Approximate Time-Parallel Simulation

Tobias Kiesling

Universität der Bundeswehr München

It is a good morning exercise for a research scientist to discard a pet hypothesis every day before breakfast. It keeps him young.

Konrad Lorenz (1903 - 1989)

Universität der Bundeswehr München

APPROXIMATE TIME-PARALLEL SIMULATION

submitted by

Tobias Kiesling

DISSERTATION

for the achievement of the academic degree
doctor rerum naturalium (Dr. rer. nat.)

at the

Fakultät für Informatik
Universität der Bundeswehr München

Supervisors:

Prof. Dr. Axel Lehmann
Universität der Bundeswehr München
Neubiberg, Germany

Prof. Richard M. Fujimoto
Georgia Institute of Technology
Atlanta, GA, USA

Neubiberg, December, 2005

Acknowledgments

Many persons directly or indirectly contributed to this thesis. I feel especially grateful for the general support of my family, most of all my beloved wife Manuela. The main person, however, who enabled this thesis was Prof. Axel Lehmann. He always supported my work and allowed me great latitude, which is a primary requirement to conduct original research. Furthermore, I am honored and thankful that Prof. Richard Fujimoto, a known expert in the field of parallel and distributed simulation, agreed to act as second supervisor of this thesis. Several colleagues deserve a special mention as well. Siegfried Pohl inspired the first ideas about approximate time-parallel simulation. Thomas Krieger helped with the derivation of the computational overhead of time-parallel queuing simulation presented in Section 7.3. Last but not least, I would like to give special thanks to Johannes Lüthi, who supported the work on time-parallel traffic simulation presented in Chapter 9, but was also constantly willing to discuss various aspects of this thesis.

Abstract

In classical parallel discrete-event simulation, the simulation state space is decomposed into a number of sub-spaces. The responsibility for the simulation of each sub-space is assigned to a separate parallel process and simulation is performed concurrently by the parallel processes. This is also referred to as spatial parallelization. To allow for an efficient parallel simulation, a suitable decomposition has to be found, where parallel processes are largely independent of one another. This is not always possible, as the spatial decomposability of a model might be limited, depending on the specificities of the model state space.

Time-parallel simulation is another approach, where each of the parallel processes is responsible for the simulation of the whole state space, but only for a part of the overall simulation time interval (also referred to as temporal parallelization). The simulated time is decomposed into a number of time slices and the calculations corresponding to each slice are assigned to a separate parallel process. The central problem of this approach are the initial states of the parallel simulations, as these are unknown prior to the simulation execution. The advantages of time-parallel simulation are the achievable parallelism and a high degree of independence between parallel processes.

Most of the existing approaches for parallel simulation only consider accurate parallelization methods, i.e. methods ensuring identical results of the parallel simulation and a corresponding sequential simulation. However, in many cases, no satisfying degree of parallelism can be achieved with accurate parallelization approaches, due to interdependencies between parallel processes. As an alternative, approximate parallel simulation methods are constructed to tolerate a deviation of parallel simulation results from those of a corresponding sequential simulation, giving the opportunity for an efficient parallel simulation. Without a fundamental analysis of the error introduced with the approximation, simulation results might be seriously compromised.

In this thesis, the applicability of approximate methods to the approach of time-parallel simulation is examined. A formal model of time-parallel simulation is introduced and used as a basis for the definition of the approximate parallel simulation method. Furthermore, the approximate time-parallel simulation approach is enhanced to a new parallel simulation approach. This progressive time-parallel simulation is intended to provide imprecise simulation results quickly, improving these results progressively in the continued simulation execution. For a thorough examination of the proposed approaches, three case studies are performed: queuing systems, simulation of caching in computer systems, and road traffic simulation. Theoretical considerations indicate the feasibility of the approximate time-parallel simulation methods, as do experiments with prototypical implementations of approximate simulators.

Zusammenfassung

In der klassischen parallelen Ereignis-basierten Simulation wird der Zustandsraum eines Simulationsmodells in Teilräume zerlegt. Bei der Simulationsausführung werden dann die Änderungen der jeweiligen Zustandsvariablen nebenläufig berechnet (räumliche Parallelisierung). Hier ist es wichtig, eine Zerlegung des Zustandsraumes zu finden, so dass die parallelen Prozesse möglichst unabhängig ausgeführt werden können. Dies ist nicht immer einfach zu erreichen, da die räumliche Zerlegbarkeit eines Modells in Abhängigkeit von der Beschaffenheit des Zustandsraumes oft nur eingeschränkt gegeben ist.

Ein anderer Ansatz ist die Zeit-parallele Simulation, bei der jeder parallele Prozess für die Berechnung der Variablen des gesamten Zustandsraumes verantwortlich ist, aber nur für einen Teil der gesamten Simulationszeit (zeitliche Parallelisierung). Die zu simulierende Zeit wird in Zeitscheiben aufgeteilt, und die Aufgabe der Berechnung jeder einzelnen Scheibe einem separaten Prozess zugewiesen. Das zentrale Problem dieses Ansatzes ist die Tatsache, dass die Zustände am Beginn der Zeitscheiben im Allgemeinen unbekannt sind. Vorteile der zeitlichen Parallelisierung sind der nahezu unbegrenzte Parallelitätsgrad und eine weitgehende Unabhängigkeit zwischen den parallelen Prozessen.

Der größte Teil der Ansätze zur parallelen Simulation beschäftigt sich mit exakten Parallelisierungsverfahren, das sind solche, die im Vergleich zur zugehörigen sequentiellen Simulation identische Lösungen liefern. In vielen Fällen kann durch eine exakte Lösung des Parallelisierungsproblems auf Grund von Abhängigkeiten zwischen logischen Prozessen jedoch kein befriedigender Parallelitätsgrad mehr erreicht werden. Falls man jedoch von der Forderung einer exakten Lösung abrückt, ist es durchaus möglich, erhebliche Leistungssteigerungen zu erzielen. Beim Einsatz approximativer Verfahren muss beachtet werden, dass Simulationsergebnisse nicht wesentlich verfälscht oder sogar unbrauchbar werden. Hierzu sollte der eingeführte Fehler untersucht und das verwendete Verfahren gegebenfalls modifiziert werden, mit dem Ziel, eine akzeptable Ungenauigkeit zu erreichen.

In dieser Dissertation wird die Anwendbarkeit approximativer Verfahren in der Zeit-parallelen Simulation näher untersucht. Hierzu wird ein formales Modell der Zeit-parallelen Simulation vorgestellt und um approximative Anteile erweitert. Ferner wird die approximative Zeit-parallele Simulation zu einem neuen Ansatz weiterentwickelt. Bei dieser progressiven Zeit-parallelen Simulation werden ungenaue Simulationergebnisse möglichst bald geliefert und diese im weiteren Simulationsverlauf sukzessive verbessert. Die vorgestellten Ansätze werden an Hand dreier Anwendungsfälle aus dem Bereich der Warteschlangensysteme, der Simulation von Caching-Verfahren und der Verkehrssimulation näher untersucht. Theoretische Untersuchungen der Anwendung der approximativen Methoden deuten auf die Umsetzbarkeit der vorgestellten Ansätze hin, genauso wie Experimente mit prototypischen Implementierungen.

Contents

List of Figures

Part I.
Background

1. Introduction

1.1. Motivation

In various fields of science, modeling is an essential technique to cope with the complexity of the real world. This spans applications as diverse as meteorological models used for weather prediction, product models used in software development projects, or performance models used in performance engineering of computer and communication systems. In each of these examples, the *model* is “a representation of an object, system, or idea in some form other than that of the entity itself” [93]. In most of the cases, a model is an abstraction of either an existing part of the physical world or of a system yet to be developed. Both of these possibilities are typically subsumed under the terms *real system* [93] or *source system* [109]. The nature of the abstraction when building a model highly depends on its intended purpose, which is the driving motivation for the development of the model. E.g., the intended purpose of a weather model is most probably an accurate prediction of the weather of the following days, but other intended purposes are possible. Due to a variety of different intended purposes, an arbitrary number of different models can be created for a single real system in most of the cases.

Many different kinds of models exist, leading to a variety of model categorizations. First of all, models are classified by the way they are used. An *illustrative model* is intended to illustrate the properties of the real system, as an aid to thought or an aid to communication [23]. One example of this category is a product model in a software development project, where standard methods like the Unified Modeling Language (UML) [37] are used to specify models. An *analytical model* is a mathematical representation of the real system, where analytical solution techniques [38] can be used to calculate properties of the modeled system. To be able to provide these solutions, the model must be described with formal methods, e.g. differential equations. A *simulation model* is a special kind of model intended for the execution on a computer system. A computer simulation is utilized to perform *simulation experiments*, i.e. executions of the simulation model with a given set of input parameters. The advantage of computer simulation, in contrast to analytical model evaluation, is its general applicability to models of varying complexity and size.

Simulation models must properly represent temporal aspects of the system [30]. The *real time* in the system is represented by the *simulation time* in the model. A third notion of time is the *wallclock time* passing during the execution of the simulation model as a computer program. Further classification of simulation models is based on the relationship between simulation time and wallclock time. In a *real-time simulation*, the passing of simulation time is directly proportional to the passing of wallclock time. This type of simulation is typically used in simulation models with continuous user interaction, intended for training or entertainment. If there is no direct relationship between simulation time and wallclock time, the model is called an *as-fast-as-possible simulation*. This is also commonly called an *analytic simulation* [30]. However, in order to avoid confusion with the notion of analytic modeling, the term is avoided in this thesis. This type of simulation is typically used for the quantitative analysis of complex systems, with no or only marginal user interaction.

As stated by its name, as-fast-as-possible simulation is intended to run as fast as is possible to achieve. If there is no explicit limit on the amount of available processing time, in many cases the simulation will be implemented as a *sequential simulation* for a single-processor machine. However, with large or complex models, processing times might increase beyond an acceptable or economically feasible limit. One technique to increase the performance of an existing model is the parallelization of the model, leading to a *parallel or distributed simulation*. In this case, parallel simulation is used as a technique to increase the performance of a model as much as possible. In the context of this thesis, the terms *parallel simulation* and *distributed simulation* are used interchangeably, although distinctions between those terms exist in the literature. There are a number of introductory texts for parallel and distributed simulation [25, 27, 74].

There are a number of different approaches to the parallelization of a given simulation model (see Section 2.2). A commonality of all approaches is the decomposition of the overall simulation task into a number of subtasks executed concurrently on parallel processing nodes.The concurrent execution of a simulation introduces an overhead in comparison to its sequential simulation. The total amount of overhead can be divided into three categories [51]. *Communication overhead* is the cost of having multiple parallel processes, which have to communicate to achieve a common goal. *Synchronization overhead* results from idle processes, having to wait for the progress of other processes before being able to continue. *Computational overhead* results from redundant computations, which might occur if the decomposition leads to an overlapping of concurrent simulation tasks. The performance and efficiency, and thus the feasibility, of a parallel simulation model depends on the amount of overhead that is introduced with the parallelization.

There are several factors influencing the amount of each type of overhead in a parallel simulation. A large influence is due to the execution environment, e.g. a shared-memory multi-processor machine vs. a distributed cluster of workstations connected via Ethernet. Another factor is the type and quality of the parallel implementation. However, this thesis is mostly concerned with the overhead induced by the nature of the model in combination with the parallelization approach. It is commonly accepted in the literature, that the parallelization of a simulation model is a non-trivial task, especially because there are a number of different strategies, none of which can be identified as the single most successful approach. Prior to the development of a parallel model, a careful analysis of the sequential model is necessary to identify suitable decompositions of the overall simulation task.

Classical parallelization approaches (see Section 2.2) are conservative in the sense that they do not change the behavior of the original simulation model. The goal of these approaches is to produce exactly the same results as a corresponding sequential simulation. This has the advantage that once the validity of a simulation model has been established, no further validation activities of parallelizations of this model are necessary. Unfortunately, with these correct methods, a satisfying decomposition of the simulation task sometimes is hard to be found [88, 100]. Therefore, alternative parallelization approaches tolerate deviations of the results of executions of the parallel simulation from that of a corresponding sequential simulation. The viability of these *approximate parallelization approaches* is confirmed by the observation that even a sequential simulation model is an abstraction of the real system, more or less deviating from the real system in its behavior. In many cases, there are uncertainties in the model, where the modeler was not able to precisely capture certain aspects of the real system. These uncertainties can be exploited to improve the performance of the simulation system [29, 86]. When applying approximate parallelization methods, the need for validation of the parallel simulation is introduced, as the parallelization influences the behavior of the resulting model.

If utilized correctly, approximate parallel simulation can be a valuable method for the performance increase of simulation models, especially for types of models, where correct parallelization methods are not viable. In principle, for every simulation parallelization approach, approximate variants can be defined, all with their strengths and weaknesses. Compared to the large amount of work on correct parallelization of simulation models, little research has been performed towards a theory of approximate parallel simulation (see Chapter 3 for a presentation of existing work in this direction). The main part of this thesis is aimed at a contribution to this area of research by the introduction of the novel approach of *time-parallel simulation with approximate state matching*.

When utilized in real-time environments, approximate parallel simulation approaches can be compared to similar methods of other research areas. E.g., in hard real-time systems, the technique of *imprecise computations* [59] has been developed. An imprecise computation is allowed to give imprecise (approximate) results if the precise (correct) results cannot be provided in time. *Progressive processing* is an extension of imprecise computations, where computations are continued after the initial determination of approximate results to improve these results *progressively*. One of the more prominent applications of this technique is *progressive image rendering* [10, 19, 44], where an imprecise (e.g. blurred) or incomplete representation of the image is provided to the user after a short answer time, after which the quality of the image is progressively improved until the best representation of the image has been produced or the user cancels the image rendering process.

Progressive parallel simulation is a novel concept, extending the idea of progressive processing to the area of parallel simulation. It can be utilized effectively in contexts where quick results are desirable, regardless of their precision, but more precise results might be needed later. A second contribution of this thesis is the definition of *progressive time-parallel simulation*, which is the corresponding extension of the approximate time-parallel simulation approach.

1.2. Contributions

As mentioned above, there are two main contributions of this thesis to the research in the area of parallel and distributed simulation. The first one is the novel concept of time-parallel simulation with approximate state matching. Although there has been some work on similar techniques utilizing time-parallel simulation (see Section 3.2.4), none of these is as generally applicable as the method proposed here. The salient features of approximate state matching in time-parallel simulation are the fine granularity of the error control and the applicability to models of various types. An important aspect of approximate parallel simulation is the error that is introduced with the approximation. The characteristics of such an error and possible ways for an error control are discussed in more detail in this thesis.

The second main contribution is the application of progressive processing, developed in other fields of research, to the area of parallel simulation. The technique of progressive time-parallel simulation is proposed and the development of the error over the execution time is discussed in detail.

An important property in the research area of modeling and simulation is the diversity of possible simulation models. Even if only a certain class of simulation models (e.g., discrete-event simulations, see Section 2.1) is considered, the specificities of two different models can be substantially different. This is a significant impediment to the development of general theories in this area of research. If a general theory for a class of models has to be developed, its substance and assertions often have to remain unsatisfyingly vague. A possible way to cope with this problem is to provide more details in conjunction with one or more examples. To generalize about the consequences of applying a parallel simulation approach, a single example is not sufficient.

In this thesis, three models from different problem domains have been chosen as proof-of-concept of the techniques of approximate state matching and progressive time-parallel simulation: queuing system simulation, simulation of caching in computer systems, and road traffic simulation. Queuing system simulation [2, 48] is an important performance modeling technique in various application areas, e.g. performance evaluation of computer and communication systems [38]. The mathematical nature of queuing systems allows for an analytical examination of the consequences of parallel simulation. Simulation of cache replacement policies [95] is used to evaluate the impact of a specific cache implementation in a computer system. Trace-driven cache simulation is characterized by very long run times, leading to a general suitability of time-parallel simulation [40]. Road traffic simulation [42] is utilized in the area of transportation research for performing analysis of traffic situations and forecasts of traffic congestions. The model chosen in this thesis [69] is an example of a simulation with a high-dimensional state space, where classical time-parallel simulation is no suitable parallelization approach.

1.3. Thesis Overview

The thesis is arranged in four parts. The rest of the introductory Part I consists of a presentation of the preliminaries for this thesis in Chapter 2 and a discussion of the context of this thesis in Chapter 3. The main parts of the thesis are Part II, where the concepts of approximate state matching and progressive time-parallel simulation are introduced, and Part III, which presents the three case studies. Part IV concludes this thesis.

Chapter 2 comprises a short introduction to discrete-event simulation and a discussion of the most prominent parallel simulation approaches: conservative and optimistic synchronization and time-parallel simulation. Although most of this thesis is not concerned with conservative and optimistic parallel simulation techniques, a basic knowledge of these methods is necessary for the understanding of Chapter 3, which gives an overview of the research context of this thesis and the related work in the area of approximate parallel simulation techniques.

Chapter 4 presents a formal model of time-parallel simulation, which is the basis of the definitions of approximate and progressive time-parallel simulation. In Chapter 5, the technique of approximate state matching in time-parallel simulation is introduced.

The case studies of Part III are presented in Chapters 7, 8, and 9, consisting of the exemplary applications of approximate state matching and progressive time-parallel simulation to queuing systems, cache simulation, and road traffic simulation.

The thesis is concluded with a summary and review of the work, as well as an indication of future work in Part IV.

2. Preliminaries

In the field of modeling and simulation, two kinds of simulation models exist. A real-time simulation model, with a high degree of user interaction, is designed to create a realistic or entertaining representation of reality. Typical applications include training simulation exercises or virtual reality computer games. As humans are actively involved with such a system, the simulation must be executed in real time. In contrast, an as-fast-as-possible simulation is utilized for an analysis of the modeled system. In the latter case, there is usually no human interaction, hence real-time support is not required. As the name of this type of simulation implies, as-fast-as-possible simulations are executed as fast as is possible to achieve, i.e. there are no run-time constraints. Nevertheless, for economical or usability reasons, execution times of simulation models should be as short as achievable. In order to decrease the runtime of such a simulation, parallel simulation methods can be applied. As this thesis is concerned with parallelization of as-fast-as-possible simulation models, real-time simulation is not considered any further. Note however, that the concepts developed in this thesis could also be applied to the area of real-time simulation.

Moreover, as-fast-as-possible simulation models can be classified by the technique that is utilized for the description of model dynamics. An important technique for model specification is in the form of differential equations. Typically, equations describe a continuous change of the system over simulation time. This is used most often by physicists and engineers to model phenomena of physical systems. Another approach is *discrete-event simulation*, where the state of the model changes on the occurrence of an event, i.e. at discrete points in time. The latter has been used in many kinds of models, e.g. for performance evaluation of computer and communication systems, manufacturing models, or road traffic simulation. Continuous simulation with the use of differential equations is a separate field of research closely related with numerical methods for the solution of differential equations. In this thesis, only discrete-event simulation systems are considered.

2.1. Discrete-Event Simulation

This section gives only a short overview of discrete-event simulation. For more information, the reader is referred to one of the various textbooks on this topic [6, 26, 53].

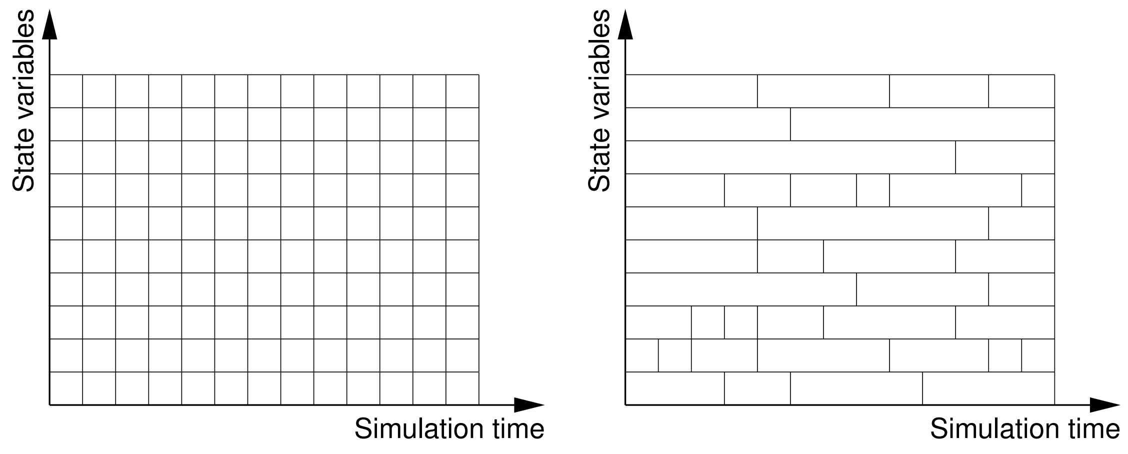
Like every model, a discrete-event simulation model is required to represent the development of the state of the real system over time. As such, there are three central aspects of a discrete-event simulation:

- At any time, the simulation is in one of a number of possible *states*. The set of all valid states of a model is called the model *state space*.
- A simulation model is of a dynamic nature. Therefore, it has to represent the changing of state over time.
- The passing of time has to be recorded in the simulation.

As mentioned in Section 1.1, different notions of time exist. Wallclock time denotes the time that passes during the execution of the simulation program. The simulation time is an abstraction used in the simulation to represent the time in the real system. In this thesis, to emphasize on the difference of wallclock time and simulation time, different symbols for variables of the corresponding types are used. An instant in wallclock time is denoted by a small case latin letter t, while an instant in simulation time is denoted by the small greek letter τ (both possibly with sub- and superscripts).

The state of a simulation is composed of a number of atomic state variables, typically real-valued. As explained above, in a discrete event simulation, the state of the system changes only on the occurrence of an event. Therefore, it is possible to use a rule-based specification of model dynamics: On the occurrence of an event, the next state of the simulation is determined by the type of event that occurred and the previous state of the simulation. Every event causes an action that consists of two parts: the modification of the simulation state and the scheduling of new events. Additionally, the process *advances* to the time of the event being processed, i.e. the simulation time of the process is set the event time. An event can be scheduled for an occurrence at any time after the current simulation

Figure 2.1. Time-advancement schemes in discrete-event simulation [30]



(a) Time-stepped execution (b) Event-driven execution

time. Therefore, at any time of the simulation execution, there are one or more scheduled events, not yet processed (i.e., scheduled for a future time). These are stored in a data structure called the *future event list*.

Two different approaches for advancing the simulation time in a discrete-event simulation execution exist. In a *time-stepped execution*, the simulation is advanced a number of equal-sized time steps. At every time step, a new state is calculated based on the current value of the state variables. However, it is not necessary, that there are state changes during a time step at all. If this occurs frequently during a simulation execution, processing cycles are wasted with unnecessary state calculations. If the times where state changes can occur are known in advance, the unnecessary calculations can be eliminated by skipping those time steps where no state change occurs. This is done in an *event-driven execution*, where time is advanced to the next event in the future event list. Figure 2.1 illustrates time-stepped and event-driven execution schemes. Vertical bars are used to represent recalculations of state variables.

2.2. Parallel Discrete-Event Simulation

To decrease run times of discrete-event simulation systems, the simulation task can be decomposed into multiple subtasks to be simulated concurrently on parallel processing nodes. If, for statistical reasons, multiple replications of an experiment with a simulation model are to be performed, these can be executed in parallel [33]. However, depending on the parameters of the simulation experiment, a direct parallelization of the simulation model can be more efficient [39], especially if there are runtime constraints for the simulation execution.

A common approach to the parallelization of a simulation model is by state decomposition, i.e. a grouping of state variables into subsets of the overall set. Each of these identified subsets is assigned to a processing node for parallel simulation. Most often, the parallel simulation processes are mutually dependent. In the case of discrete-event simulation, this is expressed by an exchange of messages between parallel processes, indicating the scheduling of one or more events for the receiving process. The efficiency of the parallel simulation decreases with an increasing message volume. Therefore, the amount of inter-process communication should be minimized by a proper state decomposition. In many cases, this can be achieved by taking the inherent parallelism of the real system to be modeled into account. In the real system, *physical processes* can be identified and these are transformed into *logical processes* in the parallel simulation model [30].

In the classical parallel discrete-event simulation theory, a central requirement is the *correctness* of the parallel simulation in comparison to a corresponding sequential simulation. The parallel simulation is said to be correct, if it is guaranteed to produce the same results as the sequential simulation. This is achieved by ensuring identical sequences of events in the parallel and sequential models. The following *local causality constraint* guarantees correctness of the parallel simulation.

Definition 2.2.1 (local causality constraint [27])**.** A discrete-event simulation, consisting of logical processes (LPs) that interact exclusively by exchanging time stamped messages obeys the local causality constraint if and only if each LP processes events in nondecreasing time stamp order.

Therefore, an important property of parallel simulation of discrete-event models is the introduction of a local simulation time for each logical process. In order to maximize the degree of parallelism of the simulation system, no strict synchronization of the local clocks of the processes is required. Completely independent time advancement of logical processes is desirable, but in general not achievable due to the local causality constraint. Two different types of approaches for the synchronization of logical processes exist: *Conservative synchronization* avoids the occurrence of causality errors by determining if the processing of an event is *safe*, i.e. it is not possible that an event with a smaller time stamp appears at a later time. *Optimistic synchronization*, in contrast, always allows the processing of an event, but if a message arrives later that violates the local causality constraint, that causality error is recovered by resetting the simulation to a state before the occurrence of the error.

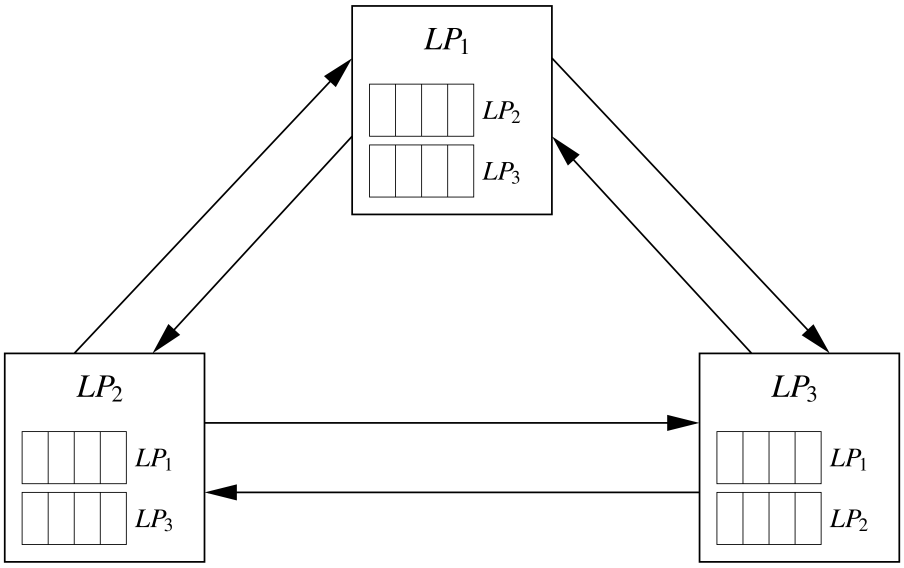
2.2.1. Conservative Synchronization

The goal of conservative synchronization algorithms is the prevention of causality errors by determining safe events. In the basic approach, this is achieved with a set of event input queues at each logical process. Each of these sets consists of an input queue for every other logical process sending messages to the local process. This is depicted in Figure 2.2, where every one of three LPs communicates with the other two. To advance the local time of a logical process, the event with the smallest time stamp is chosen from all of the event queues. The event is processed, possibly causing a new event to be sent to another process. As it is required that time stamps of new events are strictly greater than the local time of the issuing process, compliance to the local causality constraint is guaranteed for every logical process.

Time advancement is only possible if every one of the event queues of a logical process contains at least one event. If any queue is empty, the event with the smallest time stamp cannot be determined, as an event with an arbitrary time stamp might appear later at the empty queue. In this case, no safe event can be determined and the process blocks until an event arrives at the empty queue. This introduces the possibility of a *deadlock*, which occurs if all processes are blocked, waiting for an event to arrive from another process.

There are two ways to cope with deadlocks: *deadlock avoidance* and *deadlock detection and recovery*. For the latter, a central controller process is introduced, which is responsible for the deadlock detection. A moderately complex detection algorithm is presented in [22]. If a deadlock has

Figure 2.2. Structure of a conservative parallel simulation

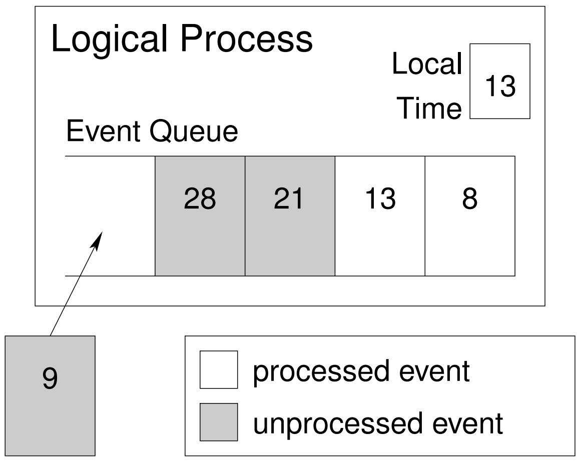


been detected, it can be broken by determining the event with the smallest time stamp of all events in the simulation, which is always safe to be processed. This can be done easily by collecting the events with the smallest time stamps from all logical processes. The parallel simulation approach with deadlock detection and recovery can be found in [16].

Another approach, which is of more relevance in parallel discrete-event simulation is the *null-message* approach due to Chandy and Misra [15] and Bryant [12]. If a process is blocked, instead of waiting until the missing event arrives with a message, a null message with a lower bound on the time stamp of future messages of that process is sent. The *lower bound on the time stamp (LBTS)* is equivalent to the local time of the logical process plus the *lookahead*, i.e. a minimum amount of time that a message is scheduled in the future. The amount of lookahead for each process is model specific, e.g. it might be derivable from physical limitations of the real system. A null message can be interpreted as a normal event with the exception that it does not lead to state changes or scheduling of new events. As soon as a null message arrives at an empty event queue and there is no other empty queue at the same logical process, it can be processed, causing the simulation time of the process to advance.

The Chandy/Misra/Bryant approach is a distributed algorithm that is easy

Figure 2.3. Straggler message arriving at a local process



to implement. However, although it ensures that no deadlocks can occur, the performance of the algorithm highly depends on the amount of lookahead in the model. With poor lookahead, it might take a long time to break an existing deadlock, as only small time advances can occur. Even if there is a modest amount of lookahead in the model, the lookahead is not always easy to identify.

2.2.2. Optimistic Synchronization

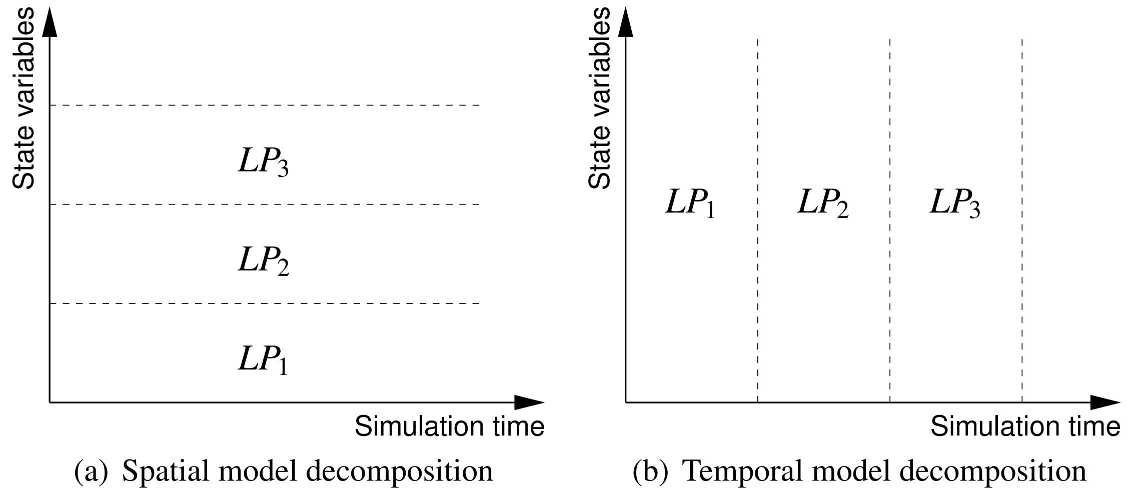
Conservative simulation approaches guarantee the satisfaction of the local causality constraint by processing only safe events. Optimistic methods, in contrast, allow the processing of any event, regardless of the value of its time stamp. Therefore, *straggler messages* may occur, i.e. messages with a time stamp t_s smaller than the local time t_l of the process (in Figure 2.3, this is illustrated with $t_s = 9$ and $t_l = 13$). As this violates the local causality constraint, a *rollback* to t_s is required. A rollback is intended to reset the state of the logical process to the correct value at time t_s. Each of the messages sent by the process after time t_s might be invalid, as the process could enter a completely different simulation state after processing the straggler message. Therefore, all of these messages have to be retracted. The most well-known optimistic algorithm is Jefferson's Time Warp method [45], which provides details about the mechanisms of rollback, state saving, message retraction, and global virtual time.

To be able to reset the state of the simulation system in case of a rollback, state saving has to be performed. There exist two widely used techniques to accomplish state saving in Time Warp. With *copy state saving*, the whole state of a logical process is saved and stored for later rollbacks. This has the advantage that rollbacks are easy and efficient to perform, but the cost in terms of memory consumption and processing time for state savings is high. In *incremental state saving*, a log is kept, recording changes for each event being processed. In the case of a rollback, the log entries of events being rolled back are scanned in order of decreasing time stamps, undoing the state changes associated with each event. In this case, the cost of state saving is decreased, but the cost of rollbacks is increased. Which of the two methods to choose depends on the frequency of rollbacks and the size of states to be saved, including the size of events themselves. If rollbacks are likely to occur, copy state saving might be more efficient. Otherwise, incremental state saving is the more viable alternative.

Message retraction is needed to ensure the consistency of the parallel simulation system. In the case of a rollback, all messages caused by events occurring after the rollback time are to be retracted. This is a complex operation, as the messages themselves might have caused other messages to be sent, which have to be retracted as well. In Time Warp, the concept of *anti-messages* is used for the implementation of message retraction. For every message to be retracted, a corresponding anti-message is sent to the original receiver. If the original message has not yet been processed by the receiver (i.e. it is waiting in the receiver's event queue), anti-message and original message annihilate each other. If the original message already has been processed, a rollback to the time of the original event is performed, which is equivalent to a rollback due to a straggler message. In the case of unreliable communication links, it might be the case that an anti-message arrives at the receiver prior to the original message, e.g. if the original message was delayed during the transmission. In this case, the anti-message is inserted into the event queue and annihilates the original message as soon as the latter one arrives. In contrast to normal messages, however, anti-messages are never processed.

To be able to send anti-messages in case of a rollback, a record of messages with receiver and time stamp has to be stored for every message that is sent by a logical process. Together with the need for frequent state savings, this incurs a high memory overhead of the parallel algorithm. Another

Figure 2.4. Decomposition of simulation models [30]



(a) Spatial model decomposition (b) Temporal model decomposition

problem of the Time Warp mechanism as presented above, is the inability to produce reliable output during the simulation execution, as every event that is processed might be retracted later. To solve these issues, the concept of *global virtual time* is used.

Definition 2.2.2 (Global virtual time [30])**.** Global virtual time at wallclock time T (GVT_T) during the execution of a Time Warp Simulation is defined as the minimum time stamp among all unprocessed and partially processed messages and anti-messages in the system at wallclock time T.

As GVT_T is the lower bound of all the time stamps of messages in the system, it is not possible that a straggler message or anti-message with a time stamp smaller than GVT_T arrives at wallclock time T. Rollback to a time before GVT_T will not be necessary in any of the logical processes. Therefore, state savings and event records can be discarded for all simulation times smaller than GVT_T. Furthermore, simulation output can be generated for the same time period.

Asynchronous computation of the global virtual time is a non-trivial task, especially due to the problem of transient messages, which might be in the system at the time of GVT computation. Examples for efficient algorithms for GVT computation can be found in [62, 92].

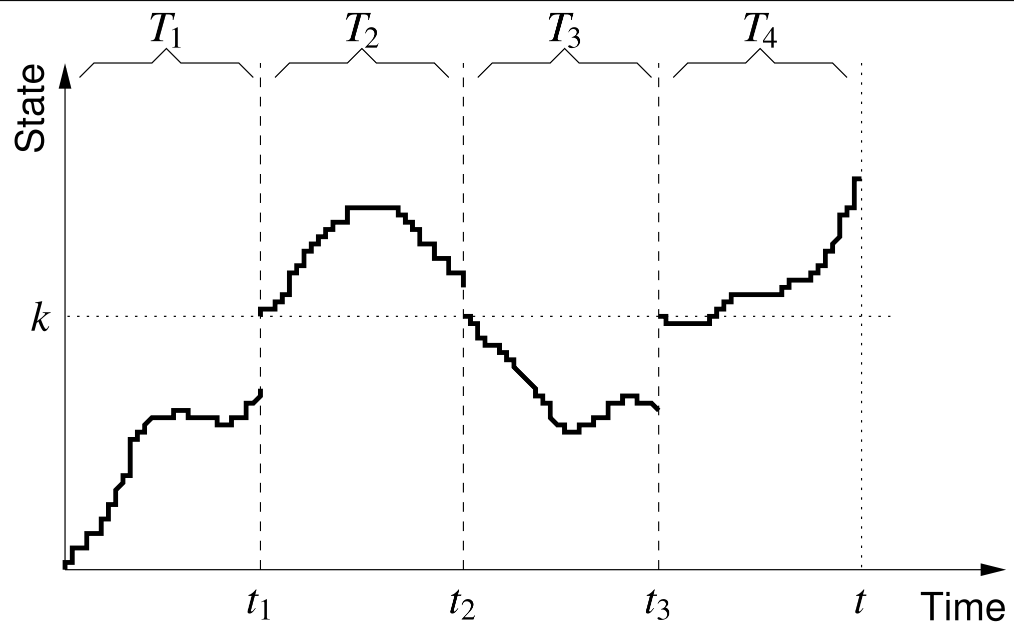
2.3. Time-Parallel Simulation

In the parallel-discrete event simulation approaches discussed above, the set of state variables of a simulation model is decomposed into subsets. Each of these is assigned to a logical process managing the corresponding part of the global state. These logical processes are then executed concurrently on parallel processing nodes. An illustration of this approach is depicted in Figure 2.4(a). Drawbacks are the introduction of an overhead for the synchronization between logical processes and the possibly limited amount of achievable parallelism, which is restricted by the number of state variables and the decomposability of states in the model. Time-parallel simulation takes a different approach by decomposing the time axis and performing simulations of resulting time intervals in parallel (see Figure 2.4(b)). Afterwards, the results of all intervals are combined to create the overall simulation result. This has the potential for massive parallelism, as the maximum number of logical processes is determined by the number of possible time intervals, which is only restricted by the granularity of the time representation in the simulation implementation. The method is first attributed to Chandy and Sherman [14], which propose a combined spatial/temporal decomposition of the simulation task, i.e. a combination of classical parallel simulation techniques with time-parallel simulation.

Without further mechanisms, the final and initial states of adjacent time intervals do not necessarily coincide at interval boundaries, possibly resulting in incorrect state changes (this situation is depicted in Figure 2.5). Several solutions of this *state-matching problem* [30] have been proposed. Lin and Lazowska [58] introduce the notion of *regeneration points*, which are states that keep reoccurring throughout a simulation execution. If such a state can be identified a priori, a number of simulations can be executed concurrently, starting from the regeneration point and continuing until the regeneration point is reached again. Afterwards, the traces of the parallel simulations are concatenated to a correct trace of the simulation over the whole time period. The drawback of this approach is the difficulty to identify regeneration points, especially for models with complex states. Nevertheless, this approach was used successfully for the simulation of ATM multiplexers [3, 4] and cascaded statistical multiplexers [76].

Another solution is to use *fix-up computations* [40], where prior to the simulation, the states at interval boundaries are guessed and a first simula-

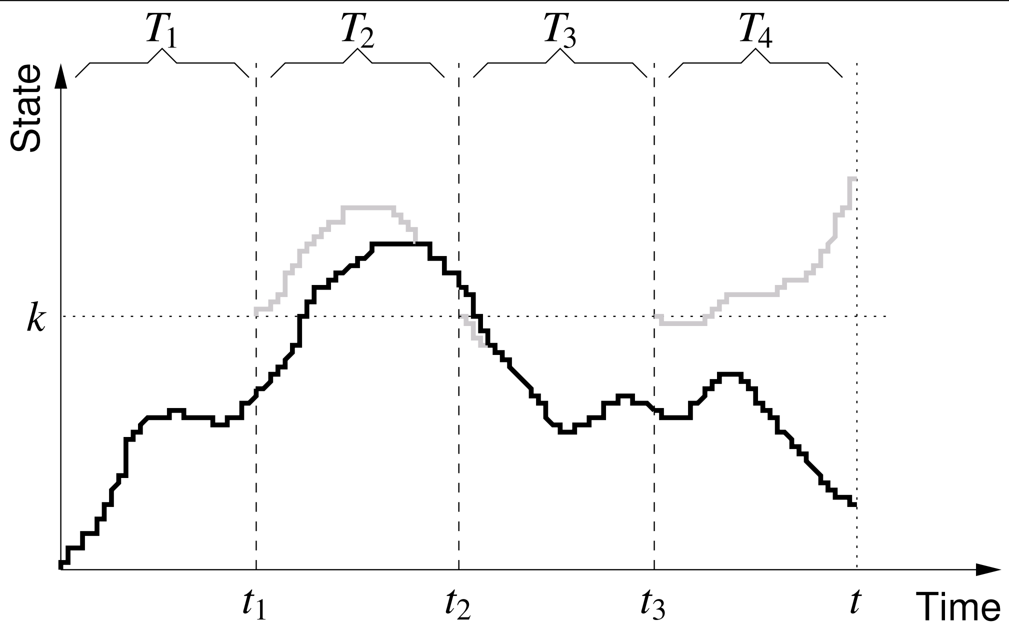
Figure 2.5. Incorrect state changes in a time-parallel simulation



tion phase is performed starting with the guessed states as initial states of the concurrent simulation executions. Without a perfect guessing scheme, this frequently results in incorrect state changes at interval boundaries. Fix-up computations are re-executions of the simulations of those time intervals that started from incorrect initial states (see Figure 2.6 for an illustration). Depending on the knowledge gained in a simulation phase, only a partial re-simulation might be necessary, but in the worst case, a fix-up computation consists of the re-simulation of a complete time interval. Without a good guessing strategy or efficient fix-up computations, this approach leads to a significant amount of computational overhead. The method of temporal parallelization with fix-up computations has been applied successfully to the simulation of caching in computer systems [40, 75] and the simulation of Ethernet [107], among others.

A completely different approach to time-parallel simulation utilizes *parallel prefix computations* [52] to solve recurrence equations describing the dynamics of a simulation model. This method is best understood in the context of an example. Consider a single-server queuing system with an arbitrary statistical distribution of job interarrival and service times. Inter-arrival and service times are presampled and stored in an input trace. Now,

Figure 2.6. Performing fix-up computations



the job arrival instants can be defined as a recursive function of interarrival times. E.g., if job $i-1$ arrived at time A_{i-1} and r_i is the interarrival time between jobs $i-1$ and i, the arrival time of job i can be calculated by $A_i = A_{i-1} + r_i$. By repeatedly solving the recurrence, we arrive at the formula $A_i = r_1 + r_2 + \ldots + r_i$. This can be solved efficiently in parallel by utilizing parallel prefix computations [34, 50]. E.g. the sum $r_1 + r_2$ can be computed on one processing node, while $r_3 + r_4$ is computed on another node, combining the results after both computations have been finished (as can be seen, parallel prefix can only be used on computations with associative operators). In the queuing example above, the departure times of jobs can be described as recurrence equations as well, and solved by parallel prefix computations. The method of parallel simulation of G/G/1 queues has been described by Greenberg et al. [35, 36], also discussing possibilities to apply this method to the simulation of more complex topologies of queuing networks. This technique fits into the category of time-parallel simulation, as the statistics of jobs arriving at different simulation time instants are computed concurrently.

3. Thesis Context

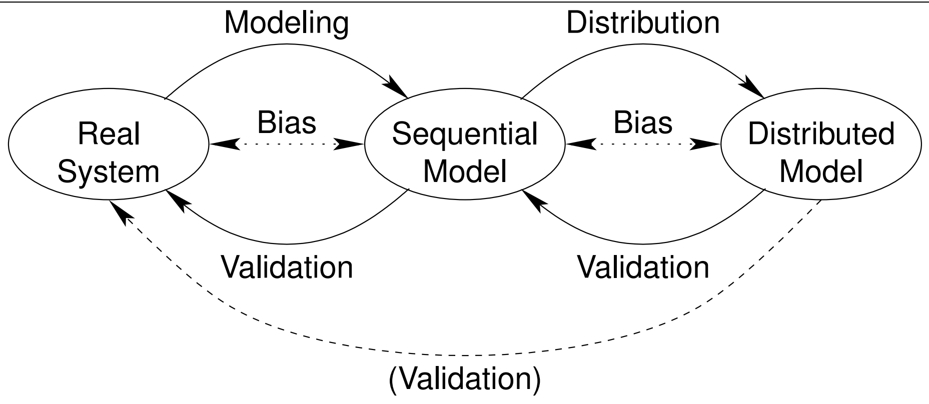
This thesis is concerned with the presentation of new approximate methods to increase the performance of parallel as-fast-as-possible simulation models. These methods introduce an error in the simulation results, i.e., the results of the approximate parallel simulations deviate from those of a corresponding sequential simulation. In the field of parallel and distributed simulation, there are a variety of other techniques directly or indirectly comparable to approximate state matching. In the first part of this chapter (Section 3.1), a broader survey of related techniques is given, in order to classify the approach introduced in this thesis in the context of parallel and distributed simulation. In the second part (Section 3.2), detailed discussions of the more directly related work are presented.

3.1. Classification of the Approach

Traditional synchronization methods for parallel simulation of discrete-event models were devised to ensure correctness of the parallel simulation, i.e., to produce results identical to those of a corresponding sequential simulation. Proofs of this property have been provided for conservative [65] and optimistic [55] synchronization, as well as time-parallel simulation methods [40, 58]. However, the utilized parallelization methods often restrict the degree of parallelism of simulation systems [88, 100]. Therefore, variants of these methods have been developed, where the correctness property does not hold [61, 87, 101] (cf. the related discussion in Section 1.1).

Figure 3.1 depicts an abstract view on the development process of a parallel or distributed simulation. Starting from a problem in the real world, a model of the real system is created. It is supposed here, that the initial model is of a sequential nature, i.e. no special precautions for the parallelization of the model have been taken. The abstraction involved in the modeling step leads to a discrepancy between the behavior of the real sys-

Figure 3.1. Distributed Simulation Development



tem and the behavior specified in the model. This discrepancy can be interpreted as a *bias* of the model with respect to the real system (the term bias is used in a non-statistical sense here). The model is considered credible if the bias is negligible according to the problem definition. To ensure this credibility, validation of the model against the real system is performed according to pre-defined criteria.

To create a parallel or distributed simulation, the sequential model is enhanced with details about the parallelization. As discussed above, this introduces another level of bias, which is caused by the utilized parallelization method. For a study of the bias introduced in the process of creating a parallel or distributed model, different aspects of interest, as well as different reference systems, may be considered. For example, the following aspects in a (distributed) simulation may be of interest:

- the trajectory of the state vector in the state space,
- an aggregated performance measure (e.g. the average utilization of a server in a queuing network model),
- events occurring in an event-driven simulation,
- the exact ordering of events in a discrete-event simulation, and
- the state of the simulation system at an arbitrary point in wall clock time during the simulation run.

A reference system is a basis against which to determine the bias in a simulation model. Typical reference systems are:

- the real/physical system (*validation of simulation models*),

- the formal and/or conceptual model the executable model is based on (*verification of simulation models*),
- another simulation run with the same simulator (*repeatability of simulation experiments*),
- a sequential version of a parallel or distributed simulation system (*synchronization and coordination in parallel and distributed simulation*), and
- an idealized parallel simulation system, e.g. parallel simulation using a reliable communication system without latencies.

There are various sources for bias in the above sense. In the sequel, reasons for uncertainties and incorrectness are classified in three categories:

1) bias that is caused independently from parallelization or distribution of the simulation model,
2) bias in parallel and distributed simulation that is caused by the technical framework applied in parallel and distributed simulation, and
3) bias that is deliberately accepted by the modeler (simulation developer, resp.) in a controlled fashion in order to increase the performance or fault tolerance of a parallel or distributed simulation run.

3.1.1. Bias Independent of Parallelization

Even in sequential simulation on a single processor machine, aspects of vagueness, inexactness, and uncertainty may occur. In order to obtain a clearer understanding of the effects of parallel and distributed simulation, aspects that occur independently of parallel or distributed simulation are discussed briefly.

Abstraction and Simplification When constructing a simulation model, the required abstractions and simplifications always cause a divergence of the model behavior from the behavior of the real system. Controlling this type of inexactness in a systematic way is dealt with by model validation [5, 11]. The degree of inexactness tolerated here is implicitly or explicitly defined via the primary objective of the simulation model. Incorrect behavior and associated inexactness of a simulation model as opposed to the corresponding formal or conceptual model may also be caused by incorrect translation and implementation. Such aspects should be dealt with by methods of model verification [5].

Stochastic Simulation Using random variables is a popular instrument to represent details of the real system that are not available to the modeler or that are too fine grained for the intended purpose of the simulation model. Consequently, in stochastic simulation, uncertainty and vagueness occur at least twofold: Firstly, repeated simulation runs differ in their behavior due to the introduced randomness. Secondly, the behavior of the model clearly differs from the behavior of the real system, since in the real system many aspects modeled via random variables are in fact deterministic. Since random behavior in simulation models is actually realized via pseudo random sequences (which causes an additional difference between the formal and the executable model), uncertainty in the first sense may be eliminated by controlling the seeds of the pseudo random number generators. This way, also (pseudo) stochastic simulation experiments can be made repeatable. Uncertainties in the second sense are an integral property of stochastic simulation. They can be dealt with by appropriate statistical methods (such as confidence intervals or the computation of higher moments of simulation results) if the intended purpose of the simulation model is system analysis. If the intended purpose is training and education, the trainee and trainer should be aware of random aspects of the model.

Discretization and Rounding Errors Usually, for the representation of continuous aspects of system behavior, sets of differential equations are used. Since such mathematical representations often do not allow for a closed form solution, numerical methods are applied that typically rely on the discretization of continuous model aspects. This usually implies a difference between numerically computed model solutions and the theoretical exact solutions of the formal model. However, many methods of computational mathematics provide an estimation or bound for the error introduced by discretization. Moreover, discretization errors may be controlled by using techniques of adaptive step-length control. Also, rounding errors introduced by the application of numerical methods may be controlled by using appropriate mathematical tools (e.g. interval arithmetic [66, 71]).

3.1.2. Bias due to Technical Conditions in Parallel Simulation

Parallel and distributed simulation is based on the principle of simulating partitions of a simulation model on a number of processing nodes. Depending on the type of synchronization method, the most relevant aspects of the technical framework for parallel and distributed simulation with respect to bias and uncertainty are communication latencies, differing hardware clocks and the necessity to order simultaneous events.

Communication Latencies Communication between processors or computing nodes is an integral part of parallel and distributed simulation. Depending on the time management of the distributed simulation model, communication latencies can be a source of additional uncertainties and inexactness. In real-time simulation (e.g. Distributed Interactive Simulation [21, 77]), communication latencies cause a delayed arrival and thus a delayed consideration of messages. This yields inexact behavior in at least two respects: Firstly, such delays cause a difference of the simulation behavior from the formal model and a corresponding sequential implementation. This can lead to time anomalies and causality violations. Secondly, communication latencies contribute to differences between repeated simulation runs, as many communication systems involve non-deterministic behavior (e.g., Ethernet-based local area networks or the Internet).

Synchronization of Hardware Clocks Even without the consideration of communication latencies, distributed real-time simulation assumes synchronized hardware clocks in order to obtain correct behavior. However, exact synchronization of hardware clocks cannot be guaranteed. This may contribute to various uncertainty aspects: a divergence from the formal model and from a sequential implementation as well as non-repeatability of simulation runs.

The mechanisms for time-coordinated as-fast-as-possible simulations, presented in Section 2.2, guarantee that parallel simulation produces the same event sequences as sequential simulation. Furthermore, repeatability is guaranteed despite the presence of non-deterministic communication latencies. Thus, those two types of uncertainties do not appear in time-coordinated simulation, but were presented here for reasons of completeness.

Simultaneous Events To obtain repeatable simulation runs, special care has to be taken on simultaneous events (i.e., events with identical simulation time stamps) even in sequential simulation. Repeatable ordering of simultaneous events is an even more critical challenge in parallel and distributed simulation. If no measures are taken in order to guarantee a repeatable ordering of simultaneous events, uncontrollable factors such as different event rates or different communication latencies in repeated simulation runs can cause a different ordering of simultaneous events. Various approaches can be used in order to enable repeatability of distributed simulations also in the presence of simultaneous events. These techniques work by ensuring uniqueness of otherwise identical time stamps. Among other approaches, this can be achieved by assigning additional information with each time stamp that allows the consideration of causal interdependencies. Similar results can be obtained by associating different priorities to events with equal time stamps. In [30], a combination of using an age field, priority, node-ID and a sequence number is suggested. An overview of different event ordering schemes can also be found in [90].

3.1.3. Bias in Order to Achieve Higher Efficiency

A third category of bias in parallel and distributed simulation considers the deliberate acceptance of bias in order to increase the efficiency or the reliability of a simulation model. Here, bias is used as a means to improve the quality of a simulation in terms of runtime or reliability. However, this is done at the cost of a potential error in the simulation results. The method of approximate time-parallel simulation introduced in this thesis fits into this category, therefore this method is discussed intensively in the next section.

3.2. Directly Related Work

There is a moderate amount of existing work deliberately accepting bias, mainly to increase the efficiency of conservative or optimistic parallel simulation approaches. As these constitute the work directly related to this thesis, a separate section is dedicated to their discussion. Section 3.2.4 summarizes the most closely related work, i.e. the application of approximate methods in time-parallel simulation.

3.2.1. Temporal Uncertainty

With many models, sufficiently high lookahead values are hard to determine, preventing high efficiency in conservative parallel simulation. In previous work, future lists have been proposed to increase lookahead capabilities [72].

In [61], within the context of a hybrid optimistic/conservative synchronization scheme, the conservative blocking behavior is relaxed by introducing a tolerance ε. Consider a synchronization point at simulation time t_0. Then, the *tolerant synchronization* approach allows simulation until time $t_0 + \varepsilon$. Events with a time stamp $t_0 + x < t_0 + \varepsilon$ are scheduled at simulation time t_0. This introduces a controllable error ε in the exact time stamps of simulated events that allows to increase the efficiency of the simulation execution significantly. The authors argue that the introduced error corresponds to factors that are unknown in the real system. Furthermore, it is reported that the effect of the time stamp errors on the error of the overall simulation result is usually smaller than the confidence interval of the result.

Fujimoto addresses a similar yet different solution to the low/zero-lookahead-problem: In [29], a partial order called *approximate time order* is introduced: in approximate time order, intervals are used to capture temporal uncertainty of the simulated real system. It is argued that temporal uncertainty can be found in basically every model because simulation is always only an approximation of the real world. Given two time intervals, an is-before relation holds if the intervals do not overlap. For overlapping intervals no ordering relationship exists between the two events. This relaxed order for overlapping time intervals is exploited in order to increase efficiency in conservative simulation. However, causal interdependence between events may be lost with that approach. Thus, another partial order is defined that considers causality as well: *approximate time causal order*. In experiments based on the PHOLD algorithm [28], it is shown that significant speedup can be achieved by using time intervals and approximate time or approximate time causal order with moderate effects on the accuracy of the simulation results.

A disadvantage of the approach proposed in [29] is the fact that in order to use the proposed algorithms for the Runtime Infrastructure (RTI) in the High Level Architecture (HLA) [79, 78, 80], message delivery mechanisms and the HLA interface specification would have to be modified [60]. To achieve compliance with the HLA specification, Loper and Fujimoto propose to use a combination of time intervals and pre-sampling of a precise time stamp within time intervals according to some specified probability distribution [60]. The effect of this algorithm is an increase of lookahead values that can directly be used in HLA-based simulation.

An application of the concept of temporal uncertainty in parallel discrete-event simulation to optimistic synchronization via Time Warp is reported by Beraldi and Nigro [8, 9]. Similar to temporal uncertainty in conservative simulation as presented in [29], Beraldi and Nigro introduce a formal event model for optimistic synchronization, where events are assigned time intervals, rather than time instants. Experiments with this approach indicate a considerable speedup without compromising the accuracy of the simulation.

3.2.2. Spatial Uncertainty

In analogy to temporal uncertainty, Quaglia [86] argues that in many real systems also spatial uncertainty exists. In parallel and distributed simulation, usually a uniquely specified logical process is responsible for the simulation of a given event. Quaglia proposes to exploit spatial uncertainty in the model in order to relax this mapping by associating a set of logical processes to every event. The event can be passed to any of the logical processes in this set. This freedom is subsequently exploited as follows: In optimistic simulation, an event that would cause a rollback at the receiver logical process may be passed on to other logical processes in the set associated to the event. This way, the number of rollbacks can be reduced and the efficiency of the optimistic simulation run is increased. Similarly, spatial uncertainty can be used to increase lookahead in conservative simulation. In [86], this technique is applied to the simulation of mobile communication systems and a moderate bias of the simulation results is reported.

3.2.3. Relaxed Ordering Mechanisms

Most synchronization schemes for time coordinated distributed simulation rely on using time stamps for events and apply a time stamp ordered event simulation and/or message delivery. However, strict time stamp ordering often decreases the efficiency of distributed simulation systems, sometimes to a level that real-time requirements are no longer met. On the other hand, pure time-stamp ordering is sometimes overpessimistic, because not every message that is delivered out of order necessarily produces a causality violation. There exists a couple of approaches to overcome this problem by relaxation of the strict time stamp ordering.

The most radical concept is that of *unsynchronized parallel discrete-event simulation* [87, 97]. In that work, the effect of dispensing with synchronization in time-warp-based simulation of queueing models is studied. For this special case, significant efficiency improvements were reported in terms of both processor time and memory usage, with surprisingly little effect in the overall simulation results. However, the general applicability of such a radical approach seems questionable.

A concept originally developed in the context of multimedia systems with real-time requirements is the notion of *delta causality* [1, 108]. In delta causality, an expiration time is associated with every time-stamped message. Messages that are received before the expiration time is elapsed are delivered and processed in time-stamp order. Messages that are delivered after expiration, may be processed out of order, eventually causing causality violations.

Time management in the high level architecture (HLA) is based on two options for message delivery: receive order and time-stamp order. In [54], the notion of causal order in the context of HLA time management is introduced. Causal order focuses on the most important aspect of time-stamp order, namely preservation of causality. However, the costly ordering of all time-stamp-order messages is reduced such that only messages with causal dependencies are ordered. One major challenge in causal ordering is to provide less costly meta information about causality constraints than event time stamps. In [110], the authors propose an enhanced version of causal ordering, namely *critical causal order*, being an additional relaxation of causal order. With the critical causal order mechanism, it is possible to preserve causality even with a relatively large number of federates, while still preserving important real-time properties of the simulation system.

Another alternative event ordering mechanism is presented by Quaglia and Baldoni [85]. The authors introduce the notion of *weak causality* that models the intra-object parallelism in parallel discrete-event simulation systems. A correct simulation run is defined as a run where events are executed at each object according to their time stamp. The weak causality relation is applied to optimistic synchronization by the way of a synchronization protocol that reduces the number and extent of rollbacks.

3.2.4. Approximation in Time-Parallel Simulation

The concept of approximate time-parallel simulation introduced in this thesis considers an approximate solution to the state-matching problem (see Section 2.3). Instead of an exact match of the states of two different processes at the time interval boundaries, a deviation of states is tolerated according to a pre-defined threshold. This introduces a bias in the parallel simulation in the form of an error in the simulation results. Details about this method are presented in Part II.

There has been few work on the utilization of approximate methods in time-parallel simulation. The only existing approaches in this respect are specially designed for queuing system simulation. Efficient correct parallel simulation methods have been devised for various types of queuing systems and networks: certain types of networks of G/G/1 queues [35, 36], arbitrary networks of lossy markovian queues [3, 4], and certain types of G/G/1 queuing networks with loss or communication blocking [17]. In order to extend the efficient approach of Greenberg et al. [35, 36] to the simulation of G/G/1/K queues, Wang and Abrams [103] use an approximate algorithm. Another approach to approximate simulation of lossy G/G/1/K queues is presented by Lin [57].

In another work on the simulation of G/G/1/K and G/D/1/K queues, Wang and Abrams [101] introduce the notion of *partial state matching* in time-parallel simulation, which is similar to the approximate state matching approach introduced in Chapter 5. In most cases, a simulation state z is composed of several sub-states. Two states $z = (z_1, z_2, z_3)$ and $z' = (z'_1, z'_2, z'_3)$ match partially, if all sub-states in a predefined subset of the set of sub-states match, e.g. $z_1 = z'_1$. However, a deviation of the sub-states z_1 and z'_1 is not allowed. With a proper definition of the deviation measure (see Section 5.1), approximate state matching can be used to perform partial state matching as well.

The original design of partial state matching was in the context of queuing simulation, but the authors mention the possibility of applying the technique to other kinds of models [102], without providing detailed information. For reasons explained above, approximate state matching is a more general concept that allows a more fine-grained definition of error control. Therefore, it is supposed to be applicable to a wider range of models.

The directly related work, which has been discussed in this section, is summarized in Figure 3.2.

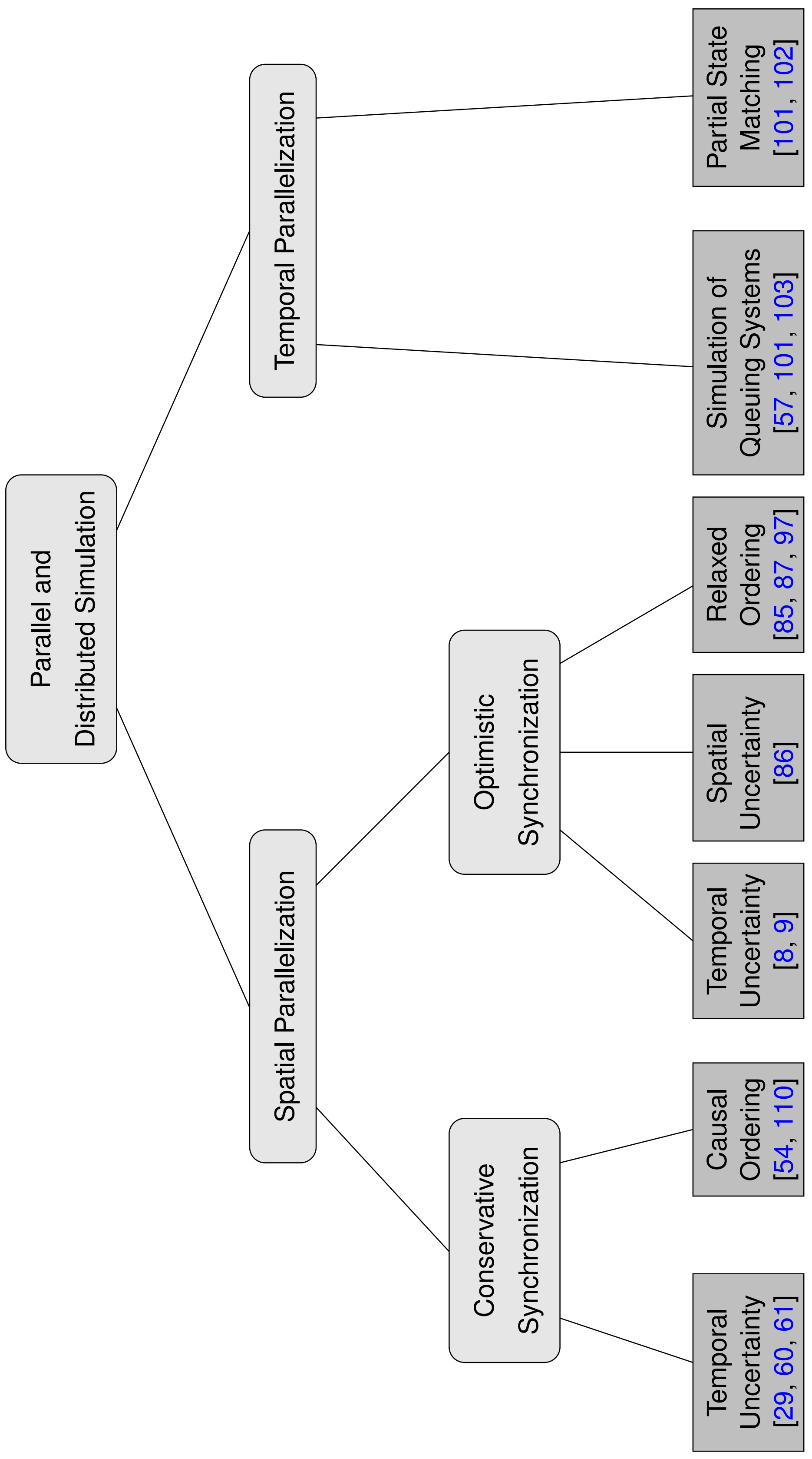

Parallel and Distributed Simulation
Spatial Parallelization
Temporal Parallelization
Conservative Synchronization
Optimistic Synchronization
Temporal Uncertainty [29, 60, 61]
Causal Ordering [54, 110]
Temporal Uncertainty [8, 9]
Spatial Uncertainty [86]
Relaxed Ordering [85, 87, 97]
Simulation of Queuing Systems [57, 101, 103]
Partial State Matching [101, 102]

Figure 3.2. Directly related work

Part II.

Approximate Time-Parallel Simulation

4. Formal Model of Time-Parallel Simulation

A discrete event simulation consists of a set of state variables that constitute the overall state of the simulation at any time during the simulation execution. The dynamic behavior is defined as state changes triggered by events occurring at specific points in simulated time.

To parallelize a simulation model, the task of computing state changes over time has to be decomposed into subtasks to be assigned to a number of logical processes for concurrent simulation. In classical parallel discrete event simulation, the set of state variables is decomposed (*spatial decomposition*) and each of the resulting subsets is assigned to a logical process, which is responsible for the calculation of states in the state space spanned by the assigned variables (see Section 2.2). E.g., a road network of a traffic simulation might be decomposed into several smaller networks simulated by logical processes concurrently.

In most cases, the subtasks created through spatial decomposition are not independent of one another. In order to achieve a causally correct behavior of the overall simulation, logical processes have to communicate. As the communication and synchronization overhead of mutually dependent processes might seriously decrease potential speedups, the efficiency of the parallelization highly depends on the level of independence of logical processes. Hence, the amount of achievable parallelism is largely determined by the decomposability of the state space.

Another possibility of discrete-event-simulation parallelization is the decomposition of the simulated time into intervals (*temporal decomposition*). For each of these intervals, the responsibility to compute the state changes at the corresponding times is assigned to a separate logical process (see Section 2.3). As a prerequisite to the introduction of approximate state matching in Chapter 5 and progressive time-parallel simulation in Chapter 6, a formal model of time-parallel simulation is presented here.

4.1. Basic Principles

To formalize the approach of *time-parallel simulation*, let $T = [0,\tau]$ be the interval of the whole simulation time. T is decomposed into the *time intervals*

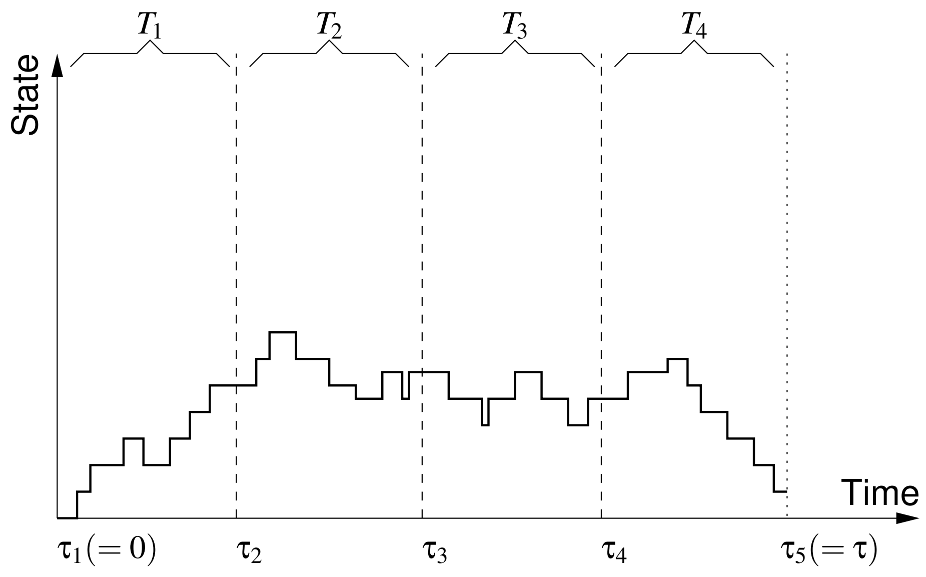
$$\begin{aligned} T_1 &= [\tau_1, \tau_2], \\ T_2 &= [\tau_2, \tau_3], \\ &\vdots \\ T_m &= [\tau_m, \tau_{m+1}], \end{aligned}$$

with $\tau_1 = 0$ and $\tau_2, \ldots, \tau_m \in (0,\tau)$ and $\tau_{m+1} = t$, such that every point in the simulated time is contained in some interval and the intervals overlap only at interval boundaries $\tau_2, \ldots, \tau_m$.

Example 4.1.1. Consider the model of a G/G/1 queue, where a stream of incoming jobs is served by a single service station. If the server is busy upon a job arrival, the job is put into a queue, where it has to wait for service. Times between arrivals of subsequent jobs have an arbitrary statistical distribution, as well as the service times. In the simplest case, the state of the system only consists of the current number of jobs in the system (i.e. the number of jobs in the queue plus the one job currently being served, if there is any). Further, the simulation is supposed to be deterministic, i.e. arrival and departure instants are predetermined and used as input of the simulation. A simulation execution of the sequential model consists of a calculation of the state trajectory over the simulated time. In this case, a spatial decomposition of the simulation model is not possible, as a one dimensional state cannot be decomposed further. However, if a very long runtime of the simulation is required, it might be reasonable to apply temporal decomposition here. Figure 4.1 shows the space-time diagram of a G/G/1 simulation, where the simulation time has been partitioned into four time intervals $T_1, \ldots, T_4$. Ideally, the state trajectory calculated by the parallel simulation should be identical to that of the sequential simulation.

Temporal decomposition of a simulation task has the advantage that it is easy to perform and that computations of state changes of different time intervals are independent of each other, with the exception of the states at intermediate interval boundaries $\tau_2, \ldots, \tau_m$. Unfortunately, the initial states

Figure 4.1. Parallel queueing simulation



at these times are not known prior to the simulation execution, although they are needed to start execution. This property of time-parallel simulation is known as the *state-matching problem* [30].

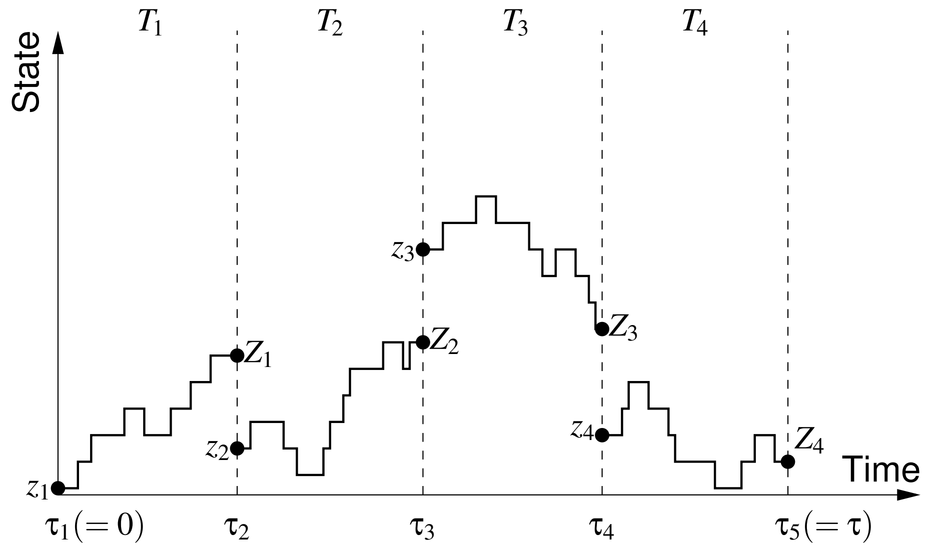
In order to apply temporal parallelization to a simulation model, the state-matching problem has to be solved efficiently. Several approaches for the solution of the problem have been proposed, as introduced in Section 2.3. In this thesis, time-parallel simulation is mainly considered in the context of fix-up computations.

4.2. Iterative Fix-up Computations

For a concurrent simulation of time intervals, the computation of every interval T_k has to start with some *initial state* z_k. At the beginning of the simulation, only the initial state z_1 of T_1 at time τ_1 is known. With fix-up computations, the initial state z_k at time τ_k of every interval T_k, with $k \in \{2,\dots m\}$, has to be guessed. After the simulation of an interval T_k has been completed, the *final state* Z_k of T_k is known.

Example 4.2.1. Continuing Example 4.1.1, to enable parallel simulation of time intervals, the number of jobs at interval boundaries $\tau_2,\dots,\tau_4$ have to be guessed. Without further information about the model, an arbitrary value

Figure 4.2. Parallel simulation with guessed initial states



must be used as initial state of an interval. Figure 4.2 shows the state trajectories, which might result from a parallel simulation with random initial values. In this case, the overall trajectory of states differs significantly from that of the sequential simulation, also exhibiting incorrect state changes at interval boundaries $\tau_2, \ldots, \tau_4$.

As illustrated in Example 4.2.1, there is no guarantee that the final and initial states of adjacent intervals match, i.e., it is possible that $Z_{k-1} \neq z_k$ at time τ_k. This situation can be interpreted as an incorrect state change, not accounted for by the model. Therefore, after a first simulation phase with guessed initial states, fix-up computations are performed to amend these incorrect state changes for all intervals T_k, where $Z_{k-1} \neq z_k$.

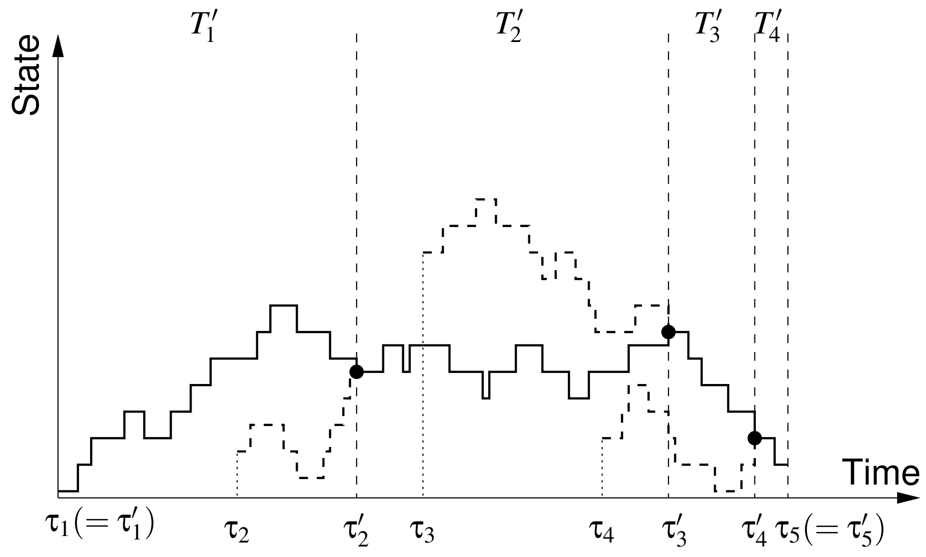
In the simplest case, a fix-up computation is just a re-execution of the simulation of a particular time interval T_k with a new initial state z'_k identical to the final state of the preceding interval Z_{k-1}. Here, even if fix-up computations have only got to be performed for a small number of intervals, there is a significant amount of computational overhead. It is desirable that only a part of the computations of a time interval have to be repeated. The parallel cache simulation approach presented in Chapter 8 is an example where only small fractions of time intervals have to be re-simulated during fix-up computation phases.

Example 4.2.2. With the parallel simulation trajectory shown in Figure 4.2, fix-up computations have to be performed to achieve correct simulation results identical to that of the corresponding sequential simulation. Using the simple fix-up computations presented above is possible, although not reasonable. A more efficient method might compute correct simulation results without performing complete re-simulations.

In general, a single fix-up computation phase might not be sufficient to achieve a completely correct simulation, as final states of intervals might change due to a fix-up computation, which would result again in incorrect state changes at interval boundaries. An additional fix-up phase is required, probably leading to yet another fix-up phase. In the worst case, the parallel simulation has a performance comparable to that of the corresponding sequential simulation, with a much higher work done. In fact, the deterministic parallel model of Example 4.2.2 exhibits this behavior if simple fix-up computations are used.

The time-parallel simulation approach presented above can be used both for stochastic and deterministic simulation models. However, when applied to stochastic models with non-repeatable simulation executions, that parallelization approach is unstable in the sense that the amount of fix-up computations depends highly on the random numbers drawn in the execution. Even worse, in models with large state spaces and high variability in simulation states, it might be unlikely that the exact same state appears multiple times during the simulation execution. In these cases, exact state matching is not supposed to occur often, leading to an unsuitability of time-parallel simulation. To cope with these problems, the *behavior* of the simulation entities must be modified to conform to a strict repeatability. Hence, special care has to be taken for stochastic aspects of the model. Fortunately, repeatability can be achieved easily by pre-computation of random number sequences used in the model, although this introduces a high memory overhead for the storage of random numbers. As a cost efficient alternative, it is sufficient to preselect/precompute the seeds for random number streams for every time interval. This allows to use the same random number sequences for original simulations and fix-up computations. In the rest of this thesis, it is supposed that the supported models exhibit a repeatable nature.

Figure 4.3. Time-parallel simulation with fix-up computations

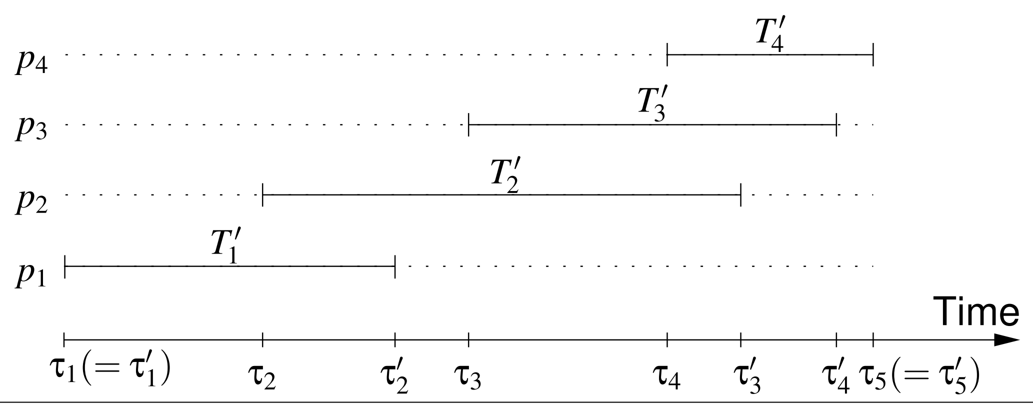


4.3. Continuous Fix-Up Computations

The original presentation of fix-up computations in time-parallel simulation [40] used an iterative view, as presented above. In general, following this approach is expected to result in a poor performance, especially if the workload of parallel processes vary greatly in terms of processing time. Here, the synchronization overhead can be decreased by breaking up the strictly iterative approach. However, this indicates that a more suitable approach to model fix-up computations can be found. An alternative view of fix-up computations in time-parallel simulation regards fix-up computations as continuous operations, which do not have to be split in phases. This also has the advantage, that a time-parallel simulation algorithm with continuous fix-up computations can be easily enhanced to provide results progressively (see Chapter 6 for more details).

In the continuous view, Fix-up computations for an interval T_{k+1} are just a continuation of the simulation of the preceding interval T_k by process p_k until a time τ_k', where the state of the initial simulation period performed by process p_{k+1} matches the corresponding state calculated during the fix-up computations by process p_k. Figure 4.3 illustrates the concept of fix-up computations. As can be seen in the figure, the correct sequence of states results from a concatenation of the sequences of states of intervals

Figure 4.4. Overlapping time intervals



$T'_k = (\tau'_k, \tau'_{k+1}]$ calculated by process p_k for all $k \in \{1, \ldots, m\}$.

Figure 4.3 also shows the case, where state matching does not occur during the fix-up computations of process p_2 for interval $T_3 = [\tau_3, \tau_4]$. In that case, process p_2 continues fix-up computations beyond τ_4. State matching is now attempted against the sequence of states calculated by process p_3 for interval T_4. The fix-up computations of process p_1 now overlap with the initial simulation period of p_2 in the interval $[\tau_2, \tau'_2]$.

Overlapping of simulation intervals is illustrated in Figure 4.4: every process computes a time interval that overlaps with the time interval of the successor. The overlap has to extend to a simulation time for which the originally computed state equals that computed by the processor producing the simulation time overlap. To be more specific, the *fix-up period* of process p_k is the time interval $[\tau_{k+1}, \tau'_{k+1}]$. Now every process p_k is responsible for calculating the simulation results of the time interval $[\tau'_k, \tau'_{k+1}]$, hence the fix-up period of p_k can also be interpreted as a *warm-up period* of process p_{k+1}. In order to produce simulation results identical to that of a sequential simulation, the times τ'_{i+1} have to be chosen such that the simulation states of each pair (p_k, p_{k+1}) of adjacent processes match at τ'_{k+1}. In the worst case, this may lead to a situation where process p_1 has to compute the evolution of state variables for the whole simulation time. In this case, time-parallel simulation degrades to sequential simulation.

An important characteristic of time-parallel simulation is the amount of computational overhead introduced with fix-up computations. Employing the continuous view, this amount can be expressed relative to the time interval boundaries. It depends on the state calculations done by the corresponding processes. Hence, the length of interval T'_k handled by a process p_k, is not known a priori and does not necessarily have the same length as other intervals. The computational overhead of the parallel simulation is given by the lengths of fix-up periods of processes. The computational overhead O due to fix-up computations can be measured as the sum of the lengths of fix-up periods and expressed relative to the length τ of the whole simulation:

$$O := \frac{1}{\tau} \sum_{k=2}^{m} (\tau'_k - \tau_k)\,. \tag{4.1}$$

For equidistant process start times τ_k, i.e. $\tau_k = (k-1)\frac{\tau}{m}$, (4.1) can be simplified to

$$O = \frac{1}{\tau} \sum_{k=2}^{m} \left(\tau'_k - (k-1)\frac{\tau}{m}\right) = \frac{1}{\tau} \sum_{k=2}^{m} \tau'_k - \frac{m-1}{2}\,. \tag{4.2}$$

The worst case for the overhead is that all processes perform fix-up computations until the end of the simulation time, i.e. $\forall k \in \{2,\ldots,m\} : \tau'_k = \tau$. The overhead $\widetilde{O}$ this case can be calculated from (4.2):

$$\widetilde{O} := \frac{1}{\tau} \sum_{k=2}^{m} \tau - \frac{m-1}{2} = \frac{m-1}{2}\,. \tag{4.3}$$

As an example, consider a simulation with $\tau = 100$ and four parallel processes ($m = 4$). In the worst case, $\widetilde{O} = 1.5$, i.e. the computational overhead for the parallel calculation is 1.5 times the length of the overall simulation interval. The total work done is 2.5 times the work in the sequential case. To determine the overall efficiency of the parallel algorithm, additional knowledge about the communication and synchronization overheads would be necessary.

Besides the overhead O due to fix-up computations, the overall computational overhead of time-parallel simulation consists mainly of the cost of state match detection. However, the efficient implementation of the latter depends on the specificities of the corresponding model. Furthermore, the efficiency of the parallel simulation also depends on the synchronization and communication overheads. Unfortunately, no general assertions can be made regarding these two types of overhead, as they are highly model and implementations specific. Hence, for discussions of the efficiency of time-parallel simulation, computational overhead remains one of the most important aspects.

A direct consequence of the observations above is that the efficiency of a time-parallel simulation model can be pre-estimated by simulating the fix-up computations to measure the presumable state match times. Simulation can be performed easily by the execution of two overlapping sequential simulations working on the same stream of random numbers, but starting at different locations of the random number stream. This approach was used in this thesis to analyze the feasibility of time-parallel road traffic simulation in Chapter 9.

5. Approximate State Matching

The state-matching problem renders time-parallel simulation for many models useless, especially for classes of models with complex states and/or large state spaces. However, in cases where final and initial states of adjacent time intervals deviate only by a marginal amount, it might be feasible to accept the corresponding simulation run, although imprecise simulation results are produced.

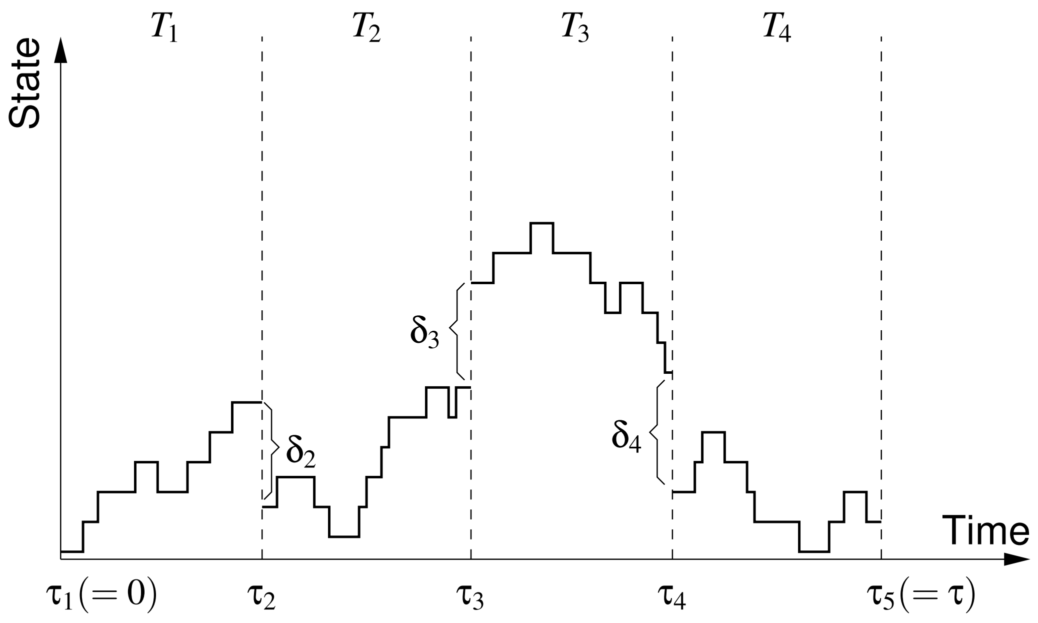
This concept of *approximate state matching* can be interpreted as a toleration of incorrect state changes at interval boundaries. The obvious advantage is the considerable gain in speedup due to a lower computation and synchronization overhead. In the case of toleration of arbitrary state deviations, the parallel model is supposed to scale linearly with the number of processing nodes, as the overhead of the parallelization is minimized.

However, the increase in performance gained with approximate state matching comes at the cost of an error that is introduced in the simulation results. If no further actions are taken, the error might seriously invalidate results without any knowledge about the level of the introduced error. In order to evaluate the impact of approximate state matching for a given model, a measure for the error has to be established. Subsequently, this measure can be used to estimate the amount of error in the simulation results in order to evaluate the quality of the parallel simulation, or even to implement an error control mechanism that limits the level of approximation to achieve more accurate results.

5.1. Basic Approach

When considering iterative fix-up computations, as presented in Section 4.2, approximate state matching is a straightforward method. Instead of trying to find exact state matches at interval boundaries, approximate state matching tolerates a deviation of the corresponding states of two adjacent time

Figure 5.1. Deviation of states δ_k with iterative fix-up

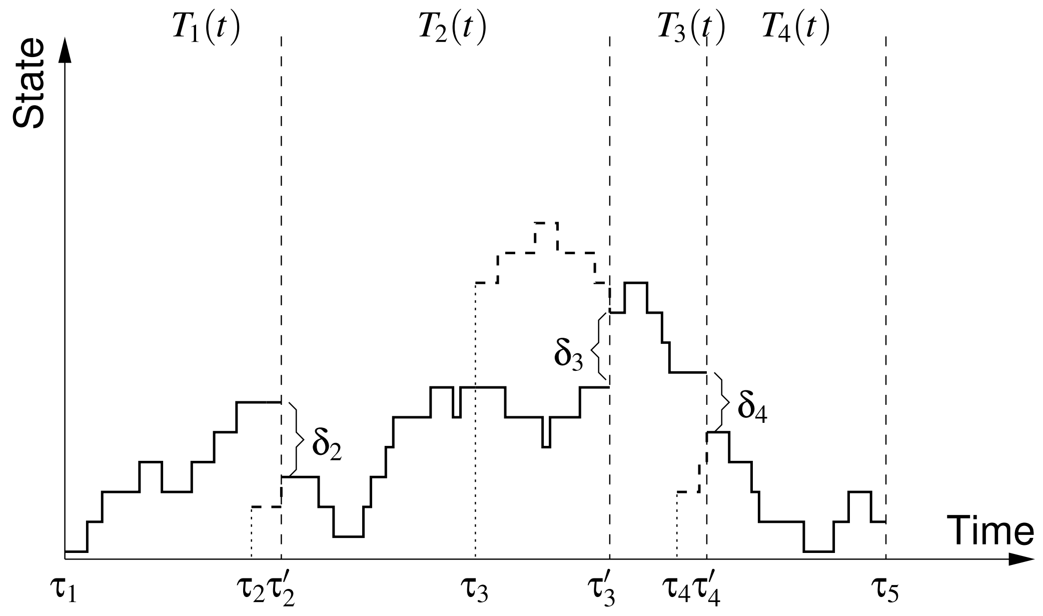


intervals. The amount of deviation at an interval boundary τ_k is measured with the *deviation of states* δ_k. This is illustrated in Figure 5.1.

The deviation of states δ_k at a boundary τ_k measures the distance between the final state Z_{k-1} of a simulation iteration of interval T_{k-1} and the initial state z_k of the same simulation iteration of interval T_k. Hence, this is a local measure, conveying only limited information of the overall accuracy of the simulation execution. However, during a typical time-parallel simulation run, no further information is available. When using approximate state matching, a proper definition of δ_k has to comply with several requirements. This is discussed extensively in Section 5.2. Furthermore, instead of tolerating any amount of deviation at interval boundaries, error control can be applied with a threshold that specifies the maximum amount of state deviations at boundaries (see Section 5.3).

The utilization of approximate state matching with continuous fix-up computations is slightly different, as processes are not constrained to execute only inside their assigned time intervals. Here, approximate state matching is applied by having a process p_k execute fix-up computations until a time τ'_k, where an acceptable level of the deviation of the states of processes p_k and p_{k-1} at τ'_k has been reached. This is approach is illus-

Figure 5.2. Deviation of states δ_k with continuous fix-up



trated in Figure 5.2. Which amount of state deviation is acceptable is again a matter of error control discussed in Section 5.3.

5.2. Error Characterization

In the basic model of approximate state matching introduced above, δ_k is a distance measure between the final state Z_{k-1} of interval T_{k-1} and the initial state z_k of the directly following interval T_k. The toleration of incorrect state changes at a number of interval boundaries induces an *overall error E* in the simulation results.

Depending on the type of model used, the nature of the error E might differ significantly. In stochastic models, where simulation results are used as statistical estimators of random variables, approximate state matching might lead to a bias and increase the variance of estimators. An appropriate error measure must be defined so as to reflect these two properties.

Example 5.2.1. Consider a queuing system where the average number of jobs in the system is to be calculated. When performing multiple replications of the simulation execution, the mean $\overline{J}$ can be determined together with its empirical variance s_J^2. To measure the error of a corresponding approximate parallel simulation, the deviation between the average number of jobs of the sequential simulation $\overline{J}_{seq}$ and the parallel simulation $\overline{J}_{par}$ is considered, as well as the increase of the variance $s^2_{J_{par}} - s^2_{J_{seq}}$.

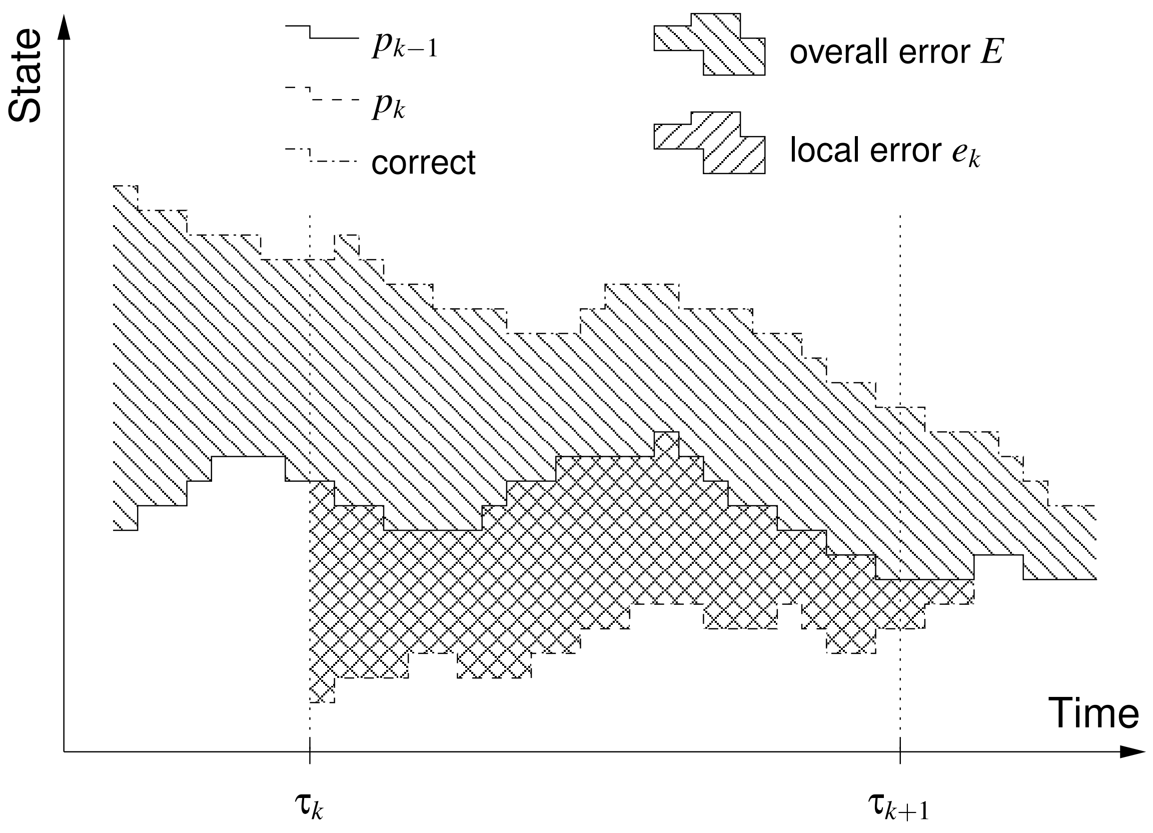
If repeatability of simulation executions has been ensured via pre-sampling of random variables (see Section 4.2), results are calculated deterministically for a given set of random numbers. Using approximate state matching with repeatable simulations, the error E in simulation results is reproducible. Nevertheless, using different sets of random numbers in separated experiments, the error E is itself an estimator of the true value inherent in the model that might not satisfy the desired statistical properties of being unbiased and consistent.

Another possibility are deterministic simulation models not exhibiting any stochastic properties (e.g. trace-driven cache simulation – see Chapter 8). In that case, results can not be interpreted as statistical estimators and approximate state matching simply leads to a deviation of approximate and exact results.

The overall error E is a result of the toleration of incorrect state changes at interval boundaries. With a proper definition of the deviation measure δ, every deviation of states δ_k at boundary τ_k contributes individually to the overall error E. The *local error* e_k due to an incorrect state change at boundary τ_k records the direct impact of the toleration of δ_k to that part of the overall result calculated by p_k. As thus, it relates the execution performed by p_k starting from an incorrect initial state at time τ_k with the execution that would be performed by the preceding process p_{k-1}, rather than the correct execution! In stochastic simulation models, the state deviations δ_k and the individual errors e_k are random variables, as they are determined by a simulation execution, which is of a stochastic nature. The following example is supposed to illustrate the relationship between overall error E and local errors e_k.

Example 5.2.2. Consider the queuing system model from Example 5.2.1, where the average number of jobs in the system is to be determined. For reasons of conciseness, the contribution of approximate state matching to the variance of the estimator is ignored, considering only the difference in the mean $\overline{J}$ of parallel and sequential simulations as overall error E. Figure 5.3 shows an extract of a sample approximate time-parallel simulation execution, concentrating on the simulation of a time interval T_k. The errors

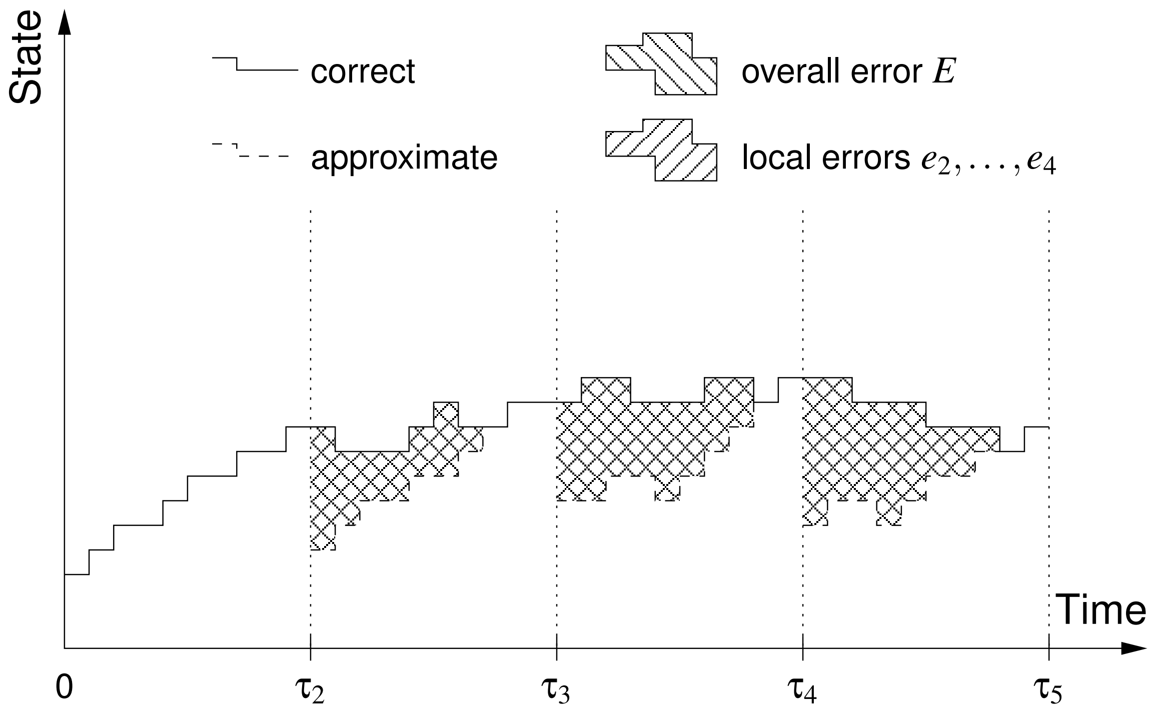
Figure 5.3. Overall error compared to local error



E and e_k are illustrated here as the area between the respective trajectories, which can be used to calculate the difference in $\overline{J}$.

As discussed above, a local error e_k does not directly reflect the part of the overall error that is caused by a deviation of states at τ_k. The definition of e_k is useful to evaluate the impact of state deviations on E, as it can be much more easily calculated and/or measured in experiments. However, for a proper analysis of the overall error by way of examining the local errors, there has to be a strong correlation between both types of error. Depending on the nature of the simulation model, different types of relationships between local and overall error exist. At best, the effect of the toleration of an incorrect state change in a time interval T_k is local to T_k, i.e. the simulation inside T_k is robust to the incorrect state change in the sense that the final state Z_k of T_k does not depend on the initial state z_k. In this case, it should

Figure 5.4. Relationship of errors in the best case



hold that

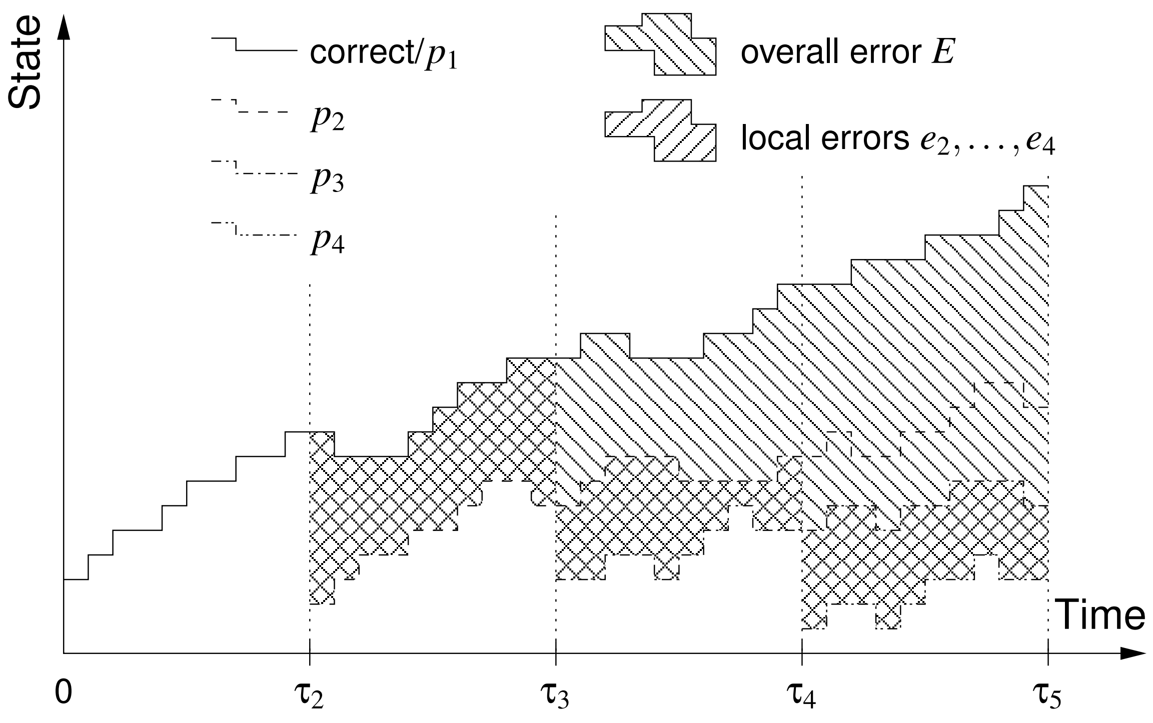
$$E = \sum_{k=2}^{m} w_k e_k,$$

which is a weighted sum of the local errors with some weights $w_2, \ldots, w_m$. Figure 5.4 illustrates this case (note the coincidence of local errors and overall error).

The worst case for the overall error E occurs if each of the errors e_k due to incorrect state changes propagates through the rest of the simulation. Utilizing the local errors to analyze the accuracy of the simulation leads to a serious underestimation of the overall error E in this case. Figure 5.5 illustrates this case.

Error measures for the local errors e_k and the overall error E can be used to evaluate the quality of an approximate simulation model relative to the corresponding sequential model. Evaluation can be twofold: *experimental* or *analytical*. In an experimental evaluation, the error E is measured by comparing simulation results of repeated experiments with the parallel and sequential models. This method is straightforward and easily usable, but

Figure 5.5. Relationship of errors in the worst case

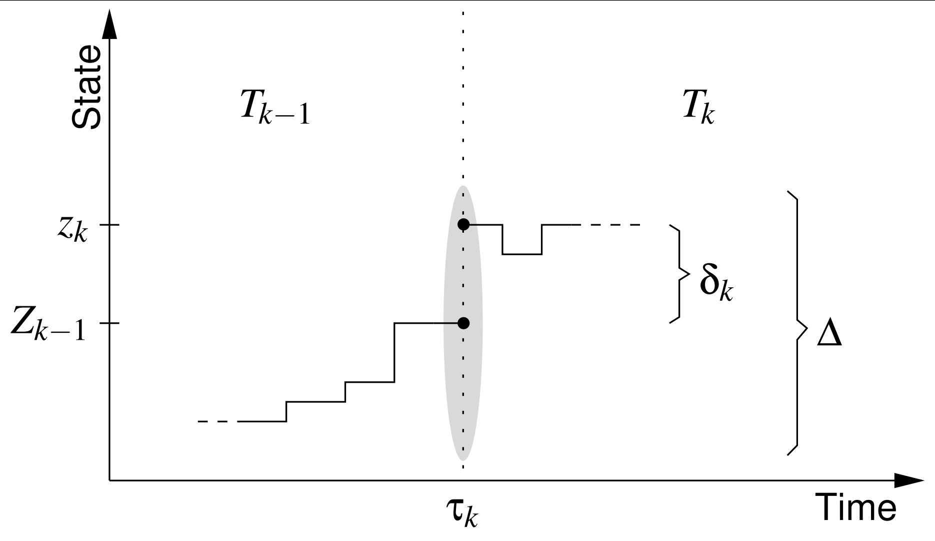


has the disadvantage that the approximate model can only be evaluated for the specific parameters used in the experiments. Extrapolating results to predict the quality of the approximation for other parameter combinations is dangerous. With analytical evaluation, in contrast, the quality of the approximate model can be examined for all possible parameter combinations. In the best case, the error can be calculated directly. Unfortunately, even for simple simulation models, like single server queueing systems, this proves to be hard.

5.3. Error Control

If a deviation of states δ_k at interval boundary τ_k can be measured during a simulation execution, it can be used to implement an error control in the approximate simulation. This is done with a tolerance Δ, where a simulation execution of an interval T_k is accepted if $\delta_k \leq \Delta$. If $\delta_k > \Delta$, fix-up computations have to be performed as in the basic time-parallel simulation approach.

Figure 5.6. Error control



In Figure 5.6, the scenario is explained in detail: a simulation execution of time interval T_k started with an initial state z_k. When the simulation of the previous interval T_{k-1} completes, it is found that z_k is an incorrect state, since it does not exactly match the final state Z_{k-1} of the simulation of T_{k-1}. Nevertheless, the simulation of T_k is accepted, as the deviation at time τ_k, which is simply defined as the distance between states Z_{k-1} and z_k, is smaller than the predefined tolerance Δ.

If the state deviations δ_k are to be used for control of the introduced error, it is important that a strong relationship between δ_k and the corresponding error e_k exists. With a definition of δ_k where δ_k and e_k are not or only weakly related, serious errors might occur due to incorrect state changes that are accepted because of a small deviation δ_k. Due to δ_k and e_k being random variables in most cases (see Section 5.2), the relationship between them can be expressed as a correlation (or covariance, if the probability distributions of δ_k and e_k are known). A weak correlation indicates that δ_k is no reasonable measure for error control in the corresponding parallel simulation. In some specific cases, like in cache simulation presented in Chapter 8, bounds on the error e_k can be calculated from the state deviation δ_k. This represents a strong relationship between both measures as well, even if correlation between them might be low. In any case, for a reasonable implementation of error control, the relationship of state deviations to introduced errors has to be analyzed.

Note that, although most of the content of this section has been explained with iterative fix-up computations in mind, all of the discussions apply equally to continuous fix-up computations. In fact, continuous fix-up is not a new concept, but rather a different view on the method of fix-up computations. The difference is mainly located in an alternative assignment of the responsibility for performing fix-up computations. With iterative fix-up computations, a process p_k is responsible for the fix-up computations of its originally assigned time interval T_k. With continuous fix-up, this responsibility is pushed to the preceding process p_k. Although continuous fix-up computations are a redundant view on the same method, it is still worthwhile to have that view in mind, as it delivers additional understanding of the method of fix-up computations and prepares the introduction of progressive time-parallel simulation in Chapter 6.

6. Progressive Time-Parallel Simulation

Parallel simulation techniques are utilized to increase the performance of complex large-scale simulation models. Classical parallel simulation methods, as presented in Section 2.2, preserve causal relationships between events, leading to simulation results identical to those of a corresponding sequential model. However, depending on the nature of the underlying model, the parallelism achievable with these approaches is limited in many cases. Therefore, several alternative approaches have been developed that increase the achievable parallelism at the cost of a loss of accuracy in the simulation results (see Section 3.2). The novel concept of progressive time-parallel simulation introduced here is a combination of precise and imprecise approaches. The basic idea is to produce approximate simulation results as early as possible and correct them progressively afterwards until correct simulation results have been calculated. The simulation user can cancel this process at any time. This is especially interesting in systems with real-time constraints.

The methods developed in the field of parallel simulation can be compared to approaches from other fields of research. In hard real-time systems, the technique of *imprecise computations* [59] has been developed. An imprecise computation is allowed to give imprecise (approximate) results if the precise (correct) results cannot be provided in time. This is comparable to the approximate parallel simulation techniques discussed in Section 3.2.

The concept of *progressive processing* is very similar to imprecise computations, with the exception that computations are continued after the initial determination of approximate results to improve results *progressively*. One of the more prominent applications of this technique is *progressive image rendering* [10, 19, 44], where an imprecise (e.g. blurred) or incomplete representation of the image is provided to the user after a short answer time.

After that time, the quality of the image is progressively improved until the best representation of the image has been produced or the user cancels the image rendering process. This is of renewed importance with the advent of the World Wide Web (e.g. progressive rendering of Scalable Vector Graphics files with SVG 1.2 [43]). Other applications include progressive information retrieval [111], progressive data stream processing [82], and progressive DBMS-based decision support [94].

Progressive processing can be utilized effectively in contexts where quick results are desirable, regardless of their precision, but more precise results might be needed later. In the field of computer simulation, an application of progressive rendering techniques to virtual environments (e.g. [106]) immediately comes to mind. However, other applications are conceivable in conjunction with as-fast-as-possible simulation, e.g. simulation-based military decision support, simulation-based traffic control, or simulation-based scheduling and control of manufacturing systems.

Progressive simulation can be used in conjunction with parallel and distributed simulation methods to provide approximate simulation results in a very short time. This is especially useful if the efficient application of traditional parallel simulation methods is hard or even impossible to achieve.

6.1. Basic Idea

In the basic time-parallel approach presented in Chapter 4, simulation results are available only after fix-up computations of all parallel processes are complete. The other alternative is to produce results as early as possible, viz. directly after the initial simulation phase is completed. As already discussed above, simulation results might be incorrect at that time, but in many cases they might still serve as valuable indications on the correct results. Further, instead of terminating the simulation at that time (having calculated approximate results), fix-up computations can be performed, while providing increasingly accurate simulation results to the user in the further simulation progress. The accuracy of results should improve continually with runtime, reaching correct results upon completion of fix-up computations.

6.2. Simulation Progress

To capture the dynamic nature of such a system, a family of *progress functions* is introduced, indicating the local simulation time of a process at a point in wallclock time.

Definition 6.2.1 (Progress functions)**.** Let $\tau_0(t) := 0$ and $\tau_{m+1}(t) = \tau$. Furthermore, for all $k \in \{1, \dots, m\}$, let $\tau_k : \mathbb{R}_0^+ \to [0, \tau]$ be a family of monotonic increasing functions with $\lim_{t\to\infty} \tau_k(t) = \tau$ and $\tau_{k-1}(t) \leq \tau_k(t)$. τ_k is the *simulation progress function* of p_k, where the local simulation time of process p_k at wallclock time t is given by $\tau_k(t)$.

For every process p_k, it is required that the corresponding progress function is monotonically increasing. Otherwise, processes might roll back their local time, which is not considered here. Further, it is assumed that processes make progress, eventually reaching the end of the simulation period and overtaking of processes is not allowed. The latter assumption is not necessary in all cases, but is provided here to simplify the following definitions.

The processing of a progressive time-parallel simulation system can be divided in two phases. The *initial simulation phase* of a process p_k lasts until it has progressed to the start time of its following process, i.e. $\tau_k(t) = \tau_{k+1}(0)$. After that time instant, the process is in its *fix-up computation phase*. At wallclock time t, a process p_k has calculated the sequence of states of interval $[\tau_k(0), \tau_k(t)]$. At any time during the fix-up computation phase of a process, this interval can be divided in two different parts: one for which fix-up computations have already been performed by the preceding process, and the second part, where this is not the case and thus results have to be provided for this interval by the process.

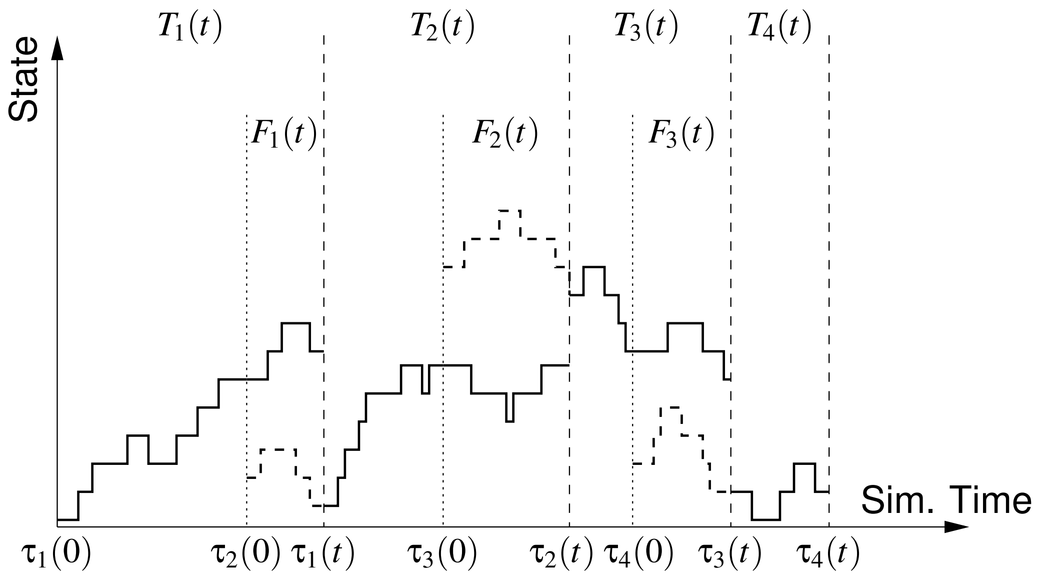
Definition 6.2.2 (Fix-up and results regions)**.** For every $k \in \{1, \dots, m-1\}$,

$$F_k(t) := [\tau_{k+1}(0), \tau_k(t)]$$

is the *fix-up region* of p_k and for every $k \in \{1, \dots, m\}$, the *results region* of p_k is defined as

$$T_k(t) := (\tau_{k-1}(t), \tau_k(t)].$$

Figure 6.1. Progressive simulation at time t



The results region of a process has a direct impact on the results generated by that process: Results and statistics are adapted to reflect only the computations performed inside the results region of the process (see Section 6.3). A snapshot of an example simulation system at wallclock time t is given in Figure 6.1.

In analogy to the overhead in basic time-parallel simulation given in Equations (4.1) to (4.3) in Section 4.3, the computational overhead in a progressive time-parallel simulation system at wallclock time t can be defined as the sum of lengths of fix-up phases:

$$O(t) := \frac{1}{\tau} \sum_{k=1}^{m-1} (\tau_k(t) - \tau_{k+1}(0)). \tag{6.1}$$

If we suppose equidistant start times of processes, i.e. $\tau_k(0) = (k-1)\frac{\tau}{m}$:

$$O(t) = \frac{1}{\tau} \sum_{k=1}^{m-1} \left(\tau_k(t) - \frac{k\tau}{m}\right) = \frac{1}{\tau} \sum_{k=1}^{m-1} \tau_k(t) - \frac{m-1}{2}. \tag{6.2}$$

It is possible to measure the maximum amount of overhead $\widetilde{O}$ by determining the limit of $O(t)$ for $t \longrightarrow \infty$:

$$\widetilde{O} := \lim_{t\to\infty} O(t) = \frac{1}{\tau}\sum_{k=1}^{m-1} \lim_{t\to\infty} \tau_k(t) - \frac{m-1}{2}. \tag{6.3}$$

This can be simplified to

$$\widetilde{O} = \frac{1}{\tau}\sum_{k=1}^{m-1} \tau - \frac{m-1}{2} = \frac{m-1}{2}, \tag{6.4}$$

which is the same result as for the worst-case overhead $\widetilde{O}$ for the basic time-parallel simulation given in (4.3).

6.3. Simulation Results

The notions of initial simulation phase and fix-up computation phase of a process p_k were introduced above. Now, similar terms are defined for the overall simulation. The overall simulation system is in the *initial simulation phase* if at least one of the processes is in the initial simulation phase, and in the *fix-up computation phase* afterwards. The identifier t_0 will be used in the rest of the chapter to denote the time instant where the system finishes the initial simulation phase.

Definition 6.3.1 (Start of fix-up computation phase)**.** The fix-up computation phase of a progressive time-parallel simulation systems starts at time

$$t_0 := min\{t \geq 0 | \forall k \in \{1,\ldots,m\} : \tau_k(t) > \tau_{k+1}(0)\}.$$

Simulation results are not available during the initial simulation, as only a part of the whole simulation interval has been processed. At any time t after t_0, preliminary simulation results are available by combining the local results computed by each process p_k in the time-interval $T_k(t)$. Let $\mathcal{R} = \mathbb{R}^n$, with an arbitrary $n \in \mathbb{N}$, denote the result space of the simulation (in general, the results of a simulation can be of an arbitrary type. However, in most practical cases, atomic results can be mapped to real numbers, so using $\mathbb{R}^n$ for the result space should be sufficient). Then the results calculated by the processes $p_1,\ldots,p_m$ are represented by the a number of result functions.

Definition 6.3.2 (result functions)**.** For all $k \in \{1,\ldots,m\}$, let $R_k : [t_0,\infty) \to \mathcal{R}$ be the *local result function* of process p_k, where $R_k(t)$ is the result computed by p_k for interval $T_k(t)$ at wallclock time t.
Furthermore, let $\widehat{R} : [t_0,\infty) \to \mathcal{R}$ be the *global result function* which is an arbitrary combination of the R_k for all $k \in \{1,\ldots,m\}$. A requirement on $\widehat{R}$ is: If $T_k(t) = \emptyset$, i.e. $\tau_{k-1}(t) = \tau_k(t)$, then $R_k(t)$ should have no impact at all on $\widehat{R}(t)$.

The preliminary simulation results computed at time t are given by the global result function. The only requirement on the global result function is, that the local result of a process p_k should have no impact at all on the global result if the results region of p_k is empty, i.e. no results should be produced by the process. This also implies, that

$$\lim_{t\to\infty} \widehat{R}(t) = \lim_{t\to\infty} R_1(t)\,, \tag{6.5}$$

i.e. if the parallel simulation is run for a sufficiently long time, the global result is completely determined by the first parallel process p_1.

The global result function $\widehat{R}$ is an estimation of the correct result R that would be provided by an equivalent sequential or basic time-parallel simulation execution. Guessing of initial states at time interval boundaries before start of the simulation is not performed for the first process p_1, because the same initial state can be used as for a sequential simulation. Therefore, the results calculated by process p_1 for its time interval $T_1(t)$ are always identical to those of a sequential simulation. Together with the assumption $\lim_{t\to\infty} \tau_k(t) = \tau$, it holds that

$$R = \lim_{t\to\infty} R_1(t). \tag{6.6}$$

These observations lead to the following important theorem, which indicates that the progressive parallel simulation eventually produces correct results.

Theorem 6.3.1. *Given a progressive time-parallel simulation execution with progress functions $\tau_1,\ldots,\tau_m$, local result functions $R_1,\ldots,R_m$, and global result function $\widehat{R}$. Then it holds, that*

$$\lim_{t\to\infty} \widehat{R}(t) = R\,.$$

Proof. Follows directly from (6.5) and (6.6). □

Theorem 6.3.1 settles the question, whether progressive time-parallel simulation eventually leads to correct simulation results. In practical cases, however, the rate of the convergence of $\widehat{R}$ and R, reflecting the development of the accuracy of the simulation, is of more interest. The *accuracy function* α is used to relate the value of the estimated result $\widehat{R}(t)$ at a time t with the correct simulation result. Different possibilities for the definition of α exist. For example α could be defined as the difference between estimated and correct result:

$$\alpha(t) := R - \widehat{R}(t)\,,$$

or as the ratio between estimated and correct result:

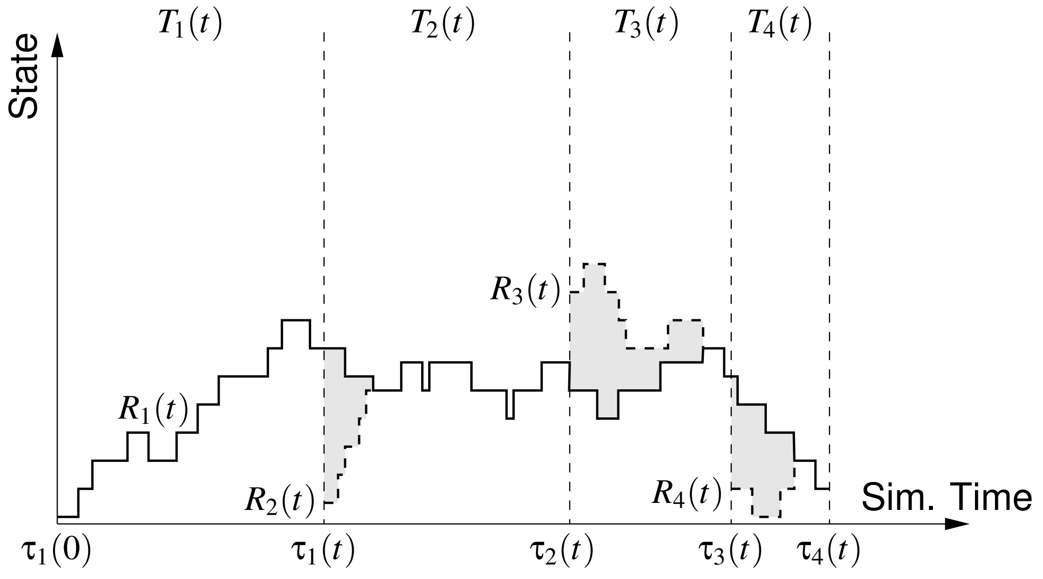
$$\alpha(t) := \frac{\widehat{R}(t)}{R}\,.$$

The precise definition of α depends on the specificities of the result functions in the model. More specifically, it is not defined here, whether an increasing value of α represents an increasing or decreasing accuracy. In most cases, the accuracy function is not available during the simulation execution, but it can be used to evaluate the feasibility of a specific progressive simulation application. Section 7.5 contains an exemplary definition of the accuracy function.

Another important aspect of the accuracy function is its monotonicity. In general, it cannot be guaranteed that α is a monotonic function. However, in such a case, the employment of progressive simulation is dangerous, as the user might cancel a simulation execution due to a sufficient accuracy of the results, which would be misleading if the accuracy degrades with further simulation. Therefore, a necessary preliminary for the reasonable application of progressive time-parallel simulation to a model is that α is found to be a monotonic function. For example, in the specific model of single-server queuing systems, this property is shown in Chapter 7.

Figure 6.2 illustrates the concepts of local results and accuracy. The figure shows a snapshot of a progressive time-parallel simulation system at wallclock time t. It is supposed, that the goal of the simulation is to determine the area below the trajectory. The correct result (which is not known at time t) is the area between the solid line corresponding to R. The local results $R_k(t)$ at time t consist of the areas below the dashed lines

Figure 6.2. Accuracy at time *t*



starting at the times $\tau_1(t), \ldots, \tau_3(t)$. The accuracy of the system at time t is illustrated here as the shaded areas between the dashed and solid lines.

Part III.

Case Studies

7. Queuing Systems

Queuing Systems are an important modeling tool due to their simple nature, as well as the substantial work that has been done in the theory of queuing phenomena. The possibility of composition of queuing systems to arbitrarily large queuing networks significantly extends the applicability of queuing theory. For the efficient simulation of queuing networks, the handling of their building blocks, the atomic queuing systems, has to be mastered. This is also true for the parallelization of queuing network simulations.

The concepts discussed in Part II are applied to the simulation of queuing systems here. Alternative schemes for the parallel simulation of queuing systems and queuing networks with varying strengths and weaknesses have been developed earlier [35, 57, 101, 103].

In order to apply approximate time-parallel simulation techniques, repeatability of the simulation has to be assured (see Section 4.2). Otherwise, no guarantee regarding the progression of fix-up computations could be given. Fortunately, repeatability of a stochastic queuing system simulation can be achieved by sampling of random numbers prior to the simulation execution. In practice, no presampling of random numbers might be necessary, the same effect probably being achievable by the utilization of pseudo random number generators with fixed seeds. Furthermore, it will be supposed that the sequence of job arrival instants is available instead of the job interarrival times. The determination of job arrival instants from interarrival times is a task that can be calculated easily and efficiently [35].

The pre-computed sequences of job arrival instants and service times are stored in an input trace, which is used to perform a repeatable trace-driven simulation. For time-parallel simulation, the responsibility for the simulation of different parts of an input trace is assigned to separate parallel processes. However, as discussed in Section 4.3, the subtraces handled by different processes overlap due to fix-up computations. Although decomposition is not directly applied to the simulation time here, the resulting parallel simulation exhibits all of the characteristics of a time-parallel simulation system.

Before introducing approximate queuing simulation, some necessary preliminaries have to be provided. First, a mathematical definition of G/G/1 queuing system simulation is given. Then, basic time-parallel queuing system simulation is presented.

7.1. Foundations

In the following, the problem of queuing simulation is reduced to the calculation problem of determining the departure times of a number of jobs with associated arrival instants and service times. Based on this sequence of departure instants, the trajectory of the number of jobs in the system over time can be reconstructed easily [35]. The calculation of the job departure instants of a G/G/1 queue can be represented by a recursive function that relates a job with its departure time [49]. The basic observation underlying this formulation is, that the departure instant $d(j)$ of job j is determined by its own service time $s(j)$ and either its arrival instant $a(j)$ or the departure instant of the previous job $d(j-1)$ (depending on which occurs later in time):

$$d(j) = max(a(j), d(j-1)) + s(j)\,. \tag{7.1}$$

This recursive formula for the calculation of departure times is easily understandable and can be implemented efficiently as a sequential computer program. However, the goal of this chapter is to illustrate the application of approximate time-parallel simulation to queuing systems. Therefore, (7.1) is modified to support the parallel calculation of departure times, defined later.

Definition 7.1.1 (departure function)**.** Let $N := \{1, \dots, n\}$ be the set of jobs to simulate. Let $s : N \to \mathbb{R}^+$ be a positive function of job service times. Let $a : N \to \mathbb{R}_0^+$ be a strictly monotonic increasing function of job arrival instants. The departure function $d_u^i : \{i-1, \dots, n\} \to \mathbb{R}_0^+$ for $u \in \mathbb{R}_0^+$ and $i \in N$, is defined as

$$d_u^i(j) := \begin{cases} u & , \quad \text{if } j = i-1 \\ max(a(j), d_u^i(j-1)) + s(j), & \text{otherwise} \end{cases} .$$

The parameter i is used to restrict the domain of the function to the interval $\{i-1,\ldots,n\}$, starting at job i, which is a part of the overall simulation domain. The function is defined for job $i-1$, as well, but this value is only to be used as the base case of the recursion. The parameter i is relevant in the context of time-parallel simulation of the queuing system, where the simulation of a parallel process is started with an initial job representing the time interval boundary. $i = 1$ is used for a sequential calculation of departure times and for the first process in a time-parallel simulation execution. The parameter u of the departure function is used to indicate an *initial delay* for the queuing system. In a sequential queuing system which is typically empty at the beginning of the observed time, $u = 0$. However, if there is an initial load of the system, the queue starts with at least one existing job. This property can be captured by introducing an initial delay with $u > 0$ that must pass until the first job can be served. Later, this is used to represent the performance of fix-up computations in a time-parallel simulation, where a process starts its fix-up computation phase with a number of jobs in its queue, which has been determined during its initial simulation phase. As can be seen easily, d_u^i is a strictly monotonic increasing function.

For time-parallel queuing simulation, it is convenient to compare the calculations of two adjacent processes, e.g. to decide on the termination of fix-up computations. The following lemmas provide a foundation for the formalization of time-parallel queuing simulation.

Lemma 7.1.1. *Let $i \in N$ and $u,v \in \mathbb{R}_0^+$ with $u \geq v$. Then for all $j \in \{i-1,\ldots,n\}$:*

$$d_u^i(j) \geq d_v^i(j)\,.$$

Proof. Proof by induction over j with the basic case $j = 0$ trivially met.
Let $d_u^i(j-1) \geq d_v^i(j-1)$ hold.
Case 1 ($d_u^i(j-1) \leq a(j) (\Rightarrow d_v^i(j-1) \leq a(j))$):
$d_u^i(j) = a(j) + s(j) = d_v^i(j) \Rightarrow d_u^i(j) \geq d_v^i(j)$
Case 2 ($d_u^i(j-1) > a(j)$ and $d_v^i(j-1) \leq a(j)$):
$d_u^i(j) = d_u^i(j-1) + s(j) > a(j) + s(j) = d_v^i(j)$
Case 3 ($d_u^i(j-1) > a(j)$ and $d_v^i(j-1) > a(j)$):
$d_u^i(j) = d_u^i(j-1) + s(j) \geq d_v^i(j-1) + s(j) = d_v^i(j)$
Case 4 ($d_u^i(j-1) \leq a(j)$ and $d_v^i(j-1) > a(j)$):
Cannot occur due to $d_u^i(j-1) \geq d_v^i(j-1)$. □

Lemma 7.1.1 indicates that for two simulation executions starting with different initial delays, the departure times of all jobs of the simulation with the lower initial delay are always dominated by the departure times of jobs in the second execution. However, it is still unknown how departure functions with different parameters i can be compared. The following lemma shows how this can be done.

Lemma 7.1.2. *Let $i, i' \in \mathbb{N}_0$ with $i \leq i'$ and $u \in \mathbb{R}_0^+$. Let $u' := d_u^i(i'-1)$. Then the following equality holds for all $j \in \{i'-1, \ldots, n\}$:*

$$d_u^i(j) = d_{u'}^{i'}(j) .$$

Proof. Proof by induction over j with the base case $j = i'-1$ met due to Definition 7.1.1. Suppose, that for any $j \in N_{i'}$, $d_u^i(j) = d_{u'}^{i'}(j)$ holds. Then $d_u^i(j+1) = max(a(j+1), d_u^i(j)) + s(j+1) = max(a(j+1), d_{u'}^{i'}(j)) + s(j+1) = d_{u'}^{i'}(j+1)$. □

Lemma 7.1.2 shows, that a departure function d_u^i is equivalent to a function $d_{u'}^{i'}$ with another parameter $i' > i$ for all $j \geq i'-1$. This can be achieved by a simple adjustment of the initial delay u. Hence, if two departure functions with differing parameters i and i' are to be compared, the function with the smaller parameter can be adjusted properly.

Now, a strong relationship between simulation executions with the same parameter i has been established and it has been shown how departure functions with different parameter i can be compared. The purpose of the next section is to show how state match detection in time-parallel queuing simulation can be performed: State matching occurs if the states of adjacent simulations are identical. In fact, it is shown in Section 7.2, that matching between the states calculated by two processes p_k and p_{k+1} occurs exactly when the queuing system gets empty the first time during the fix-up computations of p_k. In the case of calculation of departure times, as presented in Definition 7.1.1, the property of an empty queue is represented by the expression $d_u^i(j-1) \leq a(j)$, in which case the service of job j is not influenced at all by the history of the queuing system, as job $j-1$ departs from the system before job j arrives. The following lemma is an important step to the property of match detection discussed above. It gives a characterization of the state match time for two simulations with differing initial delays, but the same domain.

Lemma 7.1.3. *Let $i \in \mathbb{N}_0$ and $u, v \in \mathbb{R}_0^+$ with $u \geq v$. Then $\forall j \in \{i-1, \ldots, n\}$, the following two statements are equivalent:*

$$\begin{array}{ll} \text{(i)} & d_u^i(j) = d_v^i(j) \\ \text{(ii)} & d_u^i(j-1) = d_v^i(j-1) \vee d_u^i(j-1) \leq a(j)\,. \end{array}$$

Proof. Let $i \in \mathbb{N}$ be an arbitrary natural number. Furthermore, choose an arbitrary $j \in \{i-1, \ldots, n\}$.
Case 1 ($d_u^i(j-1) \leq a(j)$ and $d_v^i(j-1) \leq a(j)$):
$d_u^i(j) = max(a(j), d_u^i(j-1)) + s(j) = a(j) + s(j)$
$= max(a(j), d_v^i(j-1)) + s(j) = d_v^i(j)$
$\Rightarrow$ (i) always holds.
(ii) trivially holds due to $d_u^i(j-1) \leq a(j)$.
Case 2 ($d_u^i(j-1) > a(j)$ and $d_v^i(j-1) \leq a(j)$):
$d_u^i(j) = max(a(j), d_u^i(j-1)) + s(j) = d_u^i(j-1) + s(j)$
$> a(j) + s(j) = max(a(j), d_v^i(j-1)) + s(j) = d_v^i(j)$
$\Rightarrow$ (i) never holds.
(ii) never holds due to $d_u^i(j-1) > a(j) \geq d_v^i(j-1)$.
Case 3 ($d_u^i(j-1) > a(j)$ and $d_v^i(j-1) > a(j)$):
(i) is reduced to $d_u^i(j-1) = d_v^i(j-1)$.
$d_u^i(j) = max(a(j), d_u^i(j-1)) + s(j) = d_u^i(j-1) + s(j)$
$d_v^i(j) = max(a(j), d_v^i(j-1)) + s(j) = d_v^i(j-1) + s(j)$
Thus, it holds that $d_u^i(j) = d_v^i(j) \Leftrightarrow d_u^i(j-1) = d_v^i(j-1)$, which settles Case 3.
Case 4 ($d_u^i(j-1) \leq a(j)$ and $d_v^i(j-1) > a(j)$):
Cannot occur due to Lemma 7.1.1. □

7.2. Time-Parallel Queuing Simulation

With the foundations defined in the previous section, it is now possible to define time-parallel queuing system simulation.

Definition 7.2.1. Let $N := \{1, \ldots, n\}$ be a set of jobs with corresponding arrival instants $a : N \to \mathbb{R}_0^+$ and service times $s : N \to \mathbb{R}^+$. The responsibility for the calculation of departure times of jobs is assigned to m processes $p_1, \ldots, p_m$, assigning start job $j_k \in \{1, \ldots, n\}$ to every process p_k ($j_1 = 1 <$

$j_2 < \ldots < j_m \leq n$). Furthermore, each logical process p_k is assigned a simulation interval $N_k := \{j_k - 1, \ldots, n\}$ and an initial delay $u_k := a(j_k)$

The initial delay u_k of p_k is set to the arrival instant of j_k, the first job processed by p_k. This value of the initial delay represents an empty queue at the beginning of the simulation of each simulation interval, as the departure time of the first job j_k is in any case $a(j_k) + s(j_k)$ and the job is not influenced by any of the preceding jobs in the system.

Before going on, the following lemma is defined to simplify the proofs of Lemma 7.2.2 and Theorem 7.2.1.

Lemma 7.2.1. *Let $k, l \in \{1, \ldots, m\}$ with $k < l$ and $z := d_{u_k}^{j_k}(j_l - 1)$. Then the following implication holds for all $j \in \{j_l, \ldots, n\}$:*

$$z \leq u_l \Rightarrow d_{u_k}^{j_k}(j) = d_{u_l}^{j_l}(j) \, .$$

Proof. First of all, note that $k < l$ directly leads to $j_k < j_l$ due to definition, which is silently exploited in all of the following proofs. Due to definition, it holds that $u_l = a(j_l)$ and $d_z^{j_l}(j_l - 1) = z$. Hence, $d_z^{j_l}(j_l - 1) \leq a(j_l)$. Therefore, $d_z^{j_l}(j_l) = a(j_l) + s(j_l) = d_{u_l}^{j_l}(j_l)$. Repeated application of Lemma 7.1.3 leads to $d_z^{j_l}(j) = d_{u_l}^{j_l}(j)$ for all $j \in \{j_l, \ldots, n\}$. An application of Lemma 7.1.2 settles the proof. □

The value $a(j_l)$ of the initial delay u_l of a process p_l is reasonable, as it leads to the domination of $d_{u_l}^{j_l}$ by $d_{u_k}^{j_k}$ for every $k < l$. This property is shown in the following lemma and can be exploited later for the determination of the match detection criterion.

Lemma 7.2.2. *Let $k, l \in \{1, \ldots, m\}$ with $k < l$. Then the following inequality holds for all $j \in N_l$:*

$$d_{u_k}^{j_k}(j) \geq d_{u_l}^{j_l}(j) \, .$$

Proof. Let $z := d_{u_k}^{j_k}(j_l - 1)$. There are two cases for the value of z:

Case 1 ($z > u_l$):

Then, due to Lemma 7.1.1, $d_z^{j_l}(j) \geq d_{u_l}^{j_l}(j)$ for all $j \in N_l$. With an application of Lemma 7.1.2, this translates directly to $d_{u_k}^{j_k}(j) \geq d_{u_l}^{j_l}(j)$ for all $j \in N_l$.

Case 2 ($z \leq u_l$):

Follows directly from Lemma 7.2.1.

□

Every one of the functions $d_{u_k}^{j_k}$ has a domain that extends up to the last job n. It is not possible to reduce the domain further, as for any process p_k, fix-up computations might be necessary up to n. However, it is not necessary for every process p_k to perform fix-up computations up to job n, but only until a job arrives after the previous job departed, i.e. until the process has an empty queue. This is formalized in the following theorem.

Theorem 7.2.1. *Let $k,l \in \{1,\dots,m\}$ with $k<l$. Then for all $j \in \{j_l,\dots,n\}$, the following two statements are equivalent:*

$$\begin{aligned} &\text{(i)} \quad d_{u_k}^{j_k}(j) = d_{u_l}^{j_l}(j) \\ &\text{(ii)} \quad d_{u_k}^{j_k}(j-1) = d_{u_l}^{j_l}(j-1) \vee d_{u_k}^{j_k}(j-1) \le a(j)\,. \end{aligned}$$

Proof. Choose an arbitrary $j \in \{j_l,\dots,n\}$. Let $z := d_{u_k}^{j_k}(j_l-1)$. There are two cases for z:
Case1 ($z > u_l$):
Due to Lemma 7.1.3, $d_z^{j_l}(j) = d_{u_l}^{j_l}(j)$ is equivalent to $(d_z^{j_l}(j-1) = d_{u_l}^{j_l}(j-1)) \vee (d_z^{j_l}(j-1) \le a(j))$. An application of Lemma 7.1.2 settles the proof of this case.
Case2 ($z \le u_l$):
(i) holds due to Lemma 7.2.1. If $j > j_l$, (ii) holds due to Lemma 7.2.1. Hence, let $j = j_l$. Due to $z \le u_l$, $d_z^{j_l}(j_l-1) \le u_l = a(j_l)$. Thus, (ii) holds because of Lemma 7.1.2.

□

Theorem 7.2.1 has three implications:

- Once the departure times of a job match between initial simulation and corresponding fix-up computation, the departure times of all following jobs match as well.
- During fix-up computations, if a job i arrives after or at the time when the previous job departs, the departure times of i match between initial simulation and corresponding fix-up computation.
- State matching cannot occur before in the fix-up computations a job arrives at or after the time when the previous job departs.

7.3. Computational Overhead

The previous section introduced an alternative parallel processing scheme for queuing system models. An important property of that approach is the simple state match criterion, which allows for an efficient state match detection. However, as discussed in Section 4.3, another large part of the overall overhead of a time-parallel simulation execution is determined by the amount of fix-up computations that have to be performed. These in turn depend on the times of state match occurrences in the time intervals. The aim of this section is to exemplarily investigate the computational overhead O of queuing system simulation. This is done by an examination of the expectation of (the random variable) O in case of an $M/M/1$ queue with parameters λ and μ representing arrival and service rates and under the restriction $\lambda < \mu$.

7.3.1. Facts on $M/M/1$ Queues

In this section we review some formulas concerning $M/M/1$ queues, which are needed in the considerations of Section 7.3.2.

Let $N(t)$ be the random number of customers in an $M/M/1$ queuing system at time t. We consider for all $i = 1, 2, \ldots$ the busy period initiated by i customers at time $t = 0$ (cf. [84])

$$T(i) := \inf\{t \geq 0 : N(t) = 0 \quad \text{and} \quad N(0) = i\}$$

and define $T(0) := 0$. We obtain for the expectation of $T(i)$ in case of $\lambda < \mu$ [91]

$$\mathbf{E}\,T(i) = \frac{i}{\mu - \lambda}, \quad i = 0, 1, \ldots . \tag{7.2}$$

To calculate the computational overhead of the parallel simulation, we need a further definition. For $i = 0, 1, 2, \ldots$ and given $\tau_2, \ldots, \tau_m$, let

$$T_k(i) := \inf\{t \geq \tau_k : N(t) = 0 \quad \text{and} \quad N(\tau_k) = i\}, \quad k = 2, \ldots, m$$

be the busy time initiated by i customers at time τ_k. Since $\{N(t) : t \in \mathbb{R}_0^+\}$ is a continuous time Markov chain [84], it follows that $T(i)$ and $T_k(i)$ ($k =$

$2,\ldots,m$) have the same distribution. Therefore we obtain with (7.2) in case of $\lambda < \mu$ for the expectation of $T_k(i)$

$$\mathbf{E}\,T_k(i) = \tau_k + \mathbf{E}\,T(i) = \tau_k + \frac{i}{\mu - \lambda}, \quad k = 2,\ldots,m. \tag{7.3}$$

In case of $\lambda < \mu$ we know, that for all $i, j \in \{0, 1, \ldots\}$

$$P_{ij}(t) := \mathbf{P}(N(t) = j | N(0) = i) \to \pi_j := \left(1 - \frac{\lambda}{\mu}\right)\left(\frac{\lambda}{\mu}\right)^j$$

for $t \to \infty$ (cf. [84]). We now suppose that t_0 is sufficiently large to allow the assumption, that the system is encountered in steady state. Referring to this time point t_0, we obtain with (7.3) for the expectation of the busy time $T_k^{(s)}$ in steady state

$$\mathbf{E}\,T_k^{(s)} = \sum_{j=0}^{\infty} \pi_j \mathbf{E}\,T_k(j) = \tau_k + \frac{\lambda}{(\mu - \lambda)^2}, \quad k = 2,\ldots,m. \tag{7.4}$$

7.3.2. Expectation of the Computational Overhead

In Section 4.3, the computational overhead O for the parallel simulation of general systems is introduced. Now, more specialized considerations are provided by an examination of the computational overhead $O_{M/M/1}^{(s)}$ of an $M/M/1$ queue in steady state.

Let τ_k ($k = 2,\ldots,m$) be given. Then $[N(\tau_k)|N(0) = i]$ is the random number of customers in the system at time τ_k if the system had started with i customers at time $\tau_1 (= 0)$. Therefore, $T([N(\tau_k)|N(0) = i])$ is the random variable representing the first occurrence of state 0 after time τ_k under the condition $N(0) = i$. Using the total law of expectation (see [89]) we obtain with (7.3) for $k = 2,\ldots,m$

$$\begin{aligned}
\mathbf{E}\,T([N(\tau_k)|N(0) = i]) &= \sum_{j=0}^{\infty} \mathbf{E}\,T_k(j) P_{ij}(\tau_k) \\
&= \sum_{j=0}^{\infty} (\tau_k + \mathbf{E}\,T(j)) P_{ij}(\tau_k) \\
&= \tau_k + \sum_{j=0}^{\infty} \mathbf{E}\,T(j) P_{ij}(\tau_k). \qquad (7.5)
\end{aligned}$$

Together with (4.1), the definition of $T([N(\tau_k)|N(0)=i])$ leads to

$$O_{M/M/1}(i) = \frac{1}{\tau}\sum_{k=2}^{m}(T([N(\tau_k)|N(0)=i]) - \tau_k)$$

for all $i = 0, 1, \ldots$ and finally with (7.5) to

$$\begin{aligned}\mathbf{E}\,O_{M/M/1}(i) &= \frac{1}{\tau}\sum_{k=2}^{m}(\mathbf{E}\,T([N(\tau_k)|N(0)=i]) - \tau_k)\\ &= \frac{1}{\tau}\sum_{k=2}^{m}\sum_{j=0}^{\infty}\mathbf{E}\,T(j)\,P_{ij}(\tau_k)\,.\end{aligned}$$

Note that $\mathbf{E}\,O_{M/M/1}(i)$ does not depend on the time interval boundaries τ_k with the exception of the transition probabilities $P_{ij}(\tau_k)$.

If we assume that τ_2 is sufficiently large to allow for $P_{ij}(\tau_2) \approx \pi_j$ then this relation holds also for every τ_k $(k = 3, \ldots, m)$. With (7.4) we get for the expectation of the computational overhead in steady state $O^{(s)}_{M/M/1}$

$$\begin{aligned}\mathbf{E}\,O^{(s)}_{M/M/1} &= \frac{1}{\tau}\sum_{k=2}^{m}\left(\mathbf{E}\,T^{(s)}_k - \tau_k\right)\\ &= \frac{1}{\tau}\sum_{k=2}^{m}\left(\tau_k + \frac{\lambda}{(\mu-\lambda)^2} - \tau_k\right)\\ &= \frac{m-1}{\tau}\frac{\lambda}{(\mu-\lambda)^2}\,. \end{aligned} \tag{7.6}$$

The most important observation regarding the expected overhead is a strong dependency on the distance between the arrival rate λ and the service rate μ. The expected overhead tends to grow quadratically with a decreasing distance. Furthermore, increasing λ and μ while keeping a fixed distance between both of the parameters also increases the expected overhead. Finally, as could have been guessed, the overhead tends to grow linearly in the number m of time intervals. These observations can be used to evaluate the intended application of the parallel simulation approach to a given queuing system model.

Note that in order to provide a confidence interval for the overhead, the calculation of the variance of O is necessary. Unfortunately, this calculation is a complex task which cannot be solved easily.

7.4. Approximate State Matching

As stated in Theorem 7.2.1, state match detection in time-parallel queuing simulation can be performed efficiently. Nevertheless, computational overhead is introduced with fix-up computations. A measure for the computational overhead in time-parallel queuing simulation is given in (7.6). Depending on the parameters λ and μ, the computational overhead due to fix-up computations might grow to an unacceptable level, inhibiting a reasonable speedup of the parallel simulation. In these cases, the technique of approximate state matching introduced in Chapter 5 can be applied to reduce the amount of fix-up computations.

This is done using a simple definition of the state deviation measure δ that only consists of the current number of jobs in the system. Thus, with a threshold Δ, a fix-up computation can be stopped if the number of jobs in the system reaches Δ. It is suspected, that the error in the simulation results is closely related to this definition of the deviation measure for most result statistics being calculated in a queuing system simulation.

7.5. Progressive Queuing Simulation

Rather than elaborating on the characteristics of approximate state matching in time-parallel queuing simulation discussed in the previous section, this thesis continues with a more thorough examination of the application of the progressive simulation technique introduced in Chapter 6 to queuing system simulation.

7.5.1. Theory

Time-parallel queuing simulation, as discussed in Section 7.2, is extended to provide simulation results at any wallclock time instant. To be able to do this, it is necessary to relate a wallclock time instant with the simulation progress of parallel processes. This is done in the following by introducing a family of simulation progress functions.

Definition 7.5.1 (progress function)**.** Let $\tau_0(t) := 0$ and $\tau_{m+1}(t) := n$ for all $t > 0$. For all $k \in \{1, \ldots, m\}$ let the *simulation progress function* $\tau_k : \mathbb{R}_0^+ \to$

N_k with $\tau_k(0) = j_k$ be a monotonic increasing function being complete (i.e. taking all values in N_k) and converging to n, i.e. $\lim_{t\to\infty} \tau_k(t) = n$. Further it is required that $\forall k \in \{1,\ldots,m\} : \tau_{k-1}(t) \leq \tau_k(t)$.

The simulation progress function τ_k returns the next job which is to be processed by p_k after wallclock time t. There are three requirements on the progress functions: (i) No jobs are skipped during the simulation, i.e. every value in N_k is taken by τ_k. (ii) Simulations realize progress, eventually calculating the departure time of the last job, which is expressed by the convergence to n. (iii) No passing of processes is allowed, which is expressed by the last sentence in Definition 7.5.1.

In Section 6.3, the result functions of a progressive time-parallel simulation were defined with a domain starting at a time t_0, after which estimations on the correct simulation results can be provided. With time-parallel queuing simulation defined above, it is possible to return result estimators even before every job has been processed once. Thus, the following definitions of the result functions have a domain starting at time 0. Initial simulation results are refined progressively with growing accuracy until fix-up computations are finished and correct results are known. In queuing system simulation, the result and accuracy functions depend on the simulation goals, i.e. the result statistics to evaluate. In the following, it is supposed that the average job system time (i.e. the average time a job spends in the system) is to be determined. Many of the other statistics of interest in queuing simulation can be derived from this measure. Furthermore, in order to simplify the following definitions, only the calculation of the total system time (i.e. the total time all jobs spend in the system) is considered.

Definition 7.5.2 (result functions)**.** Let $\tau'_k(t) := \max(\tau_k(0), \tau_{k-1}(t))$. The *local cumulated system time* function $R_k : \mathbb{R}_0^+ \to \mathbb{R}_0^+$ of process k, the *global cumulated system time* function $\widehat{R} : \mathbb{R}_0^+ \to \mathbb{R}_0^+$, and the *total system time* of jobs $R \in \mathbb{R}_0^+$ are defined as

$$\begin{aligned} R_k(t) &:= \sum_{j=\tau'_k(t)}^{\tau_k(t)-1} d_{u_k}^{j_k}(j) - a(j)\,, \\ \widehat{R}(t) &:= \sum_{k=1}^{m} R_k(t)\,, \text{ and} \\ R &:= \lim_{t\to\infty} R_1(t) = \sum_{j=1}^{n} d_{u_1}^{j_1}(j) - a(j)\,. \end{aligned}$$

$\widehat{R}(t)$ is an estimation, available at wallclock time t, of the total system time R, which is defined as the cumulated system time which would be calculated by the first process. Some further remarks on Definition 7.5.2:

- After time t_0, as defined in Definition 6.3.1, $\tau'_k(t)$ denotes the next job that is to be processed by p_{k-1} at time t, which is identical to the first job in the results interval of p_k at time t. To be able to calculate results before t_0, the result function is adapted not to consider jobs that have not been processed yet. This is achieved by choosing the maximum of $\tau_k(0)$ and $\tau_{k-1}(t)$ for the sum in the definition of $R_k(t)$.
- A sequential queuing simulation calculates departure times of all jobs $j \in \{1,\ldots,n\}$, which is exactly what is done by p_1 if its fix-up computations extend up to the last job n. Hence, simulation results calculated by the first process p_1 are assumed to be the correct results (cf. the related discussion in Section 6.3).

As mentioned above, the objective of the simulation is supposed to be the determination of the *total system time* R of all jobs. To distinguish between estimated result and correct result, the estimation on the total system time of jobs $\widehat{R}(t)$ calculated by processes at wallclock time t is called the *global cumulated system time*. The global cumulated system time is composed of the *local cumulated system times* calculated by the parallel processes, where exactly one of the processes is responsible for the calculation of the system time of every job.

The following lemma establishes some properties of the global cumulated system time, which are used later in this section.

Lemma 7.5.1. *In the progressive queuing system simulation defined above, it holds that* $\widehat{R}(0) = 0$*,* $\lim_{t\to\infty}\widehat{R}(t) = R$*, and* $0 \le \widehat{R}(t) \le R$ *for all* $t \ge 0$.

Proof. Due to Definitions 7.2.1 and 7.5.1, $\tau_k(0) \geq \tau_{k-1}(0)$. Hence,

$$\widehat{R}(0) = \sum_{k=1}^{m} \sum_{j=\tau_k(0)}^{\tau_k(0)-1} (d_{u_k}^{j_k}(j) - a(j)) = 0\,.$$

By definition, $\lim_{t\to\infty} \tau_k(t) = n$, whereby follows for all $k \in \{2,\ldots,m\}$,

$$\lim_{t\to\infty} \tau_k'(t) = \max(\tau_k(0), \lim_{t\to\infty} \tau_{k-1}(t)) = \max(\tau_k(0), n) = n$$

and thus $\lim_{t\to\infty} R_k(t) = 0$. This reduces $\widehat{R}(t)$ to $R_1(t)$ as t tends to infinity. Hence, $\lim_{t\to\infty} \widehat{R}(t) = R$.

Due to Definition 7.1.1, it holds that $d_{u_k}^{j_k}(j) \geq a(j)$ for all $k \in \{1,\ldots,m\}$ and all $j \in \{j_k,\ldots,n\}$. Therefore, for $t \geq 0$, $R_k(t) \geq 0$ and furthermore $\widehat{R}(t) \geq 0$.

For the proof of $\widehat{R}(t) \leq R$ for $t \geq 0$, first note that for all $k \in \{1,\ldots,m\}$,

$$R_k(t) \leq \sum_{j=\tau_k(t)}^{\tau_k(t)-1} (d_{u_k}^{j_k}(j) - a(j)) =: R_k'(t)\,,$$

as $R_k'(t)$ is basically identical to $R_k(t)$, except that if $\tau_{k-1}(t) < \tau_k(0)$, then all elements between $\tau_{k-1}(t)$ and $\tau_k(0)$ are skipped in the sum of $R_k(t)$. Hence, it remains to be shown that

$$\widehat{R}'(t) := \sum_{k=1}^{m} R_k'(t) \leq R\,.$$

Due to Lemma 7.2.2, $d_{u_1}^{j_1}(j) \geq d_{u_k}^{j_k}(j)$ for all $k \in \{1,\ldots,m\}$ and all $j \in N_k$. Thus, for $t \geq 0$,

$$R_k'(t) \leq \sum_{j=\tau_{k-1}(t)}^{\tau_k(t)-1} (d_{u_1}^{j_1}(j) - a(j))$$

and furthermore

$$R = \sum_{k=1}^{m} \sum_{j=\tau_{k-1}(t)}^{\tau_k(t)-1} (d_{u_1}^{j_1}(j) - a(j))$$

leading to $\widehat{R}'(t) \leq R$ and thus $\widehat{R}(t) \leq R$. □

The *accuracy* of the simulation system at a time t is given by the accuracy function, which consists of the ratio between cumulated system time and total system time.

Definition 7.5.3 (accuracy function)**.** In the progressive queuing system simulation defined above, the *accuracy function* $\alpha : \mathbb{R}_0^+ \to [0,1]$ is defined as

$$\alpha(t) := \frac{\widehat{R}(t)}{R}\,.$$

The accuracy function is a theoretical construct and can, in general, not be determined during the runtime of the simulation. However, it can be used to evaluate the behaviour of a progressive time-parallel simulation application, which is done in the following theorem for queuing system simulation.

Theorem 7.5.1. *In the progressive queuing system simulation defined above,* $\alpha(0) = 0$, $\lim_{t\to\infty} \alpha(t) = 1$, *and* α *is monotonically increasing.*

Proof. $\alpha(0) = 0$ and $\lim_{t\to\infty} \alpha(t) = 1$ follow directly from Lemma 7.5.1. As R is constant over time, it remains to be shown, that $\widehat{R}(t)$ is monotonically increasing.

For an arbitrary $t \geq 0$, choose $t' > t$, such that for all $k \in \{1,\ldots,m\}$

(i) $\forall t'' > 0$ with $t < t'' < t'$: $\tau_k(t'') = \tau_k(t)$ and

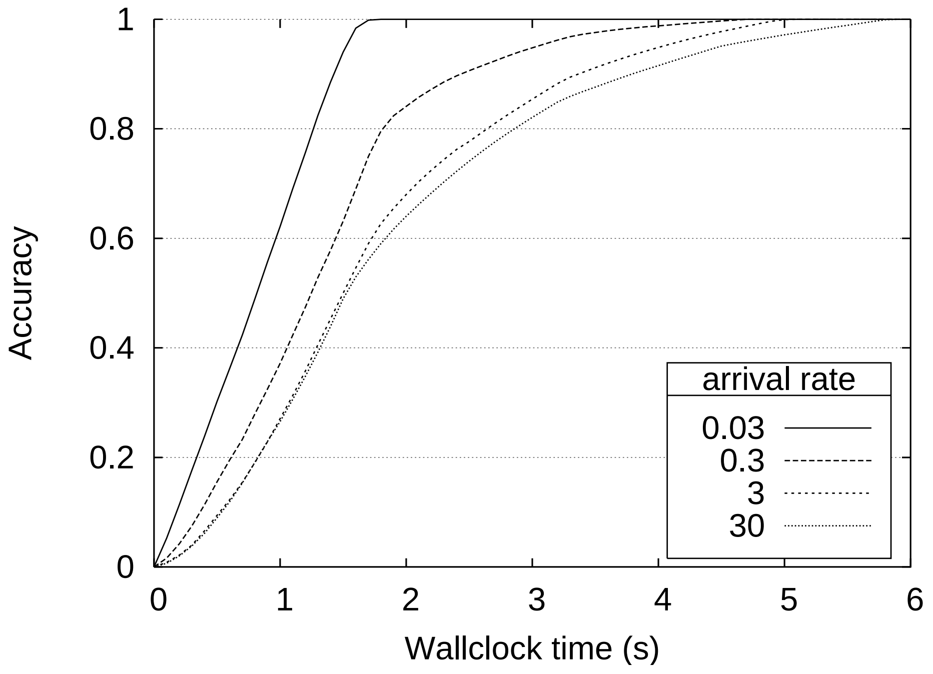
(ii) $\tau_k(t') \leq \tau_k(t) + 1$ and there exists at least one $k \in \{1,\ldots,m\}$ with $\tau_k(t') = \tau_k(t) + 1$.

Such a time t' can be found due to the monotonicity and completeness of τ_k (see Definition 7.5.1). Let $G := \{k \in \{1,\ldots,m\} | \tau_k(t') = \tau_k(t) + 1\}$, $G_1 := \{k \in G | \tau_k(t') \leq \tau_{k+1}(0)\}$, and $G_2 := \{k \in G | \tau_k(t') > \tau_{k+1}(0)\}$. Then for all $t'' > 0$ with $t < t'' < t'$, $\widehat{R}(t) = \widehat{R}(t'')$. Furthermore, with $\tau'_k := \tau_k(t') - 1$,

$$\begin{aligned}\widehat{R}(t) = \widehat{R}(t') \quad &+ \sum_{k\in G_1} (d_{u_k}^{j_k}(\tau'_k) - a(\tau'_k)) \\ &+ \sum_{k\in G_2} ((d_{u_k}^{j_k}(\tau'_k) - a(\tau'_k)) - (d_{u_{k+1}}^{j_{k+1}}(\tau'_k) - a(\tau'_k)))\,.\end{aligned}$$

Due to Definition 7.1.1, $d_{u_k}^{j_k}(\tau'_k) - a(\tau'_k) \geq 0$ for all $k \in \{1,\ldots,m\}$ and due to Lemma 7.2.2, $d_{u_k}^{j_k}(\tau'_k) \geq d_{u_{k+1}}^{j_{k+1}}(\tau'_k)$ for all $k \in \{1,\ldots,m\}$. Hence, $\widehat{R}(t') \geq \widehat{R}(t)$, and $\widehat{R}$ is monotonically increasing. $\square$

Figure 7.1. Accuracy function α with $\lambda - \mu = 0.001$ and varying λ utilizing 4 processes

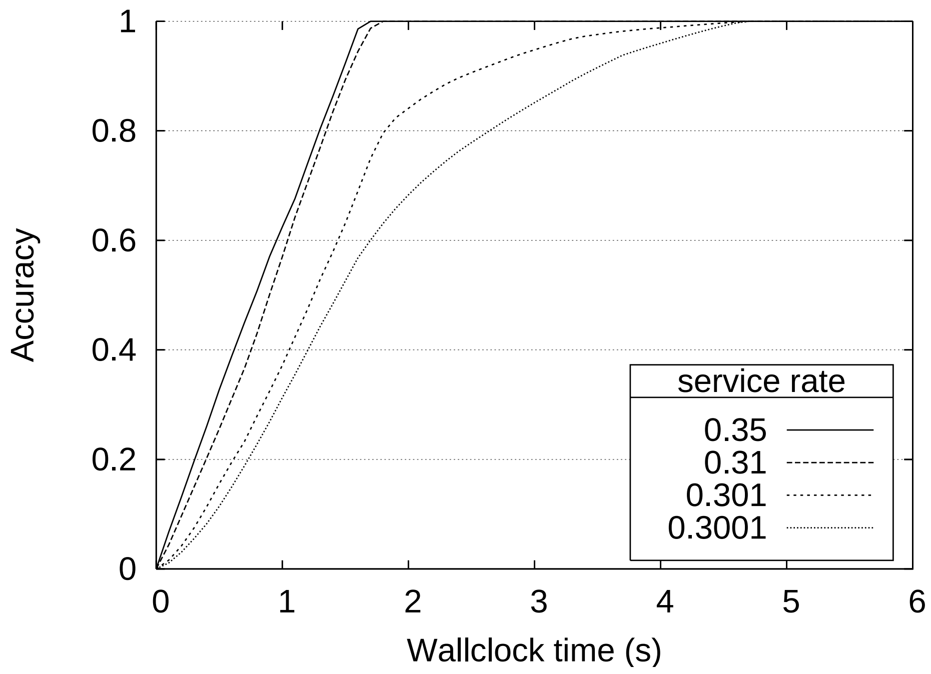


7.5.2. Experiments

To investigate the development of the accuracy of progressive queuing system simulation over time, a prototypical progressive queuing simulator has been implemented and a number of experiments have been performed on an SGI Origin multiprocessor machine. As a maximum of eight processors was available for the experiments, the number of processes was varied between one and eight. According to Section 7.3.2, it is to be expected that the accuracy of a simulation execution tends to grow slower with an increasing arrival rate λ, as well as with an increasing distance $\mu - \lambda$. Therefore, experiments were conducted with varying parameters λ and μ in order to confirm that observation. The results of the experiments are summarized in the following. Note that the values shown here are averages over 50 repetitions of the same experiment with different random numbers. All of the experiments consisted of the calculation of cumulated system times for 200000 jobs.

Figure 7.1 summarizes the results of experiments with a varying arrival rate λ while keeping a fixed distance $\mu - \lambda = 0.001$ and 4 processes. The

Figure 7.2. Accuracy function α with $\lambda = 0.3$ and varying μ utilizing 4 processes

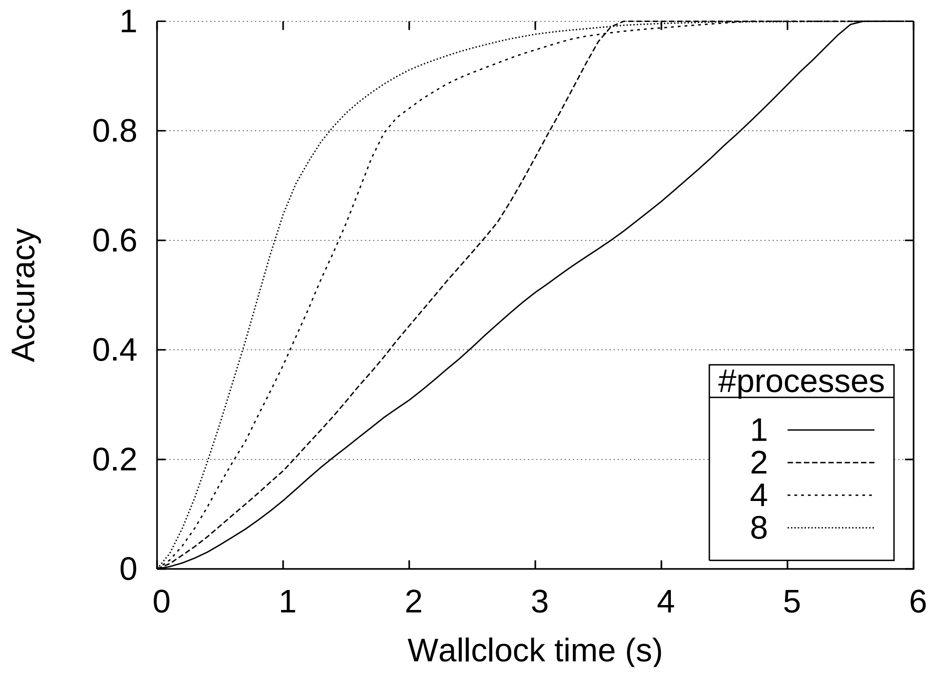


accuracy function α is plotted against wallclock time measured in seconds. As expected, accuracy tends to grow slower with an increasing value of λ. Note also, that the accuracy grows faster in the beginning, with the growth slowing down afterwards. This is a desirable property with progressive simulation, as a reasonable accuracy can be achieved after a short runtime, although the calculation of the exact result might take much longer.

Figure 7.2 shows the results of experiments with a fixed arrival rate $\lambda = 0.3$ and varying service rate μ using 4 processes. Again, the expectation of a slower growth of the accuracy with increased distance $\mu - \lambda$ is met. Note that growth is nearly identical with the larger values of $\mu = 0.35$ and $\mu = 0.31$. However, as the distance between λ and μ decreases to 0.001 and further to 0.0001, the growth of the accuracy decreases significantly.

Figure 7.3 shows the results of experiments with fixed arrival rate $\lambda = 0.3$ and fixed service rate $\mu = 0.301$ with 1, 2, 4, and 8 parallel processes. As expected, the accuracy tends to grow faster with a higher number of processes, although the increase in growth lags behind the expectation. The reason for this is the significant overhead due to the computation of simulation results,

Figure 7.3. Accuracy function α with $\lambda = 0.3$, $\mu = 0.301$, utilizing a varying number of processes



which could be ameliorated by more infrequent result computations.

7.6. Summary

The application of approximate time-parallel simulation techniques on existing simulation models is illustrated in this chapter by the example of queuing systems. The simple nature of basic queuing systems allows for a concise mathematical representation of model dynamics if the stochastic properties of queuing systems are removed by presampling of random numbers. In time-parallel simulation of single-server queues, state matching can be detected efficiently without state comparisons between adjacent processes (see Theorem 7.2.1). However, depending on the arrival and service time parameters of the model, there might be a long time until occurrence of state matching. Approximate state matching as well as progressive time-parallel simulation can be used to shorten answer times of a simulation execution. With progressive simulation, after providing estimations of the correct results early on, results are subsequently refined by the use of fix-up computations. Result and accuracy functions of a progressive time-parallel queuing simulation have been defined and it has been shown that this is indeed a feasible application of progressive techniques, as the accuracy function, being normalized to the interval $[0,1]$, is monotonically increasing and converges to one with increasing simulation runtime (see Theorem 7.5.1). Furthermore, experiments have been performed to determine the rate of increase of the accuracy function. As expected, the growth of the accuracy function depends on the arrival rate λ, the distance $\mu - \lambda$, and the number of parallel processes.

8. Cache Simulation

Simulation of computer caches, with one of the most important replacement policies being *least recently used* (LRU), has been topic of research for several decades. This lead to the development of efficient LRU simulation algorithms, where hit or miss rates are determined for given memory address traces. Different schemes for the parallelization of these algorithms exist for Single Instruction Multiple Data (SIMD) [75] and Multiple Instruction Multiple Data (MIMD) machines [40] (for an explanation of the categorizations of parallel computers into MIMD and SIMD machines, the reader is referred to one of the various textbooks on computer architecture, e.g. [20, 83]). On MIMD machines, the method of temporal parallelization is applied, by splitting a memory address trace into several subtraces that are used as input for parallel cache simulations. Empirical studies [73] show, that the SIMD algorithms exhibit the best results for small cache sizes, but that the MIMD algorithms are better in overall.

In the time-parallel approach, the cache contents at the subtrace boundaries are not known a priori, wherefore they have to be guessed initially and corrected by the use of fix-up computations after a first simulation phase. Depending on the specificities of the input trace and the size of the cache, these fix-up computations can lead to an increase of the parallel simulation runtime to that of the corresponding sequential simulation, or even worse. Experiments with this approach [73] indicate that the speedup for a small number of processes is excellent, but degenerates with a higher number. Approximation can be applied here to decrease the cost of the fix-up-computation phase or to avoid it completely.

The chapter starts with an introduction to the basic properties of LRU cache simulation in Section 8.1, before presenting parallel simulation approaches in Section 8.2 and Section 8.3. The approximate cache simulation algorithms are discussed in Section 8.4 and Section 8.5. Results of experiments with the approximate LRU cache simulation algorithms are presented in Section 8.6.

Figure 8.1. LRU stack processing in Example 8.1.1

Time	1	2	3	4	5	6	7	8	9	10	11	12
Req	a	b	c	c	d	b	b	a	b	e	c	a
Dist	∞	∞	∞	1	∞	3	1	4	2	∞	5	4
Stack	a	b	c	c	d	b	b	a	b	e	c	a
		a	b	b	c	d	d	b	a	b	e	c
			a	a	b	c	c	d	d	a	b	e
					a	a	a	c	c	d	a	b
										c	d	d

8.1. Least-recently-used caching

Basic simulation of caching with the LRU policy is straightforward. A data structure for the cache is initialized for a given cache size and an input trace is processed, updating the LRU cache appropriately and recording the number of hits.

This approach has the disadvantage, that for every cache size a new simulation execution is necessary. Therefore, an approach was introduced to calculate the hit rates for any number of cache sizes in a single pass over the input trace [63]: simulation is performed as usual, with the exception that the cache size is supposed to be unbounded (i.e. no replacement occurs). For every request, the position of the corresponding page in the LRU stack is recorded as the *stack distance* of the request, which is the minimal size of a cache, such that the request can be served from it. A *distance table* is used to record stack distances. After processing the input trace, the entries of the table can be cumulated, producing the *success function* of the simulation, which is defined as $S(c) = \sum_{i=1}^{c} D_i$, where c is the cache capacity for which to calculate the number of hits and D_i is the number of occurrences of stack distance i in the simulation of the input trace.

Example 8.1.1. The determination of stack distances for a simple input trace is illustrated in Figure 8.1. It shows the processing of the trace of page requests where in every time step the stack distance of a request is determined from the current stack and the stack is changed afterwards (either by mov-

Figure 8.2. Stack distances after processing in Example 8.1.1

Dist	1	2	3	4	5	∞
Count	2	1	1	2	1	5

ing the requested page to the top or by pushing it, if it did not yet appear). Note, that in Figure 8.1, the LRU stack is shown as it appears after processing the corresponding request. The stack distance is either the position of the requested page in the LRU stack, or ∞ if the page is not yet present. Figure 8.2 shows the distance table for the example input trace at the end of processing, which is used to determine numbers of hits for specific cache sizes. E.g. four hits are scored for a cache size of three, which is the sum of the entries of distance three and lower.

Example 8.1.1 leads to two important observations: (i) the determination of stack distances is a generalization of simple LRU cache simulation, as the number of hits for a given input trace is directly related to the stack distances of requests in the trace, and (ii) the stack distance of a request to a specific page is the number of unique requests between the current request and the previous occurrence of a request to the same page plus one. In Example 8.1.1, the stack distance of the request to page b at time 6 is 3, which is the number of unique requests (c and d) between time 6 and time 2 (the last occurrence of a request to page b) plus one.

Various ways of implementing the LRU stack have been proposed: linked lists [63], hash tables [7], and search trees [81]. An overview of the implementations together with discussions about their strengths and weaknesses can be found in [96].

8.2. Simple parallel cache simulation

Let $T = (t_1, \ldots, t_n)$ be the input trace which is a sequence of n requests. For parallel simulation, T is split into m non-overlapping subtraces $T_1, \ldots, T_m$. The subtraces do not necessarily have the same length, although this might be preferable for a balanced load distribution among processes.

In the basic time-parallel simulation approach, every subtrace is assigned to a separate process for simulation, which is performed in several phases:

in the *initial simulation* phase, the input subtraces are processed once with an empty initial cache, yielding an incorrect value of the overall number of hits; in the *fix-up computation* phases, resimulation of parts of the subtraces is performed until the correct value for the number of hits is determined. For every pass over a subtrace, the cache contents that have been determined by the simulation of the directly preceding subtrace in the previous pass are used as initial cache. The length of this phase can range from a partial pass over the subtrace to $m-1$ passes in the worst case.

Note, that it is sufficient to include those requests in the resimulation subtrace that could not be served from the cache and only up to the time where the cache is filled for the first time. Although this leads to an incorrect order of pages in the LRU stack during resimulation, the number of hits are calculated correctly.

Example 8.2.1. Let $T = (a,b,c,c,d,b,b,a,b,e,c,a)$ be a sequence of page requests, which is used as the input trace of a cache simulation for a cache size of 3. The overall trace T is split into two subtraces $T_1 = (a,b,c,c,d,b)$ and $T_2 = (b,a,b,e,c,a)$ for parallel simulation on two processes. As T_1 is the first subtrace, resimulation is not necessary. Two hits are recorded and the cache contents after processing are b,d,c in order of most recently to least recently used. At the same time, simulation of T_2 is performed with an (incorrect) empty initial cache. Therefore, all misses occurring before the cache has been filled for the first time have to be reconsidered to determine correct simulation results. As the cache of size 3 is filled after the fourth request in T_2, the subtrace b,a,e has to be resimulated with cache contents b,d,c.

In the original time-parallel simulation approach, resimulation is performed by the same process that created the resimulation subtrace with the final cache resulting from the simulation of the previous subtrace. Therefore, resimulation can occur only after a simulation phase has been completed for both subtraces. This is implemented by performing simulation in strict phases, using barrier synchronization of all processes between phases. However, synchronization among processes can be relaxed when processing is changed slightly.

Consider Example 8.2.1, where the responsibility for processing the resimulation subtrace of T_2 can be pushed to the process that is responsible for the simulation of T_1. No transfer of cache contents is necessary here. The process just continues simulation as if the resimulation subtrace is part of its initial subtrace T_1.

If subtraces are fed to processes through input queues, the fact that resimulation is performed is transparent to the processes and synchronization can be minimized: every process processes page requests from its input queue until the special symbol $\perp$, signifying end of the input trace, is read. Instead of recording page requests to be resimulated in a separate data structure, they are put directly into the input queue of the preceding process. Synchronization is performed implicitly by the blocking of processes on an empty input queue. A process knows that it has finished simulation if it encounters the $\perp$ symbol, passing it on to the input queue of the previous process. In this case, the last process never performs any resimulation steps, stopping execution as soon as its initial subtrace has been completely processed. This can easily be achieved by putting $\perp$ at the end of the input queue of the last process.

This approach is summarized in Algorithm 8.1. Parallelism is introduced by use of the construct

$$\textbf{for } l \leq i \leq u \textbf{ pardo } \mathit{stmt}$$

adopted from [46], which indicates concurrent execution of statement *stmt* for every $i \in \{l, \ldots, u\}$. Several functions are used for convenience in the algorithm. ENQUEUE and DEQUEUE execute the corresponding operations on the input queue of a process, PUSH places a request at the top of an LRU stack, REPOSITION moves a request from anywhere in a given stack to the top, and REPLACE removes the bottom request in a stack and puts a new request on top.

8.3. Parallel full LRU stack simulation

Although mentioned in [40] that it is possible to calculate stack distances with the basic time-parallel simulation approach, details were only provided at a later time [73].

With the simple time-parallel cache simulation structured as in Algorithm 8.1, it can easily be extended to calculate the success function instead of the number of page hits with three basic changes: every page that has been requested at least once is present in the LRU stack used to determine stack distances (pages are never removed); instead of the absolute num-

Algorithm 8.1 Simple time-parallel cache simulation

Input: *Subtraces $T_1, \dots, T_m$ and cache size s_{max}. The stack S_i for every subtrace T_i is initially empty. The input queue Q_i for every process is initialized with the input subtrace T_i. Additionally, $\perp$ is put at the end of Q_m. The hit counters h_i are initialized to 0.*

Output: *The sum of the number of hits h_i of every process i.*

```
begin
    for $1 \leq i \leq m$ pardo
        $req = \text{DEQUEUE}(Q_i)$
        while $req \neq \perp$ do
            if $req \in S_i$ then
                $\text{REPOSITION}(S_i, req)$
                $h_i = h_i + 1$
            elsif $|S_i| < s_{max}$ then
                $\text{PUSH}(S_i, req)$
                if $i > 1$ then $\text{ENQUEUE}(Q_{i-1}, req)$
            else
                $\text{REPLACE}(S_i, req)$
            $req = \text{DEQUEUE}(Q_i)$
        if $i > 1$ then $\text{ENQUEUE}(Q_{i-1}, \perp)$
end
```

ber of page hits, stack distances have to be recorded; and for all subtraces, except of the first, stack distances cannot be correctly determined if the corresponding request is not in the local LRU stack, wherefore the processing of these requests has to be delegated to the directly preceding process.

To see why this is correct, observation (ii) from Section 8.1, relating the stack distance to the number of different occurrences of requests between the current request and its previous occurrence, must be considered. First, the observation implies that in the parallel processing approach, all stack distances whose corresponding requests are found in the LRU stack can be determined correctly. This is due to the fact that the previous occurrence of the current request, as well as all of the different requests in between, have been processed. Second, all of the first occurrences of requests to the same page cannot be determined by the current process (except of the first), wherefore they are sent to the preceding process in correct order.

Intermediate requests with already determined distances can be skipped, as for the calculation of stack distances only the first occurrence of a request is relevant.

Example 8.3.1. Consider the sequence of page requests of Example 8.1.1, which again is split into two subsequences $T_1 = (a,b,c,c,d,b)$ and $T_2 = (b,a,b,e,c,a)$ to be fed to two processes, P_1 and P_2, for the determination of stack distances. Recall the correct sequence of stack distances for T, which is $(\infty,\infty,\infty,1,\infty,3,1,4,2,\infty,5,4)$. The distances for T_1 are correctly calculated by P_1. In the simulation of T_2, the distances for requests 3 and 6 are determined to 2 and 4. The rest of the requests is sent to P_1 for resimulation, as P_2 cannot determine the correct distances.

Algorithm 8.2 shows the parallel processing to determine the stack distances of an input trace. In addition to the functions introduced with Algorithm 8.1, here the function POSITION is used, which returns the position of a request in the given LRU stack.

8.4. Approximate Cache Simulation

As introduced above, there are two approaches for the simulation of LRU caches. The simple approach determines the hit rate for a given trace and cache size. The more general full-stack approach calculates the stack distances of the requests in the input trace, which can be used to implement a success function that returns the number of hits for a given cache size. Here, the algorithms presented above are changed to calculate intervals for the number of hits rather than exact values. Depending on simulation needs, these intervals can be used to get approximate results in a much shorter execution time. The following two sections focus on two different aspects of approximate calculation: how approximate simulation results can be determined, and which criteria can be used to decide on the appropriate time to finish simulation execution.

Few effort is required to change Algorithm 8.1 and Algorithm 8.2 to give approximate simulation results during any time of resimulation. However, the calculations presented here only work correctly when execution has been stopped, either permanently (i.e. the simulation is finished), or temporarily for the calculation of the current simulation accuracy.

Algorithm 8.2 Full LRU stack simulation

Input: *Subtraces $T_1, \dots, T_m$. The stack S_i for every subtrace T_i is initially empty. The input queue Q_i of every process is initialized with the input subtrace T_i. Additionally, $\perp$ is put at the end of Q_m. The distance tables D_i are initialized with 0 for every possible stack distance.*

Output: *The overall distance table $D_{overall}$, which is the sum of the distance tables D_i for all processes i.*

```
begin
    for $1 \le i \le m$ pardo
        $req = \text{DEQUEUE}(Q_i)$
        while $req \ne \perp$ do
            if $req \in S_i$ then
                $dist = \text{POSITION}(S_i, req)$
                $D_i[dist] = D_i[dist] + 1$
                $\text{REPOSITION}(S_i, req)$
            else
                $\text{PUSH}(S_i, req)$
                if $i > 1$ then $\text{ENQUEUE}(Q_{i-1}, req)$
            $req = \text{DEQUEUE}(Q_i)$
        if $i > 1$ then $\text{ENQUEUE}(Q_{i-1}, \perp)$
end
```

8.4.1. Simple cache simulation

It was already mentioned in [40], that upper and lower bounds on the number of hits are calculated easily at the end of a simulation iteration, where simulation can be stopped if bounds are tight enough. With the parallel simulation algorithm of the previous section, approximate results can be calculated at any time, leading to a higher flexibility for the decision on simulation termination.

In Algorithm 8.1, upper and lower bounds for the number of hits can be calculated without further mechanisms, if all processes stop execution at least until the calculation of bounds is complete. The minimal number of hits at that time $h_{min} = \sum_{i=1}^{m} h_i$ (where m is the number of processes) is just the sum of the hits recorded by all processes so far. Additionally, every request still in the input queue of a process might be a hit, although this is not known yet. Therefore, the maximal number of hits $h_{max} = h_{min} + \sum_{i=1}^{m} |Q_i|$ is derived from the number of all requests for which the status (hit or miss) could not yet be determined.

Example 8.4.1. The subtraces $T_1 = (a,b,c,c,d,b)$ and $T_2 = (b,a,b,e,c,a)$ of Example 8.2.1 are simulated by two processes P_1 and P_2. After the requests of both traces have been processed exactly once, a lower bound on the number of cache hits of 4 can be calculated, which are two hits recorded by P_1 for requests 4 and 6 of T_1 and two hits recorded by P_2 for requests 3 and 6 of T_2. As the status of requests 1,2,4, and 5 of T_2 could not be determined yet, they are put into the input queue of P_1 for resimulation, which now has a length of 4, leading to an upper bound on the number of hits of 8.

8.4.2. Full LRU stack simulation

To devise a parallel algorithm for the full stack LRU simulation providing approximate results at any time during simulation execution, the success function is modified to return an interval of the number of hits for a given cache size at any time, instead of the exact value, available only at the end of simulation. Result intervals are specified by their lower and upper bounds which have to be derived from the simulation state. For the identification of lower bounds, no additional mechanism is needed, as they can be directly calculated from the finite stack distances determined during the simulation execution (called *final stack distances* hereafter).

For the determination of upper bounds, all requests which might be hits for a given cache size must be considered, or equivalently, all requests not determined to be sure misses. As a first approximation, in addition to the sure hits, all requests with infinite stack distances (i.e. requests where the correct distances are yet unknown) might be hits for any cache size (except for the first process, where all stack distances are correctly determined, including infinite ones). However, a more precise determination of upper bounds of result intervals is possible using additional knowledge gained during simulation execution. Let r be a request to a page whose stack distance cannot be determined by a process p. Then, the correct stack distance d of r must be at least the *preliminary stack distance* $\tilde{d}$, which is the current length of p's LRU stack after pushing r. Recall that a request r with a stack

Figure 8.3. Tables in Example 8.4.2 after first iteration

Dist	1	2	3	4
Count	0	1	0	0

(a) Lower-bound distances

Dist	1	2	3	4
Count	3	3	2	0

(b) Upper-bound distances

Figure 8.4. Tables in Example 8.4.2 after second iteration

Dist	1	2	3	4
Count	2	1	1	1

(a) Lower-bound distances

Dist	1	2	3	4
Count	0	0	1	2

(b) Upper-bound distances

distance of d is a miss for a given cache size s if $d > s$. As just noted, for any preliminary stack distance $\tilde{d}$ of r, $d \geq \tilde{d}$ holds. Thus, if $\tilde{d} > s$, also $d > s$ and r must be a miss for cache size s. If preliminary distances are collected in a distance table in the same way as final distances, the same lookup in the cumulated table can be used to determine the number of possible (but not surely determined) hits for a given cache size, and hence the upper bound on the number of hits. This is due to the fact that only requests with preliminary distances lesser than or equal to the cache size might be hits, all others are known to be misses.

Stack distances are recorded in two different tables, final distances in a *lower-bound distance table* and preliminary distances in an *upper-bound distance table*. The enhanced algorithm processes requests similar to Algorithm 8.2. If the stack distance of a request can be calculated exactly by the process, it is recorded in the lower-bound distance table. Otherwise, the preliminary stack distance is determined as the number of elements in the LRU stack after pushing the current request, this distance is recorded in the upper-bound distance table, and the request is sent to the input queue of the preceding process for resimulation.

Example 8.4.2. Consider the subtraces $T_1 = (a,b,c)$, $T_2 = (c,d,b)$, $T_3 = (b,a,b)$, and $T_4 = (e,c,a)$ which were created by dividing the input trace of Example 8.1.1 into four subtraces for parallel simulation. After all requests have been processed once, only the stack distance for the second b in T_3 could be determined exactly to 2, as well as the distances of ∞ for all requests in T_1. For the rest of the requests, preliminary stack distances can be calculated as explained above. Figure 8.3(a) and Figure 8.3(b) show the lower-bound and upper-bound distance tables after this first iteration. These tables can be cumulated to give the upper and lower bounds of the number of hits that are returned by a call to the approximate success function. E.g. for a cache size of 2, the exact number of hits must be contained in the interval $[1,6]$, for a cache size of 4 in the interval $[1,8]$.

Further processing decreases the size of the returned intervals (increasing the accuracy of results). Figure 8.4(a) and Figure 8.4(b) show lower-bound and upper-bound distance tables after all requests to resimulate have been processed twice. Here, for a cache size of 2, the number of hits is known to be exactly 3, whereas for a cache size of 4, the number of hits must be contained in the interval $[5,8]$.

The preliminary stack distance of a request might be recorded multiple times by different processes without further provisions. Therefore, before updating its upper-bound distance table, a process must make sure that the record of the corresponding previously determined preliminary distance is removed. As the determined stack distances are reflected by the values of simple counters, this can be achieved by decrementing the value for the old preliminary distance in the table, ensuring consistent global values for the preliminary distances. For requests not processed before, no change of the upper-bound distance table is required. To implement this approach, requests that are kept in an input queue for processing must be annotated with their preliminary stack distance. Yet unprocessed requests are not annotated (or annotated with 0).

The overall approximate full LRU stack simulation approach is shown in Algorithm 8.3. The annotation of requests is realized here by using request/distance pairs as queue entries. Therefore, the ENQUEUE function takes a pair as second argument and the DEQUEUE function returns a pair. As termination of the algorithm is discussed in the next section, the termination of the while loop is not explicitly specified here. It depends on the return value of the opaque TERMINATE function, which might also need some parameters not shown for reasons of conciseness.

Algorithm 8.3 Approximate full LRU stack simulation

Input: *Subtraces $T_1, \ldots, T_m$. The stack S_i for every subtrace T_i is initially empty. The input queue Q_i of every process is initialized with the input subtrace T_i, where every request is annotated with 0. The upper-bound distance tables U_i and the lower-bound distance tables L_i are initialized with 0 for every possible stack distance.*

Output: *The overall upper-bound distance table $U_{overall}$ and lower-bound distance table $L_{overall}$, which are the sum of the tables U_i and L_i, resp., for all processes i.*

```
begin
    for $1 \leq i \leq m$ pardo
        $(req, odist) = \text{DEQUEUE}(Q_i)$
        while $\neg$ TERMINATE() do
            if $odist > 0$ then $U_i[odist] = U_i[odist] - 1$
            if $req \in S_i$ then
                $dist = \text{POSITION}(S_i, req)$
                $L_i[dist] = L_i[dist] + 1$
                $\text{REPOSITION}(S_i, req)$
            else
                $\text{PUSH}(S_i, req)$
                if $i > 1$ do
                    $pdist = |S_i|$
                    $U_i[pdist] = U_i[pdist] + 1$
                    $\text{ENQUEUE}(Q_{i-1}, (req, pdist))$
            $(req, odist) = \text{DEQUEUE}(Q_i)$
end
```

8.5. Simulation Results and Termination Detection

As discussed above, the approximate cache simulation algorithm provides results in the form of a success function returning intervals instead of exact values. An important property of the algorithm is the fact that a result interval contains the correct number of hits for a given cache size. Among other approaches, this can be exploited to provide an approximate simulation result that is chosen from the values of the result interval. Furthermore, the accuracy of the result can be calculated from the size of the interval. For example, if the middle value of the interval is chosen as an approximate result, the maximum deviation of this result from the correct value is half the length of the interval.

The basic algorithm for approximate cache simulation presented above can be used either with approximate state matching (see Chapter 5) or to perform a progressive simulation (see Chapter 6). For both cases, Algorithm 8.3 can be used unchanged. Differences exist in the detection of termination of the algorithm and in the calculation of simulation results.

Unfortunately, result calculation is a rather expensive operation, as it includes the collection of distance tables from all processes and the computation of prefix sums of the tables. Prefix sums can be calculated efficiently in parallel utilizing parallel prefix operations [50, 52]. Still, this collection and preparation of distance tables is too expensive to be performed in high frequency during the simulation execution, limiting the usability of progressive cache simulation. To perform approximate state matching, however, it is sufficient to collect and aggregate the entries of the upper-bound distance table. This is due to the result accuracy at a point in time only depending on the preliminary distances, hence the size of result intervals for a given cache size being determined from the upper-bound distance table.

Utilizing progressive cache simulation, termination detection is no issue, as the termination of the simulation process directly depends on the decisions of the user, which in turn needs measures for the current results accuracy. The situation is different with approximate state matching where criteria for the termination of the simulation have to be defined. Therefore, the rest of this section is dedicated to the discussion of termination detection in conjunction with approximate state matching.

The global calculations of result intervals presented above are correct if processes do not continue simulation during the time of calculation. Otherwise, hits might be lost or recorded multiple times. In cases where a global execution strategy independent of the current accuracy of result intervals is chosen, this does not pose any problems. For example, if runtime requirements exist, the simulation can be executed for a fixed amount of time and the accuracy of results can be determined afterwards. In this case, approximate state matching is applied without error control, i.e. the states of adjacent simulation processes do not match exactly, the result simulation execution is nevertheless accepted, resulting in errors which might be arbitrarily large. With the parallel cache simulation approach presented above, this still is a viable alternative, as the accuracy of results is determined during the simulation execution.

If termination of the simulation algorithm has to depend on the current result accuracy, or if termination control should be implemented locally in the processes, further mechanisms are needed. A straightforward approach to determine simulation runtime by result accuracy is to halt execution of the processes periodically in order to check the current accuracy. The whole simulation can be finished if accuracy is satisfying, or resumed otherwise.

Although the implementation of termination control locally in the processes is preferable, it is much harder to achieve. This is due to the implicit synchronization of processes, which only have limited knowledge about the status of the simulation in adjacent processes. Therefore, for a local termination control, either additional messages have to be passed between processes, or a controller process has to be introduced. Unfortunately, the latter approach introduces a bottleneck in the parallel simulation, which is supposed to limit its scalability. Hence, this is no viable option. Additional messages to communicate about the current results accuracy might range from flags that give the status of the preceding process (e.g. running or finished) to informations about its local result accuracy. For a very detailed termination control, sophisticated mechanisms are needed, which might incorporate a high overhead that is not justified by the finer granularity of the termination control.

Another aspect of approximate state matching in time-parallel cache simulation is the deviation measure δ to be utilized. As discussed above, the goal of the parallel full stack LRU cache simulation is to determine the success function of an input trace. What is calculated by the approximate Algorithm 8.3 is a function returning upper and lower bounds on the correct number of hits for a given cache size. A state deviation can be concisely defined here for the return value of the approximate success function for a given cache size. But how is the whole success function to be interpreted as a deviation of states? One possibility is to examine the size of result intervals for a single cache size. A reasonable value in this respect might be the largest size that is of relevance in a simulation study, as it gives an upper bound of the size of result intervals at a given time. Other deviation measures are thinkable, but their feasibility depends on the objectives of the corresponding simulation study.

8.6. Experiments

This section is concerned with an examination of the viability of the approximate cache simulation approach presented above and the implications of applying approximate state matching or progressive time-parallel simulation. A prototypical implementation of Algorithm 8.2 and Algorithm 8.3 was created in C using the message passing interface MPI [64] for communication between processes. For the determination of speedups, a sequential simulator was implemented. Only the more interesting full stack simulation approaches were considered, using the search-tree based implementation of the LRU stack described in [81].

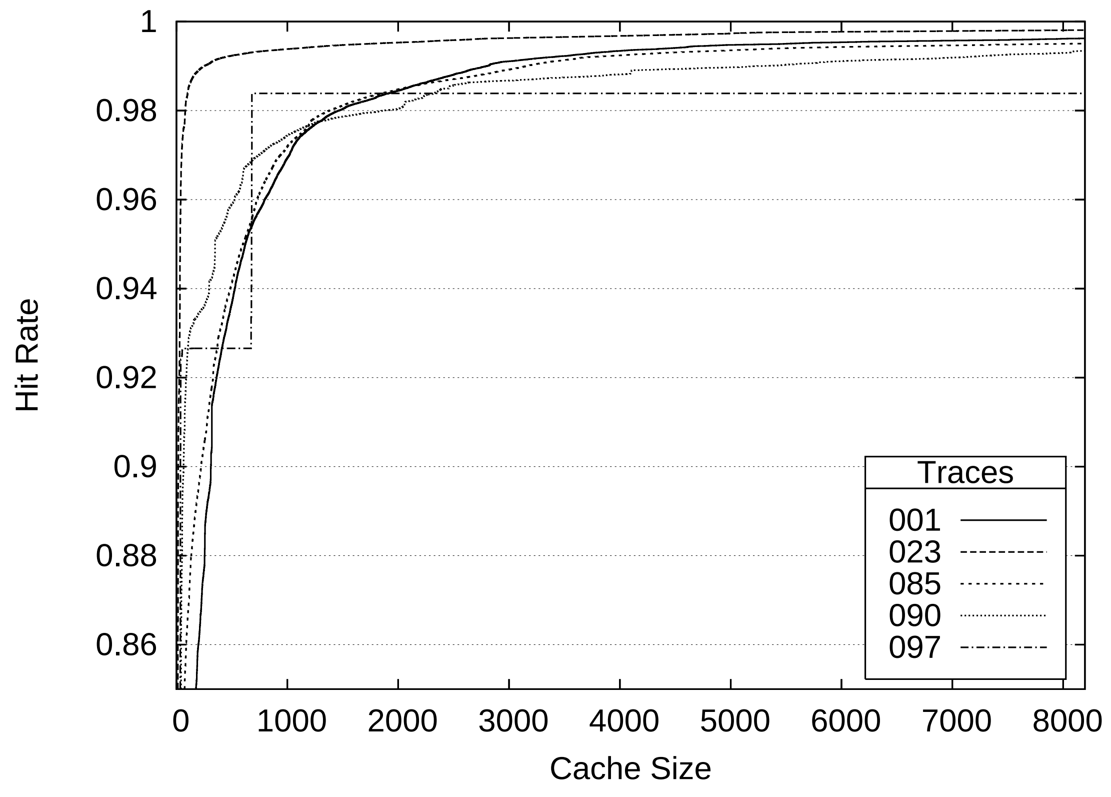
Extended experiments with these prototypes were conducted on an SGI Origin 3000 with 64 500 MHz MIPS R14000 processors, a total of 32 GB of main memory, and the IRIX 6.5 operating system. The C sources were compiled with the MIPSpro 7.2 Compiler with switches `-O3 -n32` enabled, using the MPI 1.2 implementation in the SGI Message Passing Toolkit (MPT).

The experiments consisted of the determination of the success functions of various input traces, taken from the NMSU TraceBase facility [98]. Table 8.1 summarizes the properties of the traces, which are collections of memory references from programs in the SPEC92 benchmark suite [32]. A smaller trace (001) is included with a moderate number of unique references. The remaining traces have about the same length, but differ significantly in the number of different referenced pages. From those, the 085 trace is most complex with the highest number of unique references, lead-

Table 8.1. Properties of input traces

Trace	No. of refs.	No. of unique refs.
001 cexp	18,782,144	26,198
023 eqntott	100,000,000	89,857
085 gcc	100,000,000	162,997
090 hydro	10,985,664	26,521
097 nasa 7	99,731,776	30,109

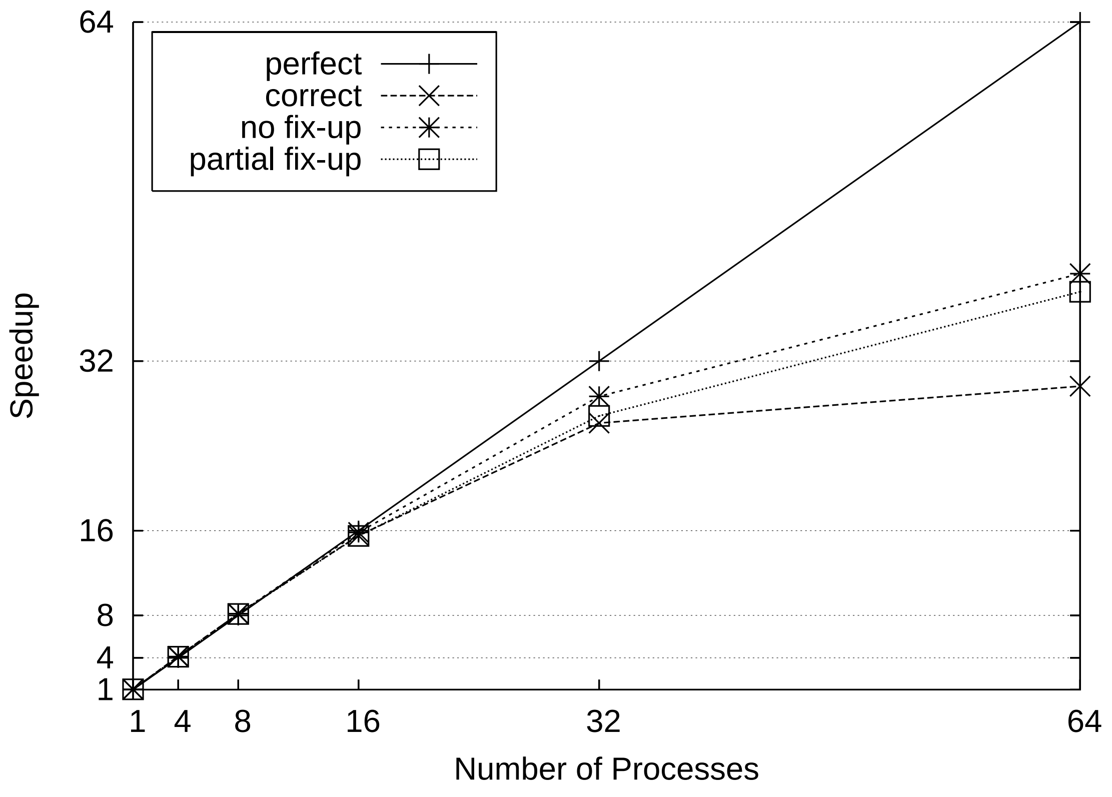
Figure 8.5. Success functions of traces



ing to interesting results discussed later. Figure 8.5 shows a comparison of the success functions determined by the sequential cache simulator. The hit rate in the interval $[0.85, 1.0]$ is plotted against an increasing cache size. Traces 001 and 085 exhibit a moderate locality, where a significant number of misses occurs even for cache sizes larger than 2000 entries. Trace 023 has a much higher overall locality indicated by the success function increasing faster than the functions of the other traces. Trace 097 shows a very special behavior, where only a limited number of different hit rates exist.

When processing the traces, no distinction was made between data and instruction references. A 16 byte cache line was supposed, masking off the last four bits of every reference. Execution times of the calculations of stack distances were recorded during the experiments of both the sequential and parallel cache simulators. The results presented here are calculated from the averages of ten repetitions of every experiment.

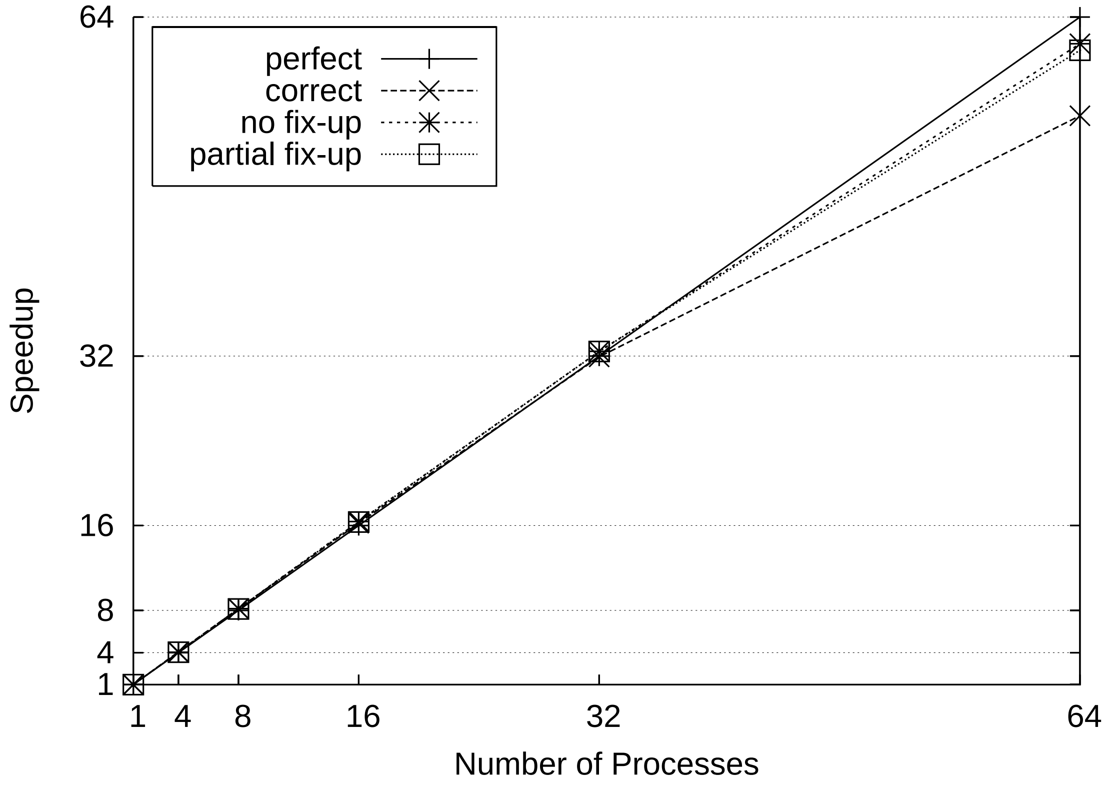
Figure 8.6. Speedups for trace 001



In the following presentation of results, the speedup of the approximate parallel simulator against the corresponding sequential simulator is discussed and the size of result intervals of the parallel simulation runs are presented for different cache sizes in order to evaluate the deviation measure for approximate state matching.

Speedups for the various traces are shown in Figure 8.6 to Figure 8.10. For all of the traces, speedup of the exact parallel cache simulation is excellent for small numbers of processes (≤ 16). However, with the number of processes increasing to 32, a degradation of performance is notable with the smaller 001 trace. With 64 processes, the degradation reaches a significant level for all of the traces. In Figure 8.6 of the 001 trace, the speedup for 64 processes even stays on about the same level as that for 32. The speedups of the 085 trace shown in Figure 8.8 exhibit a more interesting pattern. Most notably, a better-than-perfect speedup is achieved for 16 and 32 processes. This is due to the smaller size of the local LRU stacks in the

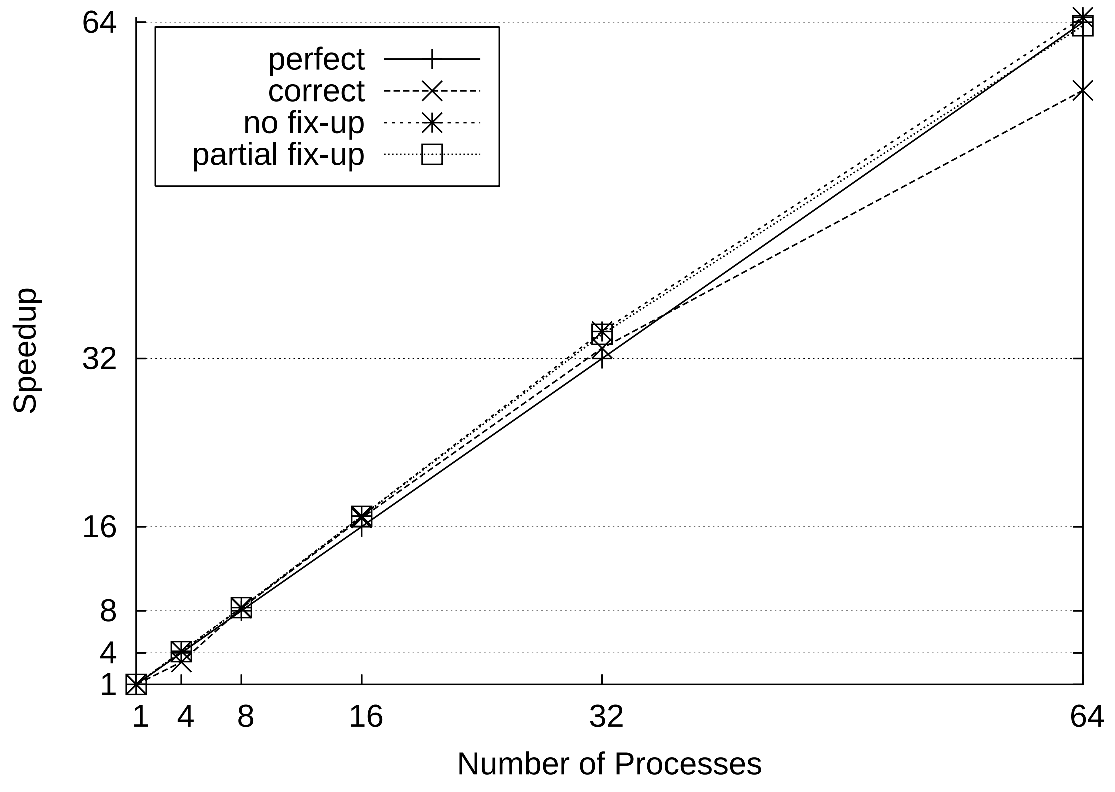
Figure 8.7. Speedups for trace 023



parallel simulation, resulting in a better cache performance and hence in a faster execution. Another observation is the bad performance of the basic parallel simulation for the 085 trace with 4 processes. This results from the size of the resimulation subtraces of a single process exceeding the limit of the MPI message buffer, which leads to a much higher synchronization overhead, as the processes have to wait until the resimulation subtraces have been fetched by the peer before being able to continue simulation.

Additionally, Figure 8.6 to Figure 8.10 show the speedups of two different types of experiments with the approximate parallel cache simulator: one labeled no fix-up with no resimulation steps at all (i.e. execution is stopped if every request from the input trace has been processed exactly once) and one labeled partial fix-up where resimulation is performed partially (more precisely, the amount of resimulation in every process is restricted to the minimal number of resimulation steps necessary in any one of the parallel processes without approximation). Both no fix-up and partial fix-up exper-

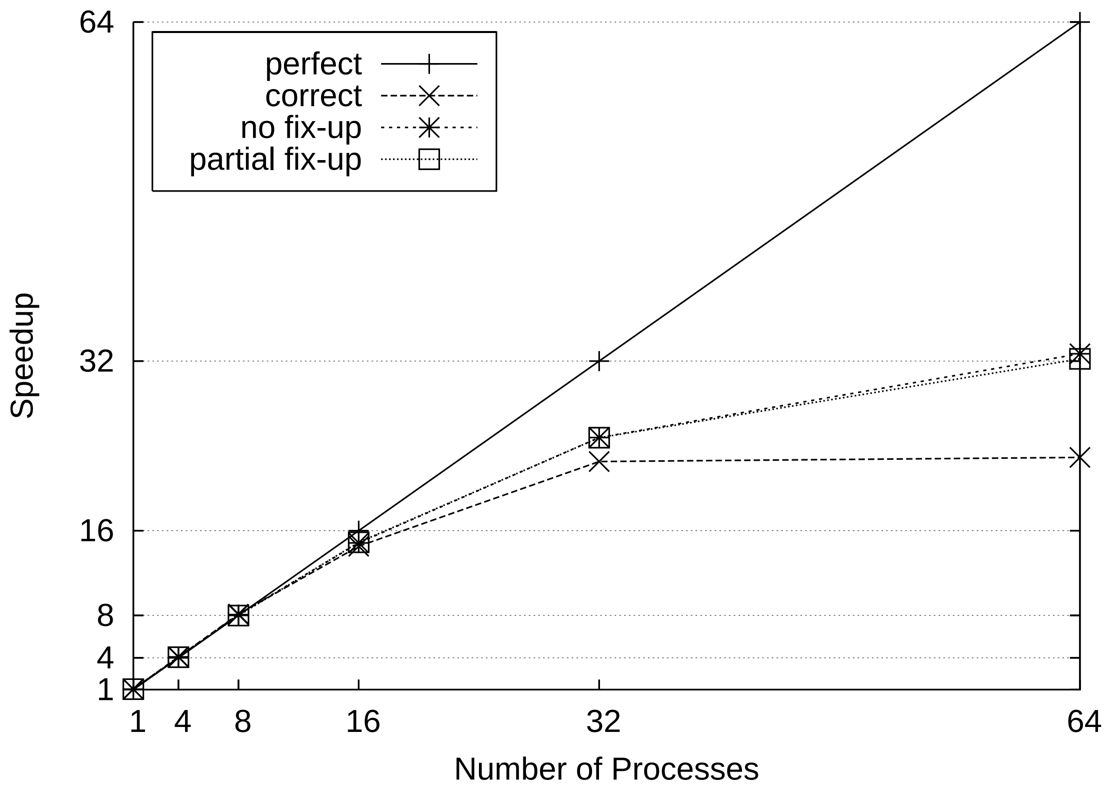
Figure 8.8. Speedups for trace 085



iments notably increase scalability, as the speedups drop only marginally for 64 processes (except for the 097 trace, where in comparison to the exact version, the speedup is not increased significantly by the approximate simulators).

As discussed above, the accuracy of results can be measured by the size of result intervals for the different cache sizes, giving the maximum range of possible results for the number of hits of an input trace. Most often, instead of the absolute number of hits, a cache simulation is concerned with the determination of the hit rate, i.e. the number of hits relative to the size of the input trace. To calculate the accuracy when determining the hit rate instead of the absolute number of hits, the size of result intervals is divided by the number of requests in the input trace to get the maximum possible deviation of the approximate hit rate for a given cache size from the correct hit rate. Figure 8.11 to Figure 8.12 present these *relative result interval sizes* in ‰ of the hit rate for the 001 and 085 traces. Accuracy is shown for

Figure 8.9. Speedups for trace 090

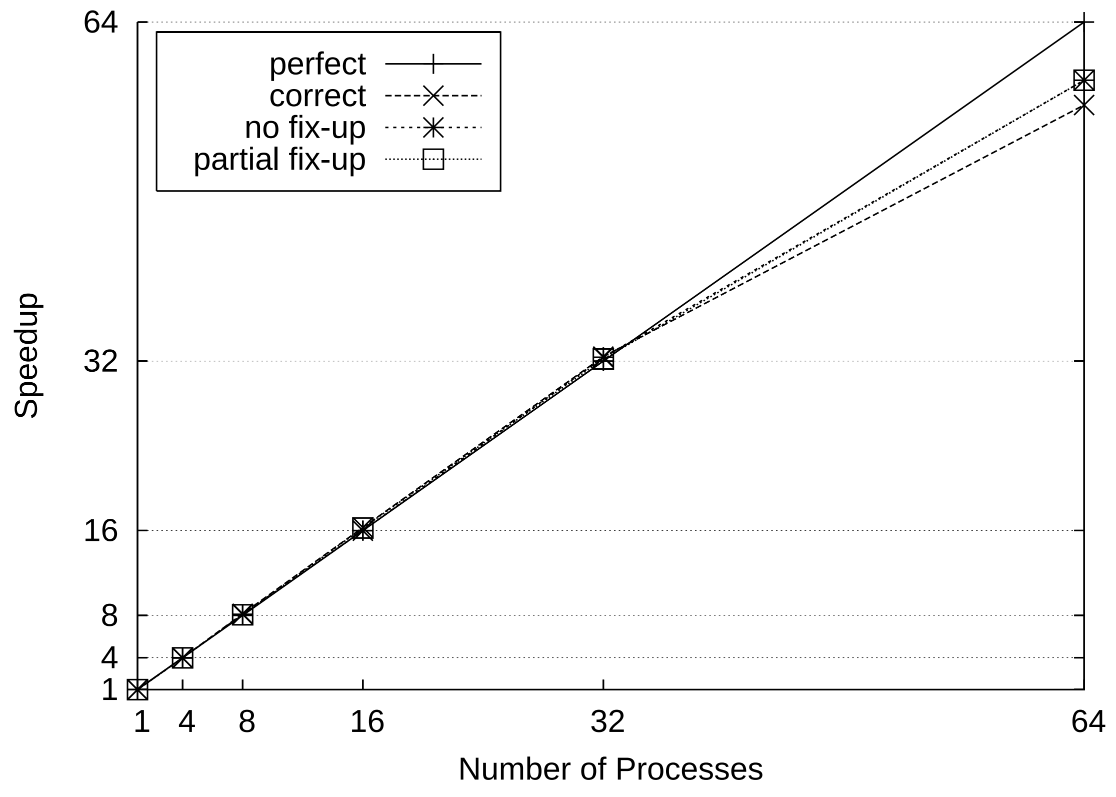


three different cache sizes for the experiments with no fix-up computations and one cache size for the experiments with partial fix-up. For the smallest cache size of 1024, the interval sizes seem to grow linearly with the number of processes, whereas for larger caches, the growth is clearly sublinear. Note also that the result interval sizes are generally low, lying in the range of a few per thousand of the hit rate. The experiments with partial fix-up, which did perform almost as well as those with no fix-up, exhibit a very high accuracy, even for larger cache sizes.

8.7. Summary

Temporal parallelization of simulation models is a promising alternative to the more traditional parallelization approaches. The difficulty in applying temporal parallelization to different application domains lies in an ef-

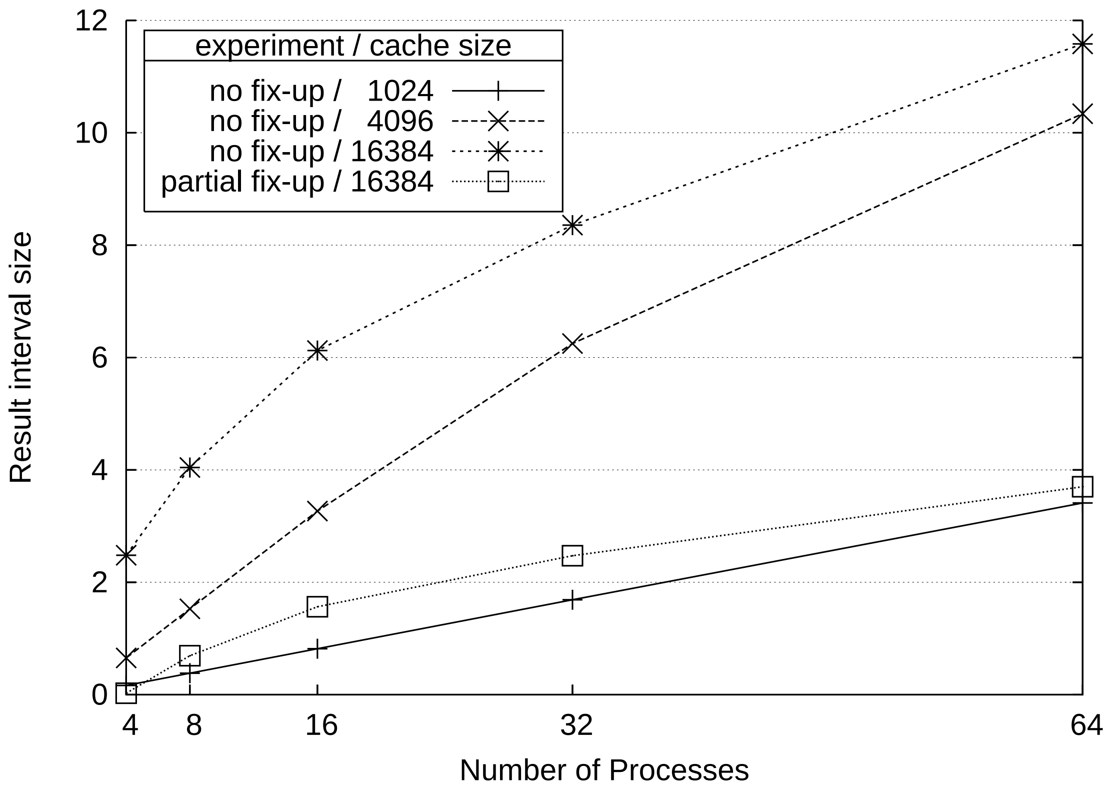
Figure 8.10. Speedups for trace 097



ficient solution of the state-match problem. This thesis proposes the usage of approximate methods for time-parallel simulation in order to extend its applicability to new classes of models, as well as a means to improve the efficiency of existing time-parallel simulation models. The focus of this chapter is the application of approximation for the parallel simulation of LRU caching.

In this chapter, existing time-parallel simulation algorithms, both for simple LRU cache simulation and full LRU stack simulation, have been presented. These algorithms have then been used as the basis for approximate simulation algorithms, where at any time during the simulation, approximate results can be calculated in the form of intervals which give the range of possible values of results. The approximate cache simulation algorithms can be used both with approximate state matching and progressive simulation without changing the basic algorithm. Differences are in the calculation of simulation results and termination detection.

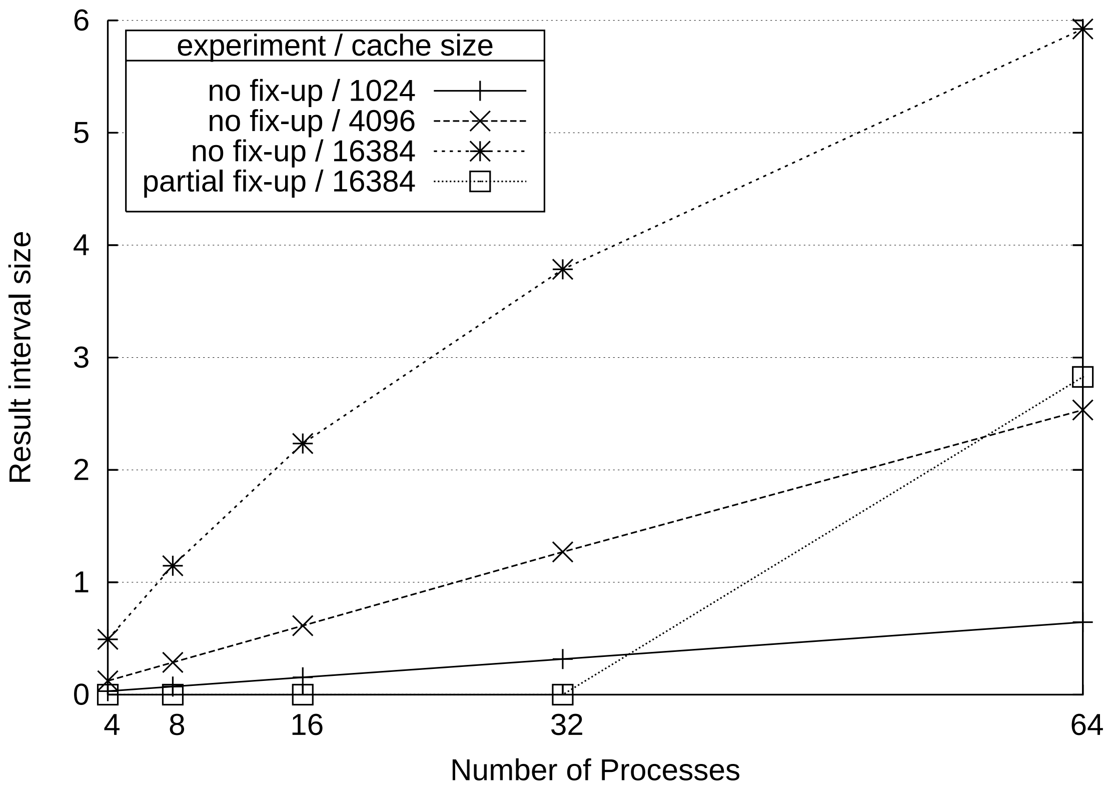
Figure 8.11. Result interval sizes for trace 001 (in ‰ of the hit rate)



The algorithms presented in this chapter (the basic as well as the approximate ones) have been prototypically implemented and experiments have been conducted. These indicate a significant increase in the speedup of the simulation with a reasonable accuracy of simulation results.

Previous work on the parallel simulation of LRU caching has either restricted itself to the simple cache simulation approach, or exhibited a serious decrease of the speedup achieved for higher numbers of processes. In this chapter, it is shown, how approximate methods can be used to increase the overall speedup, allowing the parallel simulation to scale to very high numbers of processes. Two properties of the approximation algorithms allow a direct control of the introduced error with approximate state matching and are favorable for progressive simulation: accuracy of simulation results increases monotonically with time and it can be calculated at any time of the simulation execution. In theory, linear speedup can always be achieved by allowing an arbitrary uncertainty. In practice, this is hard to achieve, due

Figure 8.12. Result interval sizes for trace 085 (in ‰ of the hit rate)



to the synchronization required for input and collection of results. However, the experiments indicate, that even with a very small error, significant increases in speedup are possible.

9. Traffic Simulation

Transportation research is concerned with the analysis of phenomena occurring in real world traffic. As the amount of traveling, as well as transport of goods, is increasing continuously, existing transportation resources get more and more overloaded, resulting in congestion or breakdown of resources. The situation is especially bad with road traffic, where a limited amount of road capacity must accommodate an increasing amount of traffic. Transportation research can help to understand the characteristics of traffic and propose solutions for existing problems. To achieve these goals, computer simulation of traffic is an important tool. A lot of work has been invested in the past to develop simulation models appropriately representing the reality of road traffic.

There are two fundamentally different types of models in traffic simulation. Macroscopic models try to characterize traffic flow with fluid-dynamical approaches [31, 56], giving an aggregated view of the problem. Microscopic simulation, in contrast, represents all entities (e.g. vehicles, traffic lights, intersections) of the simulated system as individual objects. Besides more accurate simulation results, this has the advantage that individual choices of travelers can be considered. A straightforward approach is to describe the movement of vehicles by equations, directly or indirectly derived from physical laws, e.g. [104]. In order to reduce computation time, more recent approaches model traffic as cellular automata [105], where the dynamical behavior of vehicles is discrete in time and space. An example of this is the Nagel/Schreckenberg model [69], which is recognized as simple and computationally efficient, yet exhibiting realistic behavior.

The original cellular-automaton-based model of Nagel and Schreckenberg was developed for single lane traffic. Afterwards it has been extended to support multi lane [70] and urban traffic [24]. Among other applications, it has been used for traffic forecasts [18] and as the central part of the large-scale TRANSIMS project [99].

Due to the high fidelity of the Nagel/Schreckenberg model, computational demands of large-scale traffic simulators can be high, leading to the need for efficient parallelization. Existing parallelizations [67, 68] use a domain decomposition approach where a graph representing the street network to be simulated is decomposed into subgraphs that are simulated concurrently on different processing nodes. To exchange information about vehicles traveling from one subgraph to another, the parallel processes operate synchronously in the simulated time, i.e. at every wallclock time instant, all processors execute the same simulation time step. For small numbers of processors this poses no problem and impressive performance increases can be achieved. However, due to the significant synchronization overhead, these solutions do not scale well to higher numbers of processors [13], if a fixed problem size is supposed.

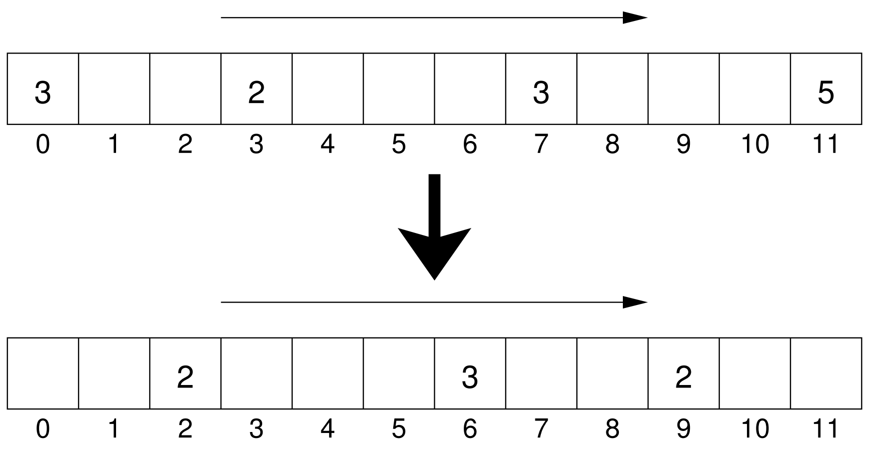
For a potential improvement of the scalability of parallel traffic simulations, the method of time-parallel simulation can be applied to reduce synchronization between processes. However, the Nagel/Schreckenberg model exhibits a rather complex state space with a large number of possible states. In that case, time-parallel simulation is supposed to produce poor results, as exact state matching is hard to perform. This problem could be weakened by the utilization of approximate state matching or progressive time-parallel simulation. This chapter is intended as a case study of time-parallel road traffic simulation, trying to extend the applicability of time-parallel simulation to models with large and complex state spaces.

As a preliminary, the utilized Nagel/Schreckenberg model is introduced in Section 9.1, before the time-parallel road traffic simulation approach is presented in Section 9.2. The feasibility of time-parallel road traffic simulation depends on the amount of computational overhead due to fix-up computations. In Section 9.3, results of experiments evaluating the extent of fix-up computations are provided.

9.1. The Nagel/Schreckenberg Model

Cellular Automata [105] are of an increasing significance in the field of experimental physics. They are most often used to describe natural phenomena with discrete and computationally efficient models. In a cellular automaton, a one-, two, or multidimensional space is divided into cells of a fixed size with a discrete state attributed to each cell. The state of an indi-

Figure 9.1. Example of vehicle movement



vidual cell changes discrete in time and depends only on the *neighborhood* of the cell which is most often defined as a set of cells located nearest in the geometrical sense.

The basic Nagel/Schreckenberg model [69] is a one-dimensional stochastic cellular automaton, where a road segment is divided into cells of a fixed size (typically 7.5m, which is enough to accommodate a single passenger car with sufficient space before and after the car). A cell can either be empty or occupied with a car traveling at a discrete velocity $v \in \{0, \dots, v_{max}\}$. The behavior of a vehicle located at a cell i is specified by four simple rules [69], which are applied in the given order to all vehicles during a time step. The rules are applied in parallel to the vehicles, operating on the old state of the simulation (i.e. the state of the simulation before any calculations of the current time step have been performed).

1) **Acceleration:** if the velocity v of the vehicle is lower than v_{max}, then it is increased by one $[v \rightarrow v+1]$
2) **Slowing down:** if there is a vehicle at site $i+j$ (with $j \leq v$), the velocity is reduced to $j-1$ $[v \rightarrow j-1]$
3) **Randomization:** if $v > 0$, then with probability p, the velocity v of the vehicle is decremented $[v \rightarrow v-1]$
4) **Vehicle motion:** the vehicle is advanced v cells

Figure 9.1 shows an example of vehicle motion in a short road segment with 12 cells. The vehicles moving from left to right are depicted by numbers inside the cells which also indicate their velocity. The state of the segment is shown before (first line) and after (second line) performing the state update of a particular time step. Note that the vehicle located in cell 9 after performing state changes chose to decrease the maximally possible velocity of 3 due to the probabilistic rule number 3).

Although extremely simple, the Nagel/Schreckenberg model exhibits realistic behavior. While at low traffic densities laminar flow of traffic can be observed, congestion clusters are formed at higher densities. Further, the formation of start-stop waves found in real freeway traffic is adequately represented by the model. This is illustrated in Figure 9.2, where the occupation of a road section (x-axis from left to right) is plotted against the simulation time (y-axis from top to bottom). A cell occupied by a vehicle is plotted as a black dot, while unoccupied cells are left blank. If the movement of a vehicle from left to right is not constricted by a congestion, its path over the road segment can be identified as a decreasing diagonal line. Further, congestion clusters (black areas) can be seen, where multiple vehicles are located close together.

A necessary prerequisite for the application of time-parallel simulation is the repeatability of simulation experiments (cf. Section 4.2). Repeatability of the microscopic traffic simulation model introduced above can easily be achieved by a presampling both of vehicle arrival instants and the random numbers for the stochastic movement of vehicles.

9.2. Time-Parallel Traffic Simulation

Parallelization approaches of traffic simulations based on the cellular automaton model using spatial decomposition, reported limited scalability with an increasing number of processors [67, 68]. In these approaches, space parallel simulation requires communication after every single time step. Thus, communication overhead becomes high for not tightly coupled processing nodes. For a similar model, Cetin et al. derive an upper bound on the number of processors that can be utilized efficiently, given the latency of the underlying communications network [13]. To be more specific, Cetin et al. report experiments using Ethernet for communication between processes, where speedup levels out at about 32 processes. This bound can be improved using faster means of communication, e.g. to about

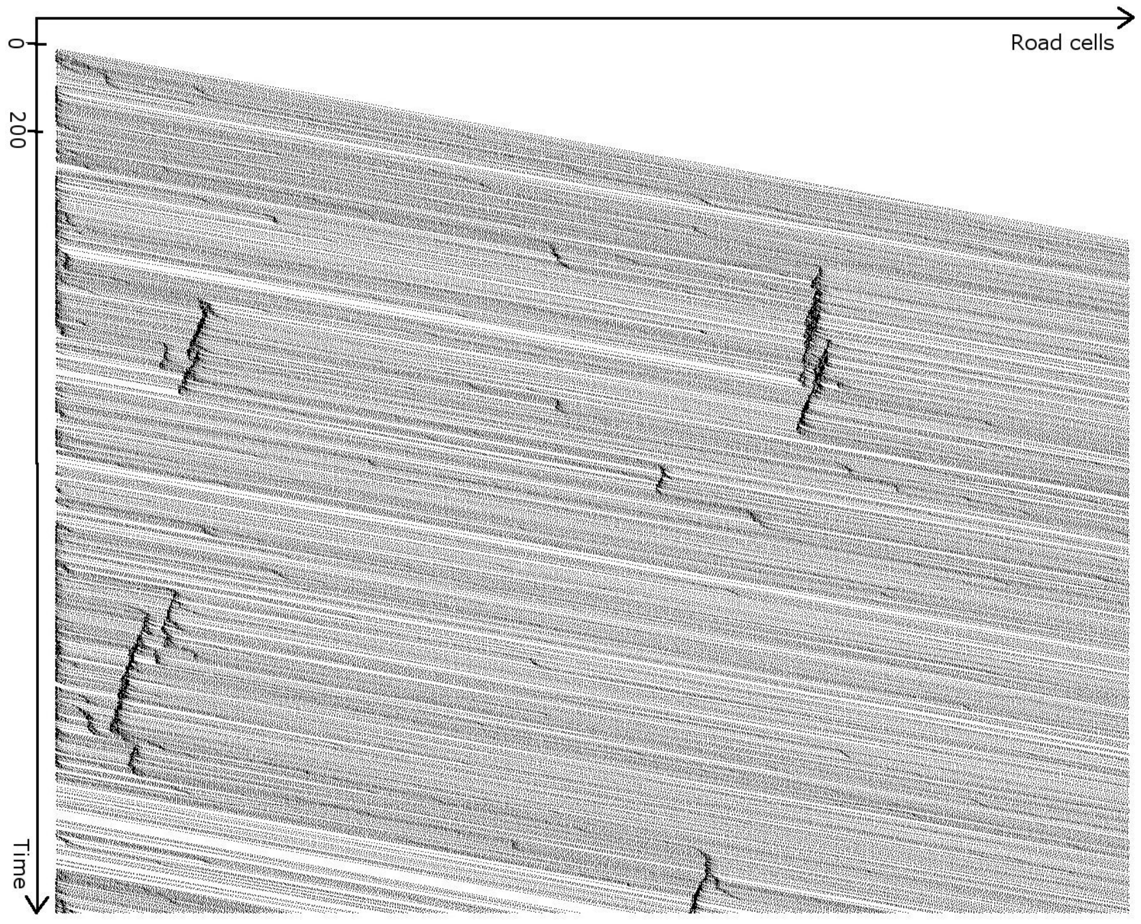
64 processors with a Myrinet network.

As an alternative parallelization approach, temporal decomposition of the traffic simulation model can be performed. As discussed in Section 4.1, the simulation time is partitioned into intervals $T_i = [\tau_i, \tau_{i+1}]$, where each interval contains a given number of time steps of the cellular automaton. Obviously, the processing done in each time step τ depends on the state of the simulation after the preceding step $\tau - 1$. Thus, the only time where the simulation state is known prior to execution is the initial state at simulation time zero. In analogy to time-parallel cache simulation (see Section 8.2), this problem can be approached by pretending that the road network is completely empty at the beginning of each time slice. This will produce incorrect simulation results which have to be corrected by the use of fix-up computations.

It appears to be manifest that this case occurs frequently in microscopic traffic simulation, rendering temporal parallelization useless here. Fortunately, traffic simulation is more robust to incorrect state changes than might be supposed. This is confirmed qualitatively in Figure 9.2 and Figure 9.3, which show the trace of vehicle movements starting at two different simulation times with an empty road section as initial state. When comparing the state development of the simulation starting at a later time (Figure 9.3) with the simulation starting at time 0 (Figure 9.2), a fast state convergence can be noted. A quantitative analysis of the state convergence behavior of microscopic traffic simulation is presented in Section 9.3.

An important aspect during fix-up computations of time-parallel simulation is to detect when two simulation states are identical (state matching). The original time-parallel simulation approach requires an exact match of the associated states. In the cellular automaton traffic model, a single simulation state is typically quite large, e.g. 2000 cell states for a road with a length of 15 km. Therefore, the comparison of two states is an expensive operation, being of the same order of complexity as the simulation of a time step. Even when only considering the computational overhead due to state comparisons, exact state matching would cut down the amount of potential parallelism significantly. Furthermore, for exact state matching, it is required to store the states of a simulation execution for each time step, extending the memory consumption by orders of magnitude. Hence, exact state matching can not be utilized, introducing the necessity of approximate state matching.

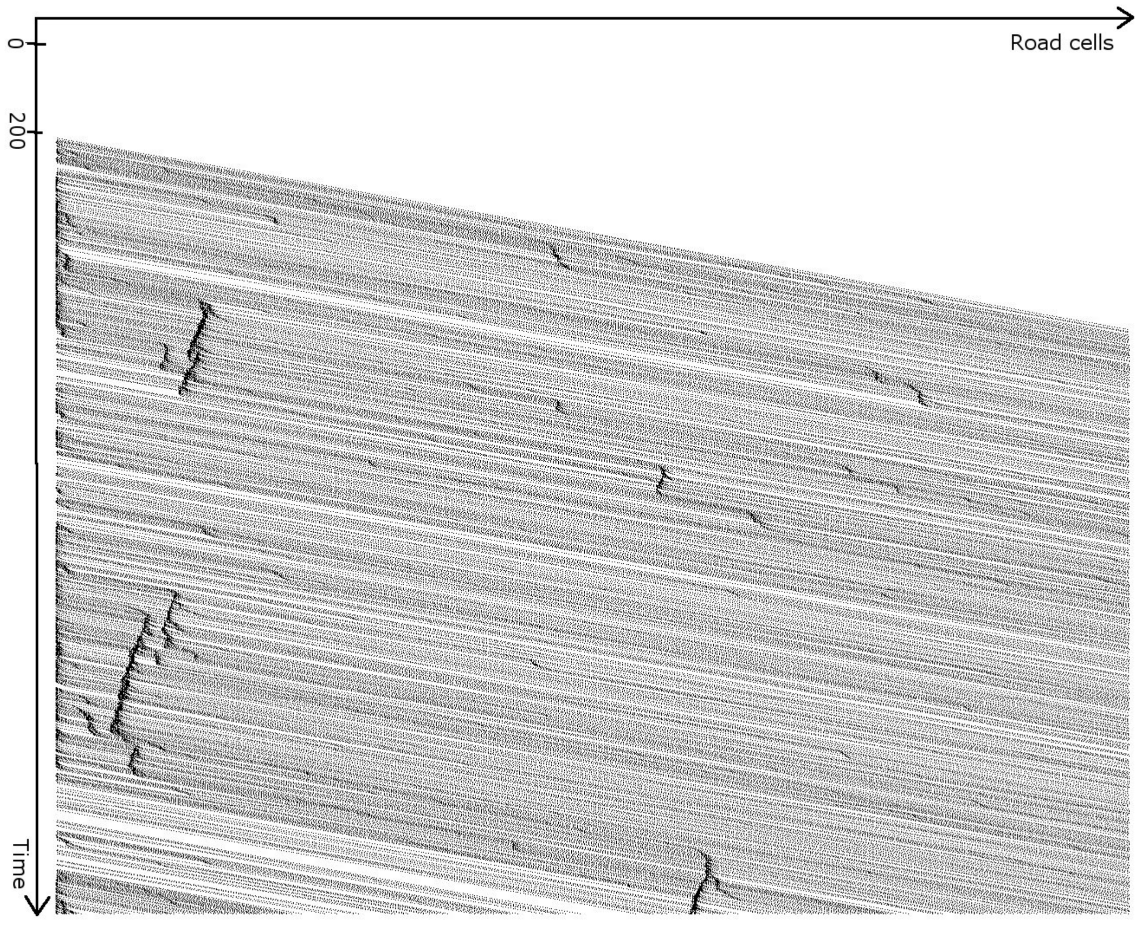
Figure 9.2. Trace of simulation starting at time 0



In order to apply approximate state matching, a deviation measure has to be defined that allows to identify *similar* states. The definition of similarity of states can only be done in the context of the objectives of the simulation study. Similarity of two simulation states should only be given if the simulation results produced as a consequence of these states are sufficiently close.

Besides the necessity of approximate state matching for efficient state match detection, it can be applied to decrease the amount of fix-up computation. This is done exactly as discussed in Chapter 5, leading to an error in the simulation results. Furthermore, such an approximate time-parallel traffic simulation can be extended to provide results progressively, as introduced in Chapter 6. In that case, the current accuracy of results during the

Figure 9.3. Trace of simulation starting at time 200



progressive simulation execution can be estimated with a deviation measure, as it would be used for approximate state matching. In both cases, the most interesting part of the application of approximate parallel simulation is the definition of a deviation measure properly representing a similarity of simulation states.

9.2.1. State Deviation Measures

In order to define and study similarity of states, two different deviation measures are introduced: (a) a microscopic measure based on the comparison of single cells and (b) a macroscopic measure based on traffic densities in road sections. A variety of other deviation measures are possible, although they are not discussed here.

The *microscopic* deviation measure compares the cells of two states s_1 and s_2 in the driving direction of the model until the state of a cell in state s_1 differs from that in s_2. Let p be the index of the first cell from the beginning of the road whose cell state is equal in s_1 and s_2, i.e. all cells before p have a different cell state in s_1 and s_2. If there is no difference at all, $p = N$, the number of cells in the model. The microscopic measure (ranging from 0 to 1) is defined to be

$$\delta_{micro}(s_1, s_2) = 1 - \frac{p}{N}.$$

This means that the microscopic measure corresponds to 1 minus the percentage of the road (starting from cell zero) where the cell states are identical. An important property of this measure is that $\delta_{micro}(s_1, s_2) = 0 \Leftrightarrow s_1 = s_2$. Thus, this measure can be used for exact as well as approximate state matching. However, the overhead for this measure is prohibitive (computational and memory complexities are identical to that of exact state matching). Hence, we only use it as a comparison basis for other aggregated deviation measures.

The *macroscopic* deviation measure that is used in Section 9.3.2 is called the *local density deviation*. It is computed as follows: the total number of cells N is partitioned into n sections (bins) of size $b = \frac{N}{n}$. In each bin, a local vehicle density is computed. This yields the following deviation measure:

$$\delta_{locdens}(s_1, s_2) = \frac{1}{N} \sum_{i=1}^{n} |\gamma(s_1, i) - \gamma(s_2, i)|$$

where $\gamma(s, i)$ is the number of vehicles in section i of state s. Note that if the number of bins is 1, two states are considered being equal if the number of vehicles in the two states is equal. On the other hand, if the number of bins equals the number of cells, two states are equal only if the occupation status of every single cell is equal in the two states.

It is important to realize that an aggregated (macroscopic) deviation measure may identify two states as being equal (i.e., yield a deviation of 0) even if they may not be exactly identical. Thus, using aggregated deviation measures, approximate state matching is performed implicitly, even if no explicit tolerance is applied. However, aggregated deviation measures are constructed in a way that gives rise to the speculation that similarity w.r.t. an aggregate measure corresponds to similarity w.r.t. the microscopic measure. This speculation is confirmed by experiments presented in Section 9.3.2 that show a significant correlation between the two proposed measures. The feasibility study in the following section also discusses the applicability of the proposed deviation measures to approximate state matching along the lines of [47].

9.3. Feasibility Study

Time-parallel traffic simulation with approximate state matching using the state deviation measures defined above, is feasible if only a small part of time intervals has to be re simulated during fix-up computations. The amount of fix-up computations is determined by the time until state matching occurs, which can be controlled with a tolerance in the case of approximate state matching. Here, the speed of state convergence in time-parallel traffic simulation is examined by simulating the fix-up computations as they would be performed by a time-parallel road traffic simulator (cf. Section 4.3). For a given time interval and trace of arriving vehicles, two different simulation executions are performed: One execution starts with an empty road segment, representing the determination of incorrect results before fix-up computations would be performed in the time-parallel simulator. The other execution starts with the *correct* state, where occupation of cells in the road segment has been determined by a warm-up phase. Now the states of both simulation executions can be compared at every time step and the convergence of states can be examined.

Note that the feasibility study documented in this section was specifically conducted with approximate state matching in mind. However, as general results about the convergence of simulation states are given, conclusions can be drawn for progressive time-parallel traffic simulation, as well.

The results of experiments presented in this section were obtained with a prototypical implementation of the Nagel/Schreckenberg model for a unidirectional one-laned road segment without intersections consisting of 2000 cells. Vehicles arrive at one end of the road according to the given trace of vehicle arrivals and are put into an input queue. If at least one cell from the beginning of the road is empty, the first vehicle in the queue enters the road section. This allows for multiple vehicle arrivals in the same time step

Table 9.1. State match times of parallel simulations

ρ	matching	mean	stdev	min	max
0.05	500	452.99	26.55	411	700
0.06	499	502.35	117.66	422	1340
0.07	360	752.61	385.75	435	1989
0.08	123	764.85	429.66	435	1993

and ensures correctness in the case of a congestion spreading back to the beginning of the segment. All of the simulations were executed for 2000 time steps.

9.3.1. Microscopic State Matching

As explained in Section 9.2, there is a prohibitively large overhead in exact state match detection. However, to gain insight into the state match behavior of time-parallel traffic simulation, a look at state match times (i.e. the times when exact state matching occurs) is worthwhile. Table 9.1 shows the results of experiments with average traffic densities ρ ranging from 0.05

Figure 9.4. Microscopic state deviation over time

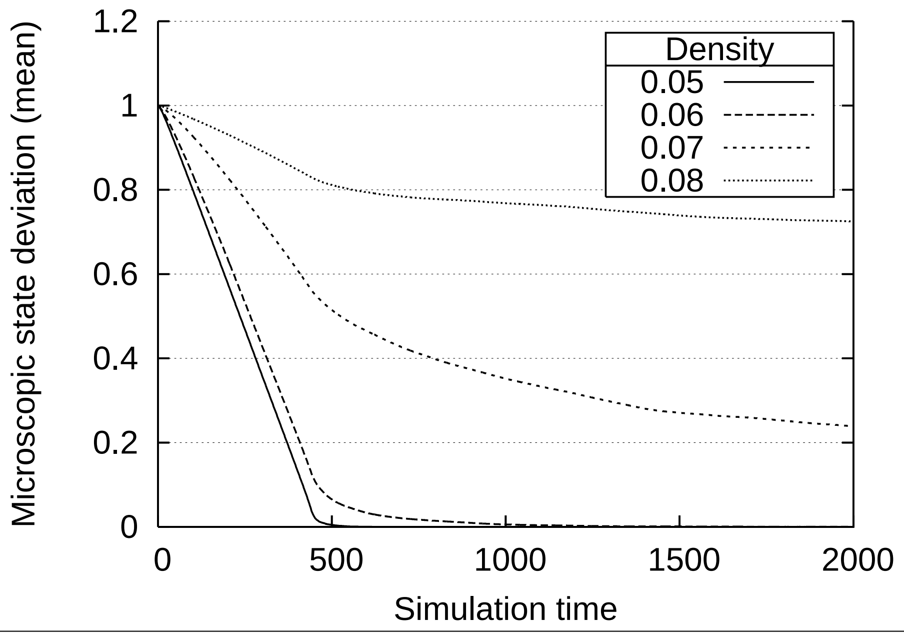
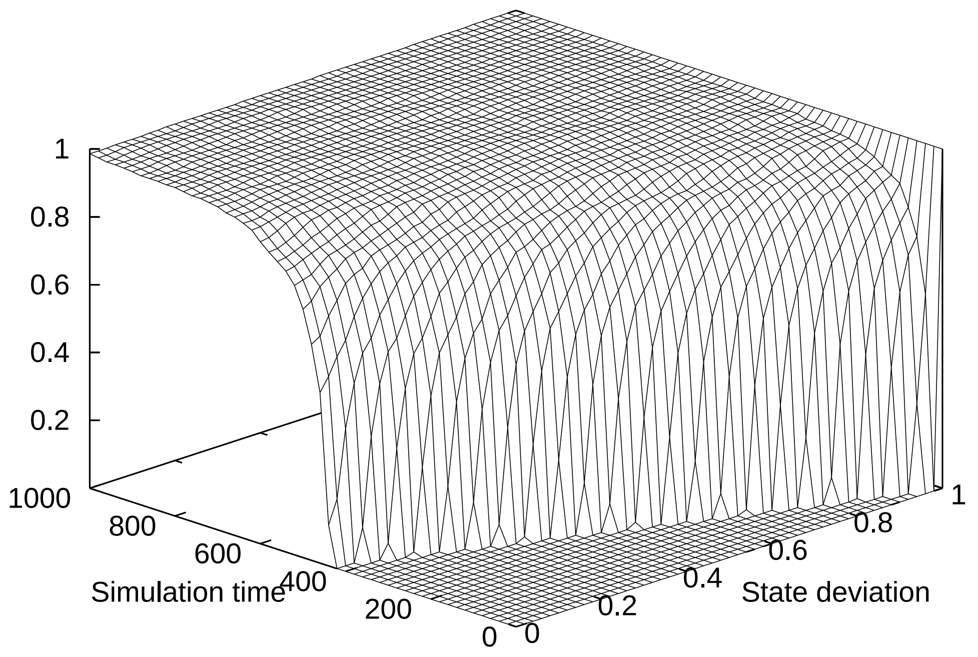


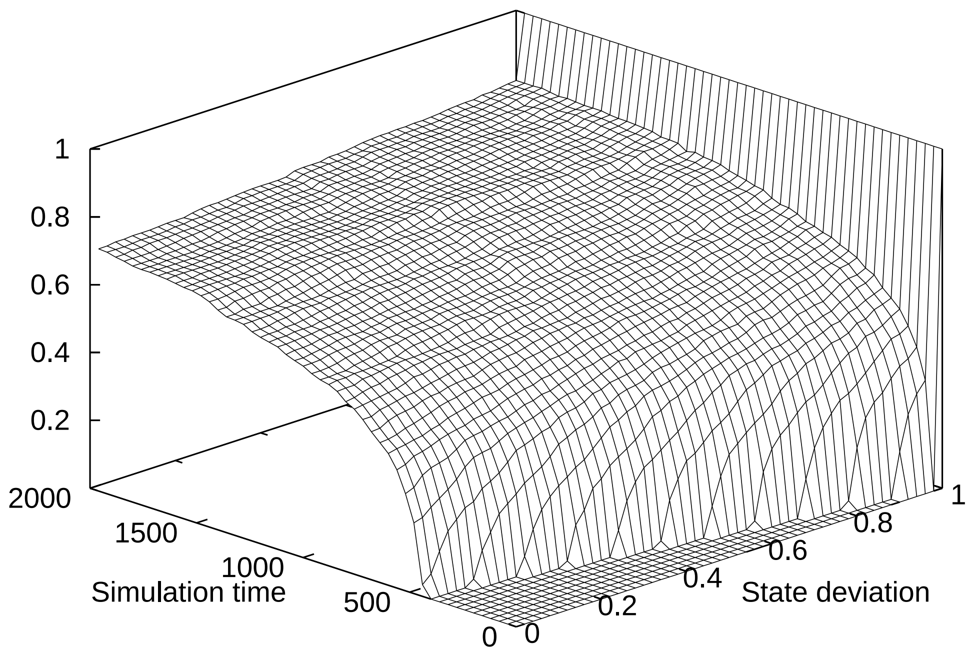
Figure 9.5. Deviation distribution over time ($\rho = 0.06$)



(i.e. 5% of available cells are occupied in average) to 0.08 and 500 repetitions for every density level. Note that in this model, saturation of the road occurs with a traffic density of 0.08 [69]. With lower traffic densities, the states of almost all of the repetitions match before the end of the observed time interval $[0, 2000]$ (column *matching*) and the average time of matching (column *mean*) is close to the average vehicle travel time. With increasing traffic density, the fraction of successful state matching before the end of simulation decreases and the average time of matching increases. Judging from Table 9.1 alone, time-parallel traffic simulation with exact state matching seems to be reasonable for traffic densities up to some value between 0.06 and 0.07.

Especially for an application of approximate state matching, the speed of convergence of simulation states is of importance. The convergence of states for 500 replications of experiments is shown in Figure 9.4 to Figure 9.6, using the microscopic state deviation measure introduced in Section 9.2.1. In Figure 9.4, the averages of all repetitions are presented for various densities. With the smaller densities of 0.05 and 0.06, the deviation

Figure 9.6. Deviation distribution over time ($\rho = 0.07$)

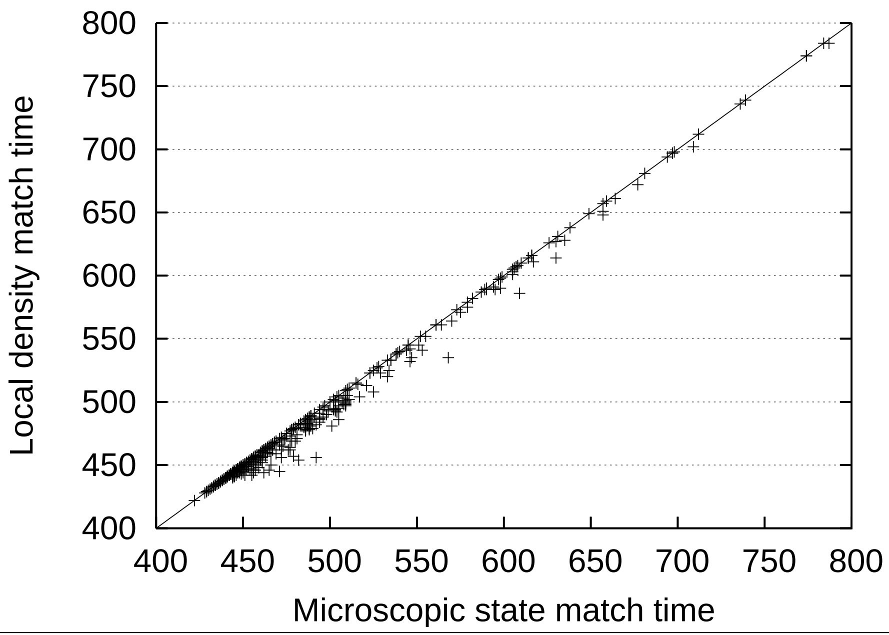


of states falls rapidly to zero, while with higher densities, only a gradual decrease can be observed.

Figure 9.5 and Figure 9.6 show the evolution of the distribution of state deviations (y and z axes) against simulation time (x-axis) of two selected densities. While convergence of states is fast for all of the repetitions with $\rho = 0.06$ (Figure 9.5), the situation is different with $\rho = 0.07$ (Figure 9.6). With this density a large part of the repetitions exhibit rapidly converging states. However, there is a significant number of repetitions with no convergence at all (a state deviation of 1 – noticeable by the almost vertical lines near the right back plane of the diagram).

Hence, using the microscopic state deviation measure, time-parallel traffic simulation seems to be feasible only for traffic densities below 0.07. However, better results might be obtained for specific simulation objectives by utilizing alternative deviation measures (see Section 9.3.3).

Figure 9.7. Microscopic state vs. local density ($\rho = 0.06$)



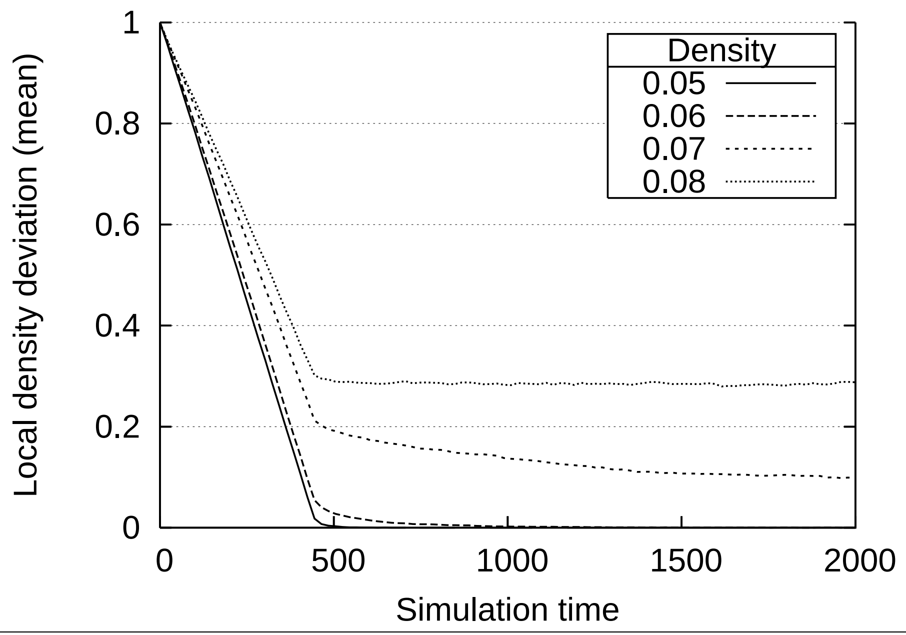
9.3.2. Aggregated Deviation Measures

Section 9.2.1 discussed the need for aggregated deviation measures to allow for an efficient detection of state matching and suggested one such measure. Here, the local density deviation measure is examined experimentally in order to validate its use in comparison to the microscopic state deviation measure. A moderate bin size of 50 cells has been used.

Figure 9.7 shows the match times of 500 repetitions with a density of $\rho = 0.06$ using the microscopic state deviation (x-axis) against the local density deviation (y-axis). It can be noted that all data points are located on or closely below the diagonal $y = x$. The situation is very similar for other densities not illustrated here. Hence, if only the times of exact matching are of interest, local density deviation seems to be an appropriate measure for state match detection, although a small bias towards the beginning of the simulation is introduced.

However, if the convergence behavior of states is of interest (e.g. if approximate state matching is to be applied), further investigation is necessary. Figure 9.8 shows the evolution of the local density deviation over simulation time (results are averages of 500 repetitions). To be able to compare experiments with different traffic densities, the graphs have been

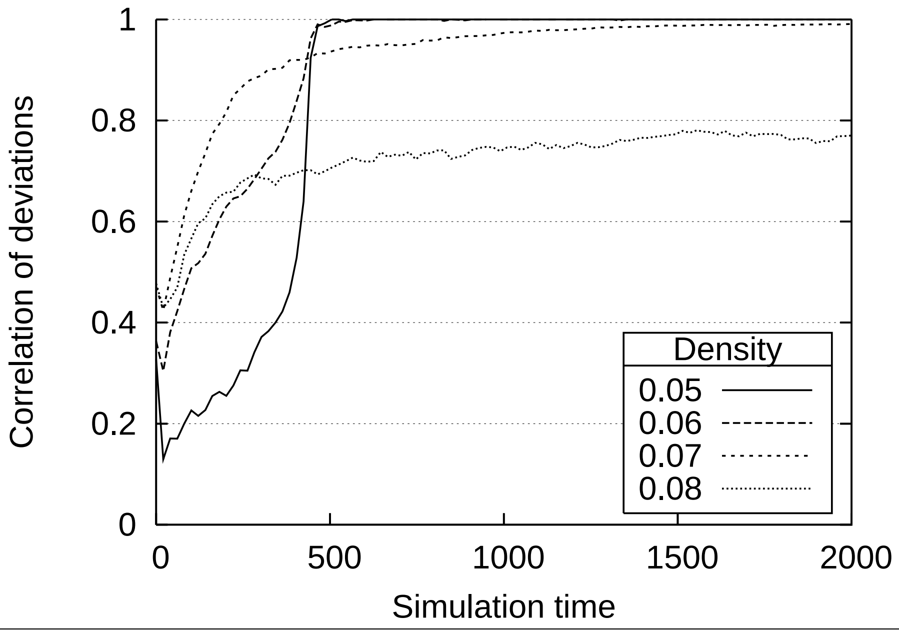
Figure 9.8. Local density deviation over time



normalized by dividing the absolute deviation at every simulation time by the maximum of all deviations (which is in fact always the deviation at the beginning of the simulation). It can be noted that the curves of the lower global densities of $\rho = 0.05$ and $\rho = 0.06$ are very similar to that of the microscopic state deviation (Figure 9.4). With a global density of $\rho = 0.07$, the local density deviation drops faster than the corresponding state deviation, but still exhibits a similar shape of the curve. Only with $\rho = 0.08$, the relationship between local density and microscopic state deviations seems to be weak.

These observations are confirmed by Figure 9.9, which shows the correlation between microscopic state and local density deviations of 500 repetitions over simulation time (Spearman's r [41] has been used as correlation coefficient, because of its resistance to outliers occurring frequently at later times during the simulations). Correlation is generally low before the first vehicle has completely passed the road, but is significant afterwards. Therefore, the local density deviation is an appropriate measure after this event. An exception is the traffic density of $\rho = 0.08$, where no significant correlation can be observed.

Figure 9.9. Correlation of microscopic state and local density



9.3.3. Travel Time Deviations

In Section 9.3.1, time-parallel road traffic simulation was found not to be feasible with a traffic density of 0.07 and above, which was determined using the microscopic state deviation measure. However, in the case of approximate state matching, it is not necessarily the case that the microscopic state deviation properly reflects the error in the simulation results introduced by the approximation. In that case, alternative deviation measures should be utilized, which may extend the applicability of time-parallel traffic simulation.

As an example of alternative measures, this section evaluates the case, where the simulation objective is to determine average travel times of vehicles over a road segment. A direct comparison between deviations of vehicle travel times and microscopic state deviation is hard to perform, as the travel time of a vehicle does not depend on a single state of the simulation alone, but rather on the set of all states of times where the vehicle is located somewhere on the road segment. On the other hand, there is only a weak dependency on the exact state of a single time step, viz. only on that part of the road where the vehicle is located during the time step. Hence, using microscopic state deviation is not reasonable in this case.

Figure 9.10. Average error due to approximate state matching

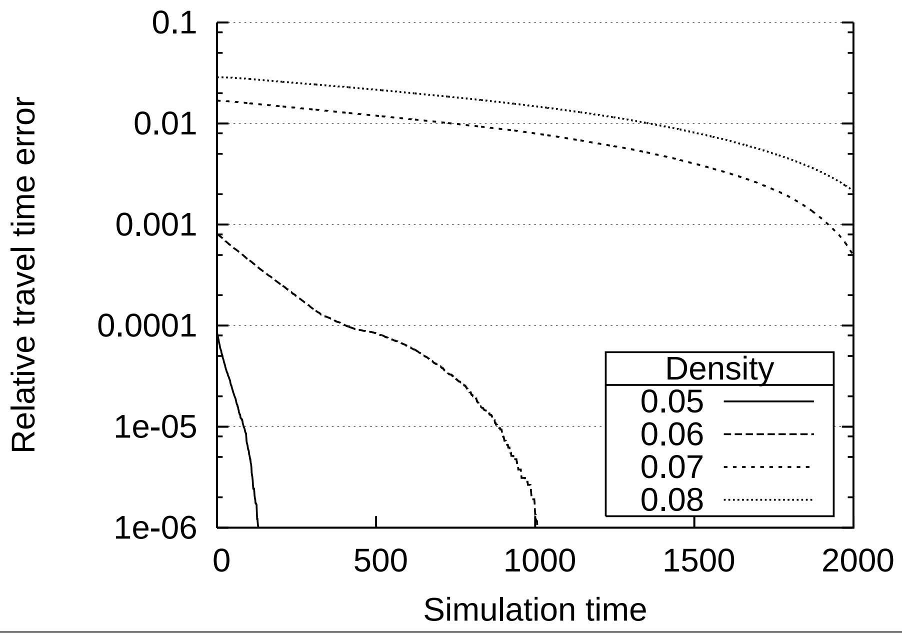
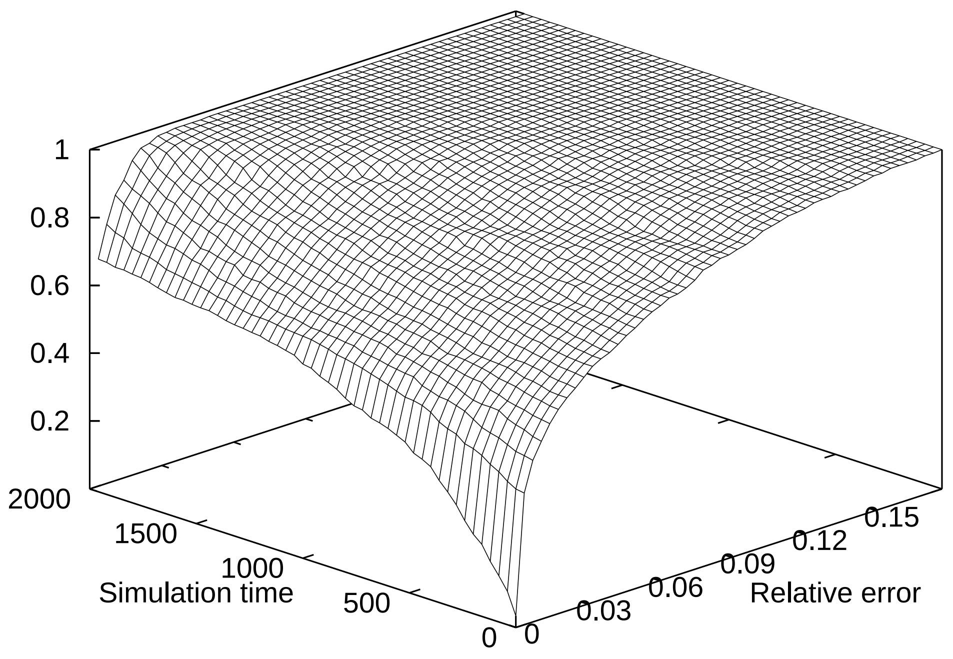


Figure 9.10 shows the relative errors in mean vehicle travel time (averaged over 500 repetitions) that would be introduced by an approximate state matching at a specific simulation time. The introduced error is generally low, not exceeding 1.5% of the correct vehicle travel time even with higher densities and it decreases monotonically with time, due to the accuracy of the simulation increasing with a growing amount of fix-up computations. For the smaller densities of $\rho = 0.05$ and $\rho = 0.06$, errors drop rapidly to zero, whereas there is a significant amount of error introduced with higher densities. However, when studying the distribution of errors over the 500 repetitions with $\rho = 0.08$, presented in Figure 9.11, it can easily be recognized that a large part of the average error is caused by a few outliers, especially at later simulation times.

9.4. Summary

To date, time-parallel simulation has mainly been applied to simulation models with rather simple complexities of simulation states. In contrast, microscopic traffic simulation exhibits high-dimensional state spaces with a high number of possible states, which cannot easily be predicted. How-

Figure 9.11. Evolution of error distributions



ever, with the approach of approximate state matching, it is possible to apply temporal parallelization. At the beginning of time intervals, the road is supposed to be empty, creating preliminary results after an initial simulation phase and correcting these afterwards by the use of fix-up computations. The difficulty here is the efficient detection of state matching, which is necessary to stop fix-up computations. This can be solved by using approximate state matching with aggregated state match measures. These measures should appropriately represent the matching of microscopic states and be efficient to calculate.

The feasibility of approximate time-parallel road traffic simulation has been examined with varying traffic densities for a simple road topology. With lower densities, the states of concurrent simulations converge rapidly, indicating a low overhead due to fix-up computations in a time-parallel simulator. With higher densities, the feasibility of time-parallel traffic simulation seems to be restricted, although for specific simulation objectives, approximate state matching can be used to achieve reasonable simulation results. The measure of local density deviation has been examined and found to appropriately represent microscopic state deviations.

The experiments indicate that approximate time-parallel road traffic simulation might be reasonable with low traffic densities. However, the model used in the feasibility study was rather simplistic, representing only a single-laned one-directional road segment. In practice, more complex topologies are supposed to occur, where the time-parallel simulation approach presented above is not a viable alternative. However, the goal of this chapter was not to establish an efficient parallel simulation technique for realistic large-scale traffic models, but rather to examine the extended applicability of approximate time-parallel simulation. In that respect, experiments were quite successful, as it could not be suspected that approximate time-parallel traffic simulation is feasible at all.

The contents of this chapter mainly apply to the technique of approximate state matching defined in Chapter 5. Moreover, the results presented here (especially those of the feasibility study) should be applicable to progressive time-parallel road traffic simulation, as well, although this remains an open issue.

Part IV.

Conclusions

Simulation is a time-consuming technique to investigate the properties of real-world systems. Parallel simulation approaches have been developed to decrease runtimes of computationally intensive simulation applications. Most of the existing approaches are precise methods, designed to guarantee exactly the same results in parallel and sequential simulations of the same model. Unfortunately, with precise parallelization techniques, the efficient parallelization of simulations was found to be a hard problem for various kinds of models. Therefore, imprecise (approximate) methods have been devised. These are intended to increase the efficiency of precise methods by accepting a deviation of simulation results from those of a corresponding sequential simulation, i.e. they introduce an error in the simulation results. Most of the approximate methods are variations of precise parallelization methods and can be classified either as spatial or as temporal parallelization approaches.

Spatial parallelization is performed by a decomposition of the model state space in several sub spaces to be simulated in parallel. In most cases, the parallel simulations are not independent of one another, hence communication and synchronization between processes is necessary. Depending on the specificities of the model, a large amount of communication and synchronization might be needed. Furthermore, the maximum amount of parallelism is restricted by the decomposability of the model state space, which might be a limiting factor in many cases. Using the alternative method of temporal parallelization, the simulation time is decomposed into a number of sub intervals and simulations of these time slices is performed in parallel. This has the advantage of reduced amounts of communication and synchronization and a high degree of potential parallelism. Unfortunately, time-parallel simulation is difficult to be applied to a simulation model due to the state matching problem, i.e. the problem of unknown initial states in the parallel simulation executions. Solutions of this problem exist in the form of regeneration points and fix-up computations, but the successful applications of these techniques were limited to a small number of different models. This thesis introduces novel approximate temporal parallelization methods, being designed to improve the performance of existing time-parallel simulation applications and to extend the class of models suitable for time-parallel simulation.

The core of this thesis are the definitions of the techniques of approximate state matching in Chapter 5 and of progressive simulation in Chapter 6, both being used in conjunction with time-parallel simulation with fix-up computations.

Approximate state matching considers the toleration of incorrect state changes at time interval boundaries to decrease the amount of fix-up computations. This introduces an error in the simulation results, which might seriously impact the validity of results. The quality of a time-parallel simulation model utilizing approximate state matching can be evaluated by analyzing the error due to incorrect state changes. Furthermore, if the error can be measured or at least estimated during the simulation execution, dynamic error control mechanisms can be implemented.

Progressive time-parallel simulation is very similar to basic time-parallel simulation with fix-up computations. The main difference is the continuously repeated output of simulation results during fix-up computations. Before fix-up computations are completed, simulation results are imprecise. Nevertheless, they might be valuable indications on the precise results with the advantage that they are available after a much shorter answer time. The simulation can be cancelled at any time during fix-up computations if result accuracy is high enough. In the best case, the accuracy of simulation results can be calculated along with the results themselves. In many cases, however, this is not possible and estimations on the accuracy have to be utilized. Possible application areas for this technique include simulation-based decision support and simulation-based traffic control, among others.

The feasibility of the novel techniques of approximate state matching and progressive time-parallel simulation has been examined in the context of three different applications: queuing systems in Chapter 7, cache simulation in Chapter 8, and road traffic simulation in Chapter 9.

The simple nature of single-server queuing systems allows for a concise mathematical representation of model dynamics. This enables the formal definition of basic time-parallel queuing system simulation, as well as approximate and progressive variants. In progressive queueing simulation, accuracy grows monotonically with time, with a rate depending on the input parameters of the simulation.

Simulation of caches utilizing the least recently used replacement policy was one of the first applications of the time-parallel simulation method, exhibiting a good parallel performance with simple cache simulation, where the hit rate for a single cache size is determined in a simulation run. However, scalability was shown to be limited with the practically more relevant full stack simulation approach, where the success function of an input trace is determined during a single simulation run. Approximate time-parallel simulation methods can be applied here to increase the scalability. Extensive experiments with this approach indicate a significant increase of the scalability, while introducing a modest amount of impreciseness in the simulation results.

While both of the other two applications mentioned above are known to be suited for temporal parallelization, the same is not true for road traffic simulation models. Especially in the cellular-automaton-based model chosen in this thesis, simulation state spaces tend to be highly dimensional, with state matching occurring only lately in time, or even not at all. Furthermore, in such a case, state match detection itself is an expensive operation, both in terms of processing time and memory consumption. Especially the latter problem can be avoided by utilizing approximate time-parallel simulation methods, as approximate state matching allows the definition of alternate state match criteria, which are much less expensive to be calculated. Experiments indicate the feasibility of approximate time-parallel road traffic simulation, although this depends on the average traffic density in a simulation experiment.

This thesis is a valuable contribution to the research in the field of parallel and distributed simulation. It shows how the general idea of imprecise/approximate parallelization can be used in conjunction with time-parallel simulation to improve the performance of existing models and discusses the consequences of such an approach in terms of the accuracy of simulation results. Furthermore, the novel approach of progressive simulation is proposed and it is shown how this approach can be achieved in conjunction with temporal parallelization. One of the salient features of this thesis is the extensive examination of the introduced approaches with three different case studies, performing both analytical and empirical evaluations of the approximate time-parallel simulation applications.

The technique of approximate state matching introduced in this thesis uses approximation with fix-up computations to solve the state-matching problem. However, approximate state matching could also be used with regeneration points. In many practical cases, the identification of regeneration points is a hard problem. Instead of an exact match of the current state with the regeneration point, an approximate match can be performed by checking whether the deviation between the current state and the identified regeneration point is smaller than a given tolerance. In this case, instead of regeneration points, *regeneration areas* are specified, which consist of a set of possible states that are regarded as sufficiently close to each other to be matched approximately. Although this approach seems to be similar to approximate state matching with fix-up computations, there might be differences in the nature of the error that is introduced, resulting in a need for future research in this direction.

The novel technique of progressive simulation is intended to be a general concept that could be exploited in various simulation applications, whether utilizing parallel simulation methods or not. It is shown in this thesis how progressive simulation can be implemented with time-parallel simulation. However, it might be interesting to investigate how progressive simulation could be achieved with spatial parallelization methods or even with sequential simulation models. This issue is open for future research.

Approximate time-parallel simulation is a class of parallelization methods which might significantly decrease parallel simulation runtimes at the cost of an error in simulation results. Although the approximate nature of such an approach prevents its use in contexts where a high accuracy of results is desirable, it is a valuable tool in contexts where run times are limited or quick results are desirable.

Bibliography

[1] F. Adelstein and M. Singhal. Real-time causal message ordering in multi-media systems. In *Proceedings of the 15th International Conference on Distributed Computing Systems*, pages 36–43, 1995.

[2] A. O. Allen. *Probability, Statistics, and Queueing Theory with Computer Science Applications*. Academic Press, Boston, 1990.

[3] S. Andradóttir and M. Hosseini-Nasab. Efficiency of time segmentation parallel simulation of finite markovian queueing networks. *Operations Research*, 51(2):272–280, 2003.

[4] S. Andradóttir and T. J. Ott. Time-segmentation parallel simulation of networks of queues with loss or communication blocking. *ACM Transactions on Modeling and Computer Simulation*, 5(4):269–305, 1995.

[5] O. Balci. Verification, validation and testing. In Jerry Banks, editor, *Handbook of Simulation*, pages 335–393. John Wiley & Sons, 1998.

[6] J. Banks, J. S. Carson, and B. L. Nelson. *Discrete-Event System Simulation*. Prentice Hall, Upper Saddle River, NJ, 4th edition, 2004.

[7] B. T. Bennett and V. J. Kruskal. LRU stack processing. *IBM Journal of Research and Development*, 19(4):353–357, 1975.

[8] R. Beraldi and L. Nigro. Exploiting temporal uncertainty in Time Warp simulations. In *Proceedings of the 4th IEEE International Workshop on Distributed Simulation and Real-Time Applications*, pages 39–46, 2000.

[9] R. Beraldi and L. Nigro. A time warp based on temporal uncertainty. *Transactions of the Society for Modeling and Simulation*, 18(2):60–72, 2001.

[10] L. Bergman, H. Fuchs, E. Grant, and S. Spach. Image rendering by adaptive refinement. *ACM SIGGRAPH Computer Graphics*, 20(4):29–37, 1986.

[11] D. Brade. *A Generalized Process for the Verification and Validation of Models and Simulation Results*. PhD thesis, Universität der Bundeswehr München, Fakultät für Informatik, 2003.

[12] R. E. Bryant. Simulation of packet communication architecture computer systems. Technical report, Computer Science Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1977.

[13] N. Cetin, A. Burri, and K. Nagel. A large-scale agent-based traffic microsimulation based on queue model. In *Proceedings of the 3rd Swiss Transport Research Conference*, 2003.

[14] K. Chandy and R. Sherman. Space-time and simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation*, pages 53–57, 1989.

[15] K. M. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering*, 5(5):440–452, 1978.

[16] K. M. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Communications of the ACM*, 24(4):198–205, 1981.

[17] L. Chen. Parallel simulation by multi-instruction, longest-path algorithms. *Queueing Systems*, 27(1-2):37–54, 1997.

[18] R. Chrobok, J. Wahle, and M. Schreckenberg. Traffic forecast using simulations of large scale networks. In *Proceedings of the 4th International IEEE Conference an Intelligent Transportation Systems*, pages 434–439, 2001.

[19] M. F. Cohen, S. E. Chen, J. R. Wallace, and D. P. Greenberg. A progressive refinement approach to fast radiosity image generation. *ACM SIGGRAPH Computer Graphics*, 22(4):75–84, 1988.

[20] D. E. Comer. *Essentials of Computer Architecture*. Pearson Education, Upper Saddle River, NJ, 2005.

[21] DIS Steering Committee. The DIS vision, a map to the future of distributed simulation. Institute for Simulation and Training, Orlando, FL, 1994.

[22] E. W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, 1980.

[23] S. E. Elmaghraby. The role of modeling in I.E. design. *Journal of Industrial Engineering*, XIX(6), 1968.

[24] J. Esser and M. Schreckenberg. Microscopic simulation of urban traffic based on cellular automata. *International Journal of Modern Physics C*, 8(5):1025–1036, 1997.

[25] Alois Ferscha. Parallel and distributed simulation of discrete event systems. In Albert Y. Zomaya, editor, *Handbook of Parallel and Distributed Computing*. McGraw-Hill, New York, 1995.

[26] P. A. Fishwick. *Simulation Model Design and Execution: Building Digital Worlds*. McGraw-Hill, New York, 1994.

[27] R. M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, 1990.

[28] R. M. Fujimoto. Performance of time warp under synthetic workloads. In *Proceedings of the SCS Multiconference on Distributed Simulation*, pages 23–28, 1990.

[29] R. M. Fujimoto. Exploiting temporal uncertainty in parallel and distributed simulations. In *Proceedings of the 13th Workshop on Parallel and Distributed Simulation*, pages 46–53, 1999.

[30] R. M. Fujimoto. *Parallel and Distributed Simulation Systems*. John Wiley & Sons, New York, 2000.

[31] N. H. Gatner and N. H. M. Wilson, editors. *Transportation and Traffic Theory*. Elsevier, New York, 1987.

[32] J. D. Gee, M. D. Hill, D. N. Pnevmatikatos, and A. J. Smith. Cache performance of the SPEC92 benchmark suite. *IEEE Micro*, 13(4):17–27, 1993.

[33] P. W. Glynn and P. Heidelberger. Analysis of parallel replicated simulations under a completion time constraint. *ACM Transactions on Modeling and Computer Simulation*, 1(1):3–23, 1991.

[34] A. G. Greenberg, R. E. Ladner, M. Paterson, and Z. Galil. Efficient parallel algorithms for linear recurrence computation. *Information Processing Letters*, 15(1):31–35, 1982.

[35] A. G. Greenberg, B. D. Lubachevsky, and I. Mitrani. Algorithms for unboundedly parallel simulations. *ACM Transactions on Computer Systems*, 9(3):201–221, 1991.

[36] A. G. Greenberg, B D. Lubachevsky, and I. Mitrani. Superfast parallel discrete event simulations. *ACM Transactions on Modeling and Computer Simulation*, 6(2):107–136, 1996.

[37] Object Management Group. Unified modeling language (uml) 2.0. `http://www.uml.org/`.

[38] B. Haverkort. *Performance of computer communication systems: a model-based approach*. Wiley, Chichester, 1998.

[39] P. Heidelberger. Statistical analysis of parallel simulations. In *Proceedings of the 1986 Winter Simulation Conference*, pages 290–295, 1986.

[40] P. Heidelberger and H. S. Stone. Parallel trace-driven cache simulation by time partitioning. In *Proceedings of the 1990 Winter Simulation Conference*, pages 734–737, 1990.

[41] G. W. Heiman. *Basic Statistics for the Behavioral Sciences*. Houghton Mifflin Academic, 4th edition, 2002.

[42] D. Helbing. *Verkehrsdynamik*. Springer, Berlin, 1997.

[43] D. Jackson and C. Northway, editors. *Scalable Vector Graphics (SVG) Version 1.2*. World Wide Web Consortium (W3C), April 2005. W3C Working Draft. Available at `http://www.w3.org/TR/SVG12/`.

[44] A. K. Jain. *Fundamentals of Digital Image Processing*. Prentice-Hall, Inc., 1989.

[45] D. R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Computer Systems*, 7(3):404–425, 1985.

[46] J. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, New York, 1992.

[47] T. Kiesling and S. Pohl. Time-parallel simulation with approximative state matching. In *Proceedings of the 18th Workshop on Parallel and Distributed Simulation*, pages 195–202, 2004.

[48] L. Kleinrock. *Queueing Systems*, volume 1. John Wiley & Sons, New York, 1975.

[49] N. K. Krivulin. Recursive equations based models of queueing systems. In *Proceedings of the 1994 European Simulation Symposium*, pages 252–256, 1994.

[50] C. P. Kruskal, L. Rudolph, and M. Snir. The power of parallel prefix. *IEEE Transactions on Computers*, 34(10):965–968, 1985.

[51] V. Kumar, A. Grama, and A. Gupta. *Introduction to Parallel Computing*. Pearson, Harlow, 2003.

[52] R. E. Ladner and M. J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27:831–838, 1980.

[53] A. M. Law and D. Kelton. *Simulation Modelling and Analysis*. McGraw-Hill, New York, 3rd edition, 2000.

[54] B.-S. Lee, W. Cai, and J. Zhou. A causality based time management mechanism for federated simulation. In *Proceedings of the 15th Workshop on Parallel and Distributed Simulation*, pages 83–90, 2001.

[55] J. I. Leivent and R. J. Watro. Mathematical foundations for time warp systems. *ACM Transactions on Programming Languages and Systems*, 15(5):771–794, 1993.

[56] W. Leutzbach. *Introduction to the Theory of Traffic Flow*. Springer, Berlin, 1988.

[57] Y. Lin. Parallel trace-driven simulation for packet loss in finite-buffered voice multiplexers. *Parallel Computing*, 19(2):219–228, 1993.

[58] Y. Lin and E. Lazowska. A time-division algorithm for parallel simulation. *ACM Transactions on Modeling and Computer Simulation*, 1(1):73–83, 1991.

[59] J. W. S. Liu, K.-J. Lin, W.-K. Shih, A. C.-S. Yu, J.-Y. Chung, and W. Zhao. Algorithms for scheduling imprecise computations. *Computer*, 24(5):58–68, 1991.

[60] M. L. Loper and R. M. Fujimoto. Pre-sampling as an approach for exploiting temporal uncertainty. In *Proceedings of the 14th Workshop on Parallel and Distributed Simulation*, pages 157–164, 2000.

[61] P. Martini, M. Rümekasten, and J. Tölle. Tolerant synchronization for distributed simulations of interconnected computer networks. In *Proceedings of the 11th Workshop on Parallel and Distributed Simulation*, pages 138–141, 1997.

[62] F. Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel and Distributed Computing*, 18(4):423–434, 1993.

[63] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[64] Message-Passing Interface Forum. *MPI-2.0: Extensions to the Message-Passing Interface*. MPI Forum, 1997.

[65] J. Misra. Distributed discrete-event simulation. *Computing Surveys*, 18(1):39–65, 1986.

[66] R. E. Moore, editor. *Reliability in Computing: The Role of Interval Methods in Scientific Computing*, volume 19 of *Perspectives in Computing*. Academic Press, Inc., San Diego, CA, 1988.

[67] K. Nagel and M. Rickert. Parallel implementation of the TRANSIMS micro-simulation. *Parallel Computing*, 27:1611–1639, 2001.

[68] K. Nagel and A. Schleicher. Microscopic traffic modeling on parallel high performance computers. *Parallel Computing*, 20:125–146, 1994.

[69] K. Nagel and M. Schreckenberg. A cellular automaton model for freeway traffic. *Journal de Physique I*, 2:2221–2229, 1992.

[70] K. Nagel, D. E. Wolf, P. Wagner, and P. Simon. Two-lane traffic rules for cellular automata: A systematic approach. *Physical Review E*, 58(2):1425–1437, 1998.

[71] A. Neumaier. *Interval methods for systems of equations*, volume 37 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, Cambridge, 1990.

[72] D. Nicol. Parallel discrete-event simulation of fcfs stochastic queueing networks. *SIGPLAN Notices*, 23(9):124–137, 1988.

[73] D. M. Nicol and E. Carr. Empirical study of parallel trace-driven LRU cache simulators. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, pages 166–169, 1995.

[74] D. M. Nicol and R. M. Fujimoto. Parallel simulation today. *Annals of Operations Research*, (53):249–285, 1994.

[75] D. M. Nicol, A. G. Greenberg, and B. D. Lubachevsky. Massively parallel algorithms for trace-driven cache simulations. *IEEE Transactions on Parallel and Distributed Systems*, 5(8):849–859, 1994.

[76] I. Nikolaidis, R. M. Fujimoto, and C. A. Cooper. Time-parallel simulation of cascaded statistical multiplexers. In *Proceedings of the 1994 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 213–240, 1994.

[77] Institute of Electrical and Electronics Engineers. IEEE standard for distributed interactive simulation — application protocols (IEEE std. 1278.1-1995), 1995.

[78] Institute of Electrical and Electronics Engineers. IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) - Federate Interface Specification (IEEE1516.1-2000), 2000.

[79] Institute of Electrical and Electronics Engineers. IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) - Framework and Rules (IEEE1516-2000), 2000.

[80] Institute of Electrical and Electronics Engineers. IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)-Object Model Template (OMT) Specification (IEEE1516.2-2000), 2000.

[81] F. Olken. Efficient methods for calculating the success function of fixed-space replacement policies. Technical report lbl-12370, Lawrence Berkeley Laboratory, 1981.

[82] D. Olteanu, T. Kiesling, and F. Bry. An evaluation of regular path expressions with qualifiers against XML streams. In *Proceedings of the 19th International Conference on Data Engineering*, pages 702–704, 2003.

[83] D. A. Patterson and J. L. Hennessy. *Computer Organization & Design – The Hardware/Software Interface*. Morgan Kaufmann Publishers, San Francisco, 1998.

[84] N. U. Prabhu. *Queues and Inventories*. J. Wiley & Sons, 1965.

[85] F. Quaglia and R. Baldoni. Exploiting intra-object dependencies in parallel simulation. *Information Processing Letters*, 70(3):119–125, 1999.

[86] F. Quaglia and R. Beraldi. Space uncertain simulation events: some concepts and an application to optimistic synchronization. In *Proceedings of the 18th Workshop on Parallel and Distributed Simulation*, pages 181–188, 2004.

[87] D. M. Rao, N. V. Thondugulam, R. Radhakrishnan, and P. A. Wilsey. Unsynchronized parallel discrete event simulation. In *Proceedings of the 1998 Winter Simulation Conference*, pages 1563–1570, 1998.

[88] D. A. Reed, A. D. Malony, and B. D. McCredie. Parallel discrete event simulation using shared memory. *IEEE Transactions on Software Engineering*, 14(4):541–553, 1988.

[89] V. K. Rohatgi. *An Introduction to Probability Theory and Mathematical Statistics*. John Wiley & Sons, 1976.

[90] R. Rönngren and M. Liljenstam. On event ordering in parallel discrete event simulation. In *Proceedings of the 13th Workshop on Parallel and Distributed Simulation*, pages 38–45, 1999.

[91] T. L. Saaty. *Elements of Queueing Theory with Applications*. McGraw-Hill, 1961.

[92] B. Samadi. *Distributed Simulation, Algorithms and Performance Analysis*. PhD thesis, Computer Science Department, University of California, Los Angeles, 1985.

[93] R. E. Shannon. *Systems Simulation: the Art and Science*. Prentice Hall, Englewood Cliffs, NJ, 1975.

[94] K.-L. Tan, P.-K. Eng, and B. C. Ooi. Efficient progressive skyline computation. In *Proceedings of the 27th International Conference on Very Large Data Bases*, pages 301–310, 2001.

[95] A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, Upper Saddle River, NJ, 2001.

[96] J. G. Thompson. *Efficient Analysis of Caching Systems*. PhD thesis, 1987. Also published as: UCB, CSD TR-87/374.

[97] N. V. Thondugulam, D. M. Rao, R. Radhakrishnan, and P. A. Wilsey. Relaxing causal constraints in PDES. In *Proceedings of the 13th International Parallel Processing Symposium*, pages 696–700, 1999.

[98] NMSU tracebase. New Mexico State University. Available at http://tracebase.nmsu.edu/tracebase.html.

[99] TRANSIMS – transportation analysis and simulation system. http://transims.tsasa.lanl.gov/.

[100] D. B. Wagner and E. D. Lazowska. Parallel simulation of queueing networks: Limitations and potentials. In *SIGMETRICS*, pages 146–155, 1989.

[101] J. J. Wang and M. Abrams. Approximate time-parallel simulation of queueing systems with losses. In *Proceedings of the 1992 Winter Simulation Conference*, pages 700–708, 1992.

[102] J. J. Wang and M. Abrams. Determining initial states for time-parallel simulation. In *Proceedings of the 7th Workshop on Parallel and Distributed Simulation*, pages 19–26, 1993.

[103] J. J. Wang and M. Abrams. Massively time-parallel, approximate simulation of loss queueing systems. *Annals of Operations Research*, 53:553–575, 1994.

[104] R. Wiedemann. Simulation des Straßenverkehrsflusses. In *Schriftenreihe des Instituts für Verkehrswesen*, 8. Universität Karlsruhe, Germany, 1974.

[105] S. Wolfram. *Theory and Applications of Cellular Automata*. World Scientific, Singapore, 1986.

[106] C. Woolley, D. Luebke, B. Watson, and A. Dayal. Interruptible rendering. In *Proceedings of the 2003 Symposium on Interactive 3D Graphics*, pages 143–151, 2003.

[107] H. Wu, R. M. Fujimoto, and M. Ammar. Time-parallel trace-driven simulation of CSMA/CD. In *Proceedings of the 17th Workshop on Parallel and Distributed Simulation*, pages 105–114, 2003.

[108] R. Yavatkar. MCP: A protocol for coordination and temporal synchronization in multimedia collaborative applications. In *Proceedings of the 12th International Conference on Distributed Computing Systems*, pages 606–613, 1992.

[109] B. P. Zeigler, H. Praehofer, and T. G. Kim. *Theory of Modeling and Simulation*. Academic Press, San Diego, CL, 2nd edition, 2000.

[110] S. Zhou, W. Cai, S. J. Turner, and F. B. S. Lee. Critical causality in distributed virtual environments. In *Proceedings of the 16th Workshop on Parallel and Distributed Simulation*, pages 53–59, 2002.

[111] S. Zilberstein and A.-I. Mouaddib. Reactive control of dynamic progressive processing. In *Proceedings of 16th International Joint Conference on Artificial Intelligence*, pages 1268–1273, 1999.