the causally dependent events is possible when either incremental state saving or reverse computation is used. Then there is a one-to-one mapping between past events and historical state. The technique relies on that the local virtual time of each basic element is known. (In [13], each basic element also has a processed events list, but this is not strictly necessary.) At the receipt of a straggler, dependencies between basic elements formed by historical events are traced, and each basic element that has a dependent event is rolled back.

3.4 Load Balancing

The purpose of *load balancing* is to distribute work to the available computing resources optimally, to maximize throughput. In PDES, load balancing has the aim to improve *event synchronicity*. We note that the slowest LP determines the speed of the simulation: if some LP advances its LVT faster than its neighbors, then it is forced to roll back its state as soon as it receives a message from any of the slower neighbors. In the case of using optimism control, the faster LP is forced to wait for some of its neighbors. Therefore, accurate load balancing minimizes the total amount of wall-clock time necessary to complete a simulation. Load balancing can be *static*, where the model is analyzed and partitioned accordingly initially, or *dynamic*, where the model load is re-balanced among processors during the simulation. In the case of dynamic load balancing, there are usually three aspects considered; a *load metric*, a *load balancing algorithm* and a *migration protocol* for moving the load, e.g., LPs or parts of an LP's model state.

Load Metric

The goal of the load metric is to define how much work a processor or LP is doing per time unit. Such a metric is then used for deciding how the load should be (re-)distributed, with the goal of giving each processor the same amount of load according to the load metric. For load balancing in PDES, and in time warp based simulators, the processor load is not a good indicator of the load; a processor may have full processor load and still not advance its simulation time, instead spending time on processing rollbacks [59]. Reiher and Jefferson [59] introduced a metric, *effective utilization*, defined as the proportion of an LPs work that is not rolled back. Given the difficulty of knowing such a metric, since the state of events is unknown until they have been committed, the authors define an approximate estimator. In [7], the LVT is used as an (inverse) load metric. The slowest LP is moved to the processor where the LVT has advanced the most. The purpose of the load balancing can then be interpreted as reducing the difference in virtual time between LPs, which optimally would lead to a reduced number of rollbacks. In [31], the rate at which LVT is advanced is used as an inverse load metric. Other factors that may be taken into account is communication [4, 73].

Load Balancing Algorithm
The load balancing algorithm should, given a list of a subset of the LPs and their respective load metrics, calculate a re-distribution of LPs/load, which typically reduces the difference in load and communication over the processors. It is common in the literature that load is only transferred between the LPs with the least and the most work, respectively. In some cases, the problem has been formulated as a minimax optimization problem, where the maximum difference in the load metric should be minimized [14]. In [31], a bin-packing algorithm is used to redistribute the work. Typically, a load-balancing algorithm tries to avoid *thrashing*, i.e., unnecessary movement of load between processors. This has been done by thresholds [14] and delaying subsequent load-balancing operations [4, 63].

Load Migration Protocol
The load migration protocol is responsible for transferring load between processors, while maintaining correctness of the simulation. In some cases, a protocol is not even necessary, e.g., when a global synchronization is used for rebalancing. Then, simulation is paused globally, work is redistributed, and the simulation is continued. In the case where the simulator has multiple processes per processor, migration consists of moving one or more processes between processors.

3.5 Challenges

The focus of this thesis is to improve the parallel performance of applications of spatial stochastic simulation in systems biology. The models that we have studied are in essence spatially extended Markovian processes. Hence, the time intervals between successive events are exponentially distributed (the exponential distribution is the only continuous distribution with the Markov property), and consequently they are highly variable and without lower bound. High variability and no lower bound of inter-event times mean there is no lookahead. No lookahead suggests spatial stochastic simulation is challenging to parallelize, with potentially high synchronization costs. The models that we have studied are also characterized by having computationally light events; the state change of an event merely consists of one or a few additions and subtractions, while computing a new timestamp for the next event is slightly more expensive. Finally, in, e.g., systems biology, the phenomena of interest that typically are being modeled, are often non-homogeneous and dynamic in behavior. The effect is that the distribution of the load on the processing elements tends to become unbalanced.

The lack of lookahead suggests that optimistic PDES should be used for parallelization of spatial stochastic simulation. In optimistic PDES, one of the central challenges for good performance is to achieve good event synchronicity, e.g., by using an optimism control protocol. The properties of spatial stochastic simulation make it reasonable to believe that several local or synchronous methods for optimism control described in Section 3.2 will not be successful. One point of view on optimism control is that, optimally each LP would advance its local simulation as far as possible without encountering a straggler. To do this, an LP needs knowledge of future incoming messages and their causal dependencies. In Paper II, we develop an optimism control technique where accurate estimates of future inter-LP events are obtained and disseminated to neighboring LPs.

Another way to achieve good event synchronicity is through load balancing. The non-homogeneous and dynamic character of spatial stochastic simulations in, e.g., systems biology, suggests that dynamic load balancing is required for good parallel performance. In Paper IV, we develop a dynamic local load balancing mechanism for PDES.

4 Event-Based Modeling

In this thesis, the focus of the discrete-event simulation has been applications to systems biology. In this chapter, this type of application is described in greater detail.

4.1 Mesoscopic Event-Based Modeling

We have considered event-based modeling of a particular class of systems, namely chemical reaction systems, which concerns the temporal evolution of a composition of molecules or entities, reacting with each other and thereby changing the overall composition. For this kind of systems, there exist a number of modeling techniques, each developed for a different level of abstraction. At the *microscopic* level, each reactant is modeled individually, including its velocity and position. At a *macroscopic* level, the amount of each type of reactant is treated as a concentration, e.g., a continuous quantity, and their time evolution is described by differential equations.

Macroscopic frameworks are adequate when the populations of reactants are big enough. Unfortunately, for systems with smaller populations, macroscopic frameworks disregard some of their important properties: when expressing populations as concentrations, the fact that populations are discrete counts of molecules and the very stochastic nature of these systems is ignored. Several phenomena actually only manifest themselves because of stochastic behavior, due to the small population, and other phenomena manifest themselves only because of the discreteness of the system. On the other hand, microscopic frameworks are intractable for more than a few number of reactants due to the high modeling complexity. *Mesoscopic* frameworks have been designed for modeling systems between the macroscopic and microscopic levels, to capture behavior that is ignored at the macroscopic level, and with a large enough number of reactants that would make it intractable for a microscopic framework.

4.2 Stochastic Simulation of Chemical Kinetics

Stochastic chemical kinetics describes the time evolution of chemical reaction systems at a mesoscopic level, and take into account that molecules exist in discrete quantities. Stochastic chemical kinetics is of special interest in modeling of cellular systems in biology, where interesting phenomena depend on stochastic effects due to low population counts, e.g., the placement of the septum (the dividing wall) in cells during cell division, whose position is decided by the oscillation of a number of proteins in some bacteria [21]. We start by looking at a model of chemical systems without spatial effects in the next section and then continue to systems with spatial effects.

The Chemical Master Equation

The *chemical master equation* (CME) [26] is a well-established formalism for describing the dynamics of a chemical system. (It is also known as the *combinatorial master equation* since it is well suited for describing any system whose state is discrete and where changes occur in discrete steps.) Such a system consists of a set of entities, e.g., molecules or proteins, of *n species* $s_1, \dots, s_n$, moving freely in some volume. Molecules may react with each other when in proximity, through a number of reactions $r_1, \dots, r_m$. It is assumed that the modeled volume is spatially homogeneous, and in thermal equilibrium; if that is the case the system is said to be *well-stirred*. The assumptions about the microscopic conditions greatly simplify the modeling. It entails that the probability of a reaction r occurring within an infinitesimal interval, the reaction's *propensity* ω_r, only depends on some constant, k_r, and, the possible combinations of the reactants, ignoring its position and velocity [29]. Hence, the state of the system at time t is described solely by the population vector $\boldsymbol{x} = (x_1, \dots, x_n)$, where x_i is the population of species s_i. Thus, the system constitutes a Markov chain $X(t)$, with transition rate ω_r for each reaction r.

As an example demonstrating the kind of systems governed by CME, we use the Lotka-Volterra predator-prey model [48, 71]. The predator-prey model describes how two species, a predator species X and a prey species Y, interact through predation and how their populations change over time. At a macroscopic level, the respective populations, in a bounded area, are described by a

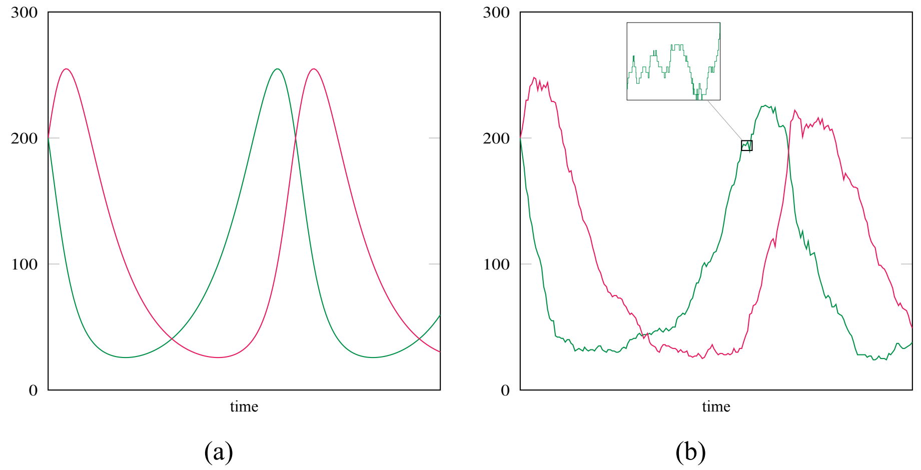

Figure 4.1. The predator-prey system. The population (y-axis) of predators (pink) and prey (green) is plotted over time. (a) The solution of the deterministic ODEs. (b) One trajectory from a simulation with Gillespie's SSA. The resemblance to the deterministic solution is evident. In the enlarged window, the discrete steps of the process can be seen.

system of ODEs,

$$\frac{dX}{dt} = k_b Y - k_p XY$$
$$\frac{dY}{dt} = k_p XY - k_d X,$$

where k_b is the rate of birth of prey, k_p is the rate at which the predators prey, and k_d is the rate of death of predators. The description is continuous and thus does not take into account that living species usually come in discrete numbers. The description is also deterministic. The solution of the ODE system for one set of parameters is shown in Figure 4.1(a). The interaction described in the figure is how the predator species prey on the prey species, leading to a growth of predators. At some point the population of prey becomes so small that it is unable to sustain the population of predators, leading to a decline in the predator population. This leads in turn to an increase in the population of prey. Expressed using the CME, we describe the system as a set of reactions instead:

$$\emptyset \xrightarrow{k_b} X, \qquad (r_1)$$
$$X + Y \xrightarrow{k_p} 2X, \qquad (r_2)$$
$$Y \xrightarrow{k_d} \emptyset. \qquad (r_3)$$

Thus, for the first reaction, we have a spontaneous generation of prey out of nothing, which merely means that we don't model the source of food of the prey. The second reaction describes that one predator and one prey in proximity (here captured by a probability of reaction) "react" (i.e., are consumed) and results in two predators. The last reaction describes the death of predators. In general, the macroscopic reaction-rate constants are not the same as the stochastic reaction constants above. However, for this example, they are the same, assuming that the reactions take place in a unit volume.

Algorithm 1 Gillespie's Direct Method for a model with m reactions.

```
Input:
$\boldsymbol{x}$ state, $t$ virtual time
$\omega_r$ propensity functions for each reaction $r$
$\boldsymbol{s}$ stoichiometry matrix, where row $\boldsymbol{s}_r$ has the discrete population changes of reaction $r$
while $t$ less than end time do
    sample $\tau \sim \text{Exp}\left(\sum_{i=1}^{m} \omega_r(\boldsymbol{x})\right)$        //Time to next reaction
    sample next reaction $r \sim \text{Disc}(\lambda_0, \dots, \lambda_m)$
    $t := t + \tau, \boldsymbol{x} := \boldsymbol{x} + \boldsymbol{s}_r$        //Update state and time
```

The traditional way to describe the time evolution of a chemical system is to solve a "master" equation of the system, which completely characterizes the system, in this case, the CME. However, the CME is prohibitively difficult to solve analytically, and even numerically, except for systems with a small number of species and reactions [30]. Therefore, various Monte Carlo simulation methods have been developed. A famous method to sample exact trajectories of such systems is Gillespie's Direct Method (sometimes just called the *stochastic simulation algorithm* (SSA), which arguably is too general) [29]. The algorithm is outlined in Algorithm 1. We use the notation $\text{Disc}(\lambda_0, \dots, \lambda_n)$ to denote a discrete probability distribution of a set of variables $0, \dots, n$ with probabilities $\lambda_0, \dots, \lambda_n$, where $\text{P}(\lambda_k) = \lambda_k / \sum_{i=0}^{n} \lambda_i$, and $\text{Exp}(\lambda)$ to denote an exponential distribution with rate parameter λ. The key point of the algorithm is that a common rate for all reactions is calculated, from which the time of the next reaction is sampled (line 5). The latter works, since for $X_0, \dots, X_n$ independent exponential random variables with parameters $\lambda_0, \dots, \lambda_n$, we have $\min\{X_0, \dots, X_n\} \sim \text{Exp}(\lambda_0 + \cdots + \lambda_n)$. It is first when the reaction event is processed that it is decided which reaction occurs, by a random draw. Hence, the method only requires sampling of two random numbers per event.

Gillespie's SSA can be used to sample a trajectory from the system described by the reactions (r_1) to (r_3), an example is shown in Figure 4.1(b). Similarly to the deterministic solution in Figure 4.1(a), we see how the population of the predators trail after the preys, but here we also have random population increases and decreases, and slight variations in relations to the deterministic solution can be seen, with respect to, e.g., the maximum population attained. We note that only a single trajectory is displayed in Figure 4.1(b), and each sampled trajectory will look different. The deterministic continuous solution fails to describe several scenarios, such as extinction, caused by the stochastic and discrete nature of the process, shown in Figure 4.2. If one wants to know

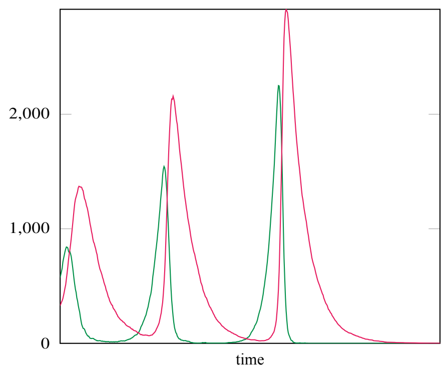

Figure 4.2. The Predator-Prey system. A trajectory of Gillespie's SSA where one species becomes extinct, followed by the extinction of the other species as a consequence. The extinction occurs due to the discrete nature of the system.

the mean extinction time, it is necessary to use a model that is both stochastic and discrete. Other stochastic phenomena not captured by macroscopic frameworks is random switching between steady states [20], as shown in Figure 4.3. Random switching occurs in, e.g., gene regulatory networks [27]. To find the mean switching time between the steady states, it is necessary to use stochastic methods.

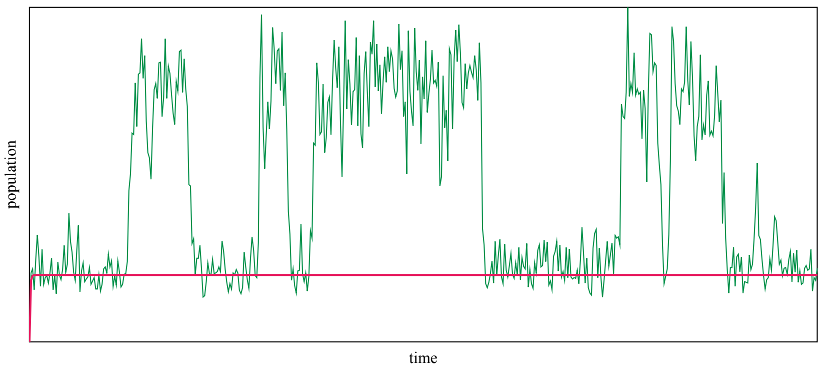

Figure 4.3. A system exhibiting random switching. The ODEs describing the system has multiple steady states. The solution (pink line) converges to one of the steady states dependent on the initial condition. A sampled trajectory of Gillespie's SSA (green line). We see that the solution initially is similar to the deterministic solution. However, the stochastic fluctuations sometimes are so strong that it pushes the systems to another steady state. The switching occurs due to the stochastic nature of the system.

Numerous refinements to Gillespie's original SSA have been proposed, and one notable efficient version is the *next reaction method* (NRM) by Gibson and Bruck [28]. Instead of sampling from a discrete distribution to select the next reaction as in the direct method, all the reactions are stored in a priority queue.

Whenever a reactions propensity is changed, the priority queue is updated to reflect the change. They observe that a reaction, when processed, only modify the populations of the reactants' species, while other species' populations are left unchanged. A dependency graph is constructed so that for each reaction, only the reaction rates of dependent species are updated. A second enhancement of the NRM is that only events that actually are processed need a new random number for the calculation of their new scheduled time, whereas for other dependent reactions' event times it suffices to perform a scaling according to the change in population. The effect is that the algorithm only needs one random number per iteration instead of two, as in Gillespie's SSA, which may have a sizeable impact on performance.

4.2.1 The Reaction-Diffusion Master Equation

When describing intracellular systems, the principles of the CME still hold, but they have to be adjusted according to the fact that molecules no longer move freely, but in some solvent inside an enclosed vessel. The molecules collide frequently with other molecules, constantly changing direction, leading to a chaotic movement. Hence, the movement of molecules, *diffusion*, is much slower than in, e.g., reactive systems consisting of a gas. The assumption of spatial homogeneity from the CME can also be considered as a relation between the movement of molecules and the rate of chemical reactions; the movement has to occur at a faster rate than the reactions so that molecules have time to "search through" the volume between reactions. Since the diffusion is slower in liquid compared to the free flight in a gas, the assumption is only valid for a small volume [18]. The solution is to subdivide the volume into *subvolumes* and apply the assumption of spatial homogeneity within each subvolume, and postulate that only reactants within the same subvolume can react.

The *reaction-diffusion master equation* (RDME) [26] is the spatial extension of the CME and is used for modeling in these cases. In addition to the reactions, it describes how an entity may diffuse in the volume being modeled, by defining actions of movement between the subvolumes. When discretizing the volume, some physical considerations of the model has to be taken into account. These physical considerations also form the basis of the assumption of the CME. The subvolumes have to be big enough to properly localize the entities, e.g., the modeled entities should fit into the space of the subvolume, and the subvolumes must be small enough to be considered well-stirred.

As an example, we take a spatial extension of the Lotka-Volterra predator-prey model [62]. Instead of assuming spatial homogeneity within the area that is modeled, the area is represented by a two-dimensional grid. The species may move (diffuse) between adjacent grid cells. Only when a predator and a prey occupy the same subvolume, predation may occur according to reaction (r_2).

Predators die off according to reaction (r_3) if they occupy a subvolume where there is no prey. The birth reaction (r_1) of prey is slightly modified, to account for the fact that birth may only occur in subvolumes occupied by prey,

$$X + Z \xrightarrow{k_b} 2X + Z.$$

Here we've added a third species Z, that can be considered to act as food for the prey species. The movement of the species is controlled by two diffusion rate constants, d_X and d_Y.

Algorithm 2 Outline of the NSM for a model with m reactions and n species. $\text{Exp}(\lambda)$ denotes the exponential distribution with intensity λ, and $\text{Disc}(\lambda_{e_0}, \dots, \lambda_{e_k})$ denotes the discrete distribution of the values $e_0, \dots, e_k$ with probabilities $\lambda_{e_0}, \dots, \lambda_{e_k}$.

```
Input: In addition to the state of Gillspie's SSA, NSM defines:
eventqueue of voxel-timestamp pairs $\langle v, t_v \rangle$
$D$ diffusion matrix of diffusion rates between subvolumes
procedure INITIALIZE
    for all subvolumes $v$ do
        $\lambda_v^r \leftarrow \sum_{i=0}^m \lambda_v^{r_i}$, $\lambda_v^d \leftarrow \sum_{i=0}^n \lambda_v^{d_i}$      //reaction and diffusion rate in v
        $\lambda_v \leftarrow \lambda_v^r + \lambda_v^d$                                 //total rate of voxel v
        sample $t_v \sim \mathbf{E}(\lambda_v)$                                  //initial event timestamp for v
        INSERT(eventqueue, $\langle v, t_v \rangle$)
procedure SIMULATION LOOP
    $\langle v, t_v \rangle$ ←DELETEMIN(eventqueue)
    sample event type ~ $\text{Disc}(\lambda_v^r, \lambda_v^d)$                  //reaction or diffusion?
    if event type = reaction then
        sample $e \sim \text{Disc}(\lambda_{r_0}^v, \dots, \lambda_{r_m}^v)$             //which reaction
    else                                                      //diffusion
        sample $e \sim \text{Disc}(\lambda_{d_0}^v, \dots, \lambda_{d_n}^v)$             //which diffusion (i.e., which species)
        sample diff. direction $v'$ according to $\boldsymbol{x}$ and $D$ //diffuse to subvolume $v'$
    for all reactions $r_i$ dependent on $e$ according to $G$ do
        update propensities $\lambda_v^{r_i}$, and $\lambda_{v'}^{r_i}$ if $e$ is a diffusion
    update $\lambda_v^d$, and $\lambda_{v'}^d$ if $e$ diffusion, according to $\boldsymbol{x}$ and $D$
    $\lambda_v^r \leftarrow \sum_{i=0}^s \lambda_v^{r_i}$
    $\lambda_v \leftarrow \lambda_v^d + \lambda_v^r$
    if $e$ diffusion then                                     //update timestamp for $v'$
        $\lambda_{v'}^{\text{old}} \leftarrow \lambda_{v'}, \lambda_{v'} \leftarrow \lambda_{v'}^r + \lambda_{v'}^r$                //update total rate $\lambda$ for $v'$
        $t_{v'} \leftarrow t + (t_{v'} - t)(\lambda_{v'}^{\text{old}} / \lambda_{v'})$                //rescale timestamp for $v'$
        update eventqueue with modified event $\langle v', t_{v'} \rangle$
    sample new time $\tau \sim E(\lambda_v)$ for voxel $v$, set $t_v := t + \tau$
    $t := t_v, \boldsymbol{x} := \boldsymbol{x} + \boldsymbol{s}_e$                                //update state and time
    INSERT(eventqueue, $\langle v, t_v \rangle$)
```

Stochastic simulation in the vein of the SSA may be used for efficiently sampling trajectories of the population in the subvolumes. One such method is the NSM [17], shown in Algorithm 2. It is essentially a combination of

Gillespie's original SSA and the NRM, but there is in addition to the reaction events, also diffusion events. A diffusion event represents the movement of a single reactant, from one subvolume to another. For each subvolume, the total rate, i.e., the sum of the rate of all reactions and the rate of all diffusions out of the subvolume, is calculated and used to sample the time of the next event of the subvolume (line 26). The next event time of each subvolume is stored in a priority queue. In the main loop of the simulation, the earliest event in the priority queue is extracted, which also decides in which subvolume the next event occurs (line 10). Then it is decided if it is a reaction or a diffusion, and which reaction or diffusion it is (lines 11, 13 and 15). In case it is a diffusion, it is also decided to which subvolume the event diffuses (line 16). After the event has been processed, affected reactions and diffusions are re-scaled, and the priority queue is updated.

In practice, all random variables are sampled using a uniform PRNG, together with a suitable technique to generate the required distribution, such as inverse transform sampling. Therefore, in some cases the pseudorandom variables may be reused more than once, e.g., for line 11 and line 15, it suffices to use a single variable.

The original NSM assumed a Cartesian structured mesh, and hence the probability of each diffusion direction is simply $1/k$ for k neighbors at line 16. Engblom et al. [19] has extended the NSM for unstructured meshes, which permits easier modeling of, e.g., curved boundaries of the modeled volume. There, the probability of a diffusion direction is given by the diffusion matrix D together with the population. The methods are implemented in the URDME framework [16], which this thesis has been based on.

The NSM has been used to study, e.g., protein fluctuations taking part in cell division [21], regulatory processes relevant for differentiation of stem cells [69], and the polarization of yeast cells [45].

4.3 Parallel NSM

Simulation of the NSM is computationally heavy and time-consuming, motivating a parallel approach. To parallelize the NSM, typically the subvolumes are distributed over the LPs, either one-to-one, or by letting each LP control a partition of the model. Diffusion events between subvolumes on different LPs are exchanged as messages.

Parallel simulation of RDME models using the NSM has previously been addressed by [15, 43, 46, 72]. Only optimistic protocols have been considered, due to the inherent lack of lookahead in RDME models, or any Markov model in general. Each LP represents a subvolume [46, 72] or a subdomain [15, 43]. The models are implemented using MPI [15, 46, 72], where LPs are mapped to MPI processes [15, 72]. Optimism control has been implemented by a static time window based on GVT [43] or by Breathing Time Warp [72].

Jeschke et al. [44] explored how a parallel NSM simulation can be run in the background on desktop computers. They note that the general challenges of the NSM which exhibits fine-grained event computations and zero lookahead, combined with a grid-computation approach, causes the overheads to outweigh any gains from the distribution of the work onto multiple computers.

4.4 Challenges

The key to the performance of the sequential NSM is the compact representation of its event queue. Events are aggregated, so that there is a single event per subvolume, instead of one separate event per reaction and per voxel neighbor. The type of transition of an aggregated event and its direction (if it is a diffusion) is generated when it is processed.

Parallelizing NSM is challenging due to the properties described in Section 3.5. One approach to achieve an efficient parallelization might be to use a suitable optimism control technique. Hence, we want to apply the *dynamic local time window estimates* (DLTWE) technique, which was developed and successfully applied in Paper II, to NSM. DLTWE requires each LP to communicate accurate estimates of the timestamps of the next outgoing inter-LP events to its neighboring LPs, but in order to do that, the timestamps of future events must be known beforehand. Thus, it would be necessary to modify the core NSM algorithm to expose timestamps of future inter-LP events. To do that, inter-LP events cannot be aggregated, resulting in a bigger memory footprint of the simulator, thereby deteriorating its performance. In exchange, more information is available.

In Paper III, we study how to make the NSM expose timestamps of future events, and investigate if the cost of maintaining the timestamps up to date is worth the performance gain allowed by the information.

5
Skiplist-based Priority Queues

Priority queues are fundamental to many applications such as event queues in DES, various graph algorithms [23], huffman encoding [39]. Skiplists [56, 57] has been shown to be suitable for implementing parallel priority queues [64] and has been used to implement, e.g., garbage collection [52].

In this chapter, we give a short background to skiplists, priority queues and their usability for parallel applications. We also introduce a correctness condition and a common progress characterization of parallel algorithms.

A skiplist is a search data structure, akin to a binary search tree, with a probabilistic guarantee of being balanced. It is built up as a collection of hierarchically arranged linked lists, where higher level lists act as shortcuts into the lowest level list. The lowest level list is a regular sorted linked list, containing all elements, or nodes, stored in the structure. Each higher level list visits fewer nodes than its lower-ranking lists. Thus, a search procedure for finding a node can start by traversing the top level list, quickly homing in on the neighborhood of the key, and then gradually descend level by level, to find

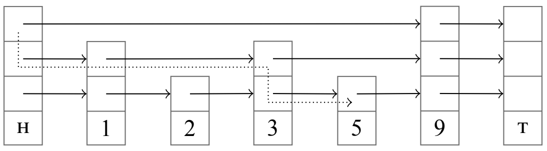

Figure 5.1. A skiplist with three levels. The skiplist contains 7 nodes, including head sentinel H and tail sentinel T. The dotted line depicts the search path for the node with key 5.

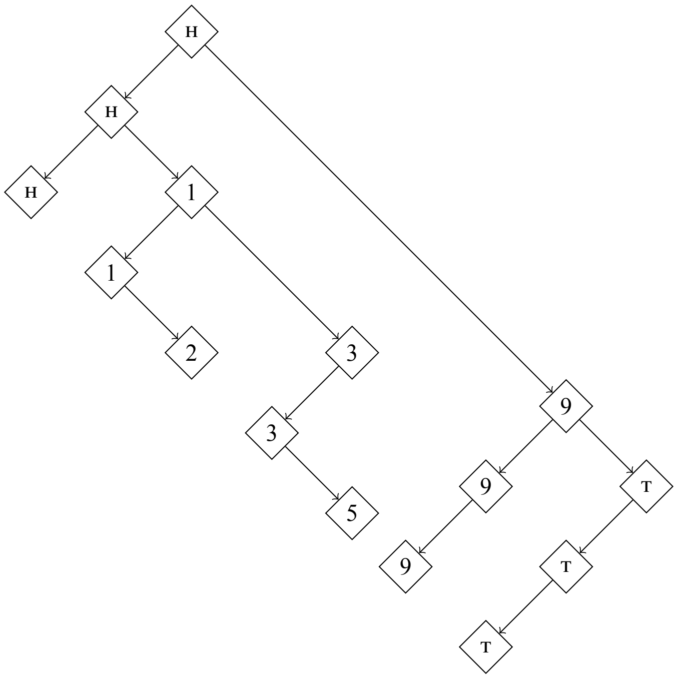

Figure 5.2. The same skiplist as in Figure 5.1, viewed as a binary search tree.

the correct node holding the key. The set of levels in which a node participates is determined at node creation: nodes are randomly assigned a *height*, and participates in levels up to that height. For convenience, a skiplist typically has a *head sentinel* node and a *tail sentinel* node, at the start and the end of the list, respectively. Both sentinel nodes are assigned the maximal height, and initially, all forward pointers of the head node point to the tail node. The keys of the head and tail nodes are defined as being smaller respectively greater than all other keys.

In Figure 5.1, a skiplist with a maximum level of three is depicted. The list consists of five elements, and the head and tail nodes. In the figure, the highest level list only consists of the head, the element 9, and the tail node. Skiplists were conceived as an alternative to balanced search trees, such as red-black trees and AVL trees. From that perspective, one can view the same skiplist as a binary search tree, as depicted in Figure 5.2.

The procedure for locating a key k in a skiplist is as follows: Starting at the head node, the highest level list is searched, and as soon as a key greater than k is encountered, the search continues on the next lower level. Finally when the lowest level is reached, either the node with key k is found, or there is no key k in the skip list. The search is depicted in Figure 5.1. The efficiency of the search depends on the distribution of the heights of the nodes. The heights of the nodes are randomly generated, typically according to a geometric distribu-

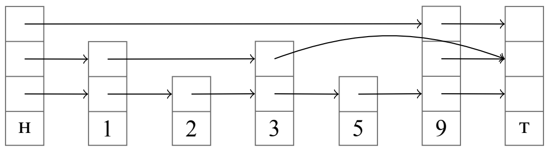

Figure 5.3. A skiplist not complying with the skiplist invariant. The third level list (the highest level) is not a subset of the second level list, since the top (level two) pointer of node 3 points directly to the tail sentinel node, and not to node 9.

tion (i.e., independent coin tosses). Hence the number of nodes at each level decreases exponentially, which results in that the expected maximum length of the search path to any node is $O(\log(n))$, for a skiplist with n nodes [57]. This property is also enjoyed by any balanced binary search tree, such as AVL trees or red-black trees, but they may need rebalancing to achieve that bound in the general case. They may not need rebalancing if the input is suitably distributed (e.g., random). In contrast, skiplists provide the same guarantee independent of input.[1]

Skiplist algorithms tend to be simpler than corresponding balanced search tree operations, while maintaining their properties of $O(\log(n))$ search length, making them an appealing option. For insertion and deletion, only pointers of the predecessor at each level has to be modified. Since the height of the tree is maintained by the randomly assigned heights of the nodes, there is no need for complex rebalancing procedures. Due to the locality of the insertion and deletion operations, skiplists have therefore been considered suitable for parallel applications, and in particular as a replacement for parallel search trees, that tend to be very complex to design [56].

The skiplist invariant states that, in a skiplist, the nodes at each level of the list is a subset of the nodes of the list below it [37]. Skiplists are almost always described and depicted as if the skiplist invariant holds, which is true for all existing sequential skiplist algorithms. However, the skiplist invariant is difficult to maintain in the concurrent case, and not all algorithms do that since it is not necessary for the algorithm to work correctly. Such situations may occur when, e.g., INSERT and REMOVE operations simultaneously modify the skiplist. In Figure 5.3, an example of a state that most concurrent skiplist algorithm can reach is shown. In the figure, the third level (the highest level) list is not a subset of the second level list, but apart from appearing differently than expected there is no effect on correctness. For correctness, it is however assumed in most cases, that all higher level lists are subsets of the lowest level list. Herlihy et al. [37] argues that such loose constraints on the structure of skiplists makes it harder to reason about their correctness, and designs a skiplist algorithm where the skiplist invariant is maintained.

[1]Skewed removals, where only higher level nodes are removed, may cause the structure of the skiplist to degenerate, resulting in a worse search time.

5.1 Lock-freedom and Correctness for Concurrent Data Structures

Concurrent data structures are accessed by multiple threads concurrently, which communicate by modifying the state of the data structure, stored in shared-memory. The threads' interactions with the data structure may be interleaved in many ways, making their behavior less straightforward to reason about. To better understand concurrent algorithms, they are often classified depending on which type of progress they guarantee. In concurrent algorithms, it is also not evident what it means for parallel modifications of a concurrent data structures to be correct. Below, we first introduce a type of progress guarantee, followed by maybe the most common correctness criterion.

Concurrent data structures are often broadly categorized as either *blocking*, where the operation of one thread may block that of another thread; and *non-blocking*, where this is not the case. The most commonly used non-blocking progress guarantee is arguably *lock-freedom*. An algorithm is lock-free if there always is some thread making progress within a finite number of steps of the execution of a concurrent algorithm, independent of the behavior of a (adversarial) scheduler. As a consequence, algorithms that uses some kind of mutual exclusion are not lock-free: if the thread currently holding a lock is suspended, then eventually all other threads may wait for the lock to be released, preventing any thread from making progress.

Linearizability [36] is a correctness criterion for concurrent algorithms. A concurrent algorithm is said to be linearizable, if, for any parallel execution of its operations,

- for each invocation of an operation, it appears as if the operation takes effect instantaneously somewhere between its invocation and its response, and
- the total ordering of these instantaneous operations corresponds to a valid sequential execution.

Most lock-free data structures rely on a specific hardware instruction called *compare and swap* (CAS) for performing their operations. The CAS operation writes to a memory location, but only on the condition that the contents of the memory location contains the expected value. The operation is atomic, i.e., the test condition and the write are performed uninterrupted in hardware, such that the memory contents cannot be modified after the test condition but before the write. The complexity of the instruction makes it expensive to use, compared to other instructions. However, not using some kind of hardware primitive is not an option; Attiya et al. [2] has shown that it is impossible to actually implement concurrent data structures without the support of some (expensive) synchronizing hardware primitive. Also, multiple threads trying to modify the same memory locations with CAS instructions may lead to severe contention. Therefore, to improve performance of concurrent data structures, the goal should be to minimize the usage of CAS.

5.2 Priority Queues

A priority queue is an abstract data type that defines at least two operations, DeleteMin and Insert. The DeleteMin operation returns the element with the smallest key in the queue. The Insert(k,v) operation inserts a value *v* associated with a key *k*, also called the priority of the value. A priority queue can be implemented efficiently by, e.g., heap or tree data structures, and by skiplists. Due to skiplists having shown to be suitable for parallelization, a number of skiplist-based priority queue algorithms have been proposed for parallel use, of which the first one was proposed by Shavit and Lotan [64]. They note that intricate parallel synchronization schemes have been developed for heap-like structures, the most prominent being the one of Hunt et al. [40], but that such structures nonetheless do not scale beyond 10 to 20 processors. They provide a skiplist-based priority queue with superior scalability when evaluated on a simulated 256 processor computer. The key technique in the paper is a separation of logical and physical deletion: a node is considered to be deleted when it is marked, e.g., by a flag, but the actual unlinking of the node from the data structure can proceed in several steps. The first lock-free skiplist-based priority queue was presented by Sundell and Tsigas [70]. That algorithm also distinguishes between logical and physical deletion, but limits the number of simultaneously allowed logically deleted nodes to one, to achieve linearizability. Hence, when an operation observes a logically deleted node, it helps to complete the physical removal of it. However, this leads to increased contention, since multiple threads try to modify the same memory operations with some synchronizing hardware primitive, e.g., CAS.

More recently, focus has shifted to priority queues with relaxed semantics of the DeleteMin operation: Instead of returning the element with *the* smallest key, an element with a “small” key (where small is not necessarily well-defined or bounded) is returned [1, 61, 74]. Notably, Rihani et al. [60] introduces multi-queues, where each processor core is assigned a sequential priority queue. Insert operations are randomly distributed over all the priority queues. For DeleteMin operations, the minimum element of two queues is selected. Interestingly, this point of view can be applied to space-parallel PDES. Removal of the earliest event from an LP’s local event queue can effectively be seen as the DeleteMin on a global event queue with relaxed semantics.

5.3 Challenges

In the share-everything approach to PDES introduced in Chapter 3, the performance of the event queue is crucial to achieve good scalability. One type of data structure that can be used as an event queue is a priority queue. Due to the intrinsic bottleneck of priority queues being the DeleteMin operation, the challenge is to minimize the contention of DeleteMin operation. In Paper I, we introduce a priority queue with minimal memory contention.

6 Summary of Papers

I A Skiplist-Based Concurrent Priority Queue with Minimal Memory Contention

Priority queues are fundamental to many multiprocessor applications, such as scheduling, graph algorithms, and discrete event simulation. For concurrent priority queues, the DeleteMin operation is an inherent bottleneck.

In Paper I, a linearizable lock-free skiplist-based priority queue is presented. The new data structure improves upon earlier skiplist-based priority queues by minimizing the contention suffered by the DeleteMin operation. The amortized number of CAS instructions per DeleteMin operation is reduced to close to one. Earlier lock-free skiplist-based priority queues have used a technique where on average 4 CAS operations are needed per node deletion [70], assuming geometrically distributed heights of the nodes with a parameter of 0.5 (which is the typical case). The high number of CAS operations, and the way the operations are performed, lead to a higher risk that an operation suffers from contention.

Our new priority queue is evaluated on a synthetic benchmark with typical insertion and deletion patterns, and a parallel simulation benchmark. In comparison with other skiplist-based priority queue algorithms, the algorithm improves performance by 30–80%.

We argue for the correctness of the algorithm and its linearizability, by proving a number of invariants on the structure of the skiplist. Additionally, we provide a model of the priority queue to be used with the SPIN model checker [38]. The model has been used to verify the linearizability of the algorithm by extensive state-space exploration. In short, a sequential priority queue is evaluated simultaneously with the parallel priority queue: every time a linearization point is reached in the parallel model implementation, the same operation is performed on the sequential priority queue, at which point the states of the parallel and sequential priority queue are compared.

II Efficient Inter-Process Synchronization for Parallel Discrete Event Simulation on Multicores

In optimistic PDES, one challenge is to make LPs advance their local simulation at approximately the same rate. To achieve a common simulation rate for all LPs, optimism control techniques have been proposed, that blocks or pauses LPs that tend to be over-optimistic. One problem with optimism control techniques is to choose which information should be used for throttling the optimism, and how that information should be used. Earlier work (e.g., [66]) observe that many optimism control techniques suffer from acting on non-exact information, retrieved from, e.g., local history, or by applying global synchronization.

In Paper II, an optimism control technique suitable for spatial stochastic simulation is introduced. The technique, called DLTWE, finds and disseminates accurate estimates of the timestamps of future messages to neighboring LPs. Neighboring LPs can then use these estimates as bounds for local simulation, reducing the risk of rollbacks. The technique is targeting stochastic simulation, without lower bound on inter-arrival times of messages. The technique is applied to the *all events method* (AEM), an algorithm for parameter sensitivity estimation of RDME models. The AEM stores all scheduled events explicitly in memory. The basic idea is to scan a limited prefix of the event queue at periodic intervals, and if any inter-LP diffusions are found, timestamps are disseminated to the corresponding receiving LPs.

An additional improvement is the selective rollback technique. It is similar to the breadth-first search proposed by Deelman and Szymanski [13]. However, since in our case there is a single event history for the whole subdomain of an LP, we don't need to store the processed diffusion event twice, as described in [13]. The reduced memory footprint results in a noticeable speed improvement.

The DLTWE algorithm is evaluated on a set of synthetic and real-world benchmarks, in different sizes, shapes and with different kind of reactions, representing some typical loads that can be expected in systems biology applications. The algorithm displays a parallel efficiency of 25–81% on larger benchmarks compared to sequential AEM. On smaller benchmarks, the corresponding figure is 8–42%.

The algorithm was also compared to another optimism control technique, the *probabilistic adaptive direct optimism control* (PADOC), proposed by Ferscha [22]. The technique was chosen as a representative of the group of optimism control techniques based on local history. The technique relies on message arrival statistics, from which estimates of future incoming messages are calculated. PADOC is compared with the DLTWE technique on a subset of the benchmark set, where the DLTWE technique is shown to outperform the PADOC technique.

III Exposing Inter-Process Information for Efficient PDES of Spatial Stochastic Systems

The NSM is one of the more popular methods for stochastic simulation of chemical systems, e.g., in cellular biology. The efficiency of NSM relies on an aggregated representation of events, thus requiring less memory. However, since events are aggregated, there is less information available that can be used to predict the future behavior of the simulation. In the previous paper, Paper II, the problem was the opposite. We had an abundance of information to act upon, but maintaining the extra information caused the algorithm to be slower.

In Paper III, we propose a refined representation for the NSM, which although slightly more expensive to maintain due to more information that has to be maintained, scales better than the direct parallel implementation of the NSM. The refinement consists of representing key parts of the simulation state explicitly while keeping the lion's share of the state in aggregated form. Thus, the refined algorithm is slightly more expensive to maintain, computation and memory-wise. However, the extra non-aggregated information enables the use of the DLTWE technique first proposed in Paper II. The refined algorithm is called *refined parallel NSM* (PNSM).

To evaluate the refined parallel NSM algorithm, we design a straightforward parallelization of the NSM, called *direct* PNSM. The core NSM algorithm in the direct parallelization is more efficient than the core NSM algorithm in the refined PNSM since all events are aggregated. However, due to the lack of accurate information, the implementation of an optimism control is less straightforward. We provide a best-effort optimism control for the direct PNSM, where future message times are estimated based on diffusion intensities and subvolume populations. The estimated message times are disseminated between LPs.

In the paper, we compare the refined and the direct PNSM algorithms on a set of synthetic benchmarks of different sizes and shapes, representing topologies that can be expected to be found in real simulations. In the evaluation, we show that not using any optimism control does not scale. Then we look at the performance of the two algorithms, direct and refined PNSM. In general, the refined PNSM achieves an average speedup of 17 when executed on 32 cores. Compared to the direct PNSM algorithm, the refined algorithm displays a superior parallel efficiency by 42%, on a reduced benchmark set (due to problems with some of the more advanced benchmarks). Compared to parallel NSM algorithms from other works, the refined PNSM show a superior parallel efficiency by a margin of 65%.

IV Fine-Grained Local Dynamic Load Balancing in PDES

There are numerous optimism control methods to improve the performance in PDES. However, perhaps the most important prerequisite for achieving high performance is an evenly distributed load among processors. Phenomena of interest in, e.g., cellular biology, are often non-homogeneous and migrate over the simulated domain. Cell division regulation and the transmission of nerve impulses are examples of such processes. Hence, the distribution of work (over the model) vary over time, although it is desirable that each processing element has a constant partition of the total work attributed to it.

In Paper IV, we provide a protocol for fine-grained dynamic load migration in PDES, developed with spatial stochastic simulation in mind, where the basic elements of computation (in this case, voxels) are rather small. In the protocol, individual voxels are migrated between LPs. The protocol is combined with a fine-grained local load balancing algorithm, which optimizes both for load and inter-LP communication. The load balancing algorithm is local, meaning that only the direct neighbors to a voxel are involved, in contrary to many other load balancing algorithms, where load balancing is done based on each LP's load metric in relation to a global calculation of the load. The load balancing in our algorithm is based on local load and communication metrics. Our metric is based on the observation that there would be few rollbacks if the simulation load were well balanced. From that point of view, rollbacks caused by stragglers are arguably a good local measure of load imbalance. Thus, we use the number of locally incurred rollbacks caused by stragglers as a load metric.

The PNSM algorithm to which we apply the load balancing algorithm uses a technique for aggregating anti-messages between LPs, to reduce the amount of communication (and thereby improve performance). The simultaneous usage of aggregated anti-messages and a load migration protocol leads to several complex interactions between anti-messages and migrations of voxels, because of how the aggregated anti-messages change in meaning when a voxel is migrated between LPs. Therefore, we provide a proof of correctness of the migration protocol, where we establish a number of invariants on the states of voxels' event histories and the states of the inter-LP communication channels.

The load balancing protocol and the load balancing algorithm are evaluated on some benchmarks taken from systems biology. The load migration protocol is shown to have a low overhead, and substantially reduces the total number of rollbacks during a simulation run.

7 Summary and Conclusions

In this thesis, new techniques for improving parallel efficiency by reducing synchronization costs in PDES were developed. This thesis shows that it is possible to achieve good parallel efficiency for optimistic PDES of spatial stochastic simulation methods such as the NSM. We have managed to improve the scalability of a parallelization of the NSM in comparison to previous work in the same field. We have observed that there is a lack of realistic benchmarks that could serve as a means for comparison between different simulator implementations.

Paper I introduced a skiplist-based priority queue, to be used, e.g., as a central event queue in share-everything PDES. The possibility of further reducing the contention of the DeleteMin operation seems limited. A microbenchmark in the paper indicates that the performance of the priority queue is essentially limited by the DeleteMin operation. In the paper, approximately one CAS instruction per DeleteMin operation is necessary, amortized over time. Recent research results indicate that it is not possible to design a linearizable priority queue without using at least one expensive synchronizing hardware primitive per operation [2].

Papers II and III introduced optimism control protocols, where accurate estimates of timestamps of future inter-LP events, called DLTWEs, are disseminated between LPs The estimates are used by LPs to throttle their local simulation, which reduces expensive rollbacks.

In Paper II, the estimates are produced by scanning a prefix of an LP's event queue. The new optimism control technique is compared to an optimism control technique where estimates are based on an LP's local history of processed events. The results indicate that the accuracy of DLTWEs over the estimates based on local history has a notable impact on the parallel efficiency of space-parallel PDES.

In Paper III, the DLTWE estimates are produced by refining the simulation algorithm in a way which makes more precise information about future inter-LP events available. The approach, called refined PNSM, is compared to a straightforward parallelization of NSM, where best-effort estimates are produced from the available non-refined information. The results suggest that for many threads, the parallel efficiency achieved by using precise inter-LP event information outweighs the cost of the increased overhead of the refined simulation algorithm.

The results from Paper III suggest that load imbalance, both dynamic and static, sets an upper limit on the parallel efficiency that can be achieved. Hence, to improve scalability, it is necessary to balance the load dynamically, an approach that is pursued in Paper IV. In the paper, a fine-grained local dynamic load balancing protocol is proposed. In its evaluation, it is observed that the number of rollbacks during several benchmarks are substantially reduced by using load balancing. However, the speedup did not increase by the same degree. The approach from Paper III still shows significantly better parallel efficiency, even when the load is dynamically changing.

Future Work

In Paper III, we saw evidence indicating that the dynamic load imbalance limited the achievable parallel efficiency in our parallelization of the NSM, for models that were not well-balanced. Hence, to improve parallel efficiency further, load balance must be improved. Thus, a good dynamic load balancing technique is needed. In addition to the migration protocol, which was the focus of Paper IV, one also needs good strategies for selecting which load to move. Not much work has been done on local load balancing algorithms, which could result in some interesting outcomes. Load balancing is also highly relevant for simulation in systems biology, since the phenomena that are studied are often non-homogeneous and dynamic — that is what makes them interesting and the reason stochastic methods are applied in the first place.

The performance results from Paper I suggest that priority queues with exact semantics should preferably be used on a single multicore or manycore processor for best performance, as the scalability deteriorates when more than one processor is used, due to the limited capacity of the bus between processors. Thus, it could be suitable to use for a hybrid PDES approach, where share-everything PDES is used internally on each multicore processor, and optimistic PDES is used between processors.

Even though initially, we discarded share-everything PDES for NSM as inefficient, it would be interesting to evaluate it more thoroughly. We note that would be suitable for share-everything PDES: In NSM, there is exactly one event per subvolume in the event queue. Hence, when a thread retrieves an event from the event queue, it implicitly "locks" access to the subvolume state, since there is no other event for the same subvolume in the event queue. However, Diffusion events modify two subvolumes, and could hence cause a causality error, unless carefully orchestrated.

8 Summary in Swedish – Synkronisering i parallell diskret händelsestyrd simulering

Datorsimulering är modellering av ett system, reellt eller imaginärt, över tid. Datorsimulering återfinns som verktyg inom en mängd skilda vetenskaper, både inom naturvetenskap och samhällsvetenskap. Systembiologi, epidemilogi och mikroprocessordesign är några av många användningsområden för simulering. Simulering används för att utvärdera system, när det till exempel när för kostsamt eller för riskabelt att utvärdera systemet i verkligheten. Det kan användas för att utvärdera egenskaper hos en design i ett tidigt skede, innan tillverkning har påbörjats, för att säkerställa att designen fungerar som förväntat.

Diskret händelsestyrd simulering är en typ av simuleringsmetod där "händelser" driver simuleringen av en modell framåt. En händelse är en diskret förändring av ett tillstånd. En händelse kan till exempel vara att en individ inom en population blir smittad av en sjukdom. Antagandes att modelltillståndet består av antalet friska samt smittade individer, så består den diskreta förändringen i att antalet friska individer minskas med en individ, och antalet smittade individer ökas med en individ. En händelse antas alltid ske vid en viss tidpunkt, och antas ske ögonblickligen. En diskret händelsestyrd simulering är en kronologisk sekvens av sådana händelser, det vill säga ett händelseförlopp. Ett händelseförlopp genereras av en simulator. En simulator består oftast av en händelsekö, som innehåller samtliga framtida händelser sorterade i kronologisk ordning, en klocka som beskriver tiden i simuleringen, och ett modelltillstånd. I simulatorn förknippas varje händelse med en *tidsstämpel*, dess tidpunkt. Simulatorn går till väga enligt följande: (i) den tidigaste händelsen i händelsekön plockas ut ur kön, (ii) modelltillståndet uppdateras i enlighet med händelsen, (iii) klockan sätts till händelsens tidsstämpel, och (iv) eventuellt nya händelser, orsakade av den utvalda händelsen, schemaläggs och sätts in i händelsekön. Dessa fyra steg repeteras fram tills att klockan når en förbestämd sluttid. I kontrast till diskret händelsestyrd simulering återfinns tidsstyrd simulering, där ett diskret tidssteg görs i varje steg av simuleringen. Vid varje tidssteg beräknas hela modellens tillstånd på nytt.

Utveckling av allt komplexare system inom områden såsom mikroarkitektur har skapat ett behov av att kunna simulera allt större system. Detta har lett till ett stort intresse för parallell simulering i allmänhet, då sekvensiell simulering, det vill säga simulering som exekveras på en enda processor, är för långsam. I parallell simulering fördelas arbetet av en simulator över flera processorer, för att på så sätt kunna slutföra simuleringen snabbare, då varje processor endast har en mindre del av simuleringen att genomföra.

Inom parallell diskret händelsestyrd simulering finns det flera sätt att fördela arbetet på flera processorer. Denna avhandlingen har fokuserat på två metoder, som båda beskrivs nedan.

Den mest utbredda metoden för parallellisering av diskret händelsestyrd simulering delar upp arbetet medelst data-dekomposition, så kallad dataparallell simulering: simuleringsmodellen partitioneras och dessa fördelas över så kallade *logiska processer*, som i sin tur fördelas över ett antal processorer. Varje logisk process ansvarar för att driva simuleringen för en del av modellen framåt längs en lokal tidslinje, oberoende av de andra logiska processerna. De logiska processerna måste kontinuerligt kommunicera med varandra, då en händelse som härrör från en delsimulering mycket väl kan påverka en annan delsimulering som utförs av annan processor. Det är ofta önskvärt att den parallella simuleringen ger ett resultat som exakt överensstämmer med den sekvensiella simuleringen. För att åstadkomma ett sådant resultat måste en mottagen händelse införlivas på korrekt position i den lokala tidslinjen, sändande och mottagande logisk process måste synkroniseras. Den *lokala kausalitetsprincipen* stipulerar att en händelse vars tidsstämpel föregår en annan händelse måste behandlas först, för att ett korrekt resultat ska erhållas. Då varje delmodell simuleras oberoende av andra delmodeller, kan det vid mottagande av ett meddelande uppstå kausalitetsfel: en händelse emottas för sent, och skulle vid införlivandet i den lokala tidslinjen leda till att en senare händelse påverkar en tidigare händelse. Vid detektion av ett kausalitetsfel initieras en korrektion av den lokala tidslinjen, och andra påverkade logiska processer informeras via meddelanden. Korrektionsprocedurerna är tidsmässigt kostsamma och det är önskvärt att i största möjliga grad undvika dem. Ju större diskrepansen är mellan två logiska processers tidslinjer, desto större sannolikhet är det att deras tidslinjer kommer att behöva korrigeras vid kommunikation. En av de centrala utmaningarna i parallell diskret händelsestyrd simulering är att få samtliga logiska processer att driva sina lokala simuleringar framåt med ungefär samma hastighet, för att reducera tidsskillnaden mellan logiska processers lokala tidslinjer. På så sätt reduceras tiden som behöver läggas på korrektion av tidslinjerna, och simulatorns skalbarhet maximeras.

En annan metod att parallellisera diskret händelsestyrd simulering är att låta samtliga processorer samverka genom en central händelsekö. Detta gör att synkroniseringen mellan processorer förenklas avsevärt, då den regleras av den centrala händelsekön. Å andra sidan så leder det till att händelsekön kan bli en flaskhals, då flera processorer samtidigt vill extrahera den tidigaste händelsen.

Denna avhandling utvecklar metoder för att reducera synkroniseringkostnader inom parallell diskret händelsestyrd simulering, med målet att förbättra prestanda. Tre områden studeras i avhandlingen.

Vid parallell diskret händelsestyrd simulering medelst en central händelsekö är det viktigt att minimera synkroniseringskostnaderna för de procedurer som modifierar händelsekön, då den är en trolig flaskhals i simuleringen. Ett av resultaten i denna avhandling är en ny algoritm som minimerar synkroniseringskostnaderna för en viss typ av händelsekö.

I dataparallell diskret händelsestyrd simulering är en vanlig teknik för att reducera synkroniseringskostnader att göra så att samtliga logiska processer driver sin lokala simulering framåt med ungefär samma hastighet. Ett sätt att åstadkomma en uniform lokal simuleringshastighet är att blockera logiska processer vars lokala simuleringstid är väsentligt större än andra logiska processers. I avhandlingen har metoder utvecklats för att exponera och kommunicera framtida händelsers tid mellan logiska processer, med fokus på spatiala och stokastiska simuleringsmodeller. Resultaten visar att metoderna bidrar till ökad skalbarhet vid simulering av den sortens modeller.

I dataparallell diskret händelsestyrd simulering är en ojämn fördelning av simuleringsarbetet en vanlig orsak till höga synkroniseringskostnader. Ojämn fördelning av arbetet kan uppstå dynamiskt under pågående simulering på grund av egenskaper hos simuleringsmodellen. Genom att omfördela arbetet kontinuerligt under en pågående simulering kan en jämn arbetsfördelning bibehållas. I avhandlingen har en metod utvecklats för att dynamiskt omfördela arbetet under en pågående simulering. Resultaten visar att antalet nödvändiga korrektioner av de logiska processernas tidslinjer reduceras substantiellt.

Acknowledgements

First and foremost, I would like to thank my supervisor Bengt Jonsson, without whom this work would never have seen the light of the day. I am truly thankful for your patience, for all your hard work, and for always having had some encouraging words at the right moments. I would also like to thank my co-supervisor Yi Wang for being positive and encouraging to my work. Second, I would like to thank my co-authors Pavol Bauer, Xiaoyue Pan and Stefan Engblom. Thank you for your collaboration and interesting ideas, I learned a lot together with you.

I would like to thank the people and colleagues of the embedded systems/real time systems group, past and current, in no particular order, for contributing to a nice work environment: Jakaria Abdullah, Peter Backeman, Gaoyang Dai, Pontus Ekberg, Jonas Flodin, Nan Guan, Kai Lampka, Xiuming Liu, Mingsong Lv, Morteza Mohaqeqi, Edith Ngai, Xiaoyue Pan, Philipp Rümmer, Martin Stigge, Wang Yi and Aleksandar Zeljić.

Many thanks go to Peter Backeman, Aleksandar Zeljić, and Stefan Engblom, who in addition to my supervisors gave me valuable feedback on this thesis.

I would also like to thank all the Ph.D. students that in some way or another have crossed my path during my time at the university and made it a better place. Special mentions go to Nikos Nikoleris, Mikael Laaksoharju and David Eklöv for giving me a great start and introducing me to what it means to be a PhD.

I would like to thank my parents for their support. This work owes to you, thank you for helping out when two people is not enough. I would also like to thank my brother for being my brother. Thank you.

Last, but not least, I would like to thank my family for making life magic. Thank you Anne, for all the wonderful journeys we have done and are doing and will do together. Thank you for being there, supporting me. Thank you Finn, for writing small books, just like your parents, and for which you need the scissors. Thank you Matilda for expressing everything with your few but wonderful words.

This work was supported in part by the Swedish Foundation for Strategic Research through the project CoDeR-MP and the Swedish Research Council within the UPMARC Linnaeus centre of Excellence.

References

[1] D. Alistarh, J. Kopinsky, J. Li, and N. Shavit. The SprayList: A Scalable Relaxed Priority Queue. *ACM SIGPLAN Not.*, 50(8):11–20, 2015.

[2] H. Attiya, R. Guerraoui, D. Hendler, P. Kuznetsov, M. M. Michael, and M. Vechev. Laws of Order: Expensive Synchronization in Concurrent Algorithms Cannot Be Eliminated. *ACM SIGPLAN Not.*, 46(1):487–498, 2011.

[3] H. Avril and C. Tropper. Clustered Time Warp and Logic Simulation. *ACM SIGSIM Simul. Dig.*, 25(1):112–119, 1995.

[4] H. Avril and C. Tropper. The Dynamic Load Balancing of Clustered Time Warp for Logic Simulation. *ACM SIGSIM Simul. Dig.*, 26(1):20–27, 1996.

[5] H. Bauer and C. Sporrer. Reducing Rollback Overhead In Time-warp Based Distributed Simulation With Optimized Incremental State Saving. In *Proc. 26th Annual Simulation Symposium*, pages 12–20, 1993.

[6] R. E. Bryant. Simulation of Packet Communication Architecture Computer Systems. Tech. Rep. MIT-LCS-TR-188, Massachusetts Institute of Technology, 1977.

[7] C. Burdorf and J. Marti. Load Balancing Strategies for Time Warp on Multi-User Workstations. *Comput. J.*, 36(2):168–176, 1993.

[8] C. D. Carothers, K. S. Perumalla, and R. M. Fujimoto. Efficient optimistic parallel simulations using reverse computation. *ACM Trans. Model. Comput. Simul.*, 9(3):224–253, 1999.

[9] C. D. Carothers, K. S. Perumalla, and R. M. Fujimoto. The effect of state-saving in optimistic simulation on a cache-coherent non-uniform memory access architecture. In *Proc. 31st Conference on Winter Simulation*. ACM, 1999.

[10] C. G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Springer, 2nd edition, 2008.

[11] K. M. Chandy and J. Misra. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. *IEEE Trans. Softw. Eng.*, SE-5(5): 440–452, 1979.

[12] S. R. Das. Adaptive protocols for parallel discrete event simulation. *J. Oper. Res. Soc.*, 51(4):385–394, 2000.

[13] E. Deelman and B. K. Szymanski. Breadth-first Rollback in Spatially Explicit Simulations. *ACM SIGSIM Simul. Dig.*, 27(1):124–131, 1997.

[14] E. Deelman and B. K. Szymanski. Dynamic Load Balancing in Parallel Discrete Event Simulation for Spatially Explicit Problems. *ACM SIGSIM Simul. Dig.*, 28 (1):46–53, 1998.

[15] L. Dematté and T. Mazza. On parallel stochastic simulation of diffusive systems. In M. Heiner and A. M. Uhrmacher, editors, *Computational Methods in Systems Biology*, number 5307 in LNCS, page 191–210. Springer, 2008.

[16] B. Drawert, S. Engblom, and A. Hellander. URDME: A modular framework for stochastic simulation of reaction-transport processes in complex geometries. *BMC Syst. Biol.*, 6(76):1–17, 2012.

[17] J. Elf and M. Ehrenberg. Spontaneous separation of bi-stable biochemical systems into spatial domains of opposite phases. *Syst. Biol.*, 1(2):230–236, 2004.

[18] S. Engblom. *Numerical Solution Methods in Stochastic Chemical Kinetics*. PhD Thesis, Uppsala University, 2008.

[19] S. Engblom, L. Ferm, A. Hellander, and P. Lötstedt. Simulation of Stochastic Reaction-Diffusion Processes on Unstructured Meshes. *SIAM J. Sci. Comput.*, 31(3):1774–1797, 2009.

[20] R. Erban, J. Chapman, and P. Maini. A practical guide to stochastic simulations of reaction-diffusion processes. *ArXiv07041908 Phys. Q-Bio*, 2007.

[21] D. Fange and J. Elf. Noise-Induced Min Phenotypes in E. coli. *PLOS Comput. Biol.*, 2(6):e80, 2006.

[22] A. Ferscha. Probabilistic Adaptive Direct Optimism control in Time Warp. *ACM SIGSIM Simul. Dig.*, 25(1):120–129, 1995.

[23] M. L. Fredman, R. Sedgewick, D. D. Sleator, and R. E. Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1(1-4):111–129, 1986.

[24] R. M. Fujimoto. Parallel Discrete Event Simulation. *Commun. ACM*, 33(10): 30–53, 1990.

[25] R. M. Fujimoto and M. Hybinette. Computing Global Virtual Time in Shared-Memory Multiprocessors. *ACM Trans. Model. Comput. Simul.*, 7(4): 425–446, 1997.

[26] C. W. Gardiner. *Handbook of Stochastic Methods*. Springer, 3rd edition, 2004.

[27] T. S. Gardner, C. R. Cantor, and J. J. Collins. Construction of a genetic toggle switch in Escherichia coli. *Nature*, 403:339, 2000.

[28] M. A. Gibson and J. Bruck. Efficient Exact Stochastic Simulation of Chemical Systems with Many Species and Many Channels. *J. Phys. Chem. A*, 104(9): 1876–1889, 2000.

[29] D. T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *J. Phys. Chem.*, 81(25):2340–2361, 1977.

[30] D. T. Gillespie. Stochastic simulation of chemical kinetics. *Annu. Rev. Phys. Chem.*, 58:35–55, 2007.

[31] D. W. Glazer and C. Tropper. On Process Migration and Load Balancing in Time Warp. *IEEE Trans. Parallel Distrib. Syst.*, 4(3):318–327, 1993.

[32] S. Gupta and P. A. Wilsey. Lock-free Pending Event Set Management in Time Warp. In *Proc. 2nd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, pages 15–26. ACM, 2014.

[33] J. Hay and P. A. Wilsey. Experiments with Hardware-based Transactional Memory in Parallel Simulation. In *Proc. 3rd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, pages 75–86. ACM, 2015.

[34] P. Heidelberger. Statistical Analysis of Parallel Simulations. In *Proc. 18th Conference on Winter Simulation*, pages 290–295. ACM, 1986.

[35] P. Heidelberger and H. S. Stone. Parallel trace-driven cache simulation by time partitioning. In *Proc. 22nd Conference on Winter Simulation*, pages 734–737. IEEE, 1990.

[36] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[37] M. Herlihy, Y. Lev, V. Luchangco, and N. Shavit. A simple optimistic skiplist algorithm. In G. Prencipe and S. Zaks, editors, *Structural Information and Communication Complexity*, number 4474 in LNCS, pages 124–138. Springer, 2007.

[38] G. J. Holzmann. The Model Checker SPIN. *IEEE Trans. Softw. Eng.*, 23(5): 279–295, 1997.

[39] D. A. Huffman. A Method for the Construction of Minimum-Redundancy Codes. *Proc. IRE*, 40(9):1098–1101, 1952.

[40] G. C. Hunt, M. M. Michael, S. Parthasarathy, and M. L. Scott. An Efficient Algorithm for Concurrent Priority Queue Heaps. *Inf. Process. Lett.*, 60(3): 151–157, 1996.

[41] S. Jafer, Q. Liu, and G. Wainer. Synchronization methods in parallel and distributed discrete-event simulation. *Simul. Model. Pract. Theory*, 30:54–73, 2013.

[42] D. R. Jefferson. Virtual time. *ACM Trans. Program. Lang. Syst.*, 7(3):404–425, 1985.

[43] M. Jeschke, R. Ewald, A. Park, R. M. Fujimoto, and A. M. Uhrmacher. A parallel and distributed discrete event approach for spatial cell-biological simulations. *ACM SIGMETRICS Perform. Eval. Rev.*, 35(4):22–31, 2008.

[44] M. Jeschke, A. Park, R. Ewald, R. Fujimoto, and A. M. Uhrmacher. Parallel and Distributed Spatial Simulation of Chemical Reactions. In *22nd Workshop on Principles of Advanced and Distributed Simulation*, pages 51–59. IEEE, 2008.

[45] M. J. Lawson, B. Drawert, M. Khammash, L. Petzold, and T.-M. Yi. Spatial Stochastic Dynamics Enable Robust Cell Polarization. *PLOS Comput. Biol.*, 9 (7):e1003139, 2013.

[46] Z. Lin, C. Tropper, M. N. I. Patoary, R. A. McDougal, W. W. Lytton, and M. L. Hines. NTW-MT: A Multi-threaded Simulator for Reaction Diffusion Simulations in NEURON. In *Proc. 3rd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, pages 157–167. ACM, 2015.

[47] J. Liu. Parallel Discrete-Event Simulation. In *Wiley Encyclopedia of Operations Research and Management Science*. Wiley, 2010.

[48] A. J. Lotka. *Elements of Physical Biology*. Williams & Wilkins Company, 1925.

[49] R. Marotta, M. Ianni, A. Pellegrini, and F. Quaglia. A Conflict-Resilient Lock-Free Calendar Queue for Scalable Share-Everything PDES Platforms. In *Proc. 2017 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, pages 15–26. ACM, 2017.

[50] M. Matsumoto and T. Nishimura. Mersenne Twister: A 623-dimensionally Equidistributed Uniform Pseudo-random Number Generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, 1998.

[51] F. Mattern. Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation. *J. Parallel Distrib. Comput.*, 18:423–434, 1993.

[52] N. Nguyen, P. Tsigas, and H. Sundell. ParMarkSplit: A Parallel Mark-Split Garbage Collector Based on a Lock-Free Skip-List. In M. K. Aguilera, L. Querzoni, and M. Shapiro, editors, *Principles of Distributed Systems*, number 8878 in LNCS, pages 372–387. Springer, 2014.

[53] D. M. Nicol and X. Liu. The Dark Side of Risk (What Your Mother Never Told You About Time Warp). *ACM SIGSIM Simul. Dig.*, 27(1):188–195, 1997.

[54] D. M. Nicol, A. G. Greenberg, and B. D. Lubachevsky. Massively parallel algorithms for trace-driven cache simulations. *IEEE Trans. Parallel Distrib. Syst.*, 5(8):849–859, 1994.

[55] J. Nutaro. Discrete-Event Simulation of Continuous Systems. In P. A. Fishwick, editor, *Handbook of Dynamic Systems Modeling*. Chapman & Hall/CRC, 2005.

[56] W. Pugh. Concurrent maintenance of skip lists. Tech. Rep. UMIACS-TR-90-80, University of Maryland at College Park, 1990.

[57] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990.

[58] H. Rajaei, R. Ayani, and L.-E. Thorelli. The Local Time Warp Approach to Parallel Simulation. *ACM SIGSIM Simul. Dig.*, 23(1):119–126, 1993.

[59] P. L. Reiher and D. R. Jefferson. Virtual Time Based Dynamic Load Management In The Time Warp Operating System. *Trans. Soc. Comput. Simul.*, 7(9):103–111, 1990.

[60] H. Rihani, P. Sanders, and R. Dementiev. Brief Announcement: MultiQueues: Simple Relaxed Concurrent Priority Queues. In *Proc. 27th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 80–82. ACM, 2015.

[61] K. Sagonas and K. Winblad. The Contention Avoiding Concurrent Priority Queue. In C. Ding, J. Criswell, and P. Wu, editors, *Languages and Compilers for Parallel Computing*, number 10136 in LNCS, pages 314–330. Springer, 2017.

[62] R. B. Schinazi. Predator-Prey and Host-Parasite Spatial Stochastic Models. *Ann. Appl. Probab.*, 7(1):1–9, 1997.

[63] R. Schlagenhaft, M. Ruhwandl, C. Sporrer, and H. Bauer. Dynamic Load Balancing of a Multi-cluster Simulator on a Network of Workstations. *ACM SIGSIM Simul. Dig.*, 25(1):175–180, 1995.

[64] N. Shavit and I. Lotan. Skiplist-based concurrent priority queues. In *Proc. 14th Int'l Parallel and Distributed Processing Symposium*. IEEE, 2000.

[65] L. M. Sokol, D. P. Briscoe, and A. P. Wieland. MTW: A strategy for scheduling discrete simulation events for concurrent execution. In *Proc. 1988 SCS Multiconference on Distributed Simulation*, pages 34–42. IEEE, 1988.

[66] S. Srinivasan and P. F. Reynolds. Elastic time. *ACM Trans. Model. Comput. Simul.*, 8(2):103–139, 1998.

[67] J. S. Steinman. SPEEDES: Synchronous Parallel Environment for Emulation and Discrete Event Simulation. *Int. J. Comput. Simul.*, 2:251–286, 1992.

[68] J. S. Steinman. Breathing Time Warp. *ACM SIGSIM Simul. Dig.*, 23(1): 109–118, 1993.

[69] M. Sturrock, A. Hellander, A. Matzavinos, and M. A. J. Chaplain. Spatial stochastic modelling of the Hes1 gene regulatory network: Intrinsic noise can explain heterogeneity in embryonic stem cell differentiation. *J. R. Soc. Interface*, 10(80), 2013.

[70] H. Sundell and P. Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. *J. Parallel Distrib. Comput.*, 65(5):609–627, 2005.

[71] V. Volterra. Variazioni E Fluttuazioni Del Numero D'individui In Specie Animali Conviventi. *Mem. R. Accad. Naz. dei Lincei.*, 2:31–113, 1926.

[72] B. Wang, B. Hou, F. Xing, and Y. Yao. Abstract Next Subvolume Method: A logical process-based approach for spatial stochastic simulation of chemical reactions. *Comput. Biol. Chem.*, 35(3):193–198, 2011.

[73] L. F. Wilson and W. Shen. Experiments in Load Migration and Dynamic Load Balancing in SPEEDES. In *Proc. 30th Conference on Winter Simulation*, pages 483–490. IEEE, 1998.

[74] M. Wimmer, J. Gruber, J. L. Träff, and P. Tsigas. The Lock-free k-LSM Relaxed Priority Queue. In *Proc. 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 277–278. ACM, 2015.

Distribution: publications.uu.se
urn:nbn:se:uu:diva-342270

ACTA
UNIVERSITATIS
UPSALIENSIS
UPPSALA
2018