Distributed Event-driven Simulation- Scheduling Strategies and Resource Management

Overeinder, B.J.

Publication date
2000

Chapter 1

Introduction

"355/113 — Not the famous irrational number π,
but an incredible simulation!"

1.1 Rationale

Simulation is an everyday activity that is indispensable in society for the last decennia. Simulation spans a spectrum of activities aimed to gain more insight in the system under study. The system under study can be anything from the aerodynamic properties of a new airplane under development, to the intrinsics of protein folding into a complex, three-dimensional shape. Or the system under study is an organization or a community, for example a company that has to manage the stock logistics efficiently, or in health care where one study the variability in the spread of HIV and hepatitis C in injecting drug users.

Although the research disciplines differ, the approach to the simulation study is similar. A model of an actual or theoretical physical system is designed, experiments with the model are executed, and the results of the experiment are analyzed. Models of a physical system come in all sorts: the most exact model is the physical system itself, a scaled model in size or dimension of the physical system, an analog (physical) model, or an abstract model written in a formal notation (e.g., mathematical formulas or a formal logic language). The type of model, or level of abstraction from the physical system, also determines the method of executing experiments with the model. For example, the tidal movement of the sea in the Schelde delta in the Netherlands are extensively studied and simulated. A scaled model of the Schelde delta is designed by the Waterloopkundig Laboratorium. The scaled Schelde basin has $l \times w \times h$ dimensions of $30 \times 22.5 \times 1.2$ meter. The Schelde basin includes a wave generator that creates waves with different characteristics, and allows to study the influence of three-dimensional wave attack on structures. Instead of a scaled model of the Schelde delta, an abstract model using partial differential equations can be constructed. In general, abstract models composed of mathematical equations or formal logic notation are realized in computer programs, also called computer simulations, such that the experiments with the abstract model can be executed on computers. This is called the computer experiment. The com-

puter experiment is more flexible than experiments with the (scaled) physical system, allows exact control over the experimental conditions, and is in general more cost effective. In case of theoretical physical systems, the real-world system is not at hand for experimentation, and computer experiments are the only alternative. However, validation of the abstract model is of greatest importance: does our computer experiment describe the behavior of the real-world or theoretical physical system? In a paper by Stevenson (1999), a critical look is presented at quality in large-scale simulations.

This thesis is about computer simulation, and in particular about methods for parallel or distributed computer simulation. The increasing complexity of the real-world and theoretical systems under study, requires enormous amounts of time on the fastest available sequential computers. The challenge to reduce the so-called turnaround time of the computer simulation—that is the time required to complete the computer experiment—is approached by exploiting the parallelism that is inherent to the system, and is made available in the abstract model of the system. Exploiting the parallelism in the simulation model requires an integrated hardware-software approach. The parallel hardware architecture consisting of processing nodes interconnected by a communication network, and the parallel software that orchestrates the concurrent activities. Parallel hardware architectures are briefly discussed in this chapter. Software methods for parallel and distributed simulation are the main contribution of this thesis.

In the following sections of this introductory chapter, we present the basic issues in modeling and simulation. In particular a specific class of models is discussed, namely discrete event systems, which find more and more application in science and engineering. The parallelization of discrete event simulations is a non-trivial task. Many problems appearing in parallel and distributed computing are present in parallel discrete event simulation, e.g., non-deterministic program execution (although the behavior is deterministic), timestamp ordering and process synchronization, and distributed coordination. Apart from the simulation specific problems, the non-simulation specific opportunities and pitfalls of parallel computing are lurking, for example, data decomposition, load mapping and balancing, etc. These opportunities and pitfalls in parallel discrete event simulation are presented in the remainder of the thesis.

After the presentation of the basic issues in modeling and simulation—see Banks et al. (1999) or Zeigler et al. (2000) for a more in-depth presentation—a short exposé is given about parallel computing. Different hardware architecture design alternatives are presented, and different objectives for parallel and distributed computing are discussed. For an extensive discussion of parallel computer architectures, see Hwang (1993). In addition to the parallel computing objectives, so-called resource management strategies, i.e., the allocation of computers and network interconnections, have to be considered.

1.2 Modeling and Simulation

1.2.1 Systems and System Environment

To model a system, it is necessary to understand the concept of a system and the system constraints. We define a *system* as a group of objects that are joined together in some interaction or interdependence toward the accomplishment of some purpose. An example is a production system manufacturing automobiles. The machines, component parts, and workers operate jointly along an assembly line to produce a high-quality vehicle.

A system is often affected by changes occurring outside the system. Such changes are said to occur in the *system environment*. Depending on the purpose of the simulation study, or the experimental framework, one must decide on the boundary between the modeled system and its environment.

1.2.2 Components of a System

In order to understand and analyze a system, a number of terms are defined. An *entity* is an object of interest in the system. An *attribute* is a property of an entity. An *activity* represents a time period of specified length. If a bank is being studied, customers might be one of the entities, the balance in their checking accounts might be an attribute, and making deposits might be an activity.

We define the *state* of a system to be that collection of variables necessary to describe a system at a particular time, relative to the objectives of a study. In a study of a bank, examples of possible state variables are the number of busy tellers, the number of customers in the bank, and the time of arrival of each customer in the bank.

We categorize systems to be one of two types, discrete or continuous. A *discrete system* is one for which the state variables change instantaneously at separated points in time. A bank is an example of a discrete system, since state variables—e.g., the number of customers in the bank—change only when a customer arrives or when a customer finishes being served and departs. A *continuous system* is one for which the state variables change continuously with respect to time. An airplane moving through the air is an example of a continuous system, since state variables such as position and velocity can change continuously with respect to time. Few systems in practice are completely discrete or completely continuous, but since one type of change predominates for most systems, it will usually be possible to classify a system as being either discrete or continuous.

With respect to discrete systems, we define an *event* as an instantaneous occurrence that may change the state of the system. The term *endogenous* is used to describe activities and events occurring within a system, and the term *exogenous* is used to describe activities and events in the environment that affect the system.

1.2.3 Model of a System

A *model* is defined as a representation of a system for the purpose of studying the system. In practice, what is meant by "the system" depends on the objectives of a particular study. For most studies, it is not necessary to consider all the details of a system; thus, a model is not only a substitute for a system, it is also a simplification of the system. On the other hand, the model should be sufficiently detailed to permit valid conclusions to be drawn about the real system.

Different models of the same system may be required as the purpose of investigation changes. For example, if one wants to study a bank to determine the number of tellers needed to provide adequate service for customers who just want to cash a check or make a savings deposit, the model can be defined to be that portion of the bank consisting of the tellers and the customers waiting in line or being served. If, on the other hand, the loan officer and the safety deposit boxes are to be included, the definition of the model must be expanded in an obvious way.

Just as the components of a system were entities, attributes, and activities, models are represented similarly. However, the model contains only those components that are considered to be relevant to the study. Important qualities of a model of a system are *realizability* and *predictability* (Misra 1986). Realizability defines that the state of the model at a certain simulation time t is a function of its initial state and the changes on the state up to and including t. It says merely that a model cannot guess any future changes to the state of the system. Predictability guarantees that the system is "well defined" in the sense that the output of the model up to any time t can be computed given the initial state of the system.

1.2.4 Experimentation and Simulation

At some point in the lives of most systems, there is a need to study them to try to gain some insight into the relationships among various components, or to predict performance under some new conditions being considered. Figure 1.1 maps out different ways in which a system might be studied.

While there may be an element of truth in the famous advice of Bratley, Fox, and Schrage (1987)* that describes simulation as a "method of last resort," the fact is that we are very quickly led to simulation in many situations, due to the sheer complexity of the systems of interest and of the models necessary to represent them in a valid way.

Given, then, that we have an abstract model to be studied by means of simulation (henceforth referred to as a simulation model), we must then look for particular tools to execute this model (i.e., actual simulation). It is useful for this purpose to classify simulation models along three different dimensions:

*"The best advice to those about to embark on a very large simulation is often the same as Punch's famous advice to those about to marry: Don't!"

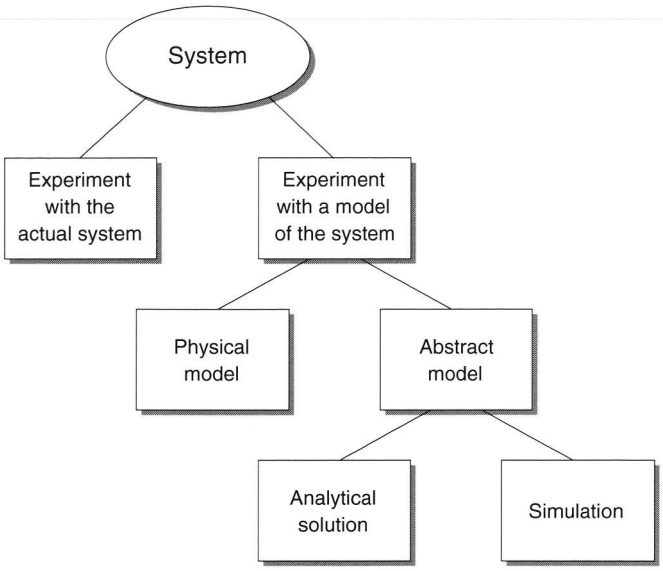


Figure 1.1: Taxonomy of methods to study a system.

Static vs. Dynamic Simulation Models A static simulation model is a representation of a system at a particular time, or one that may be used to represent a system in which time simply plays no role; examples of static simulations are Monte Carlo models. On the other hand, a dynamic simulation model represents a system as it evolves over time, such as a conveyor system in a factory.

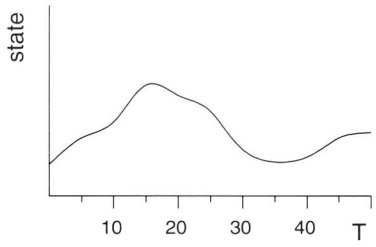
Deterministic vs. Stochastic Simulation Models If a simulation model does not contain any probabilistic (i.e., random) components, it is called deterministic; a complicated (and analytically intractable) system of differential equations describing a fluid flow might be such a model. In deterministic models, the output is "determined" once the set of input quantities and relationships in the model have been specified, even though it might take a lot of computer time to evaluate what it is. Many systems, however, must be modeled as having at least some random input components, and these give rise to stochastic simulation models. Most queueing and inventory systems are modeled stochastically. Stochastic simulation models produce output that is itself random, and must therefore be treated as only an estimate of the true characteristics of the model; this is one of the main disadvantages of such simulation.

Continuous vs. Discrete Simulation Models Loosely speaking, we define discrete and continuous simulation models analogously to the way discrete and continuous systems were defined in Section 1.2.2. It should be mentioned that

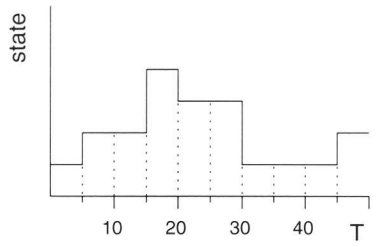
a discrete model is not always used to model a discrete system and vice versa. The decision whether to use a discrete or a continuous model for a particular system depends on the specific objectives of the study. For example, a model of traffic flow on a freeway would be discrete if the characteristics and movement of individual cars are important. Alternatively, if the cars can be treated "in the aggregate," the flow of traffic can be described by differential equations in a continuous model.

1.2.5 A Closer Look at System Models

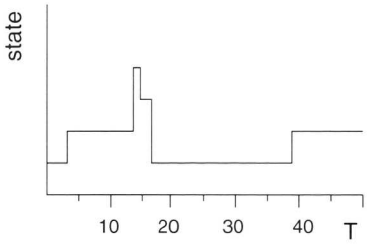
Although many attempts have been made throughout the years to categorize systems and models, no consensus has been arrived at. However, it is convenient to make the following distinction between the different models:



(a) Trajectory of a continuous-time model.



(b) Trajectory of a discrete-time model.



(c) Trajectory of a discrete event model.

Figure 1.2: Trajectory of the state vector for continuous-time, discrete-time, and discrete event models.

Continuous-Time Models Here the state of a system changes continuously over time (see Fig. 1.2(a)). These types of models are usually represented by sets of differential equations. A further subdivision would be:

- lumped parameter models expressed in ordinary differential equations (ODE's): e.g., $\dot{x} = f(x, u, t)$, where $\dot{x} = dx/dt$ is the time derivative of the systems' state x.
- distributed parameter models expressed in partial differential equations (PDE's): e.g., $\partial u/\partial t = \sigma \cdot \frac{\partial^2 u}{\partial x \, \partial y}$.

Discrete-Time Models With discrete-time models, the *time* axis is discretized (see Fig. 1.2(b)). The system state changes are commonly represented by difference equations. These types of models are typical to engineering systems and computer-controlled systems. They can also arise from discrete versions of continuous-time models. Example: $\dot{x} = f(x, u, t)$ to $f(x_k, u_k, t_k) = x_{k+1} - x_k/\Delta t$ or $x_{k+1} = x_k + \Delta t \cdot f(x_k, u_k, t_k)$. The time-step used in the discrete-time model is constant.

Discrete Event Models In discrete event models, the *state* is discretized and "jumps" in time (see Fig. 1.2(b)). Events can happen any time but only every now and then at (stochastic) time intervals. Typical examples come from "event tracing" experiments, queueing models, operations research, image restoration, combat simulation, etc.

1.2.6 Model Execution: Time-Driven versus Event-Driven

We have seen that in continuous systems the state variables change continuously with respect to time, whereas in discrete systems the state variables change instantaneously at separate points in time. Unfortunately for the computational experimenter there are but a few systems that are either completely discrete or completely continuous, although often one type dominates the other in such hybrid systems. The challenge here is to find a computational model that mimics closely the behavior of the system, specifically the simulation time-advance approach is critical.

If we take a closer look into the dynamic nature of simulation models—keeping track of the simulation time as the simulation proceeds—we can distinguish between two *time-advance* approaches: *time-driven* and *event-driven*.

Time-Driven Simulation

Continuous systems described by, for example, partial differential equations must be discretized in time and space to be solved on a computer. The execution mechanism for continuous systems is time-driven simulation, where the simulation time advances with a fixed increment. With this approach the simulation clock is advanced in increments of exactly Δt time units. Then after each update of the clock, the state variables are updated for the time interval $[t, t + \Delta t]$. This is the most widely known approach in simulation of natural systems. Less widely used is the time-driven paradigm applied to discrete systems. In this case we have specifically to consider whether:

- the time step Δt is small enough to capture every event in the discrete system. This might imply that we need to make Δt arbitrarily small, which is certainly not acceptable with respect to the computational times involved.
- the precision required can be obtained more efficiently through the event-driven execution mechanism. This primarily means that we have to trade efficiency for precision.

Event-Driven Simulation

In event-driven simulation, also called discrete event simulation, on the other hand, we have the next-event time advance approach. Here (in case of discrete systems) we have the following phases:

1. The simulation clock is initialized to zero and the times of occurrence of future events are determined.
2. The simulation clock is advanced to the time of the occurrence of the most imminent (i.e., first) of the future events.
3. The state of the system is updated to account for the fact that an event has occurred.
4. Knowledge of the times of occurrence of future events is updated and the first step is repeated.

The most important advantage of this approach is that periods of inactivity can be skipped over by jumping the clock from *event time* to the *next event time*. This is perfectly safe since—per definition—all state changes only occur at event times. Therefore causality is guaranteed.

1.2.7 World Views in Discrete Event Simulation

All discrete event simulations contain an executive routine for the management of the event calendar and simulation clock, i.e., the sequencing of events and driving of the simulation. This executive routine fetches the next scheduled event, advances the simulation clock and transfers control to the appropriate routine. The operation routines depend on the world view, and may be events, activities, or processes (Hooper 1986).

Event Scheduling In *event scheduling* each type of event has a corresponding event routine. The executive routine processes a time ordered calendar of event notices to select an event for execution. Event notices consist of a time stamp and a reference to an event routine. Event execution can schedule new events by creating an event notice and place it at the appropriate position in the calendar. The clock is always updated to the time of the next event, the one at the top of the calendar.

Activity Scanning In the *activity scanning* approach a simulation contains a list of activities, each of which is defined by two events: the start event and the completion event. Each activity contains test conditions and actions. The executive routine scans the activities for satisfied time and test conditions and executes the actions of the first selectable activity. When execution of an activity completes, the scan begins again.

Process Interaction The *process interaction* world view focuses on the flow of entities through a model. This strategy views systems as sets of concurrent, interacting processes. The behavior of each class of entities during its lifetime is described by a process class. Process classes can have multiple entries and exits at which a process interacts with its environment. The executive routine uses a calendar to keep track of forthcoming tasks. However, apart from recording activation time and process identity, the executive routine must also remember the state in which the process was last suspended.

1.3 Parallel Computing

1.3.1 Parallel Architectures

Parallel processing, the method of having many tasks solve one large problem, has emerged as an enabling technology in modern computing. The past several years have witnessed an ever-increasing acceptance and adoption of parallel processing, both for high-performance scientific computing and for more "general-purpose" applications, as a result of the demand for higher performance, lower cost, and sustained productivity. The acceptance has been facilitated by two major developments: massively parallel processors (MPPs) and the widespread use of cluster and distributed computing.

Distributed versus Shared Memory

Parallel computers consists of three building blocks: processors, memory modules, and an interconnection network. There has been a steady development of the sophistication of each of these building blocks but it is their arrangement that is the most important factor to differentiate one parallel computer from another. The *processor* used in current parallel computers are exactly the same as processors used in single-processor systems. An exemplary illustration of the use of custom processors in parallel computers is the ASCI initiative (Clark 1998). Within the ASCI project (the US Department of Energy's Accelerated Strategic Computing Initiative), a number of parallel computers are designed and constructed that must scale the performance curve to achieve 100 Tflops ($100 \cdot 10^{12}$ floating point operations per second) by the year of 2004. The ASCI machines Red, Blue Pacific, Blue Mountain, and White use respectively the Pentium Pro, PowerPC, MIPS, and POWER3-II microprocessors.

The *interconnection network* connects the processors to each other, and sometimes to memory modules as well. The major distinction between the so-called distributed-memory and shared-memory variants of parallel computer architectures is whether each processor has its own local memory, or whether the interconnection network connects all processors to one shared global memory. These alternatives are called *distributed-memory* and *shared-memory* parallel architectures respectively, and are illustrated in Fig. 1.3.

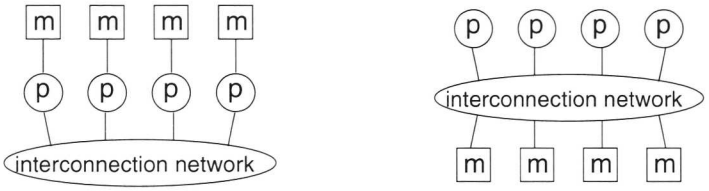


Figure 1.3: Distributed-memory and shared-memory parallel architectures.

The *memory organization model* in shared-memory parallel computers is categorized in uniform memory access (UMA) and nonuniform-memory-access (NUMA). In a UMA parallel computer, the physical memory is uniformly shared by all the processors. All processors have equal access time to all memory words, which is why it is called uniform memory access. A NUMA parallel computer is a shared-memory system in which the access time varies with the location of the memory word. The shared memory is physically distributed to all processors, called local memories. The collection of local memories forms a global address space accessible by all processors.

A distributed-memory parallel computer consists of multiple computers, often called nodes, interconnected by a message-passing network. Each node is an autonomous computer consisting of a processor, local memory, and sometimes attached disks or I/O peripherals. All local memories are private and are accessible only by local processors. However, this restriction can be alleviated by providing distributed shared memories. Internode communication is carried out by passing messages through the interconnection network.

Shared-memory systems offer the advantage of much easier programming. Building massively parallel shared-memory systems that also scale is extremely difficult however. Therefore most successful MPP architectures are distributed-memory systems.

MPP versus Cluster Computing

Massively parallel processors (MPPs) are now the most powerful computers in the world. These machines combine a few hundred to a few thousand CPUs in a single large cabinet connected to hundreds of gigabytes, or even terabytes of memory (Clark 1998). MPPs offer enormous computational power and are used to solve computational "grand challenge" problems such as global climate modeling, nuclear tests simulation, and drug design (Larzelere II 1998; Clark

2000). As simulations become more realistic, the computational power required to produce them grows rapidly. Thus, researchers on the cutting edge turn to MPPs and parallel processing in order to get the most computational power possible.

The second major development affecting scientific problem solving is *distributed computing*. Distributed computing is a process whereby a set of computers connected by a network are used collectively to solve a single large problem. As more and more organizations have high-speed local area networks interconnecting many general-purpose workstations, the combined computational resources may exceed the power of a single high-performance computer. In some cases, several MPPs have been combined using distributed computing to produce unequaled computational power. The introduction of faster interconnection networks, such as Gigabit Ethernet, HIPPI, Myrinet, or SONET/ATM high-speed networks, has resulted into so-called cluster computing. Here the distinction between MPPs and distributed computing almost disappears for the users.

The most important factor in the take up of day to day distributed computing is cost. Large MPPs typically cost more than $10 million. In contrast, users see very little cost in running their problems on a local set of existing computers. It is uncommon for distributed-computing users to realize the raw computational power of a large MPP, but they are able to solve problems several times larger than they could using one of their local computers.

1.3.2 Resource Management: Scheduling and Load Balancing

Essential to parallel computing is the efficient use of the resources, i.e., the components that make up the parallel computer such as the processors, the interconnection network, and the I/O subsystem. The allotment of resources to (parallel) applications is called *resource management*. Resource management can be directed by different strategies or objectives, such as resource utilization, fast turnaround times, and fair use of resources. There is a vast amount of literature on resource management, and some starting points can be found in Nagel and Williams (1998) and Błażewicz et al. (2000). In this section we will discuss two important resource management strategies in parallel and distributed computing, namely *high performance* and *high throughput* computing.

In the following discussion on scheduling and load balancing, we will use the terms job, process, and task to make a hierarchical distinction between the components of a sequential or parallel program. A *job* is the execution of a sequential or parallel program, and can be composed of one or more processes. A *process*, or *task*, is a logical processor executing sequential instructions and has its own state and data.

High Performance versus High Throughput Computing

The difference between high performance computing (HPC) and high throughput computing (HTC) is best illustrated by two typical (simulation) applications requiring vast amounts of computing resources (processing time), and are therefore executed on a parallel or distributed computer:

- Iteratively improve or optimize a parameter estimate in a loop of simulation runs, requiring the simulation turnaround time to be as fast as possible.
- Statistical parameter study requiring a whole series of simulation runs with identical input parameters or small changes in the parameters.

The first simulation type is a high performance computing application, where the resource management allocates resources to the application in order to minimize the turnaround time. As a consequence, resources might not be fully utilized throughout the simulation execution run. For the second simulation type, the resource management follows a different strategy. In the statistical parameter study, hundreds to thousands of independent simulation runs are necessary in order to collect sufficient data for statistical analysis. These simulation runs can be sequential or parallel execution runs, but the main resource management concern is to dispatch as much simulation runs per time unit as possible. Not the individual turnaround times of the simulation runs are important, but the number of completed jobs per time unit, thus falling into the high throughput computing category.

Scheduling and Load Balancing

One of the biggest issues in resource management is the development of effective techniques for the distribution of processes of a sequential or parallel program on multiple processors. The problem is how to distribute, or schedule, the processes among processing elements to achieve some performance goals, as discussed in the previous section.

Process scheduling methods are typically classified into several subcategories. Local scheduling performed by the operating system of a processor consists of the assignment of processes to the time-slices of the processor. Global scheduling, on the other hand, is the process of deciding where to execute a process in a parallel or distributed computer. In this discussion, global scheduling methods are classified into two major groups: *static scheduling* and dynamic scheduling (often referred to as *dynamic load balancing*).

In scheduling, the assignment of tasks to processors is done before task execution begins. Information regarding task execution times and processing resources is assumed to be known at execution time. A task is always executed on the processor to which it is assigned. Typically, the goal of scheduling methods is to minimize to overall execution time of a concurrent program while minimizing the communication delays.

Dynamic load balancing is based on the redistribution of processes among the processors during execution time. This redistribution is performed by transferring processes from the heavily loaded processors to the lightly loaded processors (called load balancing) with the aim of improving the performance of the application.

1.4 Problems and Challenges

In scientific computing, much attention has been paid to continuous-time simulation and Monte Carlo simulation, while discrete event simulation was traditionally applied in the fields of computer science, operations research and management science. In recent years, there is an increasing interest in the application of discrete event simulation to solve problems from natural sciences. In particular, problems with heterogeneous spatial and temporal behavior are, in general, most exactly mapped to asynchronous models. The interest in discrete event simulation now is motivated by the ability of this protocol to capture the asynchronous behavior that is a qualifying characteristic of these models. Also with the development of new paradigms and methodologies to solve scientific problems, discrete event simulation becomes an effective execution model for this class of problems.

Although event-driven simulation is considered an expensive execution model, in contrast to time-driven execution model used in continuous-time simulation, it is an efficient execution model for the class of asynchronous problems. Besides the aspect of asynchrony, a general tendency is the construction of more realistic models resulting in more complex and larger simulations, which require vast amounts of execution time. One fundamental method to reduce the execution time of large discrete event simulations is the exploitation of parallelism inherent in this class of simulations (Berry and Jefferson 1985; Livny 1985).

Another application area that revitalized the interest in parallel and distributed simulation in recent years, are *virtual environments* into which humans or devices are embedded. A multiple user virtual environment simulation widely used by the military today is for example battlefield training exercises. A variation on this theme is to embed into the virtual environment actual physical components, possibly in addition to human participants. This is often used to test the component, for example, to test the operability of the fire department in case of a large blaze near an oil refinery. The the U.S. Department of Defense developed a standardized framework called High Level Architecture (HLA) for this type of "mock-up" simulations (Defense Modeling and Simulation Office 1999). From a technical standpoint, HLA is important because it provides a single architecture that spans both the parallel and distributed simulations and virtual environments (Fujimoto 2000).

Our primary interest is high performance computing in scientific simulation, thus reducing the turnaround times of large scientific simulations. Our effort to reduce the turnaround times of large simulations is divided into two

strategies: *scheduling* and *load balancing*.

First, we study the parallel scheduling of events with the Time Warp parallel simulation method (Jefferson 1985). Most research in parallel and distributed discrete event simulation is focused on protocol design; and although there are encouraging advances, none of the protocols devised thus far have been shown to perform efficiently under different conditions (resulting from, for example, dynamic execution behavior and available resources). We have put our research effort into parallel simulation methods in perspective of the projected use for large, data-intensive scientific simulations. This resulted in the adaption and extension of the Time Warp method to provide efficient support for this class of simulations. In general, the execution behavior of parallel discrete event simulation is extremely complex by its asynchronous, nondeterministic concurrent characteristics. As a detailed knowledge of the execution behavior of parallel simulation methods is essential for the successful application in parallel computing, an environment to study this execution behavior to extract comprehensive information is designed.

Second, we study dynamic load balancing techniques for parallel simulation on clusters of workstations. The use of clusters of workstations becomes in a progressively increasing extent a parallel platform to solve large scientific problems. The shared resources in a cluster need an adaptive resource manager that is able to redistribute the computational load as resources become available or unavailable. Furthermore, the effectivity of the Time Warp parallel simulation method is to some extent dependent on the load balance. If the parallel Time Warp simulation computation experiences load imbalance, the performance of the simulation is not only hampered by the fact that some of the processors have more (useful) work to do than other processors, but also the Time Warp method overhead increases significantly, resulting in a severe performance drop. Hence, an adaptive runtime support environment that is able to deal with the dynamic availability of resources and the dynamic behavior of the simulation application is a valuable facility in providing a parallel simulation environment for clusters of computers.

1.5 Outline of Thesis

The thesis is composed of two parts. The first part reports on the research in parallel discrete event simulation methods as well as on the application of these methods. The second part presents the study on dynamic load balancing facilities for parallel programs, including an experimental assessment of the load balancing environment.

Part I, Chapter 2 starts with an overview of parallel discrete event simulation (PDES) methods and the current status in PDES research. The fundamental problem in PDES is formulated, and two basic approaches to solve this problem are presented. The two approaches are known as conservative methods and optimistic methods. For both basic approaches, a number of methods and extensions to these methods have been developed. In particular, optimistic

simulation methods are an active research field. The optimistic simulation methods have been used in various application areas, all with their specific requirements. These requirements has resulted in a number of extensions to the basic optimistic simulation method, making this method generally applicable to a large class of simulation problems.

The design and implementation of an optimistic simulation environment called APSIS is described in Chapter 3. The APSIS simulation environment incorporates the Time Warp optimistic simulation method, and is designed for data-intensive, scientific simulation applications. This simulation class imposes a number of design constraints that resulted in a number of extensions to the basic Time Warp optimistic simulation method. In particular, special considerations are taken for efficient data structures for the event lists and the dynamic generation and retraction of simulation events. Furthermore, effective memory management support, which is essential for data-intensive simulations, is studied.

Together with the APSIS simulation environment, the APSE parallelism analysis environment is developed. The parallel simulation execution analysis methods incorporated in the APSE environment are described in Chapter 4. The theoretical space-time model used by the analysis method is discussed, and the analysis techniques working on this space-time model are explained. The analysis method is very general, and can be applied to message-passing parallel programs, including object-oriented programs. The results of the APSE analysis can be used to assess the available parallelism that is inherent to the simulation. This inherent parallelism is a yardstick to compare the amount of parallelism that is realized, or effectively used by the Time Warp optimistic simulation method with. Chapter 4 concludes with an experimental validation and assessment of the APSIS/APSE environment for a number of relative simple queueing simulations. The execution behavior of the queueing simulations are well-understood and hence experimental APSE results can be verified with the theoretically expected parallel execution behavior.

Chapter 5 presents an extensive case study of the application of the Time Warp simulation method to asynchronous cellular automata. The concept of asynchronous cellular automata is a general solving methodology for a large class of data-intensive, scientific applications. The asynchronous cellular automata application instance used to experiment with the APSIS/APSE environment is the continuous-time Ising spin system. The Ising spin system is a fairly simple model, but exhibits complex spatio-temporal behavior. As such, the Ising spin model is an excellent application to assess the APSIS/APSE environment. The critical behavior of the Ising spin system also influences the execution behavior of the Time Warp simulation method in an unexpected, but explainable, manner.

The critical behavior of the Ising spin system and its influence on the execution behavior of the Time Warp method is further investigated in Chapter 6. Critical phenomena as observed in Ising spin systems are characterized by infinite correlation lengths (or in finite systems, by system-size correlation lengths). Particular these critical phenomena seem to influence the execution

behavior of the Time Warp method. For example, if we consider the turnaround times of the Ising spin system simulation with Time Warp, we observe a certain "constant" turnaround time if the Ising spin system is at a certain distance from the critical point. However, around the critical point in the Ising spin system, the Time Warp simulation turnaround times scale in a non-trivial way. Chapter 6 studies this remarkable phenomena, and relates this behavior to so-called self-organized criticality discovered in the Time Warp method.

Clusters of workstations, but also distributed computers interconnected by wide-area networks, are used as an alternative parallel computing platform for HPC or HTC applications, including parallel simulations. But the dynamic availability of the computing resources in such clusters necessitates an adaptive runtime support system that allows the migration of computational work from overloaded to idle resources, or from relocated resources to available resources. Part II, Chapter 7 first introduces the concepts of resource management and load balancing. Next, our research contribution to adaptive runtime support systems and task migration is presented. The task migration facilities are incorporated into the well-known PVM message-passing environment. The adaptive runtime support system is called Dynamite. The dynamic task migration facility of the Dynamite environment is evaluated by a series of experiments. The experiments are accomplished with the NAS Parallel Benchmark kernels and the GRAIL finite-element model simulation. Dynamic load balancing of optimistic parallel Time Warp simulations need extra research effort, as load balance or imbalance also influences the execution behavior of the Time Warp simulation method. Experiments with optimistic parallel simulation using the Time Warp method are not yet accomplished at the date of this writing, and are part of ongoing research.