Distributed Event-driven Simulation

Scheduling Strategies and Resource Management

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam
op gezag van de Rector Magnificus

prof. dr J.J.M. Franse

ten overstaan van een door het college voor promoties ingestelde
commissie, in het openbaar te verdedigen in de Aula der Universiteit
op donderdag 9 november 2000 te 12:00 uur

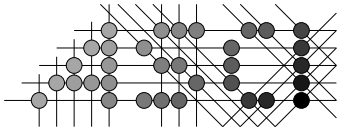
door

Benno Jaap Overeinder

geboren te Purmerend

Promotor:	prof. dr P.M.A. Sloot	(Universiteit van Amsterdam)
Co-promotor:	prof. dr M. Livny	(University of Wisconsin–Madison)
Commissie:	prof. dr ir H.E. Bal	(Vrije Universiteit Amsterdam)
	prof. dr L.O. Hertzberger	(Universiteit van Amsterdam)
	prof. dr W.G. Vree	(Technische Universiteit van Delft)
	prof. drs M. Boasson	(Universiteit van Amsterdam)
	dr M. Bubak	(University of Mining & Metallurgy, Krakow)

Faculteit: Natuurwetenschappen, Wiskunde en Informatica
Kruislaan 403
1098 SJ Amsterdam
The Netherlands



Advanced School for Computing and Imaging

The work that is described in this thesis has been carried out in the graduate school ASCI, and was financially supported by:

- Universiteit van Amsterdam
- Power Computing & Communications UvA B.V.
- Nederlandse organisatie voor Wetenschappelijk Onderzoek (NWO), under the MPR project “Parallel I/O and Imaging” (93MPR01)
- Commission of European Communities

ISBN 90-5776-058-4
ASCI dissertation series number 58.

Cover illustration: Salvador Dalí, The Soft Watch, 1950.
Printed at PrintPartners Ipskamp, Enschede, The Netherlands.

Contents

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!"

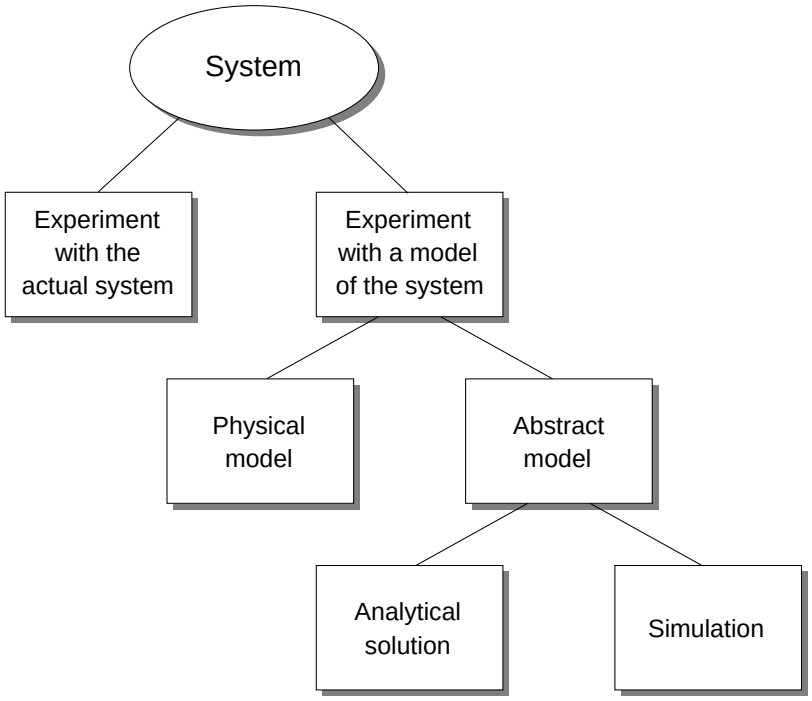


Figure 1.1: Taxonomy of methods to study a system.

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.

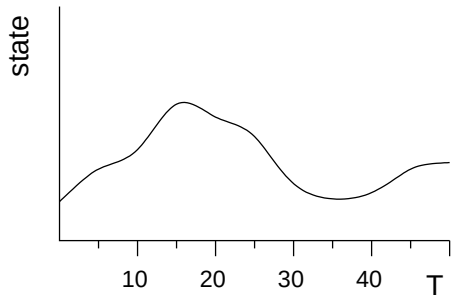
Deterministic vs. Stochastic Simulation Models If a simulation model does not contain any probabilistic (i.e., random) components, it is called deterministic; a complicated (and analytically intractable) system of differential equations describing a fluid flow might be such a model. In deterministic 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

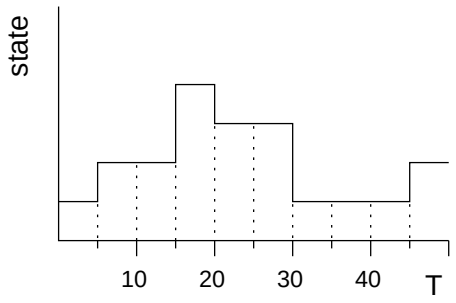
a discrete model is not always used to model a discrete system and vice versa. The decision whether to use a discrete or a continuous model for a particular system depends on the specific objectives of the study. For example, a model of traffic flow on a freeway would be discrete if the characteristics 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

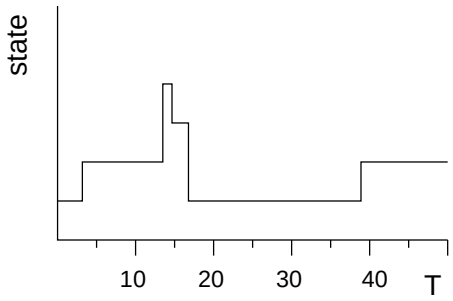
Although many attempts have been made throughout the years to categorize systems and models, no consensus has been arrived at. However, it is convenient to make the following distinction between the different models:



(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.

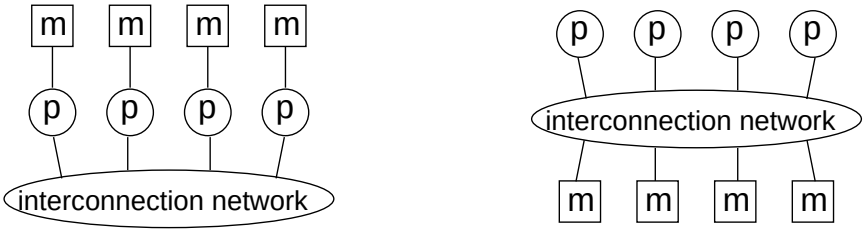


Figure 1.3: Distributed-memory and shared-memory parallel architectures.

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.

Part I

Scheduling Strategies

Chapter 2

Issues in Parallel Discrete Event Simulation

> People like us, who believe in physics, know that the distinction between the past, present and future is only a stubbornly persistent illusion.
>
> —Albert Einstein

2.1 Introduction

With the increasing complexity of the world in which we live, scientists and engineers devise simulation models that predict the complexity but require enormous amounts of time on the fastest available sequential machines. New applications that utilize all available computational resources outstrip the steady performance improvements of sequential machines, as the systems we envision are just one step bigger and more sophisticated that the current systems.

Many scientific, engineering, military, and economics projects depend heavily on simulation and the results from these simulation are often on the time-critical path of the project. One basic approach to reduce the required simulation time is the exploitation of parallelism (for discussion of high-performance computing versus high-throughput computing see Section 1.3.2). If the simulation problem is extremely regular, a time-driven approach is reasonable where the different parts of the simulation are executed synchronously in simulated time. However, for discrete event systems with a highly irregular temporal behavior, the amount of work that can be performed concurrently at a certain point in simulated time is marginal. This implies that with a synchronous time-driven execution mechanism, only a limited number of concurrent activities in the simulated system can be exploited by parallel processing. Therefore, parallel discrete event simulation methods have been developed to exploit the available parallelism in the discrete event system. Just until recently, parallel discrete event simulation was merely an academic research topic, but with the current available high-performance hardware and software platforms, a true revival of the use of parallel discrete event simulation in industrial and military application areas can be observed. Typical examples are (mobile) com-

munication network simulation (Bhatt et al. 1998) and combat simulation (Fujimoto 1998; Smith 1998).

Although discrete event simulation contains a substantial amount of parallelism, the parallelization is very difficult in practice. As in asynchronous discrete event systems few events occur at a single point in simulation time, the concurrent execution of events at different points in simulated time is required. A major drawback is the inherent complexity of this type of simulation, since the notion of a global clock that synchronizes the events is released. The absence of a global clock necessitates sophisticated synchronization algorithms to ensure that cause-and-effect relationships are correctly reproduced by the simulator. The synchronization algorithm that is essentially concerned with the correct ordering, or scheduling, of asynchronous execution of events over the distributed or parallel system is the heart of the parallel discrete event simulation problem. There are basically two methods to impose the correct temporal order of asynchronous event execution: *conservative* and *optimistic* methods. Alternative methods have been proposed but can be reduced to these two.

The conservative approach proposed by Chandy and Misra (1979, 1981), and independently by Bryant (1977), strictly imposes the correct temporal order of the events. The optimistic approach, introduced by Jefferson (1985), allows the cause-and-effect relationship to be broken but uses a detection and recovery mechanism: whenever the incorrect temporal order of events is detected a rollback mechanism is invoked to recover. Both approaches have limited scope of applicability.

This chapter presents a survey of the issues in parallel discrete event simulation, and in particular details on topics related to the optimistic simulation method. The comprehensive survey of the optimistic simulation method establishes the basis for the design considerations and decisions of the parallel simulation environment presented in Chapter 3 and Chapter 4, and the interpretation and analysis of the computer experiments presented in Chapter 5. In the following sections, we introduce the basic concepts used in parallel discrete event simulation for both the conservative and the optimistic methods. Hereafter, the conservative and optimistic methods themselves are presented, including extensions to the basic methods. Due to the complexity of the execution methods for parallel discrete event simulation it has been recognized that a structured categorization is essential (Fujimoto 1990a; Overeinder, Hertzberger, and Sloot 1991; Ferscha and Tripathi 1994; Nicol and Fujimoto 1994). The chapter concludes with a discussion of the applicability of the methods to certain classes of problems and which method offers the greatest potential as a general purpose simulation mechanism.

2.2 Basic Concepts

2.2.1 Need for Logical Processes

Asynchronous Parallel Discrete Event Simulation (PDES) strategies typically aim to decompose the simulation application into a set of concurrently executing processes, trying to exploit the parallelism inherent in the respective model components. The application system being modeled can be viewed as composed of some *physical processes*, PP_0, PP_1,..., that interact at various arbitrary points in simulated time.

The simulation model is constructed as a set of *logical processes* LP_0, LP_1,..., one per physical process. The principle difference between physical processes and logical processes is that the latter are mathematical or logical abstractions of the "real-world" physical processes. Each logical process LP_i such defined, contains a portion of the state of the system being simulated corresponding to the physical process it models, as well as a local clock that denotes the progress of this process. As a consequence, the subset of the state variables S_i is unique to a specific logical process LP_i: the state variables are not shared, i.e., $S_i \cap S_j = \emptyset$ $(i \neq j)$ for $S = \{S_1, S_2, \ldots\}$. In this context, it is possible to consider in each logical process LP_i two kinds of events: *internal events* and *external events*. The internal events only have causal effects on the local state variables S_i associated with LP_i. The external events also affect the state variables in subsets $S_j \subset S$ $(i \neq j)$ associated with other logical processes. The interaction between logical processes takes place through external events.

Now we come to one of the most fundamental concepts in discrete event simulation, namely the notion of *timestamp*. Every event (internal and external) has both a spatial coordinate, the LP_i where the event is scheduled, and a temporal coordinate, the timestamp determining when the event must be executed. Events are communicated by exchanging messages, and any logical process is free to send an event message to any logical process (including itself). Hence, the spatial coordinate of the event is specified by the destination of the message. The temporal coordinate however must be tagged explicitly to the event message by means of a timestamp indicating the clock value at which the event must be executed.

Consider for example three airports: Amsterdam, London, and Paris. We are interested in the arrival/departure throughput of airplanes of the individual airports and the direct influence the airports have on each other. Here, the airports are considered to be physical processes. The combination departure/arrival of an airplane between two of the airports is a point of interaction in simulated time. Note that the airplanes are not modeled as physical processes. The experimental frame (see Section 1.2) only considers the arrival/departure throughput of the airports.

In the simulation, the airports are implemented as logical processes: a one-to-one mapping is obtained of physical processes to logical processes, including a notion of time unique to the subset of state variables that it is responsible for (see Fig. 2.1). The airports of Amsterdam, London, and Paris are represented

by the logical processes LP_A, LP_L, and LP_P respectively. Each logical process LP_i, $i \in \{A, L, P\}$, contains a subset of the state variables S_i and a local clock T_i. The departure and the resulting arrival of an airplane is modeled by a time stamped event message that is send by the airport where the plane departs from, to the airport where the plane will arrive. In this example, the departure event is an internal event, i.e., only has causal effect on the local state variables; and the arrival event is an external event, i.e., affects the state variables associated with other logical processes. In the following discussion, the timestamp associated with a departure event is the departure time, and analogous, the timestamp associated with an arrival event is the expected arrival time of the airplane.

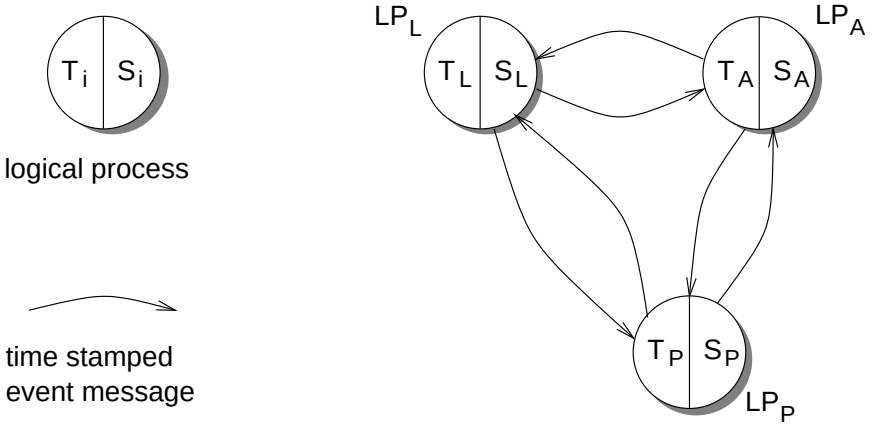


Figure 2.1: Logical processes representing the airport model. The arcs between the logical processes denote the exchange of time stamped event messages.

2.2.2 The Curse of Causality

The decomposition of the application into concurrently executing processes, introduces a complication that is the core problem all PDES strategies are trying to solve. The problem becomes clear if one examines the operation of a sequential discrete event simulator. The sequential simulator typically uses three data structures: the state variables, a future event list (or the calendar), and a global simulation clock. For the execution routine it is crucial that the earliest time stamped event (E_{min}) from the future event list is selected as the one to be processed next. If the execution mechanism would depart from this rule and select another event with a larger timestamp (E_j), it would be possible for E_j to change the state variables used by E_{min}. This implies that one is simulating a system where the future can affect the past. We call errors of this kind *causality errors*.

With the distributed execution of the simulation application, the correct causality between events must be assured. The distributed execution protocol

consists of logical processes that interact by exchanging time stamped messages. Each logical process stores incoming messages in a future event list for further processing. One can assure that no causality error occurs if each logical process adheres to the following local causality constraint:

Definition 2.1

Local Causality Constraint *A discrete event simulation, consisting of logical processes that interact exclusively by exchanging time stamped messages, obeys the local causality constraint* if and only if *each logical process executes events in nondecreasing timestamp order.*

To illustrate the causality problem, consider the following example. The two LPs representing Amsterdam and London have both scheduled an event: the departure event E_1 at logical process LP_A with timestamp 21:30*, and the departure event E_2 at LP_L with timestamp 23:15 (see Fig. 2.2(a)). Suppose now that the execution of E_1 schedules an arrival event E_3 for LP_L containing a timestamp less than 23:15, for example timestamp 22:30 (flight from Amsterdam to London takes approximately one hour) (Fig. 2.2(b)), then E_3 could affect E_2, necessitating sequential execution of all three events. If one had no information what events could be scheduled by other events, one would be enforced to process the only safe event: the one containing globally the smallest timestamp. This abolishes all parallelism and results in a sequential execution.

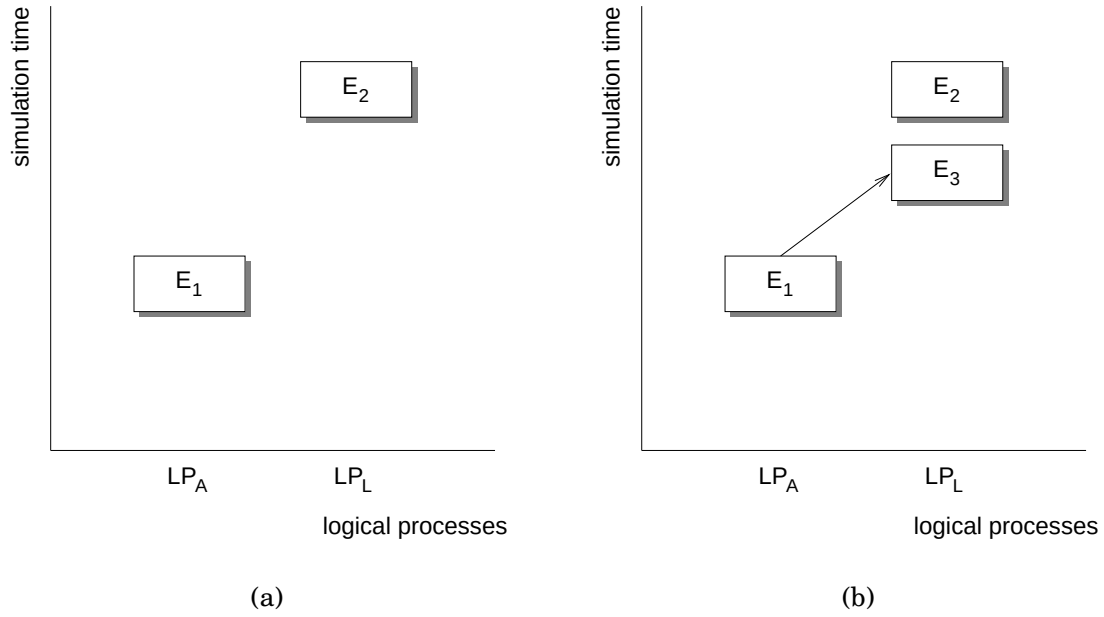


Figure 2.2: The potential causality error.

During the simulation we must therefore decide whether E_1 can be executed concurrently with E_2. But how do we know whether or not E_1 affects E_2 without

*All times are GMT.

actually performing the simulation for E_1? It is this fundamental problem the parallel discrete event simulation strategies must address.

We can classify the parallel discrete event simulation strategies by their basic approach to solve the fundamental problem, and this results in two distinct categories: *conservative* or *optimistic* methodologies. Conservative approaches strictly avoid the possibility of any causality error ever occurring. These approaches rely on a protocol to determine when it is safe to process an event. The optimistic approaches allow causality errors to occur, but use a detection and rollback mechanism to recover. In the next sections, we will describe some of the concepts and the functional behavior of conservative and optimistic simulation mechanisms.

2.3 Conservative Methods

Historically, the conservative approaches were the first distributed simulation mechanisms (Bryant 1977; Chandy and Misra 1979). The basic problem conservative mechanisms must address is to determine which event is safe to process. If a process contains an event E_1 with timestamp T_1 and the process can determine that is impossible to receive an event with a smaller timestamp, then the process can safely execute event E_1 without future violation of the local causality constraint. Processes that do not contain any safe event must block. This behavior can result in deadlock situations if no appropriate precautions are taken.

Conservative parallel discrete event simulation algorithms statically specify the links that indicate which process may communicate with which other processes. Each link has a clock associated with it that is equal to either the timestamp of the message at the front of that link's queue or, if the queue is empty, the time of the last received message. In order to determine when it is safe to process a message, two conditions are required: (i) messages from any process to any other process are transmitted in chronological order according their timestamps, and (ii) the communication link preserves the order of messages sent (FIFO). The process repeatedly selects the link with the smallest clock and, if there is a message in that link's queue, updates its local clock to the link's clock and processes the message. The order of event processing will be correct because all future messages received will have later timestamps than the local clock, since they will arrive in chronological order along each link. If the selected queue is empty, the process blocks. This is because the process may receive a message over this link with a time that is less than all the other input timestamps. Thus to insure correct order, the process is forced to wait for a message to update the clock on the link before the process can update its local clock. This protocol makes certain that each process will only process events in nondecreasing timestamp order, and thereby ensuring the chronological integrity.

Deadlock occurs when there is a cycle of blocked processes and each process is blocked due to another process in the cycle. For example consider the net-

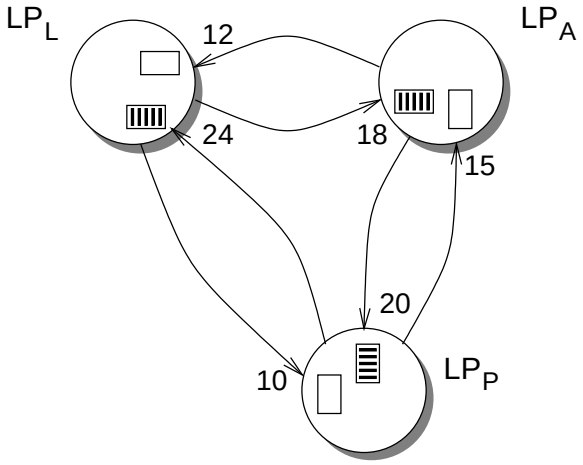


Figure 2.3: An example of a deadlock situation.

work of Fig. 2.3. Each process is waiting on the incoming link with the smallest clock value because the associated message queue is empty (the counterclockwise cycle in the figure). And although there are event messages in other input queues, the three processes are blocked due to another process in the cycle.

Several methods have been proposed to overcome the vulnerability of conservative approaches to deadlock, falling into two categories: *deadlock avoidance* and *deadlock detection and recovery*.

2.3.1 Deadlock Avoidance

The deadlock avoidance method introduces a new type of message to be used in the simulation: the *null message*. A null message with timestamp T_{null} sent by LP_i to LP_j indicates that there will be no more messages from process LP_i with timestamp less than T_{null}. Clearly, null messages have no counterpart in the physical system: a null message is an announcement of the absence of messages.

The operation of the deadlock avoidance method is a slightly modified version of the basic conservative algorithm. Whenever LP_i receives a safe event message E_i with timestamp T_i, the logical process updates its local clock accordingly and advances the simulation up to time T_i. At this point LP_i generates and sends the event messages that results from processing the event E_i. Suppose that LP_i can predict it will not send any more messages to LP_j with timestamp smaller than T_j (where $T_j \geq T_i$). Then, in the new scheme, LP_i sends a null message with timestamp T_j to LP_j to advance the clock on the link. As LP_i has progressed to simulation time T_i it can predict all event messages and the absence of messages at least up to T_i. Consequently, all outgoing links will have a clock value equal to or greater than T_i.

Reception of a null message is identical to receiving any other message. The clock value associated with the input link is updated, and possibly the local

clock value of the logical process (if the input link has the smallest clock value). The logical process can then predict the new time bounds on its outgoing links, send this information to its neighbors, and so on.

To demonstrate how the deadlock is avoided in Fig. 2.3 consider the following scenario. Assume that all LPs can predict future events minimal up to $T_i + 4$, independent from other factors that can account for future events with even larger timestamp. The local clock of LP_A is 15 (the smallest clock value of the incoming links). LP_A predicts that no event message with timestamp less than 19 $(15 + 4)$ will be sent to any other LP, and informs its neighbors by sending a null message with timestamp 19. (Note that only LP_L is informed with a null message, as the outgoing link to LP_P already reached clock value 20. See also Table 2.1.) LP_L receives the null message, updates its local clock value and computes the new lower bound on its outgoing links, and sends the null messages with timestamp 23 $(19 + 4)$ to its neighbors. Eventually, the null message with timestamp 23 arrives at LP_P, and the associated clock value of the incoming link is updated from 10 to 23. LP_P can now safely process the *genuine* event message scheduled for simulation time 20, and simulation execution proceeds.

LVT	LP_A	LP_L	LP_P
15	$(19, null) \rightarrow LP_L$		
19		$LP_L \leftarrow (19, null)$	
		$(23, null) \rightarrow LP_A$	
		$(23, null) \rightarrow LP_P$	
20			$LP_P \leftarrow (20, m)$

Table 2.1: Null message transmissions and receipts.

The deadlock avoidance scheme requires that there is a strictly positive lower bound on the *predictability* for at least one logical process in each cycle. In other words, if a null message with timestamp T is circulated through a cycle of logical processes, then after one full circulation the timestamp of the null message should be incremented to at least $T + \epsilon$, where $\epsilon > 0$. From the predictability property (Section 1.2.3) it follows that such an $\epsilon > 0$ exists if the system model is *well-defined*. After a finite number of circulations of null messages through the cycle, a safe event message will be scheduled and the simulation can continue. A more rigorous proof may be found in (Chandy and Misra 1979).

2.3.2 Deadlock Detection and Recovery

Chandy and Misra (1981) also presented a two-phase scheme where the distributed simulation is allowed to deadlock, but provisions are made to detect and resolve the deadlock. In the first phase, the parallel phase, the simulation proceeds until it deadlocks. The second phase, the interface, initiates deadlock recovery computations. The scheme involves a central controller process to

monitor for deadlock and to control deadlock recovery, thus violating a distributed computing principle. To avoid a single resource to become a performance bottleneck, any general distributed deadlock detection mechanism can be used (Chandy, Misra, and Haas 1983; Groselj and Tropper 1989). The deadlock can be broken by the observation that the message with the smallest timestamp is always safe to process; or, with use of a distributed computation, obtain a lower bound to enlarge the set of safe messages.

In an algorithm described by Misra (1986), a special message called the *marker* circulates through the network of logical processes to detect and resolve deadlock. A cyclic path traversing every link in the network is precomputed. Logical processes are colored, indicating whether the LP has received or sent a message since the last visit of the marker message. A logical process is *white* if it has neither sent nor received a message since the last visit; otherwise the LP is *black*. Initially all logical processes are black and the marker starts at some LP. If an LP receives a marker, it takes the color white and is supposed to route the marker along the cycle in finite time. The marker identifies deadlock if the last N logical processes it has visited were all white, where N is the number of links in the network. The algorithm for deadlock detection is correct if the messages arrive over the link in the same order as sent.

The scheme can be extended to detect and recover from deadlock. If the marker also administers the minimum of "next event times" of the visited white LPs, it knows upon detection of deadlock the smallest next event time and the LP at which the next events occurs. To recover from deadlock, this LP can be restarted to process its first event.

The mechanisms described above only attempt to detect and recover from global deadlocks. Prakash and Ramamoorthy (1988) suggested a hierarchical decentralized algorithm that takes advantage of the locality of these deadlocks. An alternative approach to detect and recover from local deadlocks is proposed by Liu and Tropper (1990).

2.3.3 Performance of Conservative Methods

The performance of conservative mechanisms is critically determined by the degree to which processes can look ahead and predict future events; or more importantly, what will not happen in the simulated future. As already outlined in the deadlock avoidance approach, a process with *lookahead* ϵ can guarantee that no events, other than the ones that it can predict, will be generated up to time $T + \epsilon$. The larger the lookahead, the earlier processes may be enabled to safely process future events that they have already received. Thus the null messages with a predicted timestamp $T + \epsilon$ can also be used to some extent in the deadlock detection and recovery algorithm to improve performance.

Fujimoto (1989a) describes lookahead quantitatively using a parameter called the lookahead ratio and presents empirical data to demonstrate the importance of exploiting lookahead to achieve good performance. Other studies of the performance as a function of lookahead have been published by Lin and Lazowska (1990b), Loucks and Preiss (1990), and Su and Seitz (1989). The

YAWNS algorithm (Nicol 1993) takes a synchronous approach to conservative simulation and lookahead. The YAWNS algorithm incorporates barrier synchronizations and global reductions on functions of future simulation times. An important quality of the algorithm is the little model-specific lookahead information required to achieve performance.

An important performance issue in deadlock avoidance algorithms is the overwhelming amount of null messages induced by the protocol. For example, suppose that the null message traveling through the cycle of waiting logical processes has a timestamp 10, and that the next genuine event is scheduled on simulation time 100. If with each full circulation of a null message the local clock of the logical process is increased with 1, it will take 90 circulations before the first genuine event will be processed. If an LP could deduce that it was essentially waiting for itself, the LP could just process the next event. The *carrier null message* protocol (Cai and Turner 1990) adds extra information to null messages to exploit this ability of lookahead in order to reduce the number of null messages. Other optimizations to reduce the number of null messages are presented by de Vries (1990) and Preiss et al. (1991).

2.4 Optimistic Methods

Optimistic methods do not strictly adhere to the local causality constraint as defined in Section 2.2. In this respect, optimistic methods relieve from the constraints imposed by conservative methods, among which the most prominent are the determination of safe events and the static topology of possible interactions between logical processes. A consequence of allowing causality errors to occur is that a mechanism has to be provided to detect and recover from these errors. The basic method to recover from errors is to rely on state rollback to correct the erroneous computation. The application of state rollback is also found in databases and fault-tolerant systems as the basic mechanism to recover from errors. Because state rollback is used to recover from causality errors, i.e., errors made in chronology, we can say that state rollback is the basic synchronization mechanism used in optimistic methods. By the use of the causality error detection and recovery mechanisms, optimistic methods avoid blocking and the determination of events that are safe to process, which are serious performance pitfalls in the conservative approach.

For a comprehensive understanding of the complexity of optimistic distributed simulation and the consequent problems to be solved, we will present a detailed overview of the issues in optimistic distributed simulation.

First, the basic optimistic simulation mechanism will be elaborated in larger detail. The archetype optimistic simulation protocol, proposed by Jefferson and Sowizral (Jefferson and Sowizral 1982; Jefferson 1985), is known as the Time Warp distributed simulation method. The Time Warp protocol is based on the virtual time concept. The virtual time concept is derived from the work of Lamport (1978) on logical clocks, with this difference that logical clocks are used to construct an ordering of events, while virtual time is used to

impose an ordering of events. In the following discussion, we regard the virtual time to be the simulation time as used in local clocks and timestamps.

Next, the performance improvements to the different functional components making up the distributed simulation mechanism are described. For example, rollback is an expensive operation, and one might want to prevent the recursive cancellation of processed events if the cancellation itself is unnecessary. Another issue is the overhead introduced by state saving; this can be reduced by saving not every state change. Much of the overhead of the optimistic distributed simulation protocol is a consequence of unlimited optimism of the execution mechanism. Solutions have been proposed to limit this unbridled optimism. The Global Virtual Time (GVT) computation influences the memory efficiency. The improvements described in this section are not integrated or total solutions, and hence the choice for GVT computation and state saving strategies are independent. The design and implementation alternatives of the functional components are in principle orthogonal to each other.

2.4.1 Virtual Time

The *virtual time* concept is a method to organize a distributed system by imposing a temporal coordinate system on the distributed computation (Jefferson 1985). A virtual time system is a distributed system that executes in coordination with an imaginary global virtual clock ticking virtual time. Virtual time is a temporal coordinate system used to measure computational progress and define synchronization.

The distributed system is envisioned as a collection of processes, where each process is considered occupying a point in virtual space. Every primitive action, such as changing a variable, sending a message, etc., thus has both a virtual time coordinate and a virtual space coordinate. The set of all actions that take place at the same virtual place x and same virtual time t is referred to as the event at (x, t).

The processes in the virtual time system communicate with each other by exchanging messages. Each message is stamped with four values: sender, send time, receiver, receive time. The send time indicates the virtual time at the moment the message is sent, and the receive time determines the virtual time when the message must be received. The event times are subject to the two, trivial, clock conditions as defined by Lamport (1978) and similarly by Jefferson (1985):

Definition 2.2

Clock Condition 1 *If e and e' are events in one process, and e comes before e', then the virtual time of event e must be less than the virtual time of event e'.*

Clock Condition 2 *If e is the sending of a message and e' is the receipt of that message, then the virtual send time of the message at event e must be less than its virtual receive time at event e'.*

The clock conditions guarantee the adherence to the causality constraint as defined in Section 2.2, or in other words, that the direction of information transfer is always pointing in the direction of increasing virtual time.

The concept of virtual time has many similarities with the notion of *logical clocks* (Lamport 1978). The introduction of logical clocks that adhere to the two clock conditions, allows for a total ordering of the events in the distributed system. The important difference between logical clocks and virtual time is that with logical clocks events are time stamped *a posteriori*, while with virtual time the events are time stamped *a priori*. The implementation of logical clocks assigns logical clock values to the events yielding a total order of the events of that execution. Virtual time does the reverse: all events are stamped with a clock value in such a way they comply to the clock conditions and form a totally ordered sequence. With virtual time it is up to the execution mechanism to process the events in consistence with the correct total order.

In the previous discussion of virtual time, we did not mention distributed simulation on purpose because the virtual time concept is more generally applicable to coordinate distributed computations. For example, virtual time can be used for consistent checkpointing and restarting of distributed applications (see also Chapter 7), atomic transaction processing in distributed database control (Jefferson and Motro 1986), and preservation of message order in virtual circuit communication. Li et al. (1992) presented a distributed logic programming system based on virtual time to control global backtracking in the computation. In the next section we discuss an execution mechanism called Time Warp that correctly implements virtual time, i.e., each process handles messages in timestamp order.

2.4.2 The Basic Time Warp Mechanism

The Time Warp simulation mechanism is based on the concept of virtual time. Virtual time describes how different distributed objects interact in time, and can therefore be used to serve as a basis for distributed simulation. The Time Warp mechanism implements virtual time and adheres to the temporal coordinate system imposed on a distributed simulation.

In optimistic simulation, logical processes execute events and proceed in local simulated time as long as they have any input at all. First, the logical process selects the event with the minimum timestamp of all unprocessed event messages. Next, the LP sets the local clock—also called the Local Virtual Time (LVT) of a logical process—to the minimum timestamp, and processes the selected event. After completion of the event execution, the logical process starts the following iteration by selecting the next event with the smallest timestamp.

A consequence of the optimistic execution of events is that the local clock or LVT of a process may get ahead of its neighbors' LVTs, and it may receive an event message from a neighbor with timestamp smaller than its LVT, that is, in the past of the logical process. The event causing the causality error is called a *straggler*. If we allow causality errors to happen, we must provide a mechanism to recover from these errors in order to guarantee a causally correct distributed

simulation. Recovery is accomplished by undoing the effects of all events that have been processed prematurely by the process receiving the straggler. The net effect of the recovery procedure is that the logical process *rolls back* in simulated time.

The premature execution of an event results in two things that have to be rolled back: (i) the state of the logical process and (ii) the event messages sent to other processes. In order to rollback, the mechanism requires a record of the logical processes' history with respect to internal and external events: state, messages received and sent. The rollback of the state is accomplished by periodically saving the process state and restoring an old state vector on rollback: the logical process sets its current state to the last state vector saved with simulated time earlier than the timestamp of the straggler. Recovering from premature sent messages is accomplished by sending an *anti-message* that annihilates the original when it reaches its destination. The messages that are sent while the process is propagating forward in simulated time, and hence correspond with simulation events, are called *positive messages*.

The annihilation of an anti-messages with a positive message can follow two scenarios. If upon receipt of an anti-message the matching positive message is still in the input queue, the two messages will annihilate each other and the logical process will proceed. However, if an anti-message arrives that corresponds to a positive message that is already processed, then the process has made a causality error and the logical process must also roll back. Note that this rollback is triggered by the rollback of the original logical process that sent the anti-message. A direct consequence of the rollback mechanism is that more anti-messages may be sent to other processes recursively, and allows all effects of erroneous computation to be eventually canceled. As the smallest unprocessed event in the simulation is always safe to process, it can be shown that this mechanism always makes progress under some mild constraints (Leivent and Watro 1993).

In optimistic simulation the notion of global progress in simulated time is administered by the Global Virtual Time (GVT). The GVT is the minimum of the LVTs for all the processes and the timestamps of all messages (including anti-messages) sent but unprocessed. No event with timestamp smaller than the GVT will ever be rolled back, so storage used by such event (i.e., saved state vector and event message) can be discarded. Also, irrevocable operations such as I/O cannot be committed before the GVT sweeps past the simulation time at which the operation occurred. The process of reclaiming memory and committing irrevocable operations is referred to as *fossil collection*.

The different optimistic simulation strategies from literature can all be characterized by their opportunistic processing of events that involves the occurrence of causality errors. The differences are mainly along the design axes of the functional components of the archetype optimistic simulation method, such as, cancellation strategies, state saving strategies, bounded optimism, etc. Depending on the application, these different designs may either improve or degrade performance.

2.4.3 Rollback Strategies

Rollback or cancellation strategies are specific methods to recover from a causality error and to cancel the side effects of the erroneous computation. A rollback comprises the restoration of the state vector and the annihilation of the simulation messages that are sent during the erroneous computation. The alternative cancellation strategies are primarily concerned with the limitation of the number of anti-messages and the propagation of the erroneous computation.

Aggressive Cancellation

The rollback strategy introduced with the basic Time Warp algorithm (see Section 2.4.2) is called *aggressive cancellation*. The aggressive cancellation mechanism attempts to correct the mistakes made by Time Warp as quickly as possible. Upon receipt of a straggler, the mechanism immediately recovers the state by restoring the last state vector saved with simulation time earlier than the timestamp of the straggler. The premature sent messages are directly annihilated by sending the corresponding anti-messages to the destination of the original positive message.

Lazy Cancellation

An alternative to aggressive cancellation is *lazy cancellation*, first proposed by Gafni (1988). The basic idea behind lazy cancellation is suggested by the observation that a straggler event does not sufficiently alter the simulation to change the generated positive event messages during the recomputation.

In contrast to the aggressive cancellation mechanism, the lazy cancellation mechanism does not send anti-messages immediately upon receipt of a straggler. It delays the propagation of anti-messages while the process resumes executing forward in simulated time from its new LVT. During the resimulation, the lazy cancellation mechanism checks whether the computation regenerates the *same* messages. If the same message is recreated, then there is no need to cancel the original message. An anti-message created at simulation time T is only sent if the process's LVT sweeps past time T without regenerating the same message.

The lazy cancellation mechanism avoids unnecessary canceling of correct messages at the costs of additional memory and bookkeeping overhead and delaying the annihilation of actually wrong events. This allows the erroneous computation to spread further than it would under aggressive cancellation. Depending on the application, lazy cancellation may either improve or degrade performance. One can construct extreme cases where lazy cancellation performs N times slower than aggressive cancellation when N processors are used; and vice versa where lazy cancellation achieves near N-fold speedup using N processors, while aggressive cancellation requires the same amount as the sequential execution (Reiher et al. 1990). Empirical results indicate that lazy

cancellation is slightly favorable over aggressive cancellation (Lomow et al. 1988; Reiher et al. 1990).

An interesting property of lazy cancellation is that it can, under certain circumstances, run faster than the critical path of the simulation, this is called *super-critical speedup* (see Chapter 4 for a discussion on the critical path). The lazy cancellation strategy surpasses the simulation critical path by the possibility of having a wrong computation producing a correct result.

Lazy Re-evaluation

Lazy re-evaluation is somewhat similar to lazy cancellation, but deals with the delayed discarding of state vectors instead of the delayed sending of anti-messages. Suppose the simulation process receives a straggler. If the simulation process has the same state vector after processing the straggler as the state vector logged by the state saving mechanism and no new messages have arrived, then the simulation immediately jumps forward to the LVT before the rollback occurred. (Therefore the mechanism is also called *jump forward* optimization.)

Lazy re-evaluation is promising in simulation models where events do not modify states ("read-only" or query events). However, the additional memory and bookkeeping overhead, and also the considerable complication of the Time Warp code makes that the mechanism is not commonly applied (Fujimoto 1990a).

Direct Cancellation

In optimistic simulation methods, it is important to be able to cancel the incorrect computation faster than it can spread through the system. This critical spreading behavior can be prevented by giving anti-messages a higher priority than positive messages. Fujimoto (1989b) proposed a mechanism that uses shared memory to optimize the cancellation of incorrect computations. If during the execution of an event E_1 a new event E_2 is scheduled, the mechanism associates a pointer reference to E_2 with event E_1. Upon rollback of event E_1, the pointer reference can be used to cancel E_2, using either lazy or aggressive cancellation. Good performance results have been reported on a specific version of Time Warp that uses direct cancellation (Fujimoto 1990b).

Preventing Rollback Chains

Other approaches have been proposed to limit the length of successive rollbacks as early as possible. Prakash and Subramanian (1991) attach some state information to messages, to prevent cascading rollbacks. This information allows the simulation process to filter out messages based on obsolete states that will eventually annihilated by anti-messages currently in transit. Madisetti et al. (1990) proposed within their Wolf system a mechanism that freezes the spatial spreading of the incorrect computation based upon a so-called sphere of

influence. The Wolf algorithm ensures that the effects of an uncommitted event are limited to a sphere of a computable radius around the simulation process. A disadvantage of this approach is that the set of simulation processes that might be affected by the incorrect computation is significantly larger than the actual set, and thereby leads to sending unnecessary control messages.

2.4.4 State Saving

The previous section discussed various rollback strategies, which differ in their method to restore the state vector and annihilate simulation messages. The rollback of the state vector implies that whenever an LP detects a causality error, i.e., receives a straggler, it returns to an earlier state just before the timestamp of the straggler. To restore the state vector of an LP, the simulation mechanism needs a record of the LP's state history. This is accomplished by periodically saving the process state, also called *checkpointing*. The different state saving strategies can be distinguished from each other by the method they apply to save the (partial) state vector and consequently restore the state vector to a particular simulation time to which the simulation rolls back.

The simplest method for state saving is to copy the entire state of an LP *each time* it executes an event. This is often referred to as *copy state saving* (CSS). One major disadvantage of CSS is that it becomes very expensive when the state size is large. The overhead costs of state saving consist of memory consumption and processing time. Improvements to the CSS method, such as periodic state saving, incremental state saving, and hybrid methods, have been proposed to reduce the memory consumption and/or the processing time overhead.

Periodic State Saving

The *periodic state saving* (PSS) method reduces the state saving overhead by increasing the checkpoint interval. Thus opposed to CSS, periodic state saving copies the entire state only after every χ^{th} state update, where χ is the state saving interval. However, the fact that not every state is saved counteracts some of these gains, as the reconstruction of the uncheckpointed state may be necessary. If a rollback occurs and the required state is not in the state queue, the LP must roll back to an earlier checkpointed state. The required state is recomputed from the earlier state by reprocessing the input messages. The output messages regenerated during state reconstruction must not been sent since they are an artifact of the state reconstruction phase. The process of reconstructing a missing state is called coasting forward.

The checkpoint interval is the key parameter that determines the efficiency of PSS method: it regulates the trade-off between the total cost of state saving and the amount of re-execution in coasting forward. Establishing a static value for the checkpoint interval that produces optimal performance is difficult. Furthermore, many applications show dynamic behavior where the optimal checkpoint interval is likely to vary over the runtime of the simulation. As

there is no single optimal checkpoint interval for the runtime of the application, the checkpoint interval should be adapted to the dynamic behavior of the simulation. Typical feedback effects such as *thrashing* (increase in rollbacks) and *throttling* (decrease in rollbacks) of the simulation induced by a change in checkpoint interval, also push towards adaptive checkpointing algorithms in order to achieve performance optimization (Preiss et al. 1994).

Several adaptive PSS methods have been proposed to select the optimal checkpoint interval during the execution of the simulation application. Lin et al. (1993) studied the interrelationship among checkpoint interval, the rollback behavior, and the overhead associated with state saving and restoration. Based on this model, a checkpoint interval selection algorithm which determines the optimal checkpoint interval during execution of Time Warp simulation was proposed. Fleischmann and Wilsey (1995) performed an empirical study on four adaptive PSS methods. The results show significant difference in performance, however, the performance of the adaptive PSS methods is better than the best static value for the checkpoint interval. They present a heuristic that recalculates the state saving costs (state saving and coasting forward) after every N events. If the execution time has increased significantly, the state saving interval is decreased by one, otherwise it is increased by one. Sköld and Rönngren (1996) argued that the optimal checkpoint interval also depends on the execution time or event granularity for different types of events. Their event sensitive state saving method is sensitive to which type of event the previously executed event belongs, and decide whether to save the state vector based on this information. Experimental results indicate that event sensitive state saving is a promising approach for simulation models where event granularity has large variance. Auriche et al. (1998) were successful in constructing an analytical model for checkpoint interval selection that accounts for memory management costs. The presented experimental results show that their method improves performance compared to already existing ones in some simulation scenarios.

Incremental State Saving

A different approach to reducing the state saving overhead and memory consumption is *incremental state saving* (ISS) (Overeinder et al. 1992; Bauer and Sporrer 1993; Unger et al. 1993). Many challenging real-world applications in, for example, VLSI, communication systems, and natural sciences, are characterized by LPs with very large states where only a fraction of the state is updated in each event execution. In such applications, it may be inefficient or perhaps infeasible to save copies of the complete state of the LP, which can be on the order of hundreds of kilobytes. The ISS mechanism typically exploits the partial state update due to the execution of an event by only saving each change to the state as it occurs. The incremental history record of the LP's state is created by saving the old value of a variable prior to overwriting the variable with a new value. This incremental state history record is used to restore the state of the LP by restoring each saved variable value in reverse

order they were saved.

The benefits of the ISS method are low state saving time and low memory consumption, on the condition that the LP state can be divided into small parts, with a relative small number of those parts being saved after each event execution. A typical example application can be an individual-based population dynamics model, where each LP is responsible for a subdomain with a dynamically changing population of individuals (McCauley et al. 1993; Mellott et al. 1999). With the evolution of the population dynamics model, the LP only records the changes resulting from events such as birth, death, etc., per individual. A drawback of the partial incremental state saving mechanism, however, is the increased costs for state restoration in ISS as opposed to CSS. As the state of the LP is reconstructed by restoring each saved variable in reverse order, the cost of state reconstruction is directly related to the rollback length. Therefore, the use of ISS is only effective if the rollback distance in the simulation application is sufficiently small.

Another important issue in state saving is transparency. A major advantage of CSS is that it can easily be made transparent to the programmer. However, due to problems associated with identifying which parts of the state are updated and when, makes a transparent implementation of ISS quite complex. If no special provisions for transparent ISS is taken, the application programmer is provided with support functions to save a variable. These functions must be inserted by hand, which is not a natural activity for an application programmer with no special understanding of state saving, and is therefore an error prone approach. A number of transparent ISS have been proposed that exploits the operator overloading and type parameterization capabilities in C++ (Steinman 1993b; Rönngren et al. 1996; Gomes et al. 1996). Rönngren et al. (1996) showed that their approach achieves a high degree of transparency with acceptable overhead compared to a non-transparent implementation of ISS. The ISS method integrated in the SPEEDES system (Steinman 1993b) elegantly integrates ISS with an efficient implementation of lazy cancellation, which requires *roll forward* as well as *rollback* support. West and Panesar (1996) developed a new technique that they call Automatic Incremental State Saving. This technique essentially edits the already compiled executable code directly to insert incremental state saving calls. In this way, code written and compiled by a third party may now be state saved. Bruce (1995) showed how the theory of persistence can be used as a simple yet general mechanism for performing the necessary (incremental) state saving with minimal impact on the application code. The presented results show that the performance of the persistent data structure is competitive with existing mechanisms.

The total overhead costs of incremental state saving depends to a large degree on the percentage of the state that is modified due to the execution of an event. West and Panesar (1996) find that their Automatic Incremental State Saving is beneficial if less than 15% of the state is modified in each event as compared to copy state saving. In another empirical study by Cleary et al. (1994), the results indicate that the cross-over point between ISS and CSS costs lies between 30% and 50% of state updated, which is in good agreement

with their theoretical analysis.

Hybrid State Saving

Both periodic state saving and incremental state saving have additional costs over copy state saving during state reconstruction. PSS introduces overheads in coasting forward from an earlier checkpointed state, while the state reconstruction in ISS demands large overheads in applications with large rollback distances. To solve these flaws, hybrid state saving methods have been proposed that combine PSS and ISS.

The Multiplexed State Saving (MSS) minimizes the overhead for forward execution and maintains low cost access to state at arbitrary times in the past by interleaving ISS and PSS (Franks et al. 1997). The combination yields the bounded rollback costs of checkpointing methods, with the speed of incremental methods for rollbacks of short distances. The Hybrid State Saving (HSS) is similar to the approach in MSS in its method it interleaves PSS with ISS (Soliman and Elmaghraby 1998). An analytical study of HSS shows that if 15% or more of the time to save an LP's sate is needed to save state increments after every event execution, HSS outperforms ISS.

2.4.5 Optimism Control

A serious problem hampering the effective application of optimistic simulation methods is *thrashing*. Thrashing of an optimistic simulation occurs when the system experiences excessive long and/or frequent rollbacks. This behavior is typically characterized as cascading rollbacks and echoing rollbacks, where two or more LPs initiate mutual rollbacks. This results in an inefficient execution where correcting causality errors consumes more computation time than the forward simulation. The specific thrashing behavior is induced by overly optimistic behavior of the simulation protocol. The optimistic behavior is a combination of aggressiveness and risk. Aggressiveness is the property that determines the execution of events without the guarantee of freedom of errors, and risk is the property by which the results of aggressive processing are propagated to other LPs. Besides thrashing, overly optimistic behavior is also responsible for inefficient use of memory because a certain amount of history information must be maintained to allow rollback. This results in performance degradation of the virtual memory by inducing excessive paging and/or poor cache performance (Das and Fujimoto 1997).

As uncontrolled optimism may lead to poor performance, a method to control optimism is desirable in order to adapt to the dynamic, unpredictable nature of synchronization requirements of the parallel simulation. The different approaches to optimism control, also called *optimism throttling*, can be categorized by the state information used to implement adaptivity, and by the method with which they control aggressiveness and risk.

Non-Adaptive Protocols

The first methods to control excess optimism of the simulation method used time windows. The time windows approach limits the optimism by executing events within a window of simulated time beyond the global virtual time. The events outside the time window are delayed until the time window is updated. The time window bounds the difference between the logical clocks and hence limits the lengths of rollback chains. The original simulation system exploiting this idea was the Moving Time Window (MTW) protocol (Sokol et al. 1989). A key problem with this class of non-adaptive optimistic protocols is the determination of the appropriate size of the time window. A narrow time window will limit the rollbacks, but admits a small amount of parallelism. A time window that is too large, can potentially exploit more parallelism, but the rollbacks may increase as well. A similar idea is studied by Turner and Xu (1992) in the Bounded Time Warp (BTW) protocol, where no events are processed beyond a bound in simulation time until all processes have reached that bound, when a new bound is established.

The Breathing Time Bucket algorithm (Steinman 1992) uses optimistic processing with local rollback. However, unlike other optimistic windowing approaches, anti-messages are never required. In other words, Breathing Time Buckets could be classified as a risk-free optimistic approach. Breathing Time Warp (Steinman 1993a) is an extension to the Breathing Time Bucket algorithm by allowing it to take risks. The idea is to execute the first N_1 events beyond the GVT, just as the basic Time Warp algorithm does. Then the protocol issues a nonblocking synchronization operation and switches back to the risk-free breathing time bucket algorithm to execute the next N_2 events. If all LPs reached their event horizon, that is, issued the nonblocking synchronization, a new GVT computation is started and a next cycle is issued.

The MIMDIX system (Madisetti et al. 1993) employs the ideas of probabilistic resynchronization to eliminate overly optimistic behavior. A special process called a “genie” probabilistically sends a synchronization message to all LPs, causing them to synchronize to the timestamp of the message. By keeping the timestamp of the synchronization message close to the GVT, the LPs can be kept temporally close to each other, thus reducing the risk of cascading rollbacks.

The non-adaptive protocols can be adjusted to behave like a conservative method in one extreme and like a pure optimistic method in the other extreme. For example, the width of the time window in the MTW or BTW protocol can be tightened to behave as a conservative simulation; or if the width is infinite, the protocol is equivalent to Time Warp. However, it is left to the simulation modeler to select the appropriate parameter settings. In general, the simulation modeler is not in great detail familiar with the intrinsics of the PDES protocol and the underlying parallel hardware, which makes it difficult to tune the simulation for optimal performance. Furthermore, many simulation models are dynamic in their runtime behavior, hence there is no single optimal parameter setting to control optimism. This observation motivated the design of optimism

control mechanisms that adapt themselves to changing behavior of the simulation by monitoring the state of the parallel simulation and determine the appropriate trade-off between conservatism and optimism.

Adaptive Protocols Based on Local State

Adaptive protocols are characterized by their adaptive optimism control based on the state of the parallel simulation. In the following discussion, the adaptive protocols are broadly classified according to whether the decisions are taken purely on the local state of each LP or on the global state of all LPs.

The control mechanism in Adaptive Time Warp (Ball and Hoyt 1990) uses a penalty based method to limit optimism by blocking the LP for an interval of real time (the blocking window). The blocking window is adjusted to minimize the sum of total CPU time spent in blocked state and recovery state (that is, undoing the effects of the erroneous computation). The logical process may decide to temporarily suspend event processing if it had recently experienced an abnormally high number of causality errors. The time the LP blocks is directly proportional to the width of the blocking window. To determine the optimal blocking window width, ATW assumes that the time spent either in the blocked and recovery state can be numerically approximated using a two term Taylor's series expansion.

Hamnes and Tripathi (1994) proposed a local adaptive protocol that uses simple local statistical data to avoid additional communication overhead. The protocol is designed to adapt to the application in order to maximize progress of simulation time in real time (with simulation time progress rate α). The algorithm gathers statistics on a per channel basis within each LP and uses this information to maximize α. By an interrelation of null messages, rollbacks, and blocking, the adaptive protocol retains aspects of both the optimistic and conservative protocols to provide a continuum of simulation protocols accommodated to the simulation application at hand. The probabilistic adaptive direct optimism control presented by Ferscha (1995, 1999) is similar in spirit, but adds a probabilistic component in the sense that blocking is induced with a certain probability. Several forecasting methods have been explored, such as incremental forecast methods like arithmetic mean or exponential smoothing, as well as integrated autoregressive moving average (ARIMA) models. Probabilistic optimism control was shown to outperform Time Warp for stochastic Petri net simulations, especially under load imbalance.

Optimism control mechanisms are not limited to bounded time windows and blocking windows, but scheduling and dynamic load balancing can also interact with the synchronization mechanism. High processor utilization may not imply good performance as processors may be busy with incorrect computation that will be undone later (Nicol and Fujimoto 1994). Parameterized Time Warp (Palaniswamy and Wilsey 1996) combines three adaptive mechanisms (state saving period, bounded time window, and scheduling priority) to minimize overhead and increase the performance of the parallel simulation. The measure *useful work* is defined to determine the actual amount of optimism uti-

lized by the process. Useful work is a function of a number of parameters such as the ratio of the number of committed to the total number of events executed, the number of rollbacks, the number of anti-messages, the rollback length, etc. The state saving period and the bounded time window are increased for larger values of the useful work parameter. The scheduling priority also increases with the useful work parameter, as the priority for scheduling should increase if the LP is more productive then before. Experiments with digital logic simulations demonstrated superiority of the method over ordinary Time Warp.

Adaptive Protocols Based on Global State

Global state adaptive protocols are similar to the local adaptive counterparts with respect to their mechanisms to control optimism, but they rely primarily on some aspects of the global state rather than on the local state information. The Adaptive Memory Management protocol by Das and Fujimoto (1997) uses an indirect approach to control overly optimistic event execution. It has been observed that overly optimistic Time Warp not only incurs high rollback costs, but also high memory management costs. And vice versa, that the amount of memory allocated to a Time Warp simulation automatically limits the amount of optimistic execution. The adaption algorithm attempts to minimize the total execution time rather than concentrating on one specific criteria. They argue that this approach prevents from optimizing for one aspect of the computation at the expense of a disproportional increase in another.

The Near Perfect State Information (NPSI) protocols (Srinivasan and Reynolds 1998) are a class of adaptive protocols relying on the availability of near-perfect information on the global state of the parallel simulation. As already mentioned, there are two phases in the design of adaptive protocols, and in NPSI in particular, namely the state information on which to decide and the mechanism that translates this information into control over the LP's optimism. NPSI protocols use a quantity error potential (EP_i) associated with each LP_i, to control LP_i's optimism. The protocol keeps each EP_i up to date as the simulation progresses. A second component of the protocol translates the EP_i into control over the aggressiveness and risk of LP_i. One instance of a NPSI protocol is the Elastic Time Algorithm (ETA). In ETA, the farther LP_i moves away from its predecessor, the slower its progress due to the restraining pull of the elastic band tying it to its predecessor—hence elastic time. The tension in the elastic band corresponds to the LP_i's error potential. An assumption in the applicability of NPSI protocols is that the NPSI is available at minimal costs, thus limiting the use of such protocols to shared memory multiprocessors or distributed memory systems with high-speed reduction network support.

Tay et al. (1997) proposed a throttle scheme based on the concept global progress window (GPW), which allows the individual simulation process to be positioned on a global time scale. The GPW indicates the progress status of the slowest and the fastest LPs, and is represented by $GPW = [GVT \ldots GFT]$, where GFT is the maximum of all LVT_i. Thus, GPW provides a global time scale for each LP to calibrate its simulation progress. The adaptive throttle

design regulates the number of events executed in each LP simulation cycle to achieve an in pace LVT progression. For slow LPs (close to GVT), a larger regulator value is used to accelerate the event execution. As for the fast LPs, the regulator is set to 0 to prevent it from advancing it LVT further.

2.4.6 Global Virtual Time Algorithms

The Global Virtual Time (GVT) is used in the parallel optimistic Time Warp synchronization mechanism to determine the progress of the simulation. Contrary to LVT, the essential property of the GVT is that its value is nondecreasing over real time (wall-clock time). Conceptually, the GVT is the simulated time up to which all LPs have simulated correctly and beyond which all LPs have simulated speculatively. By the property that no LP can ever rollback to a simulation time earlier than the value of the GVT, the GVT algorithm can guarantee that Time Warp eventually progresses the simulation by committing intermediate results. The progress property is also used for termination detection, as the simulation often completes when the GVT reaches a specific end time.

Another important use of GVT in Time Warp is within the fossil collection mechanism, which coordinates memory management and irrevocable operations such as I/O (including interaction with users). Optimistic simulations must save state information and positive/negative event message pairs as events are processed in order to support rollbacks. State saving and storing event messages consume valuable memory resources which must be reclaimed periodically. Because LPs never rollback to a simulation time earlier than the GVT, it is safe to reclaim memory resources for events with timestamps less than GVT. For irrevocable operations that cannot be easily rolled back this safety criteria also applies, hence the irrevocable operations are effectuated or committed as the GVT sweeps past their simulation time.

The computation of the GVT has an influence on the performance and operability of Time Warp. The exchange of information necessary to compute the GVT generates extra messages over the communication network. Furthermore, the GVT algorithm requires computational resources, thus the LPs stop simulating in order to engage in the computation of the GVT. Both the distributed nature of the information and the time-consuming GVT algorithms make that relative stale values of the GVT are computed, and consequently fossils cannot be quickly identified and collected. These performance considerations motivated the design of algorithms for accurate GVT estimation that are scalable over the number of LPs in the parallel simulation. In general, the GVT algorithms can be categorized as either centralized or distributed in nature.

Centralized GVT Computation

In centralized GVT algorithms the GVT is computed by a central GVT manager that broadcasts a request to all LPs for their current LVT and while collecting those values perform a *min*-reduction (global reduce operation selecting the

minimum value). In this approach there are two main problems that complicate the computation of an accurate GVT estimation. First, the messages in transit that can potentially roll back a reported LVT, are not taken into consideration; this is also known as the transient message problem. Second, the reported LVT values were drawn from the LPs at different real times, this is called the simultaneous reporting problem.

One of the first GVT algorithms proposed by Samadi (1985) starts a GVT computation via a central GVT manager which sends out a *GVT start* message. After all LPs have send a reply to the request, the GVT manager computes and broadcasts the new GVT value. The transient message problem is solved by acknowledging every message, and reporting the minimum over all timestamps of unacknowledged messages and the LVT of the LP to the GVT manager. Lin and Lazowska (1990a) introduced some improvements over Samadi's algorithm. In Lin's algorithm, the messages are not acknowledged but the message headers include a sequence number. The receiving LP can identify missing messages as gaps in the arriving sequence numbers. When the GVT manager starts a GVT computation, i.e., broadcast a *GVT start* message, the LPs send out to all their communication partners LP_j the smallest sequence number still demanded from this neighboring LP_j. This information is used as an implicit acknowledgement of all previous messages with smaller sequence number. The receiving LP_j can use this information to determine which messages are still in transit and compute a lower bound on their timestamps.

In the previous discussion, we assumed that the reported LVT and timestamps of unacknowledged messages are reported at some real time. However in reality, the instantaneous report, or global snapshot, is not possible, and hence the reported values are drawn at different real times. The simultaneous reporting problem is solved in the GVT algorithms described above by setting a lower and upper bound on the reporting real time at each LP such that the set of intervals $\{[start_i, stop_i] \mid i \in LP\}$ share a common real time RT. Each LP can then forward information to the GVT manager as it leaves the interval. Using this approach, calculation of the GVT involves four phases. The *start* and *stop* phases to generate $start_i$ and $stop_i$ at each process, the *collect* phase to receive the information needed to calculate the GVT and the *notify* phase to notify all LPs of the updated GVT. The stop and collect phase can often be combined, and similarly, the notify phase from a previous round can be combined with the start phase of the next round.

To reduce the communication complexity of the GVT computation, Bellenot (1990) uses a message routing graph. The GVT computation requires two cycles, one to start the GVT computation and the second to report local minima and compute the global minimum. The multi-level token passing algorithm proposed by Concepcion and Kelly (1991) applies a hierarchical method to parallelize the GVT computation. The token passing algorithm can be elegantly mapped to a hypercube topology and significantly decreases the number of messages for the computation of the GVT. The multi-level decomposition allows the parallel determination of the minimum time among the managers of each level.

Bauer et al. (1991, 1992) proposed an efficient algorithm where all event messages through a certain communication channel are numbered. The LPs administer the number of messages sent and received, and the minimum timestamp of an event message since the last GVT report. Periodically, the LPs send their local information and their LVT to the GVT manager, which deduces from the information the minimum of the transient message timestamps and LVTs.

The passive response GVT (pGVT) algorithm (D'Souza et al. 1994; D'Souza et al. 1997) is able to operate in an environment with faulty communication channels, and adapts to the performance capabilities of the parallel system on which it executes. In the pGVT algorithm, the LP considers message latency times to decide when new GVT information should be sent to the GVT manager. This allows each LP to report GVT information to arrive just in time to allow for aggressive GVT advancement by the GVT manager. A key performance improvement of pGVT is that the LPs simulating along the critical path will more frequently report GVT information than others.

An efficient, asynchronous, shared-memory GVT algorithm is presented by Fujimoto and Hybinette (1997). The GVT algorithm exploits the guarantee of shared-memory multiprocessors that no two processors will observe a set of memory operations as occurring in different orders. This property is used to solve the simultaneous reporting problem requiring only one round of interprocessor communication. Furthermore, the sequentially consistent shared memory is exploited such that the algorithm does not require message acknowledgements, FIFO delivery of messages, or special GVT messages. The transient message problem is just eliminated by allowing the sender LP copying the message into the receiver's buffer. The applications that would most benefit from this algorithm are small grain interactive simulations where GVT must be performed relatively frequently in order to rapidly commit I/O operations.

Distributed GVT Computation

Distributed GVT algorithms neither require a centralized GVT manager, nor the availability of shared-memory between the LPs. For the distributed computation of the GVT for a simulation system with FIFO message delivery, distributed snapshot algorithms (Chandy and Lamport 1985) find a straightforward application. However, due to the frequency of the GVT computation, i.e., accurate estimation of the GVT, more efficient solutions are desired. Mattern (1993) presented an efficient "parallel" distributed snapshot algorithm for non-FIFO communication channels which neither requires messages to be acknowledged. The different and rather simple solution is that the algorithm determines two snapshots, where the second is pushed forward such that all transient messages are enclosed between the two snapshots. The problem of knowing when the snapshot is complete (all transient messages have been caught) is solved by a distributed termination detection scheme.

The interference of GVT computation messages with the regular simulation messages, motivated Srinivasan and Reynolds (1993) to design a parallel reduction network (PRN)—in hardware—used by their GVT algorithm. The

LPs communicate some state information to the PRN, which is maintained by the distributed GVT algorithm. Along state information, message receipt acknowledgements are also sent over the PRN. The PRN is used to compute and disseminate the minimum LVT of all LPs and the minimum of the timestamp of all unreceived messages. The GVT is made available to each LP asynchronously of the LPs, at no cost.

The GVT algorithm in the SPEEDES simulation environment (Steinman et al. 1995) is especially optimized to support interactive parallel and distributed optimistic simulation. The SPEEDES GVT algorithm is featured in the Breathing Time Warp algorithm (see Section 2.4.5). The transient message problem, that complicates the GVT computation, is solved by flushing out all messages during the GVT update phase. The SPEEDES GVT algorithm continues to process events during this phase, but new messages that might be generated are not immediately released. Brief performance figures show that their algorithm performs well on a number of different hardware architectures.

2.5 Summary and Discussion

Parallel Discrete Event Simulation (PDES) yields a fundamental approach to reduce the required execution time of realistic simulation models. As the real-world models become increasingly more advanced and complex, PDES methods will be an invaluable technique to realize the practical simulation of certain classes of discrete event models. The parallelism that is available in DES models is exploited by decomposing the simulation model into so-called logical processes, which are the simulation equivalent of the real-world physical process. PDES methods are merely concerned with synchronization between the logical processes which execute in parallel. Two principal PDES methods can be identified: conservative and optimistic methods. Conservative methods adhere to the correct execution order of the events in the distributed simulation. Optimistic methods on the other hand are less rigid and allows causality errors to occur, but use a detection and rollback mechanism to recover.

Conservative methods offer good potential for certain classes of problems. A major drawback, however, is that they cannot fully exploit the parallelism available in the simulation application. If it is possible that event E_i might affect E_j either directly or indirectly, conservative approaches must execute E_i and E_j sequentially. If the simulation is such that E_i seldom affect E_j these events could have been processed concurrently most of the time. As a consequence, conservative algorithms heavily rely on lookahead to achieve good performance.

Optimistic methods offer the greatest potential as a general purpose simulation mechanism. A critical question faced by optimistic approaches is whether the system will spent most of its time on executing incorrect computations and rolling them back, at the expense of correct computations. Several extensions to the basic optimistic Time Warp protocol have been proposed to control this thrashing behavior and to prevent cascading rollbacks to occur. Various opti-

mism control and rollback strategies are reported that improve the efficacy of the optimistic simulation method significantly. Another serious problem with the optimistic mechanisms is the need to periodically save the state of each logical process. This limits the effectiveness of the optimistic mechanisms to applications where the amount of computation, required to process an event, is significantly larger than the cost of saving the state vector. Solutions to alleviate this problem are periodic state saving, incremental state saving, or a hybrid approach combining periodic and incremental state saving.

A number of analytical performance modeling of conservative and optimistic parallel simulation studies have been published (Nicol and Fujimoto 1994). A common characteristic among theses studies are the assumptions made for the purpose of mathematical tractability. For example, the inter-event arrival time of events is assumed to be an exponential random variable; or it is assumed that upon sending a message, it is routed to some processor randomly selected from among all processors. In general, Markov chain analysis underlies the performance studies.

A worst-case comparison of optimistic versus conservative methods reported by Lipton and Mizell (1990) shows that Time Warp is capable of arbitrarily better performance than most conservative methods, while the converse is not true. Even though the assumptions in the study describe a simulation application behavior which is rarely observed in practice (constant cost rollbacks, zero-cost message passing, and state saving), it shows how Time Warp can guess correctly while a conservative method blocks. Likewise, the proof that Time Warp performs not worse than conservative methods by a constant factor demonstrates Time Warp's resilience. The *constant factor* derived by Lipton and Mizell contains a term that is the rollback cost, so if rollback cost becomes arbitrarily large, so does the disparity in performance. Nicol (1991) studied the performance bounds on parallel self-initiating discrete event simulations. A self-initiating model schedules its own state reevaluation times (events) independently form other LPs, and sends its new state to other LPs following the reevaluation. The analysis quantifies the processor utilization to be proportional to $1/k$ for optimistic methods and $1/P$ for conservative methods without lookahead, where k is the fanout (the number of processors a message is sent to) and P is the total number of processors. The $1/P$ figure highlights the importance of lookahead for achieving performance with conservative methods. Another result of the analysis demonstrates the dependence of performance on the time increment distribution, showing that distributions with significant constant components lead to good performance. An analytical performance model by Dickens et al. (1996) compares the YAWNS conservative protocol with Bounded Time Warp (BTW). The BTW protocol performs asymptotically better than YAWNS, as the number of LPs grows. However, if many LPs are allowed per processor, YAWNS performs better than BTW under moderate levels of aggregation, or when state-saving costs are nonnegligible. A qualitative results is inferred that it is likely that limiting optimism is a good thing in a window-based framework.

The type of application is important when determining an appropriate ap-

proach to distributed simulation. For dynamic topology systems and systems with irregular interactions, Time Warp methods are preferred over conservative methods, especially if state-saving overheads do not dominate. On the other hand, if the application has good lookahead properties, conservative algorithms can exploit the special structure within a fixed topology system. If the application has both poor lookahead and large state-saving overheads all existing parallel discrete event simulation approaches will have trouble obtaining good performance, even if the application has a considerable amount of parallelism.

The performance impact of the various Time Warp optimizations is difficult to assess in general. The performance trade-off between aggressive versus lazy cancellation differs from application to application. Reiher et al. (1990) compare the two canceling strategies using a number of benchmarks and two applications. The benchmark results show that applications exist that run poorly under either method, but well under the other. Further, the two realistic applications perform reasonably well using either cancellation strategy, but somewhat better with lazy cancellation than aggressive (at most 10%). Dynamically switching between aggressive and lazy cancellation allows an optimal choice depending upon the characteristics of the application (Rajan and Wilsey 1995). Performance improvements of at most 10% are found for the dynamic cancellation approach over pure aggressive or lazy cancellation.

Periodic state saving experiments conducted by Lin et al. (1993) indicate a performance improvement of at most 10% over copy state saving. Fleischmann and Wilsey (1995) investigate dynamically adjusting periodic state saving strategies. The dynamic algorithm performs as much as 12% better than the best static periodic state saving interval value. A comparative study by West and Panesar (1996) presented the state saving costs for copy state saving and incremental state saving. Their implementation of copy state saving and automatic incremental state saving require 0.10 μsec/word (4 bytes) and 0.53 μsec/word respectively. Automatic incremental state saving is beneficial if less than 19% of the state is changed. Manual incremental state saving requires 0.42 μsec/word, which brings the ISS/CSS break-even point to 25%.

Optimism control mechanisms can substantially improve the performance of the optimistic simulation. For simulation applications that are sensitive to thrashing behavior, optimism control can show orders of magnitude improvement over unthrottled optimism in Time Warp. Choi (1998) presents the results of experiments with three different VLSI circuit simulations using the MTW optimism control mechanism. The execution times of two VLSI circuit simulations show a smooth increasing slope as the time window increases. However, the execution times of the third VLSI circuit simulation exhibit a sharp increasing slope due to an exponential growth of the number of rollbacks as the time window increases. In a study by Ferscha (1999), the adaptive optimism control mechanism halved the execution time of a Petri net “stress test” Time Warp simulation. Other studies by Palaniswamy and Wilsey (1996) and Srinivasan and Reynolds (1998) report performance improvements of 30% to 200% for various simulation applications.

Parallel discrete event simulation has been successfully applied in numerous application areas. For example, in biology with ant foraging models or population dynamics models (Deelman et al. (1996) describe the spreading of Lyme disease). In physics with colliding pucks (rigid bodies) and Ising spin systems (see Chapter 5). An important application field is computer science itself, with for example digital logic circuits and multistage interconnection networks. Public sectors such as telephone switching networks (Bhatt et al. 1998), and road and aviation traffic simulation (Wieland 1998) find also application in PDES. Typical military applications are combat simulation and military training (Smith 1998). With respect to military applications, the High Level Architecture (HLA) has been proposed as the common framework for all U.S. Department of Defense simulation applications (Dahmann 1999). The design principle of time management in HLA is transparency: the local time management mechanism used within each component (called a federate in HLA) must not be visible to other components. The broad spectrum of applications drove the design of HLA time management services, which can include event-driven, time-stepped, parallel discrete event simulation, and wall-clock time-driven mechanisms (Fujimoto 1998; Pham and Bagrodia 1999).

The successful use of PDES in the various application areas might lead one to conclude that PDES is an established methodology to parallelize discrete event simulations. However, this is not the case. As Bagrodia (1996) describes the perils and pitfalls of PDES, there are a number of issues to take care of to increase the *chance* that the parallel execution of a model will yield performance benefits. Typical pitfalls hampering the parallel performance are shared variables, poor lookahead, high connectivity, load imbalance, low event or computation granularity, and high checkpoint overheads. Unger and Cleary (1993) identified four important performance parameters for Time Warp. Two of these characterize the model: granularity and time-advance; and two characterize the Time Warp executive: state saving overhead and overhead associated with each event (event list insertion, maintenance of information, message interaction). In general, the objective in partitioning the model in parallel components is to maximize both granularity and time-advance. If granularity is large, the speedup will not be constrained by the Time Warp state saving and event overheads. The time-advance is useful as an upper bound on the achievable speedup, and hence should be maximized.

An extra complication with risk in optimistic methods is that simulation code that runs correctly on a serial machine may fail catastrophically when run in parallel (Nicol and Lui 1997). This can happen when an erroneous message (a message that will be canceled in the future) arrives at an LP where the message makes absolutely no sense given the LP's state. A possible failure is for example an array index reference out of the array bounds. And although the recovery mechanism will correct the erroneous computation eventually, the LP will be aborted by this fatal index error. If the simulation modeler did not anticipate the possibility of this inconsistency, these nonsense states can be eliminated by requiring acknowledgement of anti-messages and ordinary simulation messages.

Future directions in parallel simulation research that alleviate the problems described above, both in performance and correctness, are for example application specific libraries, new languages, support for shared state. With application specific libraries, PDES can become accessible to many simulation users. In new simulation languages, new constructs and programming paradigms can be provided that are natural and easy for simulation modelers to use, and provide the information required by the parallel simulation to obtain good performance (Bagrodia 1998; Bagrodia et al. 1998).

The various aspects of optimistic parallel discrete event simulation presented in this chapter, find their application in the sequel of this thesis. Chapter 3 presents the design and implementation of a portable Time Warp simulation kernel, and discusses the application programming interface, rollback strategy, state saving strategy, and GVT computation. In Chapter 4, a performance evaluation tool is described that enables the performance evaluation of a PDES protocol compared to a hypothetical *ideal* parallel execution of the discrete event simulation. The Ising spin experiments presented in Chapter 5, show clearly the need for incremental state saving and optimism control.

Chapter 3

The APSIS Time Warp Kernel

Main Entry: **ap·sis**
1 : the point in an astronomical orbit at which the distance of the body from the center of attraction is either greatest or least
2 : APSE 2

—Merriam-Webster Dictionary

3.1 Introduction

The development of Parallel Discrete Event Simulation (PDES) applications is a complex design and implementation activity. Besides the complexity of parallel program development, the simulation modeler also has to think about the intrinsics of the PDES method, e.g., conservative versus optimistic synchronization and the consequent design and implementation details. For example, with conservative simulators the developer must be familiar with the issues of lookahead and known Logical Process (LP) connectivity, and with optimistic simulators he must be familiar with the issues of state saving and rollback. The lack of versatile parallel simulation environments or languages and performance analysis tools to simplify the development of PDES simulations, has hampered the acceptance of PDES in the simulation community.

The Amsterdam Parallel Simulation System (APSIS) addresses some of the problems identified above, by providing a platform that supports for experimental development of optimistic simulation protocols and applications, and the subsequent performance analysis. Specifically, requirements for computational science applications are taken into consideration to assess the potential of PDES methods to solve problems originating from, e.g., physics, chemistry, or biology. These requirements put special constraints on the design of the simulation environment and necessitate new extensions to the basic Time Warp method. Performance analysis should be an integrated part of PDES application development. At any instant, the PDES protocol or application developer must be able to validate the efficiency of his parallel design and identify performance bottlenecks. In Chapter 4, the design and implementation of the performance analysis tool is presented, including its integration with the APSIS

simulation environment, which is described in this chapter.

Section 3.2 presents PDES languages and environments reported in literature, and discusses the merits of simulation languages versus simulation libraries. The APSIS design requirements and decisions are discussed in Section 3.3, together with a description of the functional design of the application programming interface, the software architecture, and the hardware architecture. In Section 3.4, we describe the necessary extensions to the basic Time Warp method that are introduced to efficiently support simulations stemming from computational science. The specific implementation details of the key features of the Time Warp simulation kernel are presented in Section 3.5.

3.2 Parallel Discrete Event Simulation Environments

The migration process from sequential discrete event simulation to parallel discrete event simulation should be as smooth as possible. The user should concentrate his effort on the modeling process instead of being bothered with the details of parallel synchronization protocols. One of the most important design goals of parallel discrete event simulation environments is to provide a level of abstraction from these synchronization details in order to enhance the usability of PDES. The PDES environment hides the complexity of the synchronization protocol by providing pre-built simulation kernel(s) as well as development tools.

The various PDES environments can be described and compared by their constitutional components, namely modeling capability, programming framework, language features and library API, synchronization protocols, and system support and environment. The modeling capability determines how the physical system is modeled, i.e., the world view presented to the user (see Section 1.2.7). The programming framework (e.g., structured or object-oriented) incorporated by the PDES environment affects the development time and maintenance effort. For example, an object-oriented programming framework will reduce the development time of the simulation application. However, in general the object-oriented approach comes at a price of increased runtime overhead, and often results in slower execution speed as compared to structured languages such as C. Language features and library API provide a set of constructs to design simulation models. Runtime system support and simulation environment comprise different aspects such as logical process to processor mapping, performance evaluation, statistical information collection, and visualization and debugging capabilities.

An important feature of parallel simulation environments is whether the PDES facilities are incorporated in a (new) simulation language or in a runtime support library (Bagrodia 1998; Low et al. 1999). Simulation languages provide a full set of well-defined language constructs to design simulation models, whereas a library only provides a group of routines to be used with a

general-purpose programming language. Consequently, the conceptual modeling framework offered by the PDES environment is more prevalent in simulation languages than in libraries. Furthermore, a simulation language with the relatively high-level interface to the user allows for optimizations by a compiler that are cumbersome at the low-level interface of a library. PDES libraries, on the other hand, give the user more flexibility in controlling the simulation application in terms of the behavior of the underlying synchronization control. A knowledgeable user may fine-tune the options provided, but users must also note that if the options are not set correctly, the performance of the simulation may degrade significantly.

3.2.1 Languages

The influence of simulation-language research on the evolution of programming languages, indicates the importance of simulation to computing (Nance 1993). For example, the concept of object-oriented program design was first incorporated in the discrete event simulation language Simula 67 (Nygaard and Dahl 1978), the first object-oriented programming language. The object-oriented design methodology naturally accommodates the modeling activity in simulation, but appeared to be successful in a much broader modeling and design perspective to tackle the complexity of large software systems. The simulation languages discussed in this section are all object-oriented languages, with the exception of Parsec.

One of the most important benefits of simulation languages over simulation libraries is that simulation languages provide a more consistent framework or world view that typically makes it easier for the user to design a model. The simulation language constructs and semantics reflect the intended use and allow for a coherent transition from simulation model to simulation application. All languages discussed in this section support the process-oriented world view. A disadvantage of simulation languages is, however, that the user often needs to learn a new programming language, although there are simulation languages that are enhancements of familiar general-purpose languages. An additional advantage of enhanced general-purpose languages is their portability and a richer program development environment support.

The Yaddes system (Preiss 1989) provides an environment for constructing discrete event simulations. The principle features of the Yaddes system are the Yaddes simulation specification language and compiler, and the runtime libraries. The Yaddes language is a specification language in the style of Lex (Lesk and Schmidt 1979) and Yacc (Johnson 1979). The basic components of a Yaddes program are *model* specifications (describe general state machine), *process* specifications (create logical processes by instantiating models), and *connections* specifications (describe connections between logical processes). The Yaddes specification files are translated into C language programs that are then compiled and linked to the simulation runtime library. The runtime simulation libraries support the simulation execution mechanisms: sequential discrete event simulation, Chandy-Misra distributed discrete event simulation,

and Time Warp distributed discrete event simulation.

ModSim (Rich and Michelsen 1991) is an object-oriented simulation language based on Modula-2. It was developed under contract with the Army Model Improvement Program (AMIP) Management Office using the Jet Propulsion Laboratory's TWOS (Reiher 1990). A sequential version of ModSim, MODSIM II, was later developed and released by CACI Products. The ModSim simulation kernel was originally the TWOS operating system, while later development included also the SIM++ environment (see next section). Both simulation kernels for ModSim support exclusively optimistic simulation based on the Time Warp protocol. APOSTLE (Wonnacott and Bruce 1996) is a high-level object-oriented simulation language for PDES. APOSTLE runs on top of an existing optimistic simulator written in C++. The optimistic simulator currently used is based on the Breathing Time Buckets synchronization protocol (Steinman 1992). The APOSTLE language has support for granularity control that allows multiple events to coalesce so that the overhead of a single event is spread over many changes of state.

Bagrodia et al. (1998) developed the simulation language Parsec that provides an easy path for the migration of simulation models to operational software prototypes, implementation on both distributed- and shared-memory platforms, and support for visual and hierarchical model design. The simulation development environment supports a number of front ends for programming models: the C-based Parsec simulation language; a C++ library that can be interfaced with native C++ code; and the Parsec Visual Environment (Pave). A portable kernel executes Parsec programs on sequential and parallel architectures. Parsec programs may be executed in two modes—as (ordinary) parallel programs or as simulation models. The simulation kernel supports a sequential, three parallel conservative, and an optimistic synchronization algorithm.

The Fornax simulation language (van Halderen and Overeinder 1998; van Halderen et al. 1998) is a Java-based discrete event simulation language. The versatility of Java enables the expression of additional semantics to offer the same conceptual framework as simulation languages do. The object-oriented programming constructs in Java are extended to implement entity, event, and simulation time control objects. By extending the Java language a process-oriented simulation language is constructed, where method calls (also called event method calls) are now time-stamped interactions between entities. The interaction between entities by event methods requires that the method call (scheduling an event) and the actual method execution (handling an event) is decoupled. A number of simulation kernels are available: a sequential simulation kernel, a parallel simulation kernel exploiting parallelism on a multiprocessor using multiple lightweight processes, and a parallel simulation kernel with a distributed global clock. An optimistic simulation kernel is under development.

The performance of the sequential simulation kernel and the parallel simulation kernel using multiple lightweight processes in Fornax compares favorable with other Java-based simulation libraries and MODSIM III. A ring-

topology queueing network and an Ising spin system simulation were used for a quantitative performance evaluation. For the ring-topology queueing network simulation, the performance of Fornax is superior to MODSIM III up to 200 concurrently active queues (entities). The Fornax performance decreases for more than 200 concurrently active entities; this is due to increased multithreading overhead costs. For the Ising spin system simulation, Fornax outperformed the other Java-based simulation libraries. In the Ising spin system simulation there was no performance breakdown observed for more than 200 entities, as there are only a limited number of entities *active* at any instance of time.

Additional to the simulation kernel, the Fornax simulation environment provides a framework for visual modeling. For example, in computer architecture modeling and simulation, a set of predefined (functional) components, such as processor, memory, cache, bus, etc., can be made available in the visual modeling environment. The user designs the computer architecture using a graphical user interface and specifies the component parameters, e.g., memory size and access times. From this visual design a Fornax simulation is distilled and executed. Other promising features that are incorporated in the Fornax environment are the capabilities for Web-based simulation (Fishwick 1997) and agent-based simulation (Joshi et al. 1997).

3.2.2 Libraries

Libraries are typically composed of a simulation kernel implementing a (number of) synchronization protocol(s) and an application programming interface (API). The API provides an interface to the simulation kernel to create and initialize logical processes, to schedule events or interactions, and to finalize the parallel simulation. In this respect, simulation libraries fulfill a dualistic role. The library as such, allows a user to implement a parallel simulation in a general-purpose language such as C or C++, by calling the appropriate library functions via the API. The other role of libraries is its use to supply a simulation kernel for parallel simulation languages. Therefore, libraries and their functionality play a central role in parallel discrete event simulation.

Most PDES library environments reported in literature are research-oriented and based on optimistic protocols. In general, compared with optimistic protocols, conservative protocols can be implemented with less effort, and the optimization of the protocol is more application specific. For instance, lookahead information, which is crucial for a conservative protocol to be effective, is application dependent. Therefore, the most effective approach is for the programmer to use a general purpose parallel runtime system, and implement optimizations specific to the simulation application. The optimistic protocols are far more difficult to implement than conservative ones, due to the inherently complex and intractable nature of the Time Warp mechanism. Furthermore, the genericness of optimistic protocols makes that the method's effectiveness is less application independent, and can be optimized over a broad range of application behaviors. Thus, contrary to conservative protocols, the most effective

way is for the programmer to use a parallel simulation library that hides the implementation details of the Time Warp mechanism.

The list of PDES library environments discussed in this section is far from complete. In the presentation, the libraries are selected by their distinctiveness, and by the amount to which they integrate the various concepts and their use in PDES.

The Time Warp Operating System (TWOS) (Jefferson et al. 1987; Reiher 1990) is historically one of the first optimistic simulation environments based on the Time Warp protocol. TWOS is composed of two layers: the Time Warp layer and the kernel layer. The TWOS program interface hides the underlying kernel layer and hardware. Thus, the programmer cannot access raw hardware, nor can he use the underlying kernel layer beneath the level of the TWOS interface. Although all interactions with the virtual machine are performed through TWOS, it is not an interactive operating system. In its current form, it is linked with the simulation to form a single executable. Interesting features of the TWOS environment are dynamic object creation and dynamic load management. Dynamic object creation is a complicated problem. Objects are created dynamically upon requests of other objects in the simulation. Such a request may be part of an erroneous computation that will eventually be rolled back. Therefore, the dynamic creation may need to be undone. Either the actual creation must be delayed until the commit point, or the entire creation must be able to be undone. Dynamic load management allows the simulation to obtain good performance by dynamically balancing computational work over the nodes based on the effective utilization, which is the fraction of useful, committed work on the simulation.

SPECTRUM (Reynolds and Dickens 1989) is a test bed for designing and evaluating parallel simulation protocols. The test bed supports experimentation on a full range of protocols in a common environment by relying on *filters* exclusively to implement parallel protocols. By specifying filters for the actions like initialization, get-next-event, post-event, advance-time, and post-message, a specific protocol instantiation can be implemented. With respect to performance comparisons, it is recognized that the SPECTRUM test bed is no substitute for carefully crafted implementations of protocols for a given architecture.

The Georgia Tech Time Warp (GTW) (Das et al. 1994) system is specifically designed for cache-coherent shared-memory multiprocessors. The simulation kernel is based on the Time Warp mechanism, and is designed to support efficient execution of small granularity discrete event simulation applications. This design objective necessitates a simple program interface that can be efficiently implemented. More sophisticated mechanisms can be implemented as library routines. For example, GTW supports an event-oriented world view, but more complex world views such as process-oriented simulation can be built on top of the GTW kernel. A number of techniques are incorporated in GTW to enable efficient parallel execution of small-grained simulation programs. Some techniques are also applicable to message-based machines, but the most important, e.g., buffer management mechanism, GVT algorithm, and maintaining locality of state vector information, are specific to cache-coherent shared-memory

machines.

ParaSol (Mascarenhas et al. 1995) is a multi-threaded system for shared-memory and distributed-memory parallel simulation. ParaSol is designed to support active-transactions flow in process-interactive simulations. The idea here is to develop a process-style description of a transaction's activity as it flows through a system (similar to popular sequential simulation languages such as CSIM, GPSS, or SIMAN). Here, transactions are implemented via time-stamped threads which transparently migrate between processors to access model resources. The use of mobile threads has several advantages: locality of reference, potential for load balancing, and simplicity in application-level coding. On the other hand, the complexity of working with threads instead of messages is significant, and poses serious challenges to efficient kernel operation.

The primary objective of the Parsimony Project (Preiss and Wan 1999) is the development of a Java-based test bed for distributed, network centric simulation. Parsimony exploits the features of Java to provide a flexible distributed simulation environment while hiding many of the simulation details from the user. However, different than Fornax, Parsimony requires explicit send and schedule method calls to model time-stamped interactions between entities. The Parsimony package includes three different distributed simulators. A distributed simulator is one in which a given user-defined simulation is executed using multiple Java Virtual Machines running on different host computers.

Other parallel simulation libraries not discussed here are SPEEDES (Steinman 1992), SIM++ (Baezner et al. 1994), SimKit (Gomes et al. 1995), WARPED (Radhakrishnan et al. 1996).

3.3 Design of the APSIS Environment

3.3.1 Requirements and Design Goals

The projected use of the APSIS simulation environment is primarily in the field of dynamic complex systems (Sloot et al. 1997) such as asynchronous cellular automata. Typical examples of asynchronous cellular automata are continuous-time Ising spin systems or population dynamics models. Without going further into the details of these application classes, we can characterize the application classes as data intensive, thus requiring efficient memory management, and in need of dynamic entity creation and deletion. These application requirements put constraints on the design of the APSIS environment.

The APSIS environment is a research vehicle for both optimistic simulation protocol design and evaluation, and parallel simulation development of dynamic complex systems. For effective parallel simulation development we need to hide the complexity of the optimistic synchronization protocol by providing a simulation language or library. The protocol design and evaluation asks for a flexible environment that is easily adapted to incorporate new concepts and is extendible to interface with the environment for evaluation of the

concepts. From a simulation modeling perspective, the model validation and simulation verification are important. This requires determinism: given the same input, a simulation should produce identical results, regardless of the number of processors or the mapping of the processes to processors used.

Operational requirements are transparent scalability and parallel efficiency. The envisaged parallel platforms range from massively parallel processors to networks and clusters of workstations. The simulations should be executable on varying number of processors and with different mappings of processes to processors without source code modifications. Over the various parallel platforms, efficient execution of the simulation should be guaranteed. Closely related to the transparent scalability and parallel efficiency, is the portability of the parallel simulation. Portability and efficient network support allows the development of the parallel simulation on a workstation, while ensuring that the simulation can be moved to a parallel processor for production runs.

3.3.2 Overview

The APSIS system contains a development and execution environment for parallel simulations. The parallel runtime executive is an optimistic simulation kernel based on the Time Warp protocol. Simulation of applications stemming from dynamic complex system requires efficient execution of small grain events. The basic world view supported by the simulation environment is event scheduling. More complex world views such as process-oriented (see Section 1.2.7) can be implemented on top of the simulation kernel. Furthermore, the data intensive characterization of dynamic complex systems motivates support for aggregation of entities in subdomains, allowing for data decomposition which closely relates to the modeling practices of the problem field. Although special considerations are taken for dynamic complex systems, APSIS is a general parallel discrete event simulation system that can be used for computer network simulations, personal communication services simulations, or aviation applications.

The simulation model adopted in APSIS is the well-known model where physical processes of a real-world system are represented by logical processes that interact with each other via time stamped event messages (see Section 2.2). A model of a physical process is the description of the state and the behavior of the associated logical process. The simulation system is than a network of instantiated models. The runtime support system of APSIS must therefore support the instantiation and the concurrent execution of models. For the instantiation of simulation models, the simulation environment provides initialization and virtual topology constructor functions. Problems with a regular interaction between the logical processes are easily instantiated with environment provided virtual topology constructors for, e.g., grid or torus topologies. Irregular topologies can be specified by the application user. The topology constructor also maps the logical processes to the parallel processors. For the concurrent execution, the simulation runtime support relies on message pass-

ing libraries such as MPI, PVM, etc. The message passing libraries abstract from the underlying hardware, MPP or cluster of workstations, and allows for portable program development that is scalable over the number of processing nodes. The supported programming model is single program, multiple data (SPMD) (Hwang 1993).

The PDES facilities of the APSIS environment are supported by a library. As APSIS is a research vehicle, the environment must be flexible and easily extendible. Languages provide a higher abstraction level, definitely preferable from a modeling perspective, but are less pliant to changes than libraries. Furthermore, knowledgeable users can fine tune the library parameters, such as state saving method, cancellation strategy, or optimism control, to attain the best performance with respect to the application class. The relative simple interface of the library can be efficiently implemented. If the various design dimensions have been crystallized, a production simulation kernel can be accommodated by a simulation language like Fornax (van Halderen and Overeinder 1998).

In Fig. 3.1 the APSIS architecture is shown. Four layers are depicted in the figure. The hardware layer is the parallel platform, which can range from a cluster of workstations to a massively parallel processor architecture. The communication libraries MPI, PVM, CK*, Panda†, etc., abstract from the specific hardware by providing a uniform, portable interface to the underlying parallel platform, i.e., the communication infrastructure. The APSIS library incorporates a number of interfaces to the various message passing layers. The simulation kernel schedules the events over the parallel architecture and coordinates the distributed logical processes using the message passing interface. The instantiation of the simulation model and the mapping of the logical processes by the topology functions makes use of the message passing layers. To the upper layers, the APSIS library supports an application programming interface for simulation initialization, event scheduling, and simulation finalization. The library is implemented in the C language, and supports both an API for C as well as C++.

3.3.3 The Application Programming Interface

The APSIS application programming interface includes a set of routines to design and implement parallel simulation applications. We organize the description of the interface routines according to their functional use: simulation instantiation, event scheduling and execution, virtual time management, memory management, and environment configuration and information.

*The Communication Kernel is a lightweight communication layer designed for simplicity and efficiency (Overeinder et al. 1995).

†In the APSIS environment, the PVM communication library (called PANPVM) implemented on top of Panda is used.

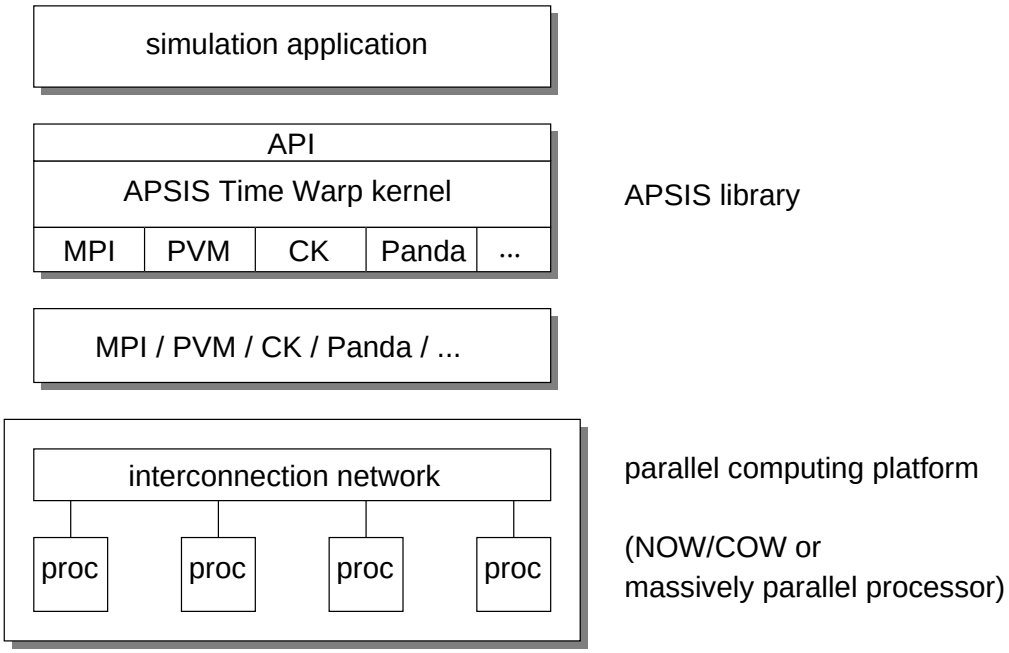


Figure 3.1: Overview of the APSIS parallel simulation environment architecture.

Simulation Instantiation

The APSIS initialization function instantiates the simulation model by creating the logical processes (LPs) making up the the parallel simulation. The virtual topology constructor functions allows the simulation programmer to describe the LP interaction pattern that is logically similar to the simulation model, thus abstracting from the particular parallel architecture, which can have different physical interconnection topologies. Tables 3.1 and 3.2 summarize the instantiation functions.

`apsis_init`	initialize the APSIS execution environment
`apsis_finalize`	terminates APSIS execution environment
`apsis_abort`	aborts APSIS execution environment
`apsis_pid`	process identifier of calling process

Table 3.1: Initialization, finalization, and environment routines.

The initialization function `apsis_init(int *argc, char ***argv)` initializes the APSIS execution environment and creates the logical processes (currently Unix processes, future development will include threads). The programming model is single program, multiple data (SPMD), hence all logical processes start the same executable. `apsis_init` accepts the `argc` and `argv` that are provided by the arguments to the ANSI C `main` function. The first argument in `argv` must be the number of logical processes in the parallel

simulation. This argument is used and discarded by `apsis_init`. The other remaining arguments are available to the simulation program.

The function `apsis_finalize(void)` cleans up all APSIS state and starts a termination detection algorithm to finalize the parallel simulation. The function `apsis_abort(int errorcode)` aborts all the logical processes. The error code is returned to the Unix or POSIX environment.

The process identifier of a logical process can be determined with the function `apsis_pid(void)`. The function returns an integer number i, where $0 \leq i \leq N-1$ and N is the number of logical processes.

`apsis_cart_create`	create Cartesian virtual topology
`apsis_cart_free`	free Cartesian virtual topology data structure
`apsis_cart_pid`	determines LP pid given Cartesian location
`apsis_cart_coords`	determines Cartesian coords given LP pid
`apsis_graph_create`	create graph virtual topology
`apsis_graph_free`	free graph virtual topology data structure
`apsis_graph_neighbors`	return the neighbors of an LP within a graph

Table 3.2: Virtual topology routines.

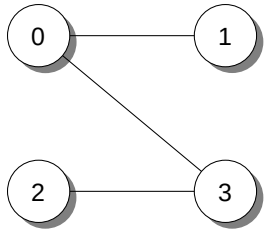
The APSIS virtual topology functions are very similar to the MPI virtual topology functions. The generic virtual topology functions can be efficiently implemented on top of a message passing layer and allow the definition of a rich set of topologies.

Function `apsis_cart_create(int ndims, int *dims, int *periods)` can be used to describe Cartesian structures of arbitrary dimension. The number of dimensions is specified in `ndims`, and the number of logical processes in each dimension in array `dims`. For each coordinate direction one specifies whether the process structure is periodic or not. For a 1D topology, it is linear if it is not periodic and a ring if it is periodic. For a 2D topology, it is a rectangle, cylinder, or torus as it goes from non-periodic to periodic in one dimension to fully periodic. The topology translation functions `apsis_cart_pid(int *coords, int *pid)` and `apsis_cart_coords(int pid, int ndims, int *coords)` provide a mean to determine the Cartesian coordinates of an LP and the pids of its neighbors.

`apsis_graph_create(int nnodes, int *index, int *edges)` creates a graph topology. The three parameters `nnodes`, `index`, and `edges` define the graph structure. `nnodes` is the number of nodes of the graph. The nodes are numbered from 0 to `nnodes-1`. The ith entry of array `index` stores the total number of neighbors of the first i graph nodes. The lists of neighbors of nodes 0, 1, ..., `nnodes-1` are stored in consecutive locations in array `edges`. The array `edges` is a flattened representation of the edge lists. The total number of entries in `index` is `nnodes` and the total number of entries in `edges` is equal to the number of graph edges.

The definitions of the arguments `nnodes`, `index`, and `edges` are illustrated in the following example.

process	neighbors
0	1, 3
1	0
2	3
3	0, 2



Then, the input parameters are:

```
nnodes = 4
index =  (2, 3, 4, 6)
edges =  (1, 3, 0, 3, 0 , 2)
```

Thus, `index[0]` is the degree of node zero, and `index[i] - index[i-1]` is the degree of node `i`, `i = 1, ..., nnodes-1`; the list of neighbors of node zero are stored in `edges[j]`, for $0 \leq$ `j` $\leq$ `index[0] - 1` and the list of neighbors of node `i`, `i` $>$ `0`, is stored in `edges[j]`, `index[i-1]` $\leq$ `j` $\leq$ `index[i] - 1`.

The information function `apsis_graph_neighbors(int pid, int maxneighbors, int *neighbors)` returns the array `neighbors` with pids that are neighbors to the specified logical process.

Event Scheduling and Execution

The basic simulation world-view incorporated by the APSIS environment is event-scheduling. In event-scheduling, a model's execution is viewed as a sequence of events, where an event can be represented by a message. The APSIS environment provides two functions to schedule and execute the events of the parallel simulation in causal order. A third (de-) scheduling function, `apsis_cancel`, is discussed in Section 3.4.1.

`apsis_send`	schedule an event for future execution
`apsis_recv`	receive an event for execution
`apsis_cancel`	cancel an event

Table 3.3: Event scheduling and execution routines.

The event scheduling function is `apsis_send(int dest, vtime ts, void *buf, int count)`, which sends a message `buf` of size `count` to destination `dest` with timestamp `ts`. The timestamp of type `vtime` can be either an integer or a floating point value. The net effect of a call to this function is the scheduling of an event at LP `dest` for execution at simulation time `ts`. The contents of the event message `buf` is non-specified by APSIS and depends on the simulation application.

A call to `apsis_recv(void *buf, int count)` retrieves the next pending event for execution. Indirectly, the retrieval of the pending event sets the

LVT of the LP to the timestamp of the event message. The event messages received over successive calls to `apsis_recv` are guaranteed in non-decreasing timestamp order, as the LPs must adhere to the local causality constraint.

The simplicity of the two routines `apsis_send` and `apsis_recv` is almost misleading, as it is the full complexity of the Time Warp protocol that realizes the local causality of the individual LPs. The details of the subtle interplay between the two routines are presented in Sections 3.3.4 and 3.5.1.

Virtual Time Management

Table 3.4 shows the virtual time information inquiry functions. The function `apsis_gvt(void)` returns the current global virtual time known by the Time Warp kernel of the calling LP. `apsis_lvt(void)` returns the current local virtual time of the LP, that is the timestamp of the current event message being processed by the LP.

`apsis_gvt`	global virtual time
`apsis_lvt`	local virtual time

Table 3.4: Virtual time management routines.

Memory Management

The APSIS environment does not include transparent, or implicit, state saving, thus the simulation application must explicitly save state changes for potential future rollbacks. The interface function to the APSIS state saving mechanism is shown in Table 3.5. The APSIS environment incorporates copy state saving and incremental state saving. (The design of incremental state saving in APSIS is further discussed in Section 3.4.2.)

`apsis_state_save`	incremental state save

Table 3.5: Memory management routines.

Independent of the selected state saving mechanism, copy or incremental state saving, the interface function is `apsis_state_save(void *buf, int count)`. The function saves the (partial) state vector `buf` of size `count` bytes into a simulation kernel data structure.

Environment Configuration and Information Routines

To allow for a flexible and extendible simulation kernel, a generic interface function to the Time Warp simulation kernel is provided by the `apsis_attr_set(int attr, void *value)` and `apsis_attr_get(int attr, void *value)` pair (see Table 3.6). These interface functions enable

knowledgeable users to fine tune the simulation kernel to their simulation application. The simulation kernel also has four debugging modes (in increasing detail of debugging messages), which can be set with `apsis_debug(int mode)`. (The debugging mode can also be set via `apsis_attr_set`, but for convenience and historical reasons the `apsis_debug` function is included.)

`apsis_attr_get`	retrieve library information
`apsis_attr_set`	set library configuration
`apsis_debug`	set library debug level

Table 3.6: Environment configuration and information routines.

Currently, the following simulation kernel attributes can be set: state saving method and virtual time window. The state saving method can be set to copy state saving or incremental state saving (default). The virtual time window controls the optimism of the Time Warp protocol by limiting the execution of events within a window of virtual time beyond the global virtual time (see also Section 2.4.5). Other simulation kernel attributes that can be requested are the number of processed events, committed events, and rolled back events.

3.3.4 The Software Architecture

The API – Time Warp Kernel Interaction

The application programming interface (API) described in the previous section, interacts with the Time Warp simulation kernel to orchestrate the distributed execution of events over the parallel platform. The instantiation functions are the interface to the Time Warp kernel module for logical process creation and topology definition. The event scheduling and execution functions, and the state saving function interact with the synchronization module of the simulation kernel. The functions to set kernel attributes work on specific parts of the Time Warp kernel, and the inquiry functions (including `apsis_gvt` and `apsis_lvt`) do not change the state of the simulation kernel.

The Time Warp Kernel

The APSIS Time Warp kernel is composed of three functional modules, namely the instantiation, synchronization, and GVT computation module (see Fig. 3.2). The modules work independent from each other; the instantiation module is only accessed during simulation startup and initialization. The modules do not interfere with each computation, but do asynchronously exchange information between each other. From Fig. 3.2 one can see that the instantiation and synchronization modules can be accessed via the API. The GVT computation module cannot be accessed by the application (with the exception of the `apsis_gvt(void)` routine, which only returns the current GVT). All modules make use of the underlying message passing layer.

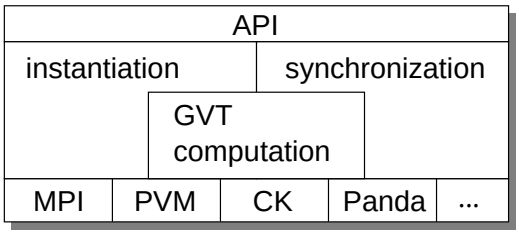


Figure 3.2: Overview of the modules in the APSIS Time Warp kernel.

Instantiation The instantiation of the simulation model consists of APSIS environment initialization and logical process creation. The creation of logical processes relies on the facilities supported by the underlying message passing layer. Although the intrinsics of process creation are different for the various message passing layers, APSIS provides one single method for process creation that is translated to the particular message passing layer, see the next section on the message passing interface. With process creation, the message passing layer is also initialized. Apart from process creation, the instantiation initializes the data structures used by the synchronization and GVT computation modules.

The virtual topology construction, if requested by the simulation application, is translated to the underlying message passing layer. For message passing layers that incorporate the concept of virtual topologies, the virtual topology construction is effectively implemented using the message passing layer topology constructors. For message passing layers without the notion of virtual topologies, such as PVM, the virtual topology construction is implemented by translating the mapping of the logical processes to the processors according to the defined virtual topology.

Synchronization The synchronization module implements the forward simulation and rollback protocol. The essential data structures for realizing the transparent rollback-based synchronization protocol for parallel simulation are the *input queue*, *output queue*, and *state queue*. The input queue, or event queue, data structure contains the events of the simulation in timestamp order. New scheduled events are inserted at the appropriate place, in timestamp order, in the queue. Closely associated with the input queue are the output and state queue, which contain respectively the event message sent and state changes due to the execution of an event (see for details of the data structure Section 3.5.1).

The interplay between scheduling and execution functions, and the local data structures of the Time Warp kernel is shown in Fig. 3.3. The receipt of an event (process recv in Fig. 3.3), retrieves the next pending event from the input queue. The execution of this event (process exec) can result in a number of state changes, which are stored in the state queue. The execution of the event can also induce a number (zero or more) of new events to be scheduled,

by sending event messages to their destination (process send). Copies of the event messages are stored in the output queue. All side effects resulting from the execution of an event are directly associated with that event. Thus, all state change and event message entries in their respective queues are linked with the "responsible" event in the input queue.

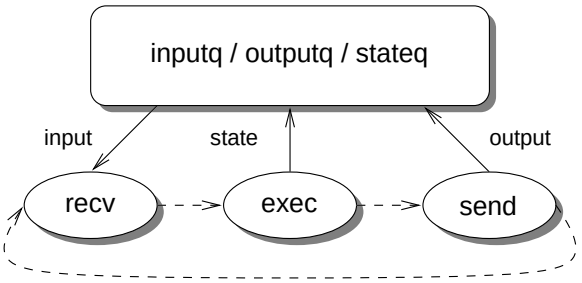


Figure 3.3: Scheduling and execution of events, and the interaction with the input, output, and state queue.

The rollback mechanism efficiently annihilates the scheduled events and undoes the state changes by using the associated data structures. Upon detection of a causality error, that is the receipt of a straggler event with a timestamp smaller than the current LVT, the rollback mechanism resets the LVT to the timestamp of the event directly before the straggler. All premature executed events with their associated side effects are nullified by running down the output and state queue, and sending out the anti-messages and replacing the state changes with their original values from the state queue. The rollback mechanism can be seen as the reverse loop of the normal forward simulation as shown in Fig. 3.3.

Global Virtual Time Computation The Global Virtual Time (GVT) computation module coordinates a number of activities. First, of course, the GVT computation itself, that is the estimation of the smallest timestamp of the unprocessed events in the system. The second important task of GVT computation is fossil collection. And finally, GVT computation is used for distributed termination detection of the parallel simulation.

The GVT algorithm used in the APSIS environment is based on the algorithm proposed by Bauer et al. (1991, 1992). The underlying message passing layers provide error-free communication channels with *first in, first out* (FIFO) behavior. The APSIS message passing interface to the underlying message passing layer such as MPI, PVM, etc., enumerates all event messages through a certain communication channel, resulting in a unique event message number per destination. The GVT module locally administers the number of messages sent and received, and the minimum timestamp of an event message since the last GVT report. Periodically, the LPs send their local information and their LVT to the GVT manager, which deduces from the information the minimum of the transient message timestamps and LVTs (see Section 3.5.4). The GVT

manager is a designated GVT computation module, for example the logical process with process identifier equal to zero.

The fossil collection task of the GVT module consists of freeing unused memory and committing irrevocable operations such as I/O. With each GVT update, the memory resources consumed by the input, output, and state queue are reclaimed. Because the LPs never rollback to a simulation time earlier than the GVT, it is safe to reclaim memory resources for events with timestamps less than GVT, together with their associated output and state queue entries. An important performance parameter of the GVT computation and fossil collection is the GVT update frequency. The three factors determining the optimal GVT update frequency are computational overhead, communication overhead, and memory usage. Computational overhead and communication overhead are minimal at low frequency, while optimal memory usage requires high frequency such that unused memory is reclaimed regularly. From experiments it appears that the communication overhead, that is the communication latency, is the most important factor in the determination of the optimal GVT update frequency. In case of low communication latency ($\pm 20\,\mu$sec), update frequency can be as high as 20 Hz, while in case of high communication latency (± 1 msec) the frequency can be as low as 0.5 Hz. Sensitivity analysis shows that frequency parameter setting is robust, that is, for large parameter ranges the GVT update performance is (sub-) optimal.

The GVT computation is also involved in termination detection of the parallel simulation. With the Time Warp protocol, the parallel simulation continues as long as there are events to be processed. If no events are pending for execution at an LP, the LVT of the LP will be set to $+\infty$. If all LPs will have their LVTs set to $+\infty$, and there are no transient event messages, then no more events are pending in the parallel simulation. With the next GVT computation, the new GVT value will be $+\infty$. This will trigger the termination phase of all LPs to finalize the parallel simulation.

The Message Passing Interface

The design of the APSIS message passing interface abstracts from the specific message passing layer used on the parallel platform. All Time Warp kernel routines access the message passing layer via the APSIS message passing interface. To support the APSIS library for a specific message passing layer, eight functions have to be supplied for program initialization and finalization, send and receive, and creation and freeing of Cartesian and graph topologies. The required send primitive is asynchronous (or buffered-mode in MPI terminology) and the receive primitive is nonblocking.

3.4 Extensions to the Time Warp Kernel

In this section we introduce two extensions to the Time Warp kernel that enlarge the effective application of the Time Warp protocol to highly dynamic

systems and simulations with arbitrary large states. First, we introduce a new primitive to retract scheduled future events. This allows us to actively remove simulation entities including their scheduled events. Second, the complication with applications with large states is solved by providing an alternative mechanism for copy state saving (checkpointing), called incremental state saving, which only saves the difference between two states.

3.4.1 Event Retraction

Many challenging simulations are not efficiently supported by the original formulation of the Time Warp protocol. For example, consider a population dynamics simulation model consisting of predators hunting for preys. Both predators and preys schedule their events for future activities, such as move, eat, and breed. Part of the modeled behavior of the predator is to catch a prey, resulting in the removal of the prey from the simulation. The elimination of the prey invalidates the previously scheduled events that initiate an activity of the non-existent prey. A convenient way of modeling this behavior is to have the predator "catch" event retract the original prey events in order to cleanup any trace of the prey. Although it is possible to simulate such activities without the ability to retract events, the availability of an event retraction primitive simplifies the simulation model, making it easier to understand and maintain (Overeinder and Sloot 1993).

At first glance, event message retraction seems to be quite similar to message annihilation in rollback, so message retraction could be implemented by sending the corresponding anti-message for the positive message that is retracted. However, there is one important difference: message retraction is part of the optimistic simulation and can potentially be rolled back, while message annihilation in rollback is part of the synchronization protocol which removes *any* reference to the event message in the parallel simulation. Thus, although the mechanism for message retraction is rudimentary available in the Time Warp protocol, we need to extend the annihilation mechanism to allow for rollback of a message retraction.

The new event retraction primitive is designed to interact with the existing annihilation mechanism without increasing the overhead of the Time Warp mechanism where the primitive is not used. Similar to rollback, the retraction primitive cancels an event message by sending the corresponding anti-message to the logical process that received the original message. But, in addition, a *positive* copy of the message is placed into a new data structure called the *cancel queue* that is associated with the logical process that invoked the retraction primitive. If the event that invoked the retraction primitive is rolled back, we send the positive copy of the message to the receiving process. Just as the output queue maintains the information necessary to roll back previous sent event messages, the cancel queue maintains the information to undo invocations of the retraction primitive.

Lomow et al. (1991) proposed a similar design for a mechanism for user-invoked retraction of events in Time Warp as described in this section. Agre

and Tinker (1991) introduced a retraction mechanism that interacts with the GVT computation and fossil collection mechanism. In their approach, event retraction is an irrevocable operation and is committed during fossil collection. Committing the event retraction consequently delays the execution until the GVT sweeps past the timestamp of the event retraction. An advantage of this approach is that a cancel queue is not necessary because the event retraction cannot be rolled back.

The Retract Primitive

The event retraction function `apsis_cancel` is conceptually the inverse operation of the event scheduling function described in Section 3.3.3. The schedule primitive `X = apsis_send(dest, ts, buf, count)` sends a positive copy of message `buf` with timestamp `ts` to logical process `dest`. The schedule function is extended to return an identifier `X`, the descriptor of the message, which can be used to refer to the generated message, e.g., to retract it.

The function `apsis_cancel(X)` retracts a previously sent message whose descriptor is `X`. The retract function discards all effects of the event simulation. If the message is not yet executed, the event message is removed from the input queue. In case the event is executed, the destination logical process is rolled back, effectively undoing all side effects, the corresponding event message is removed, and the simulation continues as if the event was never scheduled. Because retraction is the inverse of event scheduling, `apsis_cancel(apsis_send(dest, ts, buf, count))` is equivalent to a no-op. The `apsis_cancel` is only meaningful from a modeling perspective if the virtual time at which the retraction was issued is smaller than the virtual time at which the event is scheduled for execution. A sanity check tests whether this precondition is met before the retraction is executed.

3.4.2 Incremental State Saving

The design and implementation of incremental state saving (ISS) in the APSIS environment is an essential feature to effectively support the simulation of dynamic complex systems. The motivation and merits of ISS are extensively described in Section 2.4.4. In this section we present the details of the incorporation of ISS in the APSIS environment.

One of the design goals formulated in Section 3.3 specified efficient support for simulation applications stemming from dynamic complex systems, and in particular the class of asynchronous cellular automata (see for definition of asynchronous cellular automata Section 5.2). Asynchronous cellular automata are data intensive applications that easily consume all available physical memory. In terms of parallel performance, this application class falls in the category of memory-bounded speedup models (Sun and Ni 1993): the idea is to solve the largest possible problem limited by memory space. For data intensive simulation applications it is inefficient and even often infeasible to save copies of the complete state (even with periodic state saving). However, the execution of

an event in this application class only induces small changes local to the state vector, thus it is appropriate to save only the updated parts of the state using the incremental state saving technique.

The incremental state saving mechanism developed and implemented in the APSIS environment exploits the Markovian behavior of the state evolution. The parallel simulation of the dynamic complex system is spatially decomposed into a number of subdomains. Each subdomain is an aggregation of cellular automata represented by a logical process. The state of the logical process can be seen as a vector of the states of all the cellular automata it contains. The state of the logical process can now be written as $s = (s(a_1), \ldots, s(a_N))$, where N is the number of aggregated cellular automata in the subdomain.

The execution of an event $e_i^t, 1 \leq i \leq N$, scheduled at automaton a_i for virtual time t, only changes the state of automaton a_i by the locality of the transition rules of the cellular automata. Thus the execution of the event results in the new state vector $s' = \big(s(a_1), \ldots, s'(a_i), \ldots, s(a_N)\big)$. Instead of saving the complete state vector, it is sufficient to save the old state $s(a_i)$. The incremental state saving mechanism associates state $s(a_i)$ now with event e_i^t.

The APSIS environment incorporates copy state saving (CSS) and incremental state saving (ISS). The state saving method can be set with the `apsis_attr_set` simulation kernel interface function. With the `apsis_state_save` function a (partial) state can be saved to the state queue. This function is both used for CCS and ISS. The generic functions `apsis_attr_set` and `apsis_state_save` enable the simulation application to switch dynamically between CSS and ISS. However, it is the responsibility of the simulation application that with the specific state saving method the proper state information is saved with `apsis_state_save`. If the Time Warp kernel is in CSS mode, state recovery during rollback is accomplished by restoring the state vector directly form the state queue. In ISS mode, the simulation kernel reconstructs the state by restoring each saved state variable in reverse order, until the last event with a timestamp just before the event that caused the rollback.

Incremental state saving requires less state saving time and memory, at the cost of state reconstruction. The efficiency of the incremental state saving method compared to the copy state saving method depends on the rollback length and the percentage of the state that is modified due to the execution of an event. The tradeoff between copy and incremental state saving is discussed in detail in Section 2.4.4.

3.5 Implementation Aspects of the Time Warp Simulation Kernel

3.5.1 Simulation Kernel and Data Structures

The central activity of the Time Warp simulation kernel is the management of the *input queue*, *output queue*, *state queue*, and *cancel queue* data structures. All synchronization, i.e., forward simulation and rollback, and fossil collection

operations work on these data structures. Figure 3.4 shows the overall design of the queue data structures in the simulation kernel.

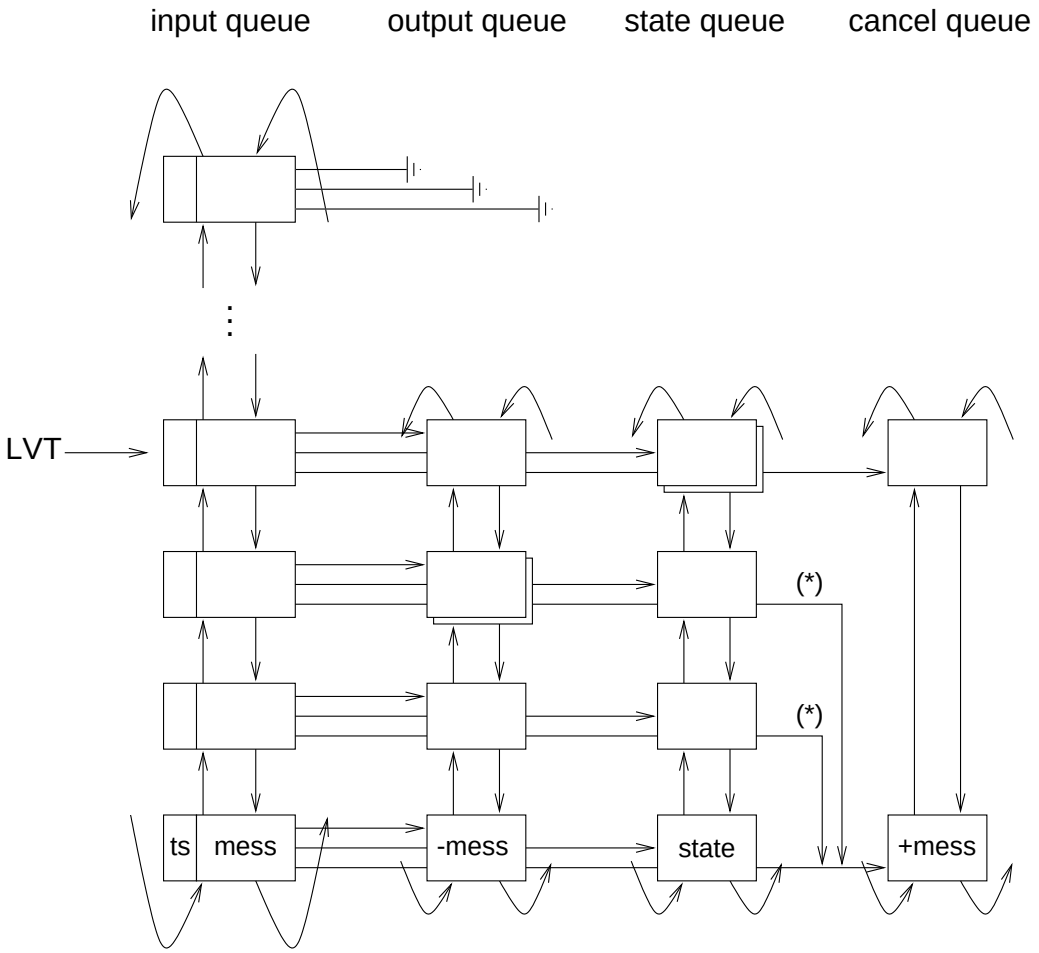


Figure 3.4: The APSIS Time Warp kernel input, output, state, and cancel queue data structure.

The input, output, state, and cancel queues are priority queues. A priority queue is a data structure for maintaining a set of elements, sorted according to an associated key value. Conceptually the queues are doubly linked lists, as shown in Fig. 3.4, although the implementation can be a calendar queue (Brown 1988), a heap (Cherkassky et al. 1996), or another data structure (Rönngren and Ayani 1997), as long as there is a previous–next relation between the queue elements. The doubly linked list data structure allows for almost effortless traversal of the queues in both directions, which is essential in forward simulation and rollback processing.

The input queue is the defining data structure, which maintains the event messages in timestamp order. The input queue is called *defining* with respect to the other queues because during insertion or deletion operations on the input queue, the elements in the input queue are maintained in timestamp order while the other queues are manipulated according to the changes to the input

queue. An input queue element has three references to its associated output messages in the output queue, its saved state changes in the state queue, and its event retractions in the cancel queue. In this data structure it is possible that a multiple number of output message, saved state changes, or event retractions are associated with one event execution. On the other hand, if the execution of an event does not result in any output message, state change, or event retraction, the respective reference directs to the element of the previous event. For example in Fig. 3.4, the second and third event message in the input queue have their cancel queue reference directing to the cancel queue element of the first event message in the input queue (see references labeled with (*) in Fig. 3.4).

The LVT points to the current event message being processed. All event messages received but not yet processed, have their associated output queue, state queue, and cancel queue references set to *nil*.

3.5.2 Synchronization

The simulation synchronization operations of the Time Warp kernel are forward simulation (event scheduling and execution), event retraction, and rollback. All operations manipulate the queue data structures in one way or another. Forward simulation and event retraction construct the data structures progressively, and rollback consistently reconstructs the data structures to an earlier virtual time.

Forward Simulation

Forward simulation consists of event scheduling and execution. The `apsis_recv` retrieves the next pending event for execution from the input queue. The operation sets the LVT (see also Fig. 3.4) to the current event timestamp. If the execution of the event results in one or more state changes, the original values of the state variables are copied to the state queue. The scheduling of a new event is accomplished with the `apsis_send` function, which sends an event message to the destination LP and stores a negative copy of the event message in the output queue. If the execution of the event is completed, the next pending event is selected with `apsis_recv`. The semantics and logical "correctness" of simultaneous events are defined and resolved by a tie-breaking mechanism imposed by the simulation kernel (Mehl 1992; Wieland 1997). Multiple events scheduled by an identical process for the same simulation time are executed in a FIFO fashion. For ties which occur as a result of scheduling from two different processes, priority is given to the event scheduled by the process with the smallest APSIS process identifier (`apsis_pid(void)`).

Optimism control (see Section 2.4.5) is also part of forward simulation. The optimism of the Time Warp protocol is controlled by limiting the execution of events within a window of virtual time beyond the global virtual time. The time window can be set with the `apsis_attr_set` simulation kernel interface. If the timestamp of the next pending event falls outside the virtual time window,

the execution of the event is throttled by blocking the `apsis_recv` function until the virtual time window is advanced (after a GVT update).

Event Retraction

Although event retraction can be considered to be part of the forward simulation, that is, it is part of the simulation application rather than the "invisible" simulation protocol, it is discussed separately for clarity.

For the implementation of the event retract function `apsis_cancel`, we can identify three situations. First, the LP which retracts the event is also the originator of the positive event message. Second, the LP which retracts the event is the destination of the positive event message. And third, the LP which retracts the event is neither the originator nor the destination of the positive event message.

In the first situation, with the message descriptor the original message can be found in the output queue (negative copy). In the second situation, the original message can be found in the input queue (positive message). In both situations, a negative copy of the event message is sent to the destination LP and a positive copy is stored in the cancel queue. The third situation is not implemented in our scheme, as there is no efficient method to retrieve a copy of the original message, and from a modeling perspective it is quite unlikely there is any need to support this situation.

We can give examples of these three event retraction situations in an individual-based population dynamics simulation (similar as discussed in Section 2.4.4). The population dynamics simulation, where predators and preys struggle for life, is spatially decomposed over a number of subdomains (assigned to different logical processes). The predators and preys have a certain (predefined) range of interaction, that is the distance a predator can see or smell a prey and hunt it down to eat it. The subdomains have a boundary region, which width is equal to the range of interaction distance. The first type of event retraction happens when a predator kills a prey within one subdomain: all future events scheduled for the prey have to be retracted. The second type of event retraction occurs when a predator in the boundary region kills a prey in the boundary region of another subdomain. Since changes in the boundary regions are communicated to neighboring subdomains, positive or negative copies of the event messages scheduled for the prey are available, and hence are retracted. The third situation, which is not implemented in the APSIS simulation kernel, would occur when a predator detects and kills a prey on a distance that is larger than the predefined range of interaction: this situation is precluded by the simulation model.

Simulation Kernel Send/Receive Pair

The APSIS interface functions for scheduling, receiving, and retracting events are handled by the send and receive functions in the simulation kernel. The simulation kernel send function routes the event message scheduled by the

`apsis_send` function call to the destination LP. The event message routing is optimized such that the delivery of messages at local destination LPs is actually a storage operation of the event into the input queue. Event messages with remote destinations are sent by the underlying message passing layer such as MPI or PVM. The simulation kernel receive function receives the event messages from the underlying message passing layer and stores the event into the input queue. The kernel receive function is implicitly called each time the `apsis_recv` function is called from the simulation application.

With the delivery of an event message, the event is stored into the input queue. The delivery of a local event message (or internal event) is directly handled by the send function, and the delivery of a remote event message (or external event) is processed by the receive function. The effect of input queue addition depends on the sign of the event message. A positive event message is inserted into the input queue at the appropriate place according to its timestamp. If a negative message (or anti-message) is inserted into the input queue, the negative message annihilates with the positive message in the input queue, thus removing the positive message. If the timestamp of the event message (either positive or negative) is smaller than the current LVT, a causality error has occurred and the rollback mechanism is triggered.

Rollback

Rollback is the basic synchronization mechanism to recover from causality errors. First the input queue is rolled back by resetting the LVT to the timestamp of the straggler event. Next, all side effects are undone by rolling back the output, state, and cancel queues. Rollback of the output queue is accomplished by sending the anti-messages to annihilate with the premature sent positive event messages. Similarly, rollback of the cancel queue incorporates sending the positive message, but an optimization is possible. If both the event retraction and the scheduling of the original event are rolled back, the positive message in the cancel queue and the anti-message in the output queue are removed and annihilated. Neither message needs to be sent to the receiving LP.

The rollback method of the state queue depends on the state saving mode: CSS or ISS (see Section 3.4.2). If the simulation kernel is in CSS mode, the rollback is accomplished by copying the saved state vector back in one single operation. If the simulation kernel is in ISS mode, the rollback of the state queue consists of running down the state queue and copying back each saved variable in reverse order.

3.5.3 Fossil Collection and Irrevocable Events

With every GVT update, the simulation kernel reclaims memory resources for events with timestamps less than GVT and commits irrevocable events for execution. With fossil collection, the memory resources for the events in the queue data structures are freed up to the last event with a timestamp *smaller* than the current GVT. The irrevocable operations are buffered during simulation

and are committed and executed if the GVT sweeps past their simulation time. In Fig. 3.5 the circular I/O buffer is shown, with the two pointers GVT and LVT indicating the start and the end of the buffered data. During simulation all I/O is buffered into this circular buffer. If the simulation rolls back, the LVT pointer is moved back up to the buffered operation with timestamp smaller than the straggler timestamp, and hence the premature buffered operations are rolled back. If the GVT value is updated, the buffered events with timestamp smaller than the GVT are committed and executed, and the GVT pointer of the circular buffered is moved forward.

Figure 3.5: The circular I/O buffer for irrevocable operations. The two pointers GVT and LVT indicate the start and end of the buffered data. In (b) the LVT pointer swept past the circular buffer size.

3.5.4 The Global Virtual Time Computation

The APSIS global virtual time algorithm is based on the GVT algorithm proposed by Bauer and Sporrer (1992). The GVT algorithm is an asynchronous centralized algorithm where the GVT is computed by a central GVT manager that broadcasts a request to all LPs for the current LVT and while collecting those values perform a *min*-reduction. The GVT algorithm assumes error free communication channels with *first-in*, *first-out* behavior.

Let S be the set of logical processes in the simulation, and $T_{ch}^{b \triangleright a}(t)$ the minimum of timestamps of messages sent by b but not yet received by a (the transient messages over a channel). The GVT at any real time t is defined as

$$\mathrm{GVT}(t) = \min\left(\min_{k \in S} \mathrm{LVT}^k(t),\ \min_{a,b \in S} T_{ch}^{b \triangleright a}(t)\right).$$

In other words, GVT is the minimum of any LVT and any message that has already been sent but was not yet received up till now.

The definition of the extended local virtual time ELVT includes the minimum timestamp of transient messages for the logical process:

$$\mathrm{ELVT}^a(t) = \min\left(\mathrm{LVT}^a(t),\ \min_{b \in S} T_{ch}^{b \triangleright a}(t)\right).$$

Using ELVT, we can now express the GVT as:

$$\mathrm{GVT}(t) = \min_{k \in S} \mathrm{ELVT}^k(t).$$

To keep account of the transient messages, all messages through a certain channel are numbered, so we can use the following notation:

- $n^{b \triangleright a}(t)$: serial number of a message sent from b to a at real time t, increasing with each sent message.
- $r^{a \triangleleft b}(t)$: number of messages received by a from b during time interval $[0, t]$.
- $T_m^{b \triangleright a}(n)$: timestamp of event message m with serial number n.

The minimum timestamp of the messages sent by logical process b to logical process a during some time interval $[t_1, t_2]$ is given by

$$T_{min}^{b \triangleright a}(t_1, t_2) = \min_{n^{b \triangleright a}(t_1) < n \leq n^{b \triangleright a}(t_2)} \left(T_m^{b \triangleright a}(n) \right) .$$

We can now rewrite ELVT for real time t^a for logical process a, as

$$\mathrm{ELVT}^a(t^a) = \min \left(\mathrm{LVT}^a(t^a), \min_{b \in S} T_{min}^{b \triangleright a}(t^b, t^a) \right) . \tag{3.1}$$

Equation 3.1 states that we can determine ELVT^a at real time t^a, if we know $\mathrm{LVT}^a(t^a)$ and the value of $T_{min}^{b \triangleright a}$ during the interval $[t^b, t^a]$ (with $n^{b \triangleright a}(t^b) = r^{a \triangleleft b}(t^a)$) from each logical process $b \in S$ that sends messages to a.

The simultaneous reporting problem deals with information messages from different times t (the time for which to determine GVT). Given Eq. 3.1, we can determine a lower bound of GVT at real time t_2^b despite the fact that we have not received any information from logical process a after t_2^a:

$$\begin{aligned} \mathrm{GVT}(t_2^b) &= \min_{k \in \{a,b\}} \left(\mathrm{ELVT}^k(t_2^b) \right) \\ &\geq \min \left(\mathrm{LVT}^a(t_2^a), T_{min}^{b \triangleright a}(t_1^b, t_2^b), \mathrm{LVT}^b(t_2^b), T_{min}^{a \triangleright b}(t_1^a, t_2^a) \right) . \end{aligned} \tag{3.2}$$

In the APSIS GVT algorithm, each logical process k sends an information message to the GVT manager consisting of its LVT^k, $r^{k \triangleleft a}$ for each incoming channel, and $n^{k \triangleright a}$ and $T_{min}^{k \triangleright a}(t_{i-1}^k, t_i^k)$ for each outgoing channel (t_i is the current time and t_{i-1} the time of the previous information message). The GVT manager then applies Eq. 3.2, determining the t_1 ("message sent") times by comparing $n^{a \triangleright b}$ and $r^{b \triangleleft a}$ for each channel ($n^{a \triangleright b}(t_1^a) \leq r^{b \triangleleft a}(t_2^b)$ will give a valid t_1 time). The new GVT estimation will be distributed to the logical processes, which in turn can start their fossil collection and commit irrevocable operations.

3.6 Summary and Discussion

The APSIS simulation environment is designed and implemented to provide an experimental platform for design and evaluation of the Time Warp simulation protocol. The APSIS environment incorporates a number of extensions to efficiently support data intensive applications stemming from the research

field of dynamic complex systems. For ease in flexibility to introduce new protocol extensions and to realize an efficient interface to the parallel simulator, we have chosen to incorporate the parallel simulation functionality into a library instead of a simulation language. Moreover, interface functions to the library parameters enables knowledgeable users to select the appropriate state saving method, cancellation strategy, or optimism control attain the best performance with respect to the application class. APSIS is by design a portable simulation environment, which runs currently on a number of Unix platforms, such as Solaris, Linux, and BSD/OS, and with various communication libraries, like PVM, MPI, and the lightweight Communication Kernel (Overeinder et al. 1995). The APSIS application programming interface includes an interface to the C and C++ programming languages.

Event retraction and incremental state saving are introduced extensions to the Time Warp simulation protocol to accommodate the APSIS library for dynamic complex system applications. Event retraction enables the dynamic creation and deletion of simulation entities. A complication with dynamic deletion of simulation entities is the removal of scheduled events for those entities in the simulated system. The event retraction function is an API function that retracts, or cancels, a scheduled event similar to a rollback, but with the difference that the retract can also be rolled back, resulting in the reschedule of the original event. The incremental state saving facility provides a memory and time efficient state saving mechanism that is suited to data intensive applications which are spatially decomposed over the logical processors. The incremental state saving mechanism only saves the state changes due to the execution of an event, and upon rollback the saved variables are copied back in reverse order to reconstruct the original state vector.

Similar to other PDES libraries mentioned in Section 3.2.2, APSIS is research-oriented and based on an optimistic protocol. The event-oriented world view can be efficiently implemented such that minimal overhead is introduced. The APSIS library has efficient data structures to allow for fast and associative access to the input, output, state, and cancel queues. Together with other optimizations, such as local message delivery and efficient support to underlying message passing layers, an effective simulation environment for solving large scale problems is realized. Future enhancements can include adaptive optimism control, hybrid state saving, and simulation language support, for example by Fornax (van Halderen and Overeinder 1998).

Chapter 4

APSE: Average Parallelism, Profile, and Shape Evaluation

Main Entry: **apse**
1 : APSIS 1
2 : a projecting part of a building (as a church) that is usually semicircular in plan and vaulted

—Merriam-Webster Dictionary

4.1 Introduction

Two measures of particular interest in parallel performance evaluation are *speedup* and *efficiency*. Notwithstanding the importance of these two measures, they do not reveal how well the available potential parallelism of the application is exploited. Nor do they help to understand why the performance may not be as good as expected. In particular, with the performance analysis and evaluation of PDES protocols we need a measure to compare the effectiveness of the different protocols. This can be done by a *relative* criterion that qualitatively compares the effectiveness of the protocols by measuring the speedup or the turn-around time. However, apart from a ranking of the different protocols, this does not answer the question how well we perform our task, that is, to comprehend how much of the potential parallelism is actually realized and what is the lower bound on the execution time. This lower bound on the execution time is the ultimate goal that the PDES protocols strive to achieve or at least to approach.

The potential or inherent parallelism of an application can be quantified as the *average parallelism* metric (Eager et al. 1989), which is a non-trivial upper bound to the asymptotic speedup of the software system (i.e., the speedup of the application with infinite resources and no synchronization costs). Thus, the average parallelism allows us to express the ability of a PDES protocol to exploit the potential parallelism in quantitative terms. For example, a statement that 80% of the available parallelism has been realized, indicates that 20% of the available parallelism has been wasted due to synchronization overhead. Note

that synchronization overhead has two components: communication overhead and PDES protocol overhead. The communication overhead is given by the communication substrate (hardware and software), while the PDES protocol overhead is determined by its efficacy to schedule simulation events in causal order.

In this chapter we propose to obtain the average parallelism of a simulation application by *critical path* analysis techniques. In this analysis, the discrete event simulation execution run is described by a task precedence graph, and well known techniques from graph theory and timing analysis are applied to determine a critical path in this graph. Besides the evaluation of the PDES protocol, the critical path analysis gives insight into the amount of available parallelism, and allows one to use the results of bottleneck analysis to improve the potential performance of the simulation application in a parallel environment.

The critical path analysis methodology is integrated within the APSIS simulation environment. A new and noteworthy feature is that the critical path analysis is performed on the task precedence graph that is obtained from a parallel execution. This is an important feature for spatially decomposed parallel simulations, where the decomposition strategy and the mapping of the application to the parallel architecture determine the available potential parallelism.

4.2 Characterization of Parallelism in Applications

The characterization of parallelism in applications can be described on different levels. At one extreme, the complete characterization of the parallelism in an application can be expressed in a *data dependency graph* (Veen 1986). In a data dependency graph, the concurrency is described on the level of arithmetic operations and assignments. Unfortunately, due to the level of detail, it is not practical to specify or analyze a full data dependency graph for programs of interest. At a less detailed level, portions of an application that are sequential (no internal parallelism) can be treated as tasks in a *task precedence graph* (Coffman 1976). Task precedence graphs are higher level than data dependency graphs, and are therefore more manageable.

At the opposite extreme, single parameter characterizations are very high level descriptions of the parallelism in an application. The *sequential fraction*, which is the fraction of the overall execution time that cannot be executed in parallel, was proposed by Amdahl (1967). Gustafson has argued that the limitation on the parallelism by the sequential fraction can be misleading, since in many applications, the parallel part of the computation grows with the number of processors, while the sequential portion does not grow, and hence the fraction typically decreases (Gustafson 1988).

Another single parameter characterization, the *average parallelism*, has been investigated by Eager, Zahorjan, and Lazowska (1989). The average par-

allelism gives a realistic upper bound to the asymptotic speedup of the parallel application when an unlimited number of resources are available. In the next sections, the average parallelism metric is defined formally. From this formal definition, we describe a method to obtain the average parallelism from the task precedence graph (also known as an event precedence graph in space-time diagrams) by critical path analysis.

4.2.1 The Average Parallelism Metric

The average parallelism metric can be defined in different equivalent ways. The specific definition of the average parallelism depends on the usage of the metric, that is, for the evaluation of the turnaround time, the speedup, or the efficiency. The common denominator in the equivalent definitions is the abstraction from the parallel hardware and the omission of all influences of system and communication overhead.

Definition 4.1 *The average parallelism, A, is defined as the average number of busy processors during the execution of the application, given an unbounded number of processors and no communication latency and other system overhead.*

Other equivalent definitions of the average parallelism measure are:

- The ratio of the total amount of work (expressed in time units as the total service time) and the theoretical turnaround time (without overhead and with ample processors allocated).
- The asymptotic speedup figures, if a hypothetical machine contains an unbounded number of available processors and there is no communication latency and other system overhead.

By deliberately keeping both the processor availability and the communication overhead out of the definition, the average parallelism reflects only the software characterization of parallelism. More precisely, we neglect the performance degradations caused by machine issues (lack of processors, communication delay and contention) to focus on the software factors of performance (non-optimality of the algorithm or program and software overhead). Just as the parallel machine size can be used as the hardware bound on parallelism, the average parallelism metric can be used as the software bound.

In addition to the average parallelism, there are several other parameters that provide some information about the available parallelism in the application. For example in Fig. 4.1, a graph is shown of the number of busy processors over the execution time of the application. We will refer to this as the *parallelism profile* of the application (Sevcik 1989). From the profile, the *shape vector* of the application is defined as the vector $p = (p_1, p_2, p_3, \ldots)$, where each p_i denotes the normalized fraction of execution time spent with degree of parallelism of i. Following the first alternative definition of Def. 4.1, the average parallelism can now also be determined by $A = \sum_i i \cdot p_i$.

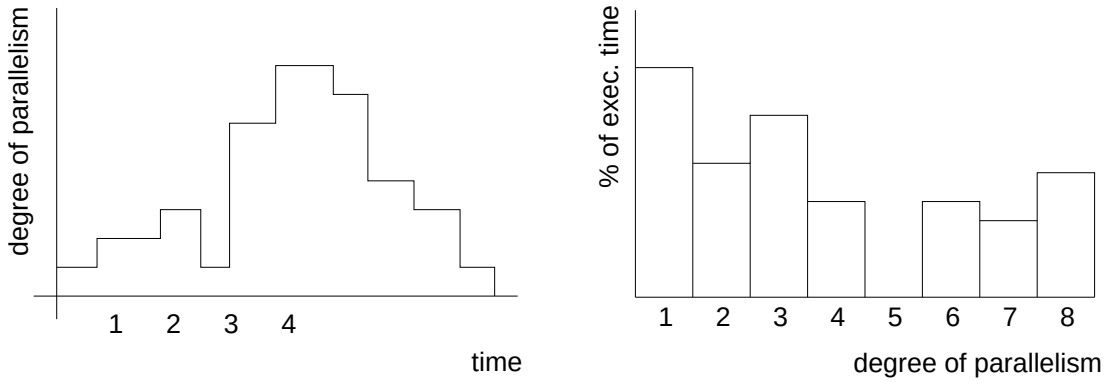


Figure 4.1: The parallelism profile of an application and its corresponding shape.

The simplicity of a single parameter characterization like the average parallelism is intuitively appealing. However, the parallelism profile and shape vector provide more information about how the parallelism is available in the application. From the parallelism profile and shape vector, additional parameters such as the minimum and maximum parallelism, and the variance of parallelism can be determined. These additional parameters are very valuable for scheduling decisions of how many processors to allocate to an application. Depending on the load of the system, the policy can determine the optimal number of processors allocated to the application while maximizing the speedup or the efficiency, or makes a trade-off between speedup and efficiency (Sevcik 1989).

By the definition of the average parallelism metric, and the other characterizations like parallelism profile and shape vector, the hardware component is totally absent. As a consequence, these parallelism characterizations cannot *directly* be obtained from the execution of the parallel application under discussion. As all influences of hardware components are ruled out from the characterizations, we need to ensure that during the analysis of the parallelism only the software components of the execution run are included. The first alternative equivalent definition of Def. 4.1 indicates that the average parallelism is ratio of the total service time to the theoretical turnaround time. The theoretical turnaround time is typically the longest weighted path in the task dependency graph. Thus, given a graph representation of the software component of the parallel execution, we can apply well-known critical path analysis methods to obtain the average parallelism.

4.2.2 The Space-Time Model

For the analysis of the parallelism in the simulation application, we need a sufficient detailed representation of the execution of the simulation run. From the previous discussion, it appears that a task precedence graph, or more appropriate for discrete event simulation, an *event precedence graph* or *space-time*

diagram, suffices for our analysis. The space-time diagram representation of the execution of the simulation run is intrinsic to the software component of the simulation application, that abstracts from the execution mechanism that drives the discrete event simulation. In this respect, the space-time diagram of a sequential execution run is identical to the space-time diagram of a parallel execution run, conservative or optimistic.

The space-time diagram describes the execution run of both sequential and parallel discrete event simulations that adhere to the process-oriented world view, such as described in Section 1.2.7 and Section 2.2. For convenience, we recapitulate the essential characteristics. In process-oriented discrete event simulation, the system being modeled is viewed as being composed of a set of physical processes that interact at various points in real time. The simulation is constructed as a set of logical processes, where each specifies the behavior of some physical process in the system. All interactions between the physical processes are modeled by time stamped event messages sent between the corresponding logical processes.

Given a particular decomposition of the simulation into logical processes, the execution of events must follow two fundamental precedence constraints, also known as causality constraints (which are a further specification of the local causality constraint as defined in Section 2.2).

Definition 4.2 *We define the two precedence constraints in terms of predecessors and antecedents:*

(a) Event e is the (immediate) predecessor *of event e' if (1) they are scheduled for the same logical process, and (2) timestamp $V(e) <$ timestamp $V(e')$, and (3) there is no other event e'' for the same logical process such that $V(e) < V(e'') < V(e')$.*

(b) Event e is the antecedent *of event e' if the execution of event e causes the scheduling of event e'. Note that e and e' may be scheduled for the same logical process.*

The space-time diagram describing the event precedences resulting from the process-oriented simulation is similar for sequential and parallel executions. As any discrete event simulation execution mechanism, sequential or parallel, must obey the causality constraint, the execution of the simulation application results in one unique and correct event order. An alternative view is that different execution mechanisms are different ways of "filling-in" the space-time diagram (Chandy and Sherman 1989). The various execution mechanisms differ in their strategy to accomplish the partial ordering of the event execution as defined by the precedence constraints. In the evaluation of the various parallel discrete event simulation protocols, we effectively determine the ability of the protocols to exploit the available parallelism in completing the space-time diagram as given for a particular simulation run. We can now use this space-time diagram to derive a protocol independent measure to quantify the effectiveness of the different protocols.

In the two dimensional space-time diagram, each event in the simulation, an independent sequential amount of work, is represented as a vertex (see Fig. 4.2). The two coordinates of each event consist of a spatial coordinate and a temporal coordinate. The spatial coordinate is the logical process LP_i where it is executed. Thus events placed on the same spatial position (vertically aligned in Fig. 4.2) occur in one logical process. The temporal coordinate is the simulation time at which the event occurs, the timestamp $V(e)$.

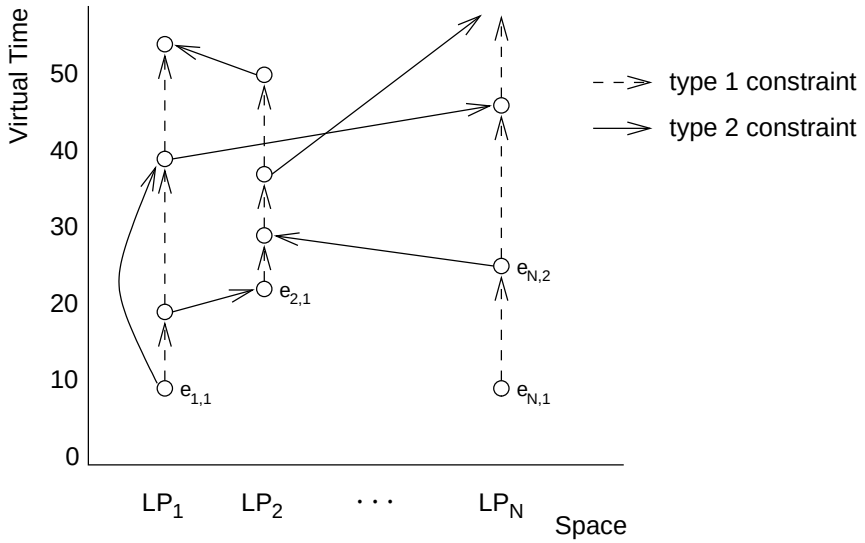


Figure 4.2: Space-time diagram depicting the events (vertices) with their dependency constraints (edges).

The precedence constraints formulated above, are represented by directed arcs. The dashed arcs in Fig. 4.2 represent the predecessor precedence constraint as defined in Def. 4.2(a): if two events e and e' are scheduled for the same logical process with timestamps $V(e)$ and $V(e')$ respectively, and $V(e) < V(e')$, then event e must be executed before e'. A continuous arc represents the exchange of an event message that must obey antecedent precedence constraint as defined in Def. 4.2(b): if event e causes the scheduling of event e', and consequently $V(e) < V(e')$, then event e must be executed before event e'. Notice that a process is allowed to send an event messages to itself.

There are some events in the simulation that have no predecessors; these events are called *initial* events (labeled $e_{1,1}$, $e_{2,1}$, and $e_{N,1}$ in Fig. 4.2). The events without antecedents are called *starting* events and are prescheduled before the execution of the simulation starts. The *terminal* events are the last scheduled events by their respective *LP*.

The resulting space-time diagram is a full description of the available parallelism in the discrete event simulation. Different analyses are possible, such as critical path computation, bottleneck analysis, etc. From these analyses, more high level parameters to characterize the parallelism can be obtained.

4.2.3 Critical Path Analysis

The first step in the determination of the average parallelism, parallelism profile, and shape vector, is the critical path analysis to determine the longest path in the space-time diagram. From the precedence constraints we construct a space-time diagram of the simulation run. The activities between the events are represented by the directed edges. We can label all activities in the space-time diagram with a weight $T(e_1, e_2)$, where e_1 is the event that preceded the activity, and e_2 is the event resulting from the activity. If e_1 is the immediate predecessor of e_2, then $T(e_1, e_2)$ expresses the service time for calculation; if e_1 is the antecedent of e_2, then it expresses the service time for communication. Note that in this scheme, the event vertices are *not* labeled with weights, but are rather instantaneous synchronization points in space-time.

By the construction of the space-time diagram with the weights along the edges, the hardware component of the system is implicitly modeled as an infinite number of identical processors, each of unit speed. The synchronization between processors has zero overhead and the entire parallel computer is devoted to one single task. With these assumptions, the hardware component is neutral to the critical path analysis such that the resulting metric is a characteristic of the software component. By mapping cost functions to the weighted graph, the available parallelism on specific hardware platforms can be obtained for a specific process to processor allocation. For example, the influence of communication costs can be studied, maybe depending on processor distance, or the consequences of oversubscribing of processes to processors, and hence load balance or imbalance.

The extended space-time diagram with associated weights to the edges, is an acyclic directed graph, or *program activity graph* (PAG), in which a longest weighted path can be found. This path is called the critical path, and its length is the minimal time required to complete the execution of the parallel simulation.

With all the edges of the space-time diagram labeled with a weight, we can associate a *critical time* with each event.

Definition 4.3 *The critical time of an event e,* $\mathrm{crit}(e)$*, is defined by:*

$$\mathrm{crit}(e) = \begin{cases} 0 & \textit{iff } ANCE(e) = \emptyset \\ \max_{e' \in ANCE(e)} \{\mathrm{crit}(e') + T(e', e)\} & \textit{otherwise} \end{cases}$$

where the ancestor set is:

$$ANCE(e) = \{e' \in E \mid e' \text{ is predecessor or antecedent of } e\}$$

It is clear that $\mathrm{crit}(e)$ is the earliest time the event e can complete execution under the assumption that no dependencies are violated. Consequently, the largest value of $\mathrm{crit}(e)$ among all the events in the simulation run will give us the lower bound on the completion time of the simulation run. The method for finding the $\mathrm{crit}(e)$ is essentially a topological sort. That is, each $\mathrm{crit}(e)$ can be calculated after all the $\mathrm{crit}(e')$ of its ancestors have been computed. An algorithm to calculate the critical times in a PAG is presented in Alg. 4.1 in Section 4.3.2.

Having determined the critical time for each event in a start-finish sequence, the critical path can be defined in reverse order:

- e with $max_{e \in E}\{\text{crit}(e)\}$ is on the critical path;
- if e is on the critical path, then $e' \in ANCE(e)$, resulting in the maximal critical time according to Def. 4.3, is on the critical path.

This method underlies both the computation of the average parallelism metric and the path enumeration for bottleneck debugging, that is an optional analysis part of the APSE tool.

4.3 Design and Implementation of APSE

The Average Parallelism, Profile, and Shape Evaluation (APSE) methodology is designed and implemented as a stand-alone tool. This allows for larger flexibility, and makes the APSE tool generally applicable for the analysis of any parallel program (see the discussion in Section 4.6).

4.3.1 Conceptual Tool Structure

We designed the APSE tool following the conceptual framework outlined by McKerrow (1988). In this conceptual framework, the performance measurement tool can be comprised of four sections:

Sensor Section This is the interface between the target process(es) and the measurement tool. The function of a *sensor* or *probe* is to detect events of interest and/or measure the magnitude of the quantities to be monitored, and store this data in some internal buffer. When these buffers fill up, it may become necessary to write their contents to secondary storage.

A software probe is typically a subprogram or a procedure call inserted in the target process. In the case of a software tool, the sensor section can be seen as the tools front-end. Sensors can be *internally driven*, meaning that the target process triggers the sensor to some accounting action, or *externally driven*, meaning that the sensor is triggered by the tool rather than the target process. An important advantage of internally driven sensors is that the data they collect is synchronized to the internal operations of the target process.

Transformer Section The transformer section performs essentially two functions. First, the (typically huge amount of) data coming from the probes is reduced to a subset of relevant data. What data is and is not relevant depends on the scope and goal of the experiment at hand. The word *reduce* in the preceding sentence may be misleading, for the amount of data reduction is strongly dependent on the requirements of the subsequent sections of the tool. The transformer may just store the data without change.

The second function of the transformer is to rearrange the data coming from the probes. The reduced set of data is structured to fit the requirements of successive phases of the tool.

Analyzer Section The data stored by the transformer is processed to produce the final output of the experiment, e.g., tables, graphs, etc. The analysis to be performed on the condensed and structured sensor data is determined by the experimental framework. For simple experiments the analysis may be not needed and can than be skipped (see Fig. 4.3), for very complex experiments the analysis can be done only after the data of several experiments is available.

With respect to the analyzer section, we can further classify tools either as *rigid* or *flexible*. A rigid tool has limited and fixed analysis capabilities, which cannot easily be changed or extended. The methods of Lin (1992) for computing the most critical path in a program activity graph is an example of a rigid analysis. On the other hand, the performance measurement environments described by Mink et al. (1990), Mohr (1990), or Reed et al. (1991, 1994, 1998) provide a generic toolbox of analysis methods, from which a specific analyzer section can be assembled.

Indicator Section The function of the indicator is to show the results of the experiment in a convenient way. Depending on the amount of sensor data, data reduction and the character of the analysis, the indicator may be less or more complex. In general, the more information must be presented by the indicator, the more visualization is needed to produce comprehensible results.

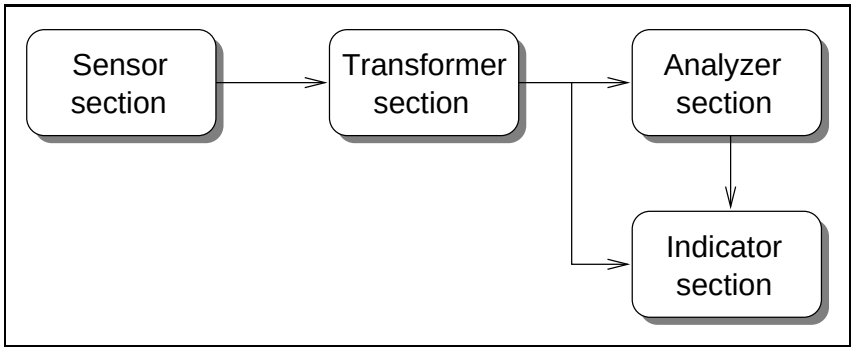


Figure 4.3: Conceptual structure of a measurement tool.

4.3.2 Overview of APSE

The overall design of the Average Parallelism, Profile and Shape Evaluation (APSE) tool is organized in according with the conceptual sections as described in the previous section. The functional structure is shown in Fig. 4.4.

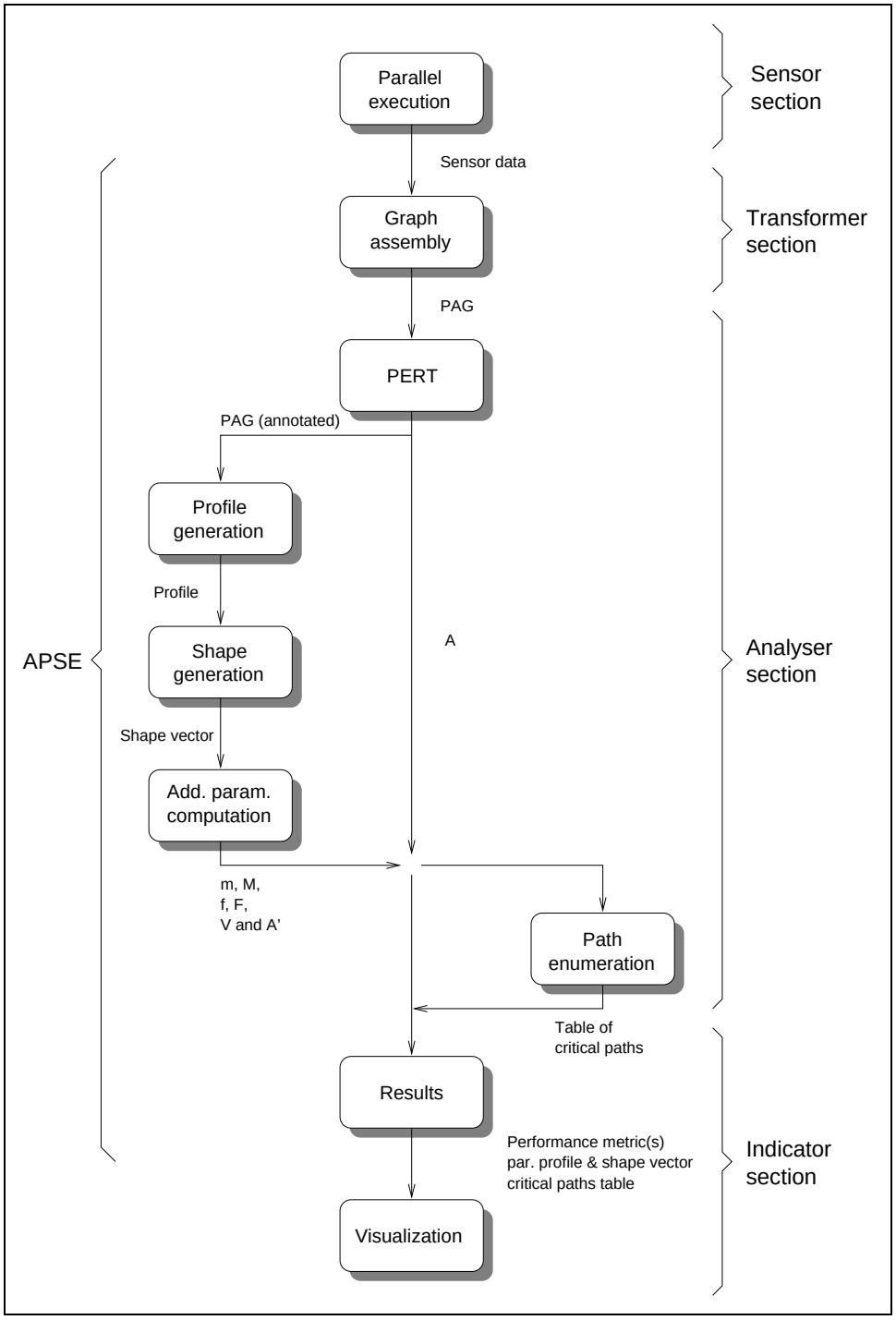


Figure 4.4: The functional structure of APSE (see for details, the text in Section 4.3.2).

The sensor section of the tool is located in a separate module and integrated with the parallel program. To keep interference (that is, program behavior perturbation due to monitoring code) as low as possible, we have isolated the recording of events from the process of analyzing them (see further on this

section). Thus, the average parallelism analysis is done postmortem, after the experimental run has finished.

The transformation section comprises the graph assembly module that constructs the program activity graph (PAG) from the sensor data. The analyzer section applies the Program Evaluation and Review Technique (PERT) algorithm to the PAG. Depending on user requests, the average parallelism metric is computed and optionally the profile and shape of the parallelism is generated. For performance bottleneck debugging, it is also possible to include the generation of a table of the most critical paths in the analysis. The indicator section includes the modules that present the results and provides an interface to visualization tools. The visualization of complex data sets such as the parallelism profile or the critical path in the PAG is not considered to be a part of APSE. However, an interface to Gnuplot is provided and more elaborate visualization with for example Tk/Tcl can be easily incorporated.

Sensor Section and Software Probes

The APSE interface with the experimental environment, in this discussion the APSIS simulation environment, is implemented by the software probes. The software probes are inserted to the Time Warp kernel (see Fig. 4.5); it is up to the experimentalist to identify the relevant trace events within the Time Warp kernel. In our study, the relevant trace events are: start and finish of a simulation event execution and the scheduling of a new simulation event. Although the number of software probes is very limited (currently six), it allows us to construct the PAG for further analysis.

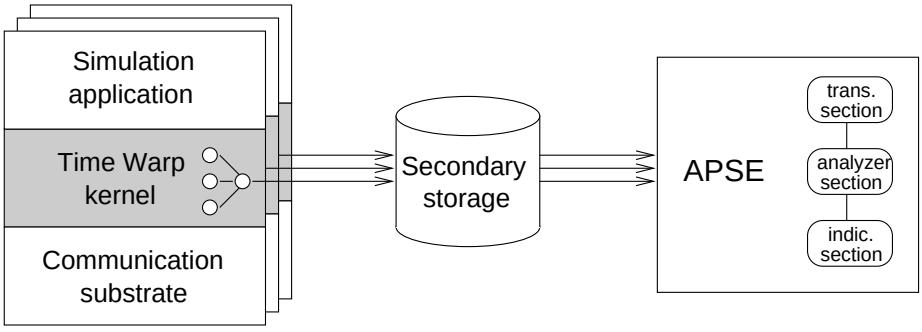


Figure 4.5: The sensor section of the APSE tool is integrated with the APSIS Time Warp simulation kernel. The trace data is written to the file system, and is analyzed off-line by the APSE tool.

The trace data gathered by the software probes is stored in a circular buffer data structure in main memory. If the circular buffer has reached its capacity, the trace data is written in a binary form to the file system. The interference of the periodic file system I/O to the program behavior and consequently the trace data is limited, as all the Time Warp protocol overhead including APSE buffer management is outside the traced program area. That is, only the committed

events in the simulation application are traced and recorded, and all erroneous events that are rolled back, and other overheads induced by the Time Warp protocol, are excluded from the trace data. This also implies that the trace data has to be committed in a similar way as events have to be committed. Only if the GVT swept past the events, the associated trace data is allowed to be written to the file system.

PAG Generation

The graph assembly module constructs the PAG from the trace event records in the input. This graph (see Fig. 4.2) contains two types of dependences among the events: intra-process and inter-process dependences, as described in Section 4.2.2. Every event record from the execution trace is converted into a PAG vertex, and the causality constraints are the edges in the graph. The graph is completed by adding two special vertices: the *source* (denoted by s) and the *sink* (denoted by t). The source is connected to all initial events and the sink is connected to all terminal events.

PERT Algorithm

The program activity graph is annotated by the Program Evaluation and Review Technique (PERT) algorithm. This algorithm annotates each vertex in the graph with its *maximal delay to sink* value. The original PERT algorithm computes for all vertices the *maximal delay from source* value rather than the maximal delay to sink, similar to the definition of critical time of an event (Def. 4.3). However, the path enumeration algorithm, as discussed in the next section, assumes the graph to be annotated with maximal delay to sink values. Since the objective is not to obtain the critical time for a specific event, but to look for the critical path, the choice is arbitrary.

Let $\mathrm{crit}(e)$ denote the "max-delay-to-sink" label of event e, and let $SUCC(e)$ be the set of successor events of e,

$$SUCC(e) = \{e' \in E \mid e' \text{ has } e \text{ as predecessor or antecedent}\}.$$

Furthermore, assume that each edge between e and e' in the PAG has a weight (or length) $T(e, e')$.

The PERT algorithm in Alg. 4.1 reflects a topological sort on the graph, since the sink is the start vertex, whose maximal delay to sink value is zero, and for each vertex in the graph that is not yet annotated, the maximal delay to sink value depends both on this value for all its successors, and on the weights of the connecting edges. Upon completion of the PERT algorithm, the maximal delay to sink value of the source vertex represents the weight of the most critical path in the graph.

By the construction of the PAG, the PERT algorithm computes the "max-delay-to-sink" time of the events while it adheres to the precedence constraints as defined in Section 4.2.2. If the user requested that a path enumeration be part of the analysis, the successors of each vertex are sorted in non-decreasing

maximal delay to sink value. This sorting is necessary in order to apply the path enumeration algorithm later on.

Algorithm 4.1 The PERT algorithm: critical time computation.

```
{s = source; t = sink;}
∀e ∈ E − {t} : crit(e) := undefined; {initialization}
crit(t) := 0;
repeat {topological sort}
  for all e ∈ E do
    if crit(e) = undefined ∧ ∀e′ ∈ SUCC(e) : crit(e′) = defined then
      crit(e) := max_{e′∈SUCC(e)}{crit(e′) + T(e, e′)};
    end if
  end for
until e = s;
{crit(s) represents the length of the most critical path}
```

Path Enumeration Algorithm

The critical path enumeration algorithm used in APSE finds the K most critical paths in the annotated PERT PAG (Yen et al. 1989). Path extracting algorithms are a very important part of parallel performance bottleneck debugging. The path enumeration algorithm is conceptually a K iterative process. In the ith iteration, the ith most critical path is expanded in the graph. The algorithm as presented in Alg. 4.2 uses a table (*PATHS*) to store the most critical paths, a dynamic threshold (T) to be able to prune paths that will not reside in the paths table, and an overloaded function nextnode:

1. nextnode(e): returns the first vertex in the sorted successor list of e;
2. nextnode(e, e'): returns nil, when e' is the last vertex in the successor list of e, or returns the vertex next to e' in the sorted successor list of e otherwise.

Let crit′(e) denote the "delay-from-source" label of vertex e. This value is easily computed for each element of any (partial) path, since it is just the sum of all the weights associated with the edges leading to the element. Therefore, we have left out the computations of the crit′(e) in the pseudo-code.

Each vertex has its successors sorted in non-increasing maximal delay to sink value, hence the most critical path can be found by starting at the source, and always taking the first successor in each vertex successor list, until the sink is reached.

Path enumeration is realized by creating variants of the most critical path. Such a variant is realized by making a copy of the current path, and then applying a process of tracing backward and forward on the copy. A new critical path is completed when the forward trace arrives at the sink.

The backward trace starts at the sink and traces the path in the direction of the source, until it arrives at a vertex where an alternative route can be

taken. Comparing the sum of the two delays $\text{crit}'(e)$ and $\text{crit}(e')$ and the weight $T(e, e')$ with the threshold provides the answer whether the alternative path is suitable. The backward trace has two termination conditions:

1. a vertex has been found in the graph from which a variant path can be followed;

2. no vertex has been found, and the algorithm terminates—this happens when the backward trace has proceeded all the way to the source.

The forward trace consists of following the first successor of each vertex every time, until the path is complete again (i.e., the last vertex of the path equals the sink vertex). The new path that is created this way is inserted in the paths table.

Algorithm 4.2 The path enumeration algorithm.

$T := 0$; {initialize threshold}
$P := (s = e_0, e_1, e_2, \ldots, e_q = t)$, where $e_{i+1} = \text{nextnode}(e_i)$;
$j := q - 1$;
while $j > 0$ **do**
  insert P to *PATHS*;
  $T = \min_{P \in PATHS}\{\text{crit}(P)\}$;

  {backward trace}
  find largest $j, 0 \leq j < q$ such that:
    $\text{nextnode}(e_j, e_{j+1}) \neq \text{nil}$ and
    $\text{crit}'(e_j) + T(e_j, e_k) + \text{crit}(e_k) > T$, where $e_k = \text{nextnode}(e_j, e_{j+1})$;

  {forward trace}
  $P := (s = e_0, e_1, e_2, \ldots, e_q = t)$, where e_i is:
    $0 \leq i \leq j : e_i$ as in the existing P,
    $i = j + 1 : \text{nextnode}(e_j, e_{j+1})$,
    $i > j + 1 : \text{nextnode}(e_{i-1})$;
end while
{*PATHS* contains the K most critical paths}

Profile and Shape Generation

The parallelism profile is a plot of the degree of parallelism versus time. The profile can be generated by rearranging and reducing the information from the PAG. For each activity in the PAG, i.e., an intra-process dependency edge, two entries are inserted in the profile list: one for the activation of the activity and one for its termination. When the list is sorted on times of activation and termination, the parallelism profile can be obtained by counting activation and termination events: parallelism increases with an activation event and

decreases with a termination event. From this profile the shape vector is generated by associating the normalized fraction of execution time spent with a certain degree of parallelism in a histogram. Given the notation in Def. 4.2.1, we can compute the following metrics from the shape vector:

Minimum parallelism $m = \min\{i \mid p_i \neq 0\}$

Maximum parallelism $M = \max\{i \mid p_i \neq 0\}$

Fraction sequential $f = p_1$

Average parallelism $A = \sum_{j=m}^{M} j \cdot p_j$

Variance in parallelism $V = \sum_{j=m}^{M} j^2 \cdot p_j - \left(\sum_{j=m}^{M} j \cdot p_j\right)^2$

The advantage of the presentation of the execution trace as a parallelism profile over a PAG comes from the compactness of the representation and its resulting in more comprehensible results. For huge execution traces even visualization will not suffice to provide insight into the trace structure if the PAG is considered. This is due to both the amount of information contained in the PAG, and the complexity of the structure of the PAG. Actually, the parallelism profile can be used complementary to the PAG in order to identify performance bottlenecks. Periods with a relative small degree of parallelism are readily recognized and direct the detailed study of the complex PAG to areas where performance bottlenecks appear.

4.4 Experiments, Validation, and Assessment

In this section we present some experiments to validate the correctness of the APSE critical path analysis, and to assess the use of critical path analysis in average parallelism evaluation. The validation and assessment is shown by two simple simulations: unidirectional and bidirectional message routing over a ring embedded in a two-dimensional torus topology. In Section 5.5 the APSE analysis will be applied to a complex problem, namely the Ising spin simulation.

4.4.1 Unidirectional Ring

The unidirectional ring simulation is realized using the APSIS environment. To generate an event trace of the parallel simulation, an instrumented version of the Time Warp simulation library is made available (see also Fig. 4.5). The event trace of a logical process is buffered in memory, and during the fossil collection phase of committed events, the buffers are flushed to the file system. In this respect, the event trace represents the sequence of correct events that are executed in causal order, thus the event in the event trace *must* be committed before they are written to file.

The embedding of the ring into the two-dimensional torus is shown in Fig. 4.6. From the starting LP 0, the event messages are routed to their east neighbor for odd rows in the torus, and to their west neighbor for even rows in the torus. If the message hits one of the east or west boundary LPs, it is routed to the south neighbor. Note that if the message arrives at the south-west corner LP, the message is routed south to LP 0, which completes one single round through the ring. The unidirectional ring simulation routes at every instance of time *one single message* through the embedded ring. A parameter determines the number of times the message is routed through the ring. As at each instance only one single message is present in the system, and hence only one LP can be active, we expect an average parallelism of one.

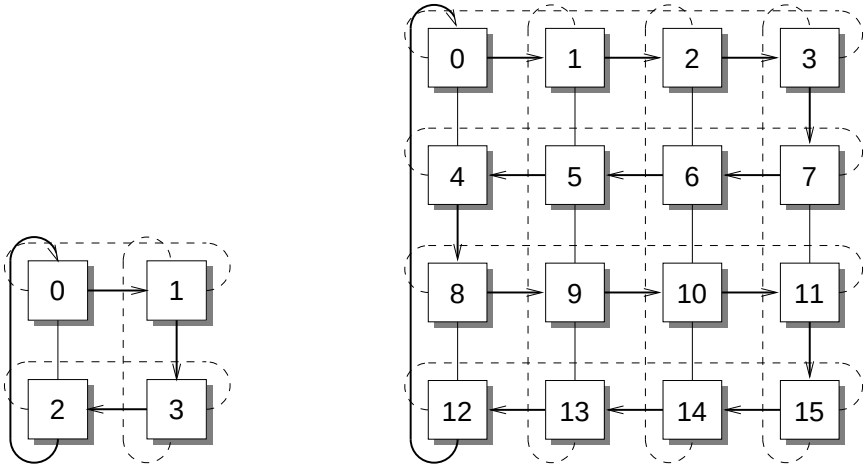


Figure 4.6: Ring mapped on two-dimensional torus topology.

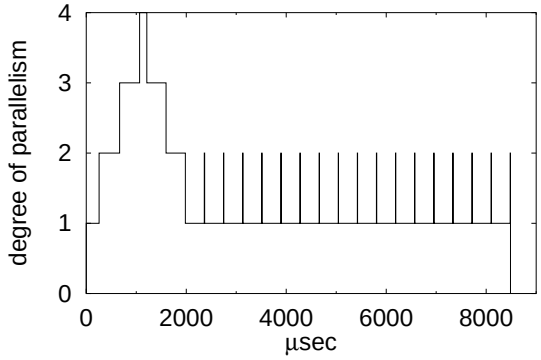
The experiments are executed on the Distributed ASCI Supercomputer (DAS)*, a 200-node parallel platform composed of four distributed clusters connected by ATM (Vetter 1995), and where the nodes within each cluster are connected by a Myrinet System Area Network (SAN) (Boden et al. 1995). All experiments are performed on one single cluster.

The results of the parallel simulation on four processors of the unidirectional ring routing of five messages are shown in Table 4.1 and Fig. 4.7. From Table 4.1 we can observe that although we expect an average parallelism $A = 1$, we have measured an average parallelism $A = 1.34$. Similarly remarkable is the fraction sequential $f = 0.79$ and fraction maximum parallelism $F = 0.02$. This can be explained by Fig. 4.7(a), the parallelism profile, that shows a transient behavior in the degree of parallelism during the initialization phase of the simulation. The initialization of the LPs, not the initialization of the simulation library, is also accounted for in the event trace. The initialization is not dependent of any event and occurs in parallel. After real time 2000 μsec the parallelism profile in Fig. 4.7(a) shows a degree of parallelism of 1, as expected.

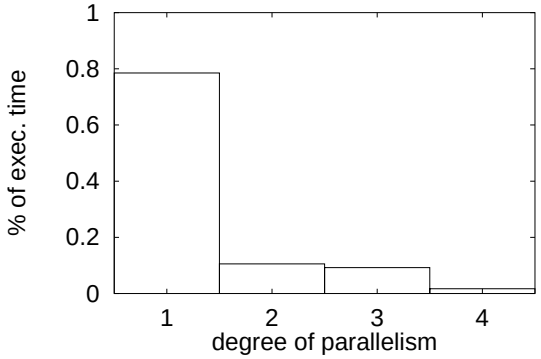
*http://www.asci.tudelft.nl/das/das.shtml or http://www.cs.vu.nl/das/

$A =$	1.34	$f =$	0.79
$m =$	1	$F =$	0.02
$M =$	4	$\sigma^2 =$	0.51

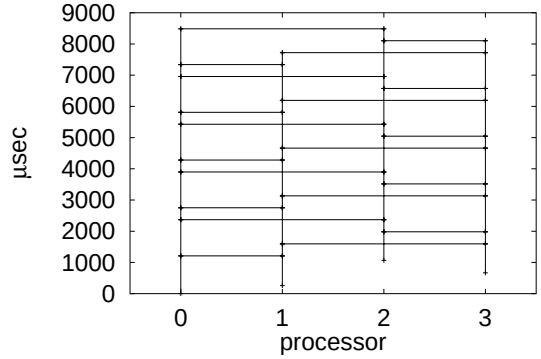
Table 4.1: Average A, minimum m, and maximum parallelism M, fraction sequential f, fraction maximum parallelism F, and variance in average parallelism σ^2 for simulation of five messages over unidirectional ring on four processors.



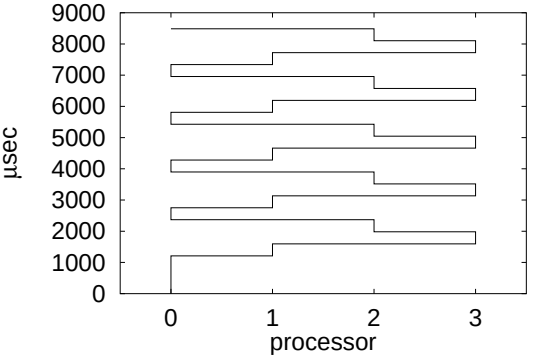
(a) Parallelism profile of unidirectional ring simulation.



(b) Shape of unidirectional ring simulation.



(c) Program activity graph of unidirectional ring simulation.



(d) Critical path of unidirectional ring simulation.

Figure 4.7: The parallelism profile, the shape, the program activity graph, and the critical path of the parallel simulation of the unidirectional ring on four processors.

The program activity graph in Fig. 4.7(c) depicts the activities and intra- and inter-dependencies between the events that trigger the activities. From Fig. 4.7(c) one can see how the event message travels from LP 0 to LP 1, LP 3,

LP 2, and back to LP 0. The y-axis in the figure shows the progress in real time, i.e., the real time accounted for the activity by the LP. The critical path in Fig. 4.7(d) shows the activities and the intra- and inter-dependencies that are along the critical path in the parallel simulation. The figure shows clearly how the critical path follows the message routing through the torus. As there is only one event message in the system, the critical path should follow the event message through the parallel simulation.

In Table 4.2 and Fig. 4.8 results are presented of the parallel simulation on sixteen processors of unidirectional ring routing of 100 messages. The average parallelism and fraction sequential in Table 4.2 are (almost) reflecting the sequential behavior of the simulation. The average parallelism $A = 1.04$ and the fraction sequential $f = 0.98$. The effects of the transient behavior in the degree of parallelism are nearly nullified by the long simulation run, see also the small peak around 0 in Fig. 4.8(a). In Table 4.2 the maximum parallelism, M, of five instead of sixteen can be explained by the gauging of the initialization activities that are relative to the first event message exchange (send or receive). Depending the duration of the initialization activity and the event inter-dependencies, the activities will overlap with each other in real time and hence accounts for the degree of parallelism. This effect can be seen in Fig. 4.7(c).

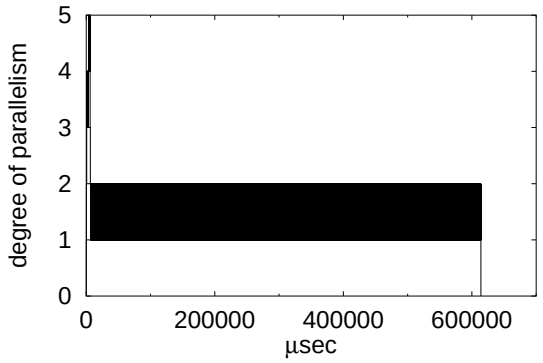
$A =$	1.04	$f =$	0.98
$m =$	1	$F =$	$0.45 \cdot 10^{-3}$
$M =$	5	$\sigma^2 =$	0.9

Table 4.2: Average, minimum, and maximum parallelism, fraction sequential, fraction maximum parallelism, and variance in average parallelism for simulation of 100 messages over unidirectional ring on sixteen processors.

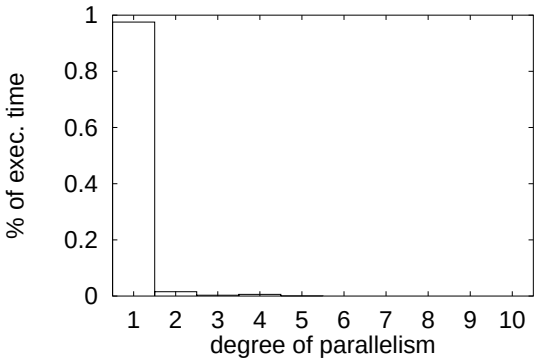
4.4.2 Bidirectional Ring

The framework of the bidirectional ring experiment is similar to the unidirectional ring simulation. The bidirectional ring through which messages are routed is embedded into a two-dimensional torus, as shown in Fig. 4.6. The bidirectional ring differs from the unidirectional ring in that the initiating LP 0 routes two messages to its neighbor: one message to its east neighbor, and one message to its west neighbor. On arrival of a message, it is forwarded along the same direction as it was received from, to the neighboring LP. At every instance the bidirectional ring simulation routes two messages through the embedded ring, and hence we expect an average parallelism of two.

The results of the APSE analysis of the parallel simulation on four processors of the bidirectional routing of five messages in both directions are shown in Table 4.3 and Fig. 4.9. The average parallelism $A = 2.30$ is larger than the expected $A = 2.0$, but again this can be explained by the transient behavior of the degree of parallelism during initialization of the LPs, see also Fig. 4.9(a).



(a) Parallelism profile of unidirectional ring simulation.



(b) Shape of unidirectional ring simulation.

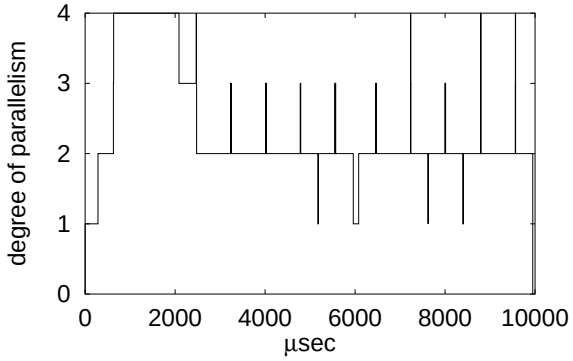
Figure 4.8: The parallelism profile and the shape of the parallel simulation of the unidirectional ring on sixteen processors.

The shape of the bidirectional ring in Fig. 4.9(b) shows a peak at degree of parallelism of two, but shows also a large percentage of execution time a degree of parallelism of four. This is due to the initialization phase.

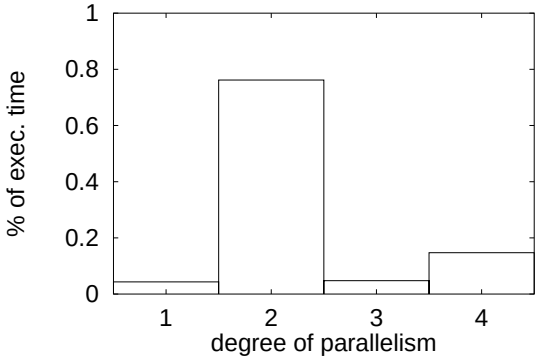
$A =$	2.30	$f =$	0.04
$m =$	1	$F =$	0.15
$M =$	4	$\sigma^2 =$	0.59

Table 4.3: Average, minimum, and maximum parallelism, fraction sequential, fraction maximum parallelism, and variance in average parallelism for simulation of five messages over bidirectional ring on four processors.

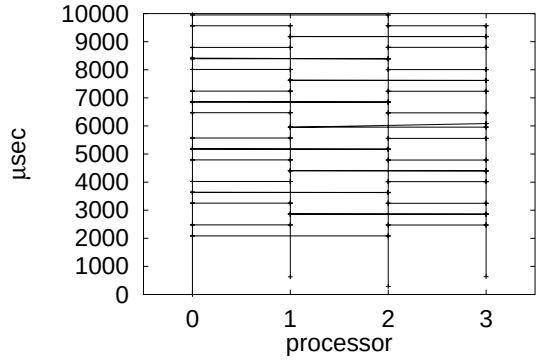
Interesting to note is the relatively large period of degree of parallelism of one at execution time $6000\,\mu$sec. If two messages pass each other halfway on their round through the bidirectional ring, the simulation of the two messages are sequentialized. This is enforced by the timestamps of the event messages. The forwarding activity of the LPs takes approximately the same amount of time, as can be seen in Fig 4.9(c) by the vertical segments between message departure and arrival. However, due to some perturbation in the activity execution time (due to underlying operating system activities, virtual memory management, etc.) one of the two messages lags behind. When the event messages meet each other halfway, the first message must wait for the second to arrive before it can be forwarded. The simulation of the lagging event message accounts for the degree of parallelism of one for a period. Note also the short period of inactivity of LP 3 in Fig 4.9(c) at $6000\,\mu$sec, and how the send and receive times differ in the figure to compensate for the difference in activity execution times. In the ideal case the send and receive times overlap with each



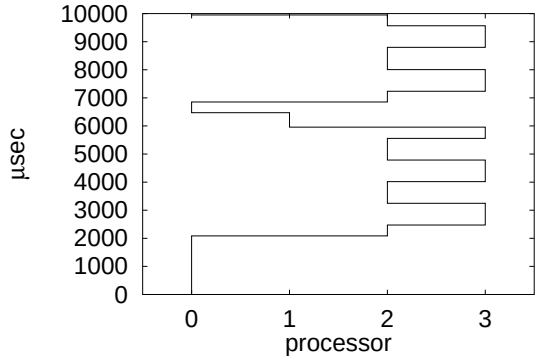
(a) Parallelism profile of bidirectional ring simulation.



(b) Shape of bidirectional ring simulation.



(c) Program activity graph of bidirectional ring simulation.



(d) Critical path of bidirectional ring simulation.

Figure 4.9: The parallelism profile, the shape, the program activity graph, and the critical path of the parallel simulation of the bidirectional ring on four processors.

other as each event activity execution takes the same amount of real time.

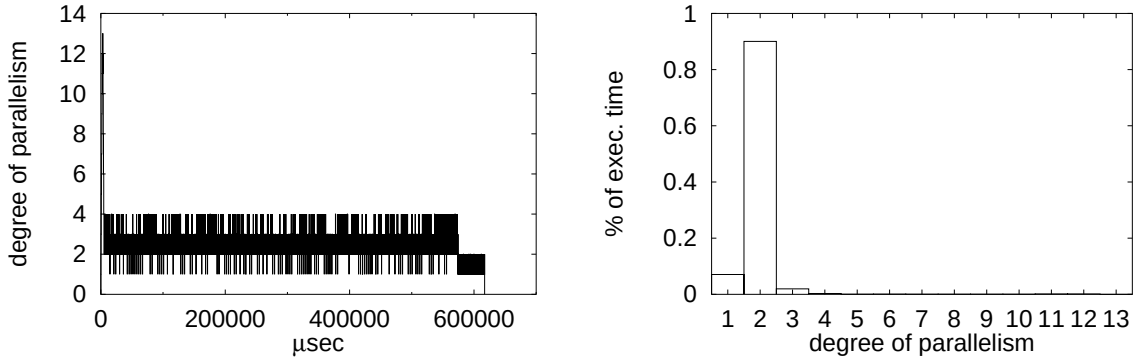
Figure 4.9(d) depicts the critical path through the bidirectional ring simulation. In the bidirectional ring simulation the critical path does not necessarily follow one single event message, but can switch between the event messages. See for example how the critical path switches between LP 2 and LP 3 during execution time interval $[2000, 6000]$ μsec.

The results for the parallel simulation on sixteen processors of the bidirectional ring routing of 100 messages are presented in Table 4.4 and Fig. 4.10. The average parallelism $A = 2.0$ is in correspondence with the theoretical expected value. The maximum degree of parallelism is thirteen, which can be explained in a similar way as for the unidirectional ring study, i.e., gauge effect.

$A =$	2.0	$f =$	0.07
$m =$	1	$F =$	$0.9 \cdot 10^{-4}$
$M =$	13	$\sigma^2 =$	0.22

Table 4.4: Average, minimum, and maximum parallelism, fraction sequential, fraction maximum parallelism, and variance in average parallelism for simulation of 100 messages over bidirectional ring on sixteen processors.

The parallelism profile in Fig 4.10(a) shows how after the short initialization phase, the degree of parallelism is two with many spikes to three and four due to small perturbations in the event activity execution time. Although the parallelism profile shows almost black areas between degree of parallelism of two and four, the contribution to the average parallelism is small as can be seen in Fig. 4.10(b), where the degree of parallelism of three is 2% and degree of parallelism of four is 0.3% of the execution time. The relative large period of degree of parallelism of one at the end of the execution is resulting from the finalization phase.



(a) Parallelism profile of bidirectional ring simulation.

(b) Shape of bidirectional ring simulation.

Figure 4.10: The parallelism profile and the shape of the parallel simulation of the bidirectional ring on sixteen processors.

4.5 Related Work

Critical path analysis has shown to be a comprehensive method to quantify the inherent parallelism or average parallelism of a parallel program. The critical path analysis technique, or more generally event tracing, is incorporated in a wide range of performance evaluation environments such as Pablo (Reed et al. 1993) and Paradyn (Miller et al. 1995). The basic critical path analysis of the

event trace does not differ for the various analysis tools, but the manner to incorporate the analysis with the parallel program execution does. Event tracing is not limited to simulation application. Any program can define "events" such as a function call, interrupt, or trap to a kernel, and record this event with a timestamp (in this case the real time). Although the APSE performance analysis is presented in the context of parallel discrete event simulation, any parallel program can be instrumented for performance analysis with APSE (Overeinder and Sloot 1995).

With respect to the specific parallel simulation instrumentation for event trace generation and critical path analysis, we can distinguish between (i) sequential versus parallel execution event traces, and (ii) on-line versus off-line critical path analysis. Sequential execution event trace generation supports early performance evaluation of the potential parallelism available in the simulation, even before a parallel program is implemented. On the other hand, parallel execution event trace generation allows for the computation of a more realistic bound that takes into account all the overheads associated with the parallel simulation of the model, except those specific to different simulation protocols. On-line versus off-line critical path analysis is a practical trade-off (Hollingsworth 1998). As the computation of the critical path is expensive, off-line (post mortem) approaches minimize the intrusiveness of the analysis onto the execution behavior. However, the off-line approach requires memory and disk space proportional to the number of events and communication operations performed. Therefore, to make critical path analysis practical for long running simulations, on-line analysis is necessary.

Berry and Jefferson (1985), and later Salmi et al. (1994), presented an off-line method for critical path analysis of sequential event traces. The inherent parallelism computation is included in the modeling process of the (sequential) simulation to assess the parallel potential. The APSE critical path analysis is similar, but within the APSIS environment, the event trace is collected from the parallel simulation execution. The analysis approach of Jha and Bagrodia (1996) is orthogonal to the previous ones. They describe an Ideal Simulation Protocol (ISP), based on the concept of critical path, which experimentally computes the best possible execution time for a simulation model on a given parallel architecture. The ISP requires an event trace from a sequential execution to eliminate protocol specific overheads, that is, the ISP knows the identity of the next message it is going to execute, thus no rollback or unnecessary blocking.

On-line critical path analysis has been used in a number of parallel simulators. Livny (1985) incorporated an on-line critical path algorithm in the Distributed System Simulation (DISS)[†] simulator. In the study it is assumed that all events have the same execution time, which is defined to be the unit time. During the distributed simulation, the global optimal execution instance of an event is computed, and the total events that have been executed is counted. The inherent parallelism is than defined as the ratio between the total events executed and the optimal execution instance of the last event in the simula-

[†]DISS is a predecessor of HLA

tion. Lin (1992) proposed a critical path analysis algorithm that is integrated with the sequential simulation. The method described by Lin is similar to the algorithm proposed by Livny, but the algorithm of Lin can be used to study load balancing under different event scheduling policies. These results are extended by Wong et al. (1995) by considering both the inherent parallelism and the parallel simulation protocol overhead. Lim et al. (1999) developed a set of performance prediction tools, including a critical path analyzer. In their approach the sequential simulator informs the analyzer about the execution time, whereupon the analyzer models the progress of the parallel execution.

Jefferson and Reiher (1991) and Srinivasan and Reynolds (1995) reported about super-critical speedup of optimistic simulation protocols. Although critical path analysis establishes a lower bound on the completion times of parallel discrete event simulations, at least one optimistic protocol can complete in less than the critical path time in a nontrivial way. For example Time Warp with lazy cancellation can achieve super-critical speedup, although this is of more theoretical than practical interest. The parallel simulation using Time Warp with lazy rollback might include erroneous causal execution of events which are accepted as the incorrect execution order does not influence the correct result of the simulation. This incorrect execution order can potentially be shorter than the (correct) critical path through the simulation, and hence beating the critical path time.

Other techniques than critical path analysis are also used in performance evaluation of parallel simulations. Ferscha and Johnson (1996) present an incremental code development process that supports early performance of Time Warp protocols and several of its optimizations. The set of tools represent a test bed for a detailed sensitivity analysis of the various Time Warp execution parameters. The performance engineering activities range from performance prediction in the early development stages, to measurements of performance metrics of the preliminary or final program. Liu et al. (1999) propose a Scalable Simulation Framework (SSF) to predict the performance of a given model, using given features of the simulator, without having to run, or even build, the model. Balakrishnan et al. (1997) present a framework for performance analysis of parallel discrete event simulators which is based on a Workload Specification Language (WSL). WSL is a language that allows the characterization of simulation models using a set of fundamental performance-critical parameters. The WSL presentation can be translated (using a simulator-specific translator) to different simulation back-ends. The ultimate goal of this project is to provide a standard benchmark suite that studies the performance space of the simulators using realistic models.

4.6 Summary and Discussion

The average parallelism analysis and the associated parallelism characterization obtained from the analysis, such as parallelism profile, shape, and critical path, provides a deeper understanding of the behavior of parallel programs in

general, and parallel simulations in particular. The APSE framework is an environment independent, stand-alone parallelism analysis workbench, which analyzes program traces generated by instrumented message passing libraries or simulation libraries. The instrumentation of a library is fairly simple as special APSE monitor functions are provided to record events into a buffer and to flush the buffer to the file system. For example, for the instrumentation of the APSIS Time Warp simulation library, we had to add four appropriate calls at three places: one place to record the start of an event, one place to record the finish of an event, and finally at one place to omit rollback activities from the event trace.

The APSE analysis framework has been applied to two fairly simple simulation problems: unidirectional ring simulation with one message, and bidirectional simulation with two messages. For long running simulations, the measured average parallelism was in correspondence with the theoretically expected value. For short runs the average parallelism was higher due to the transient behavior of the degree of parallelism during LP initialization that dominates the results. The critical path figure of the unidirectional ring simulation depicts how the critical times of the events in the parallel simulation depend on the single event message that is routed through the system. For the bidirectional ring simulation we see that in the general case where a number of events are executed in parallel, the critical path meanders through the program activity graph and not necessarily follows one single activity through the system.

The inherent parallelism made available in the software system should not be a goal itself. Given two different implementations of one simulation application, the execution of these simulations can result in different inherent parallelism measures. This fact does not imply that the implementation with a higher degree of inherent parallelism will perform better than the implementation with a lower degree of parallelism. Independently of the ability of the PDES protocol to exploit the available parallelism, the parallel architecture must be able to bring this parallelism to expression. An important characterization of the available parallelism is the grain size or granularity of the parallelism: the amount of computation involved between two synchronization points. For example, if a high degree of parallelism indicates a fine grain level of parallelism, it is difficult for a distributed memory parallel architecture to achieve proper speedup figures. On the other hand, the implementation with a lower degree, but coarse grain, parallelism can behave well on a distributed memory parallel architecture and outperform the first implementation. Resuming, the performance measures made to evaluate the effectiveness of the PDES protocol are also influenced by the granularity of the parallelism in relation to the parallel architecture. It is also the responsibility of the parallel program designer to tailor the granularity of the parallelism to the architecture.

The parallelism evaluation by APSE is also applicable to parallel programs in general. For example, by instrumentation of communication libraries such as MPI or PVM, the parallel application generates program traces that can

be analyzed by the APSE tool. For example, the APSE parallelism evaluation has been successfully applied in a study of parallel sorting algorithms (van den Brink 1997).

Chapter 5

Parallel Asynchronous Cellular Automata

> **Monte Carlo Method** [Origin: after Count Montgomery de Carlo, Italian gambler and random-number generator (1792–1838).] A method of jazzing up the action in certain statistical and number-analytic environments by setting up a book and inviting bets on the outcome of a computation.
>
> —S. Kelly-Bootle, The Computer Contradictionary

5.1 Introduction

Many fundamental problems from natural sciences can be modeled as a *dynamic complex system*. A dynamic complex system is defined to be a set of unique elements with well defined microscopic attributes and interactions, showing emerging macroscopic behavior. This emergent behavior can, in general, not be predicted from the individual elements and their interactions. In many cases, dynamic complex systems cannot be solved by analytical methods, but require explicit simulation of the system dynamics to obtain insight into the system. A distinguished computational solving method for a large class of dynamic complex systems are (synchronous) *cellular automata* (CA). The CA model is in itself a set of dynamic systems where space, time, and variables are discrete. Cellular automata exhibit remarkable self-organization that can be used in models for real-world systems. For instance, the CA technique has proven to be useful for direct simulation of fluid flow experiments in both two and three dimensions. Other applications of CA in natural sciences can be found in lattice spin models such as the Ising model, or, in biology for example, in immune deficiency in cancer tissue simulations.

A commonly made assumption is that the update strategy of a CA is synchronous, i.e., the CA system evolves in discrete time. In the *asynchronous cellular automata* (ACA) model the synchronous cell update is relaxed to allow for independent, asynchronous cell updates. The asynchronous update strategy results in a more generic approach to CA, where also continuous-time models can be described conveniently (Bersini and Detours 1994; Lumer and Nicolis

1994).

The APSIS simulation environment is designed to effectively support the parallel execution of dynamic complex systems, and in particular spatially decomposed asynchronous cellular automata. The use of PDES to solve this class of problems is a challenge as the problem sizes in terms of memory and computation time can fill any parallel supercomputer. To validate the design and assess the practical usability of optimistic simulation methods in ACA models, we designed and implemented a well-defined and well-understood problem, namely the Ising spin model. The fairly simple Ising spin model shows a complex behavior that is parameterized by essentially one degree of freedom. The different behavior characterization of the Ising spin model put also different requirements to the PDES simulation kernel, and is as such an ideal vehicle for the validation and assessment of the APSIS environment. For example, the computational load of the Ising spin model is easily adjusted by the problem size, and the communication intensity of the simulation application is determined by a single model parameter. Besides the characteristic computational load and communication intensity, the Ising spin model also exhibits long-range correlations for certain model parameter ranges, where these long-range correlations induce time-dependent (evolution of the) computational behavior. One of the first questions that is raised is how the PDES protocol behaves with respect to this class of asynchronous systems. What are the limitations of the application of Time Warp to spatial decomposed regular problems, and in what way is the execution behavior influenced by the application parameters that determine the spatial interaction (synchronization) and computational load over the parallel processes?

The asynchronous cellular automata model will be introduced in Section 5.2, including the transition from synchronous to asynchronous cellular automata update strategy. In Section 5.3 the Ising spin model is presented and the Monte Carlo simulation method is briefly explained. Further, the asynchronous, continuous-time Ising spin model is introduced. The design and parallel implementation of the continuous-time Ising spin model is presented in Section 5.4, and the results of performance and scalability experiments are reported in Section 5.5.

5.2 Asynchronous Cellular Automata

5.2.1 Cellular Automata

Cellular automata are *discrete*, *decentralized*, and *spatially extended systems* consisting of large numbers of simple identical components with local connectivity. The meaning of discrete here is, that space, time, and features of an automaton can have only a finite number of states. The rational of cellular automata is not to try to describe a dynamic complex system from a global point of view as it is described using for instance differential equations, but modeling this system starting from the elementary dynamics of its interacting parts.

In other words, not to describe a complex system with complex equations, but let the complexity emerge by interaction of simple individuals following simple rules. In this way, a physical process may be naturally represented as a computational process and directly simulated on a computer. The original concept of cellular automata was introduced by von Neumann and Ulam to model biological reproduction and crystal growth respectively (von Neumann 1966; Ulam 1970). Since then it has been applied to model a wide variety of (complex) systems, in particular physical systems containing many discrete elements with local interactions. Cellular automata have been used to model fluid flow, galaxy formation, biological pattern formation, avalanches, traffic jams, parallel computers, earthquakes, and many more. In these examples, simple microscopic rules lead to macroscopic emergent behavior.

The locality in the cellular automata rules facilitate parallel implementations based on domain decomposition. The locality combined with the inherent parallelism of cellular automata make the design and development of high-performance software environments possible on parallel architectures. These environments exploit the inherent parallelism of the CA model for efficient simulation of complex systems modeled by a large number of simple elements with local interactions. By means of these environments, cellular automata have been used recently to solve complex problems in many fields of science, engineering, computer science, and economy. In particular, *parallel* cellular automata models are successfully used in fluid dynamics, molecular dynamics, biology, genetics, chemistry, road traffic flow, cryptography, image processing, environmental modeling, and finance (Talia and Sloot 1999).

5.2.2 Asynchronous Cellular Automata

In the previous section, we have seen the dualistic functionality of cellular automata in modeling and simulation. From a *modelers perspective*, a CA model allows the formulation of a dynamic complex system (DCS) application in simple rules. From a *computer simulation perspective*, a CA model provides an execution mechanism that evaluates the temporal dynamic behavior of a DCS given these simple rules. An important characteristic of the CA execution mechanism is the particular *update scheme* that applies the rules iteratively to the individual cells of the CA. The different update schemes impose a distinct temporal behavior on the model. Thus we must select the proper update mechanism that aligns with the dynamics of the model.

The update mechanism of CAs is described as being synchronously in parallel. However, for certain classes of DCS, the temporal dynamic behavior is asynchronous. In particular, systems with heterogeneous spatial and temporal behavior are, in general, most exactly mapped to asynchronous models (Bersini and Detours 1994; Lumer and Nicolis 1994). In case asynchronous models are solved by CA, the asynchronous temporal behavior must be captured by the update mechanism. This class of CA is called asynchronous cellular automata (ACA) (Ingerson and Buvel 1984; Lubachevsky 1987; Overeinder et al. 1992; Overeinder and Sloot 1993; Sloot and Overeinder 1999). The ACA model incor-

porates asynchronous cell updates, which are independent of the other cells, and allow for a more general approach to CA. With these qualifications, the ACA is able to solve more complicated problems, closer to reality.

Dynamic systems with asynchronous updates can be forced to behave in a highly inhomogeneous fashion. For instance in a random iteration model it is assumed that each cell has a certain probability of obtaining a new state and that cells iterate independently. As an example one can think of the continuous-time probabilistic dynamic model for an Ising spin system (Lubachevsky 1988).

5.2.3 The Asynchronous Cellular Automata Model

The model for an asynchronous cellular automata is given by its constituents: the cellular automata, the definition of the neighborhood, the transition rules, and—this is exclusively for asynchronous cellular automata—a time evolution function. We define a deterministic asynchronous cellular automata by

$$Z = (I^d, N, V, v_0, f, F, T),$$

where:

- I^d is the set of d-tuple integers, called an array of the cellular space. An element of I^d represents the coordinate of a cell or a cell at that coordinate. The positive integer d is called the dimension of the cellular space.
- N is an n-tuple of different elements of I^d, called the neighborhood index. With $N = (n_1, \ldots, n_n)$ and given a cell a, an element of the set $N(a) = \{(a+n_1), \ldots, (a+n_n)\}$ is called a neighborhood cell of a, and a is the center cell of those neighborhood cells. The set consisting of elements of N is simply called the neighborhood.
- The cellular space is homogeneous. Thus for all cells, V is a nonempty finite set, called a state set.
- The state set V has an element v_0 in which the cell is at rest, called the quiescent state.
- The local function f is a mapping from V^n to V and satisfies the specific property $f(v_0, \ldots, v_0) = v_0$.
- The next state of a is given by $s_t(a) = F(a, N, f, t)$, where f is instantaneous applied to the neighborhood N of a at time t.
- The time of the next state change evaluation of a is described by $t' = T(a, N, t)$, where $t' > t$.

A *nondeterministic* asynchronous cellular automata can be obtained by introducing a random experiment and defining a random variable on the sample space of the experiment. The nondeterministic local function f is augmented

with ξ, which is a realization of the random variable. The local function can be written as f_ξ, where one can think of f_ξ as a tabulated function depending on ξ, or if the random variable is discrete, ξ can be considered as an argument to the function.

Similarly, the time evolution function can be written as T_ξ, where T_ξ can be a tabulated function depending on ξ, or, for both continuous and discrete random variables, ξ can be considered as an argument to function T_ξ. The use of ξ for continuous random variables is valid since time is continuous in asynchronous cellular automata.

A more general system can be modeled with an asynchronous cellular automata where the transition rule operates on the active cell *and* its neighborhood (Priese 1978; Lubachevsky 1988). We can write the local transition function f as a mapping from V^n to V^n and the next state is given by $s_t(N(a)) = F(a, N, f, t)$.

The concurrent update of the neighborhood $N(a)$ of cellular automaton a introduces ambiguity if two distinct neighbors a and a' decide to change their overlapping neighborhoods $N(a)$ and $N(a')$ at the same simulation time. If two events with the same timestamp are scheduled for the same cellular automaton, the feature of *instantaneous update* that is characteristic to discrete event simulation, does not preclude this ambiguity in update order. Priese indicates how one may devise a computation and construction universal, concurrent, asynchronous cellular automaton where no overlapping can possibly occur. Another practical solution is to prohibit coincidence of update time in pairs of different cells, which is part of the predictability requirements as stated by Misra (1986), or to provide tie-breaking rules for simultaneous event updates (Wieland 1997).

5.2.4 Parallel Simulation of Cellular Automata Models

The parallelization of a CA, both for synchronous and asynchronous models, is realized by spatial decomposition. That is, the individual cells of the CA are aggregated into sub-lattices, which are mapped to the parallel processors. As we will see, parallel synchronization between the sub-lattices is very different for synchronous and asynchronous CA models.

Parallel Synchronous Cellular Automata Simulation

Similar to the sequential execution of synchronous CA, the cells in a parallel synchronous CA simulation undergo simultaneous state transitions under direction of a global clock. All cells must finish their state transition computations before any cell can start simulating the next clock tick.

The parallelization of the discrete-time simulation is achieved by imitating the synchronous behavior of the simulation. The simulation is arranged into a sequence of rounds, with one round corresponding to one clock tick. Between successive rounds, a global synchronization of all cells indicates that the cells

have finished their state change at time step t and the new time step $t + \Delta t$ can be started.

Generally, the simulation proceeds in two phases, a computation and state update phase, and a communication phase. The progress of time in this time-driven simulation is illustrated in Fig. 5.1.

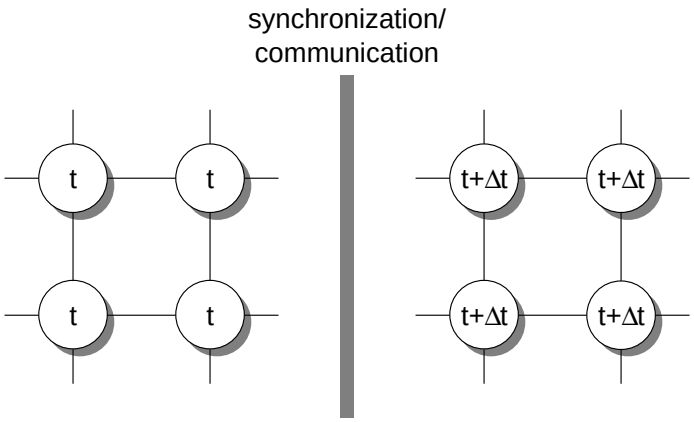


Figure 5.1: Time-driven simulation of a synchronous CA model, where computation and communication phases succeed each other.

Parallel Asynchronous Cellular Automata Simulation

In parallel ACA simulation, state transitions (further called events) are not synchronized by a global clock, but rather occur at irregular time intervals. In these simulations few events occur at any single point in simulated time and therefore parallelization techniques based on synchronous execution using a global simulation clock perform poorly. Concurrent execution of events at different points in simulated time is required, but this introduces severe synchronization problems. The progress of time in event-driven simulation is illustrated in Fig. 5.2.

The absence of a global clock in asynchronous execution mechanisms necessitates parallel discrete event simulation algorithms to ensure that cause-and-effect relationships are correctly reproduced by the simulator.

To summarize, the parallel *synchronous* execution mechanism for discrete-time models mimics the sequential synchronous execution mechanism by interleaving a computation and state update phase with a synchronization and communication phase. The parallel execution mechanism is fairly simple and induces a minimum of overhead on the computation. The parallel *asynchronous* execution mechanism for discrete event models, in our discussion the optimistic Time Warp simulation method, is more expensive than its sequential counterpart. The synchronization mechanism in optimistic simulation requires extra administration, such as state saving and rollback. Despite this overhead, op-

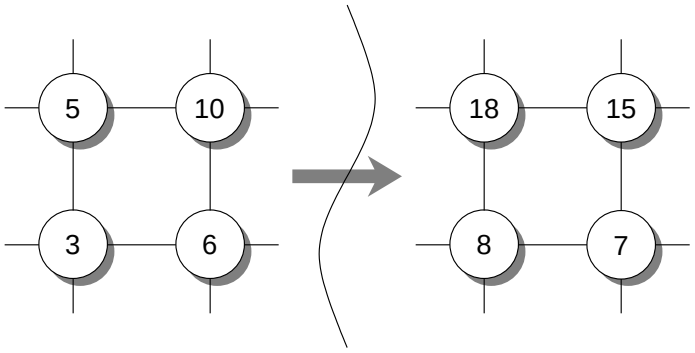


Figure 5.2: Progress of simulation time in event-driven simulation. As the cells evolve asynchronously in time, the simulation time of the individual cells are different.

timistic simulation is an efficient parallel execution mechanism for discrete event models.

5.3 Ising Spin Systems

The Ising spin model is a model of a system of interacting variables in statistical physics. The model was proposed by Wilhelm Lenz and investigated by his graduate student, Ernst Ising, to study the phase transition from a paramagnet to a ferromagnet (Brush 1967). A variant of the Ising spin model that incorporates the time evolution of the physical system is a prototypical example how asynchronous cellular automata can be used to simulate asynchronous temporal behavior.

A key feature in the theory of magnetism is the electron's spin and the associated magnetic moment. Ferromagnetism arises when a collection of such spins conspire so that all of their magnetic moments align in the same direction, yielding a total magnetic moment that is macroscopic in size. As we are interested how macroscopic ferromagnetism arises, we need to understand how the microscopic interaction between spins gives rise to this overall alignment. Furthermore, we would like to study how the magnetic properties depend on temperature, as systems generally loose their magnetism at high temperatures.

5.3.1 The Ising Spin Model

To introduce the Ising model, consider a lattice containing N sites and assume that each lattice site i has associated with it a number s_i, where $s_i = +1$ for an "up" spin and $s_i = -1$ for a "down" spin. A particular configuration or microstate of the lattice is specified by the set of variables $\{s_1, s_2, \ldots, s_N\}$ for all lattice sites (see Fig. 5.3).

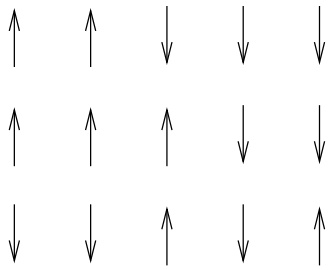


Figure 5.3: Schematic spin model for an Ising spin system.

The macroscopic properties of a system are determined by the nature of the accessible microstates. Hence, it is necessary to know the dependence of the energy on the configuration of spins. The total energy of the Ising spin model is given by

$$E = -J \sum_{i,\, j=\mathrm{nn}(i)}^{N} s_i s_j - \mu_0 H \sum_{i=1}^{N} s_i \,, \tag{5.1}$$

where $s_i = \pm 1$, J is the measure of the strength of the interaction between spins, and the first sum is over all pairs of spins that are nearest neighbors (see Fig. 5.4). The second term in Eq. 5.1 is the energy of interaction of the magnetic moment, μ_0, with an external magnetic field, H.

Figure 5.4: The interaction energy between nearest neighbor spins in the absence of an external magnetic field.

If $J > 0$, then the states $\uparrow\uparrow$ and $\downarrow\downarrow$ are energetically favored in comparison to the states $\uparrow\downarrow$ and $\downarrow\uparrow$. Hence for $J > 0$, we expect that the state of the lowest total energy is *ferromagnetic*, i.e., the spins all point to the same direction. If $J < 0$, the states $\uparrow\downarrow$ and $\downarrow\uparrow$ are favored and the state of the lowest energy is expected to be *paramagnetic*, i.e., alternate spins are aligned. If we add a magnetic field to the system, the spins will tend to orient themselves parallel to H, since this lowers the energy.

The average of the physical quantities in the system, such as energy E or magnetization M, can be computed in two ways: the time average and the statistical average. The time average of physical quantities are measured over a time interval sufficiently long to allow the system to sample a large number of microstates. Although time average is conceptually simple, it is convenient to formulate statistical averages at a given instant of time. In this interpretation,

all realizable system configurations describe an ensemble of identical systems. Then the ensemble average of the mean energy E is given by

$$\langle E \rangle = \sum_{s=1}^{m} E_s P_s \,,$$

where P_s is the probability to find the system in microstate s, and m is the number of microstates.

Another physical quantity of interest is the magnetization of the system. The total magnetization M for a system of N spins is given by

$$M = \sum_{i=1}^{N} s_i \,.$$

In our study of the Ising spin system, we are interested in the equilibrium quantity $\langle M \rangle$, i.e., the ensemble average of the mean magnetization M.

Besides the mean energy, another thermal quantity of interest is specific heat or heat capacity C_v. The heat capacity C_v can be determined by the statistical fluctuation of the total energy in the ensemble:

$$C_v = \frac{1}{kT^2} \left(\langle E^2 \rangle - \langle E \rangle^2 \right) \,.$$

And in analogy to the heat capacity, the magnetic susceptibility χ is related to the fluctuations of the magnetization:

$$\chi = \frac{1}{kT} \left(\langle M^2 \rangle - \langle M \rangle^2 \right) \,.$$

For the Ising model the dependence of the energy on the spin configuration (Eq. 5.1) is not sufficient to determine the time-dependent properties of the system. That is, the relation Eq. 5.1 does not tell us how the system changes from one spin configuration to another, therefore we have to introduce the dynamics separately.

5.3.2 The Dynamics in the Ising Spin Model

Physical systems are generally not isolated, but are part of a larger environment. In this respect, the systems exchange energy with their environment. As the system is relatively small compared to the environment, any change in the energy of the smaller system does not have an effect on the temperature of the environment. The environment acts as a heat reservoir or heat bath at a fixed temperature T. From the perspective of the small system under study, it is placed in a heat bath and it reaches thermal equilibrium by exchanging energy with the environment until the system attains the temperature of the bath.

A fundamental result from statistical mechanics is that for a system in equilibrium with a heat bath, the probability of finding the system in a particular microstate is proportional to the Boltzmann distribution (Reif 1965)

$$P_s \sim e^{-\beta E_s},$$

where $\beta = 1/k_B T$, k_B is Boltzmann's constant, E_s is the energy of microstate s, and P_s is the probability of finding the system in microstate s.

The Metropolis Algorithm

To introduce the dynamics that describe the system changes from one configuration to another, we need an efficient method to obtain a representative sample of the total number of microstates, while the temperature T of the system is fixed. The determination of the equilibrium quantities is time independent, that is the computation of these quantities does not depend on simulation time. As a result, we can apply Monte Carlo simulation methods to solve the dynamics of the system. The well-known Metropolis algorithm uses the Boltzmann distribution to effectively explore the set of possible configurations at a fixed temperature T, see for instance Binder and Heermann (1992). The Metropolis algorithm samples a representative set of microstates by using an *importance sampling method* to generate microstates according a probability function

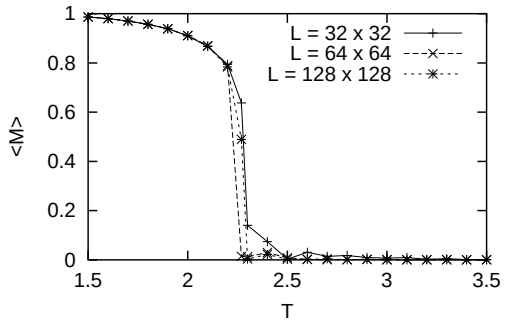
$$\pi_s = \frac{e^{-\beta E_s}}{\sum_{s=1}^{m} e^{-\beta E_s}}.$$

This choice of π_s implies that the ensemble average for the mean energy and mean magnetization can be written as

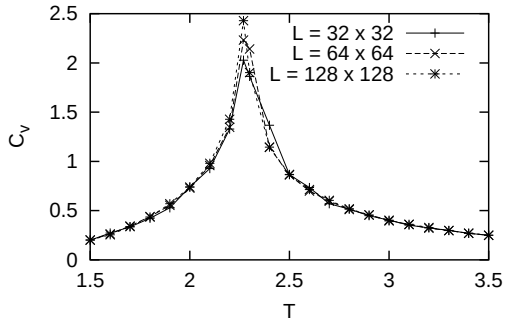
$$\langle E \rangle = \frac{1}{m} \sum_{s=1}^{m} E_s \quad \text{and} \quad \langle M \rangle = \frac{1}{m} \sum_{s=1}^{m} M_s .$$

The resulting Metropolis algorithm samples the microstates according to the Boltzmann probability. First, the algorithm makes a random trial change (a spin flip) in the microstate. Then the energy difference ΔE is computed. The trial is accepted with probability $e^{-\beta \Delta E}$ (note that for $\Delta E \leq 0$ the probability is equal to or larger than one and the trial is always accepted). After the trial, accepted or not accepted, the physical quantities are determined, and the next iteration of the Metropolis algorithm can be started.

The number of Monte Carlo steps per spin (or in general per particle) plays an important role in Monte Carlo simulations. On the average, the simulation attempts to change the state of each particle once during each Monte Carlo step per particle. We will refer to the number of Monte Carlo steps per particle as the "time," even though this time has no obvious direct relation to physical time. We can view each Monte Carlo time step as one interaction with the heat bath. The effect of this interaction varies according to the temperature T, since T enters through the Boltzmann probability for flipping a spin.



(a) Temperature dependence of the mean magnetization per spin for lattice size 32×32, 64×64, and 128×128.



(b) Temperature dependence of the specific heat for lattice size 32×32, 64×64, and 128×128.

Figure 5.5: Ising spin temperature dependence of the mean magnetization and specific heat.

The temperature dependency of the physical quantities $\langle M \rangle$ and C_v are shown in figures Fig. 5.5(a) and Fig. 5.5(b) respectively. For temperature $T = 0$, we know that the spins are perfectly aligned in either direction, thus the mean magnetization per spin is ± 1. As T increases, we see in Fig. 5.5(a) that $\langle M \rangle$ decreases continuously until $T = T_c$, at which $\langle M \rangle$ drops to 0. This T_c is known as the *critical temperature* and separates the ferromagnetic phase $T < T_c$ from the the paramagnetic phase $T > T_c$. The singularity associated with the critical temperature T_c is also apparent in Fig. 5.5(b). The heat capacity at the transition is related with the large energy fluctuations found near the critical temperature. The peak becomes sharper for larger systems but does not diverge because the lattice has finite sizes (singularities are only found in an infinite system).

Continuous-Time Ising Spin System

The standard Ising spin model represents a certain *discrete-time* model, as Monte Carlo steps are regarded to be time steps. However, the discrete evolution of the Ising spin configurations is considered an artifact. Glauber (1963) introduced *continuous-time* probabilistic dynamics for the Ising system to represent the time evolution of the physical system.

The Ising spin model with continuous-time probabilistic dynamics cannot be solved by Monte Carlo simulation, since time has no explicit implication on the evolution of the system in the Monte Carlo execution model. To capture the asynchronous continuous-time dynamics correctly, the problem is mapped to the ACA model and is executed by event-driven simulation.

In the continuous-time Ising spin model, a spin is allowed to change the state, a so-called spin flip, at random times. The attempted state change arrivals for a particular spin form a Poisson process. The Poisson arrival pro-

cesses for different spins are independent, however, the arrival rate is the same for each spin. Similar to the Monte Carlo simulation, the attempted spin flip, or trial, is realized by calculating the energy difference ΔE between the new configuration and the old configuration. The spin flip is accepted with the Boltzmann probability $e^{-\beta \Delta E}$.

The discrete-time and continuous-time models are similar. They have the same distribution of the physical equilibrium quantities and both produce the same random sequences of configurations. The difference between the two models is the time scale at which the configurations are produced: in discrete-time, the time interval between trials is equal, and in continuous-time, the time intervals are random exponentially distributed.

5.4 Optimistic Simulation of Continuous-Time Ising Spin Systems

A parallel model of the continuous-time Ising spin system is designed and implemented with use of the APSIS simulation environment. For the design and implementation of the parallel model a number of issues are important: Metropolis algorithm, simulation time advancement, random number generation, and the spatial decomposition parallelization strategy. Both the Metropolis algorithm and the time advancement (Poisson arrival process) require a pseudo-random number generator, which has to meet certain requirements to preclude undesired correlations if the pseudo-random generator is used in parallel processing. Spatial decomposition has consequences for the embedding of the parallel model in the APSIS simulation environment.

The algorithm that computes the dynamics of the Ising spin system at a temperature is the Metropolis algorithm, as discussed in the previous section. For each successful attempt, thus actual spin flip, the energy quantities E, $\langle E \rangle$, and $\langle E^2 \rangle$, and magnetic quantities M, $\langle M \rangle$, and $\langle M^2 \rangle$ are recomputed. Note that $\langle E \rangle$ and $\langle M \rangle$ are the *ensemble means*, thus not the mean energy of magnetization of the system at that simulation time, but rather the mean value of all ensemble configurations probed by the Metropolis algorithm up to the current simulation time. From the energy and magnetization values we can compute the specific heat C_v and magnetic susceptibility χ. The values E, $\langle E \rangle$, $\langle E^2 \rangle$, M, $\langle M \rangle$, and $\langle M^2 \rangle$ are typically values that are state saved by the optimistic simulation protocol.

The trials to update a spin in the continuous-time Ising spin system occur at random times. The interarrival times of the trials for a particular spin are independent and exponentially distributed, thus forming a Poisson arrival process with rate λ. In the discrete event simulation, upon the execution of a trial event at simulation time t, the spin flip is accepted according to the Metropolis algorithm, and the next trial event is scheduled at simulation time $t' = t - 1/\lambda \log U$, where U is a uniform distributed random variable in the interval $(0, 1]$.

In the PDES implementation of the Ising spin model we have two choices to schedule the trial events for the spin updates. With spatial decomposition of the Ising spin model, we aggregate a large number of spins into one sub-lattice and assign this sub-lattice to one LP. For example, the two-dimensional Ising spin system of size 128×128 is partitioned in eight sub-lattices of size 32×64. With this decomposition, each LP simulates the dynamics of $32 \cdot 64 = 2048$ spins. As each spin generates its own Poisson stream with rate λ, the LP can schedule for each individual spin a trial event. The execution of a trial event at a spin schedules a new future trial event for that particular spin. In this approach the number of events pending for execution at an LP is equal to the size of the sub-lattice, e.g., with the 32×64 sub-lattice the LP has 2048 events scheduled for future execution. However, we can do better since the Poisson asynchrony in the aggregated algorithm is a special case: the sum of k independent Poisson streams with rate λ each, is a Poisson stream with rate λk. In the event scheduling algorithm, k is the size of the sub-lattices. In the second approach, we neither maintain individual Poisson streams, nor future trial events for individual spins. Instead, a single cumulative Poisson stream is simulated, and spins are delegated randomly to meet these trial events.

In parallel simulation, and in particular parallel Monte Carlo simulation, special care should be taken with the generation of random numbers. Both the Metropolis algorithm and the Poisson stream need a uniform random variable for their operation. For sequential architectures, good random number generators exist. However, it is not at all trivial to find high-quality random number generators for parallel architectures. It should be noted that highly correlated and statistically dependent parallel random number generators originating from bad parallelization or distributed strategies may destroy or dramatically forge simulation results.

There are two basic parallelization techniques to produce random numbers. The first approach assigns different random number generators to different processors, and the second approach assigns different substreams of one large random number generator to different processors. The first approach suffers form intrinsically bad scalability (this approach requires thousands of different high-quality random number generators on massively parallel architectures such as the ASCI Option Red or ASCI Option Blue systems; Dyadkin and Hamilton (2000) presented approximately 2100 good 128-bit multipliers for congruential pseudo-random number generators). Additionally, there might also be unknown correlations between the different random number generators we use. The second approach can be controlled better, although its risks should not be forgotten.

There are two methods for splitting a given stream of random numbers into suitable parallel streams (Hellekalek 1998). The first method, the "leap-frog technique", assigns the substream $(x_{kn+i})_{n\geq 0}$ to the ith processor, $0 \leq i \leq k-1$. In other words, we use substreams of lag k of the original sequence $(x_n)_{n\geq 0}$, see Table. 5.1(a). The second method, the "splitting technique", partitions the original sequence into k (very long) consecutive blocks, see Table. 5.1(b). Each of the k processors is assigned a different block, where every block is defined by

a unique seed. This approach is very efficient way to assign different streams of random numbers to different processors.

p_0	p_1	p_2	p_3
x_0	x_1	x_2	x_3
x_4	x_5	x_6	x_7
x_8	x_9	x_{10}	x_{11}
$\vdots$	$\vdots$	$\vdots$	$\vdots$

(a) The leap-frog technique.

p_0	p_1	p_2	p_3
x_0	x_L	x_{2L}	x_{3L}
x_1	x_{L+1}	x_{2L+1}	x_{3L+1}
$\vdots$	$\vdots$	$\vdots$	$\vdots$
x_{L-1}	x_{2L-1}	x_{3L-1}	x_{4L-1}

(b) The splitting technique.

Table 5.1: Parallel random number generation: leap-frog and splitting.

The Mersenne Twister MT19937 random number generator has an extremely long period of $2^{19937}-1$ and an extensive theoretical background (Matsumoto and Nishimura 1998). Due to its long period, we can choose the initial value, the seed, randomly and obtain as many substreams as we need. It is highly improbable that two substreams will overlap. Other solutions are offered by parallel random number generator libraries such as the PRNG library (Entacher et al. 1998) or the SPRNG library (Ceperley et al. 1999). These libraries provide implementations of various parallel random number generators. The user can initialize the parallel random number generator by specifying all the parameters to the parallel random number generator, including the splitting method and the number of parallel streams.

An additional complicating factor in optimistic PDES is that due to the rollback synchronization, the sample path generated according to the desired statistics can be altered, unless some precautions are taken (Tsitsiklis 1989). In particular, if part of the simulation is performed for a second time, due to a rollback, one should use the same random numbers that were used the first time. Suppose that the dynamics of an LP has been formulated so that the statistics of the random variable x_i corresponding to the ith event has a prescribed distribution depending only on i. We can then generate random variables $x_0, x_1, \ldots$ and the the value x_i will be the one to be used for the simulation of the ith event, no matter how many times the ith event has to be simulated (due to rollbacks) and even if different simulations of the ith event corresponds to different simulation times.

To make random number generators rollback proof, the APSIS environment encapsulates the random number generators into the simulation kernel. For each ith event, the corresponding random variable x_i is generated once and stored in a data structure by the simulation kernel. Upon rollback from the kth event to the jth event, the next random variables x_i, $j \leq i \leq k$, are retrieved from the data structure before new random variables are generated. This can be efficiently implemented in a circular buffer (see Section 3.5.3, Fig. 3.5). Besides, the reuse of random variables is profitable as random number generation is relative expensive.

The resulting continuous-time Ising spin model is parallelized by spatial decomposition. The Ising spin lattice is partitioned into sub-lattices, and the sub-lattices are mapped onto parallel processors. To minimize the communication between sub-lattices, local copies of the neighbor boundaries are stored locally (see Fig. 5.6). By maintaining local copies of neighbor boundaries, spin values are only communicated when they are actually changed, rather than when they are only referenced. A spin flip along the boundary is communicated to the neighbors by an event message. The causal order of the event messages, and thus the spin updates, are guaranteed by the optimistic simulation mechanism.

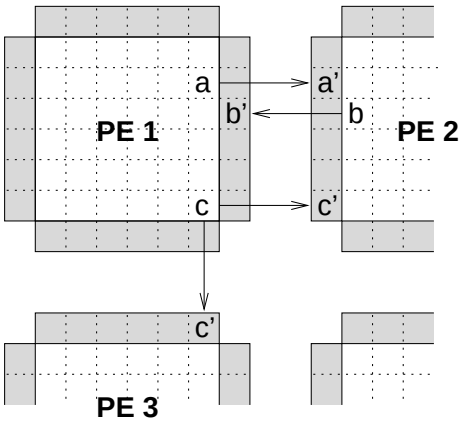


Figure 5.6: Spatial decomposition of the Ising spin lattice. The grey areas are local copies of neighbor boundary strips. For example, processor PE 2 has a local copy of spin “a” owned by processor PE 1. Processors PE 2 and PE 3 both own a copy of spin “c”. The arrows in the figure indicate the event messages sent upon a spin flip.

Asynchronous cellular automata, and thus also the Ising spin model, put efficient memory management requirements on the original formulation of the Time Warp method. The state vector of a spatial decomposed ACA can be arbitrarily large, that is, all the cells in the sub-lattice are part of the state vector. Hence, the incremental state saving method of the APSIS environment is used during the simulation of the Ising spin model. Although incremental state saving requires less state saving time and memory, there is an increased cost of state reconstruction. In general, the number of rolled back events is a fraction of the number of events executed during forward simulation. In this respect, the state recovery overhead of incremental state saving and copy state saving are in the ratio of 10^2 bytes to an order of 10^6 bytes, therefore incremental state saving is favorable in spatial decomposed ACA applications.

5.5 Parallel Performance and Scalability

A series of experiments were executed, for different problem sizes and Ising spin parameter settings, to get insight into the efficiency and scalability behavior of Time Warp. The experiments with the parallel Ising spin simulation were performed on the Distributed ASCI Supercomputer (DAS)*. (Note that ASCI stands for Advanced School for Computing and Imaging—a Dutch research school.) The DAS consists of four wide-area distributed clusters of total 200 Pentium Pro nodes. ATM is used to realize the wide-area interconnection between the clusters, while the Pentium Pro nodes within a cluster are connected with Myrinet system area network technology. The experiments are performed within one single cluster, thus all communication is via the 1.28 Gbit/s Myrinet. The communication layer is an efficient implementation of PVM on top of Panda (Ruhl et al. 1996). Panda is a virtual machine designed to support portable implementations of parallel programming systems. The efficient communication primitives and thread support in Panda allows for low latency, high throughput communication performance over the Myrinet network. The measured Panda/PVM (null message) latency is $\pm 17\,\mu$sec, and the throughput ± 60 MB/s.† For random number generation, we make use of the Mersenne Twister MT19937 random number generator, where each LP initializes the random number generator with a different seed to get parallel random number substreams.

5.5.1 Relative Parallel Performance and Scalability

In the first series of experiments, we study the *relative efficiency* and *scalability* of the parallel Ising spin simulation. This is done by comparing the execution time of the parallel simulation on one processor, $T_p(1)$, with the execution time on different number of processors, $T_p(P)$. The relative efficiency is now defined as

$$E = \frac{T_p(1)}{T_p(P) \cdot P}\,.$$

The parameters to the Ising spin experiment are the lattice size $L \times L$, the temperature T, the number of simulation steps, and the number of processors. The lattice size $L \times L$ and the number of processors determine the granularity of the computation, or the computation to communication ratio. Given the decomposition shown in Fig. 5.6, the boundary lattice points are potentially communicated to the neighboring LPs. The ratio of boundary lattice points to the total number of local lattice points is $4/M$, where $M \times M$ is the sub-lattice size after decomposition. The temperature T of the Ising spin system determines also in part the granularity of the LPs: as the temperature increases, the behavior of the system becomes more dynamic and hence more communication occurs between the nodes. The temperature of the system also influences

*http://www.asci.tudelft.nl/das/das.shtml or http://www.cs.vu.nl/das/

†Performance experiments with MPI are presented by al Mourabit (2000).

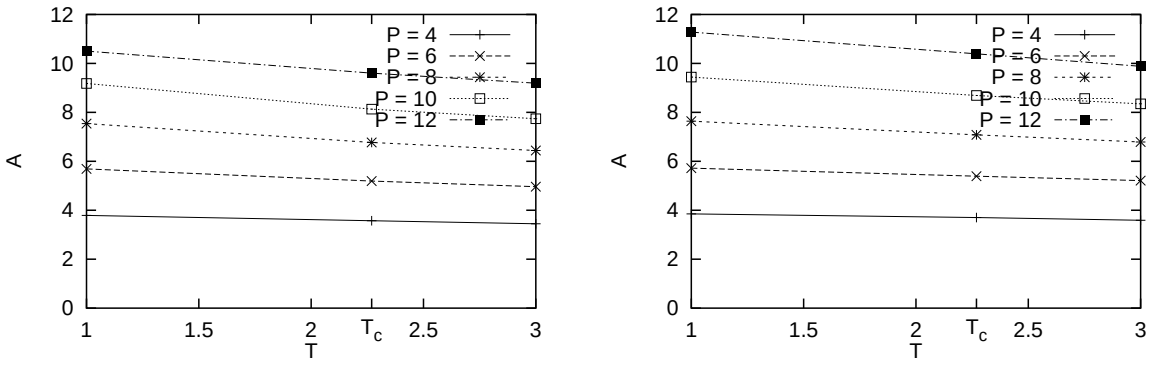
the computational behavior of the simulation in a more subtle way, which is presented in Chapter 6.

The simulation time is denoted by a derivative of Monte Carlo time steps. A Monte Carlo time step embodies $L \times L$ spin update attempts such that all the lattice points in the system have potentially got the opportunity to change their state. In the continuous-time Ising spin simulation we still use the notion of Monte Carlo steps to specify the duration of the simulation, as it conveniently indicates both the statistical evolution of the system and the expected amount of computational work. Of course, it has no (direct) relation to the simulated time, as simulation time progress is determined by the aggregated Poisson arrival process.

APSE Analysis

The influence of temperature and lattice size on the average parallelism inherent to the Ising spin simulation is studied by use of the APSE analysis framework (see Chapter 4). The APSE analysis allows one to study the scalability behavior of the simulation application without any assumption on the simulation protocol, or stated differently, with an ideal, omniscient simulation protocol. The results from the APSE analysis supports the interpretation of the experimental results in the next section. For example, if the APSE analysis of a simulation application results in a limited average parallelism, this should be attributed to the bounded inherent parallelism of the simulation application software, rather than the inability of the simulation protocol to exploit the available parallelism.

The dependency of the average parallelism on the Ising spin temperature is shown in Fig 5.7 for lattice sizes $L = 32$ and $L = 64$. From the figures we learn that the average parallelism decreases for increasing temperature.



(a) Average parallelism versus temperature for $L = 32$.

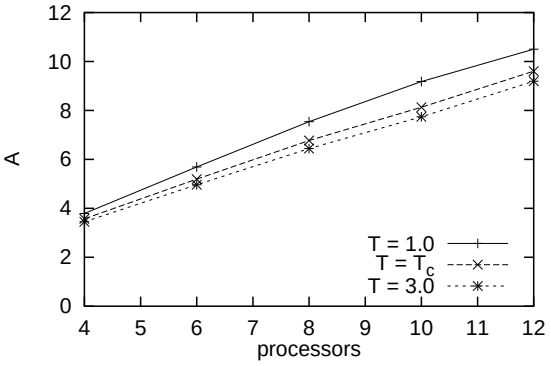
(b) Average parallelism versus temperature for $L = 64$.

Figure 5.7: APSE average parallelism versus temperature analysis.

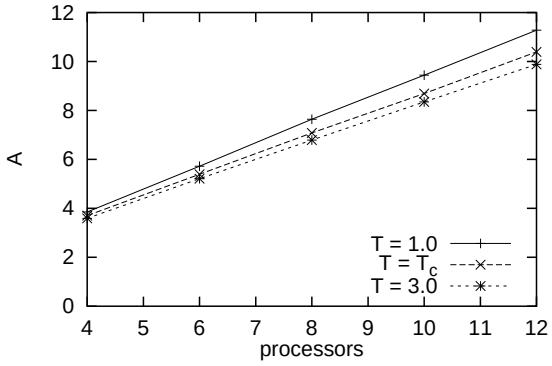
This can be explained by the higher rate of successful spin flips for higher temperatures. Spin flips on the sub-lattice boundaries must be synchronized with the neighboring sub-lattices, resulting in sequentialization of (part of) the spin flip trials.

The impact of lattice size on the sequentialization of boundary spin flips appears also from the figures for lattice sizes $L = 32$ and $L = 64$. The average parallelism is smaller for smaller lattice size L, and vice versa. For constant temperature, the ratio of boundary spin flips is larger for small lattice sizes than for large lattice sizes. Hence, for small lattice sizes the ratio of sequentialized spin flips is larger, which depresses the average parallelism.

Figure 5.8 depicts how the average parallelism scales with the number of processors for the temperatures $T = 1.0$, T_c, and 3.0. The average parallelism figures show the temperature and lattice size dependency as the number of processors increase. Again, low temperature and large lattice size enhance the average parallelism in the Ising spin simulation.



(a) Average parallelism versus number of processors for $L = 32$.

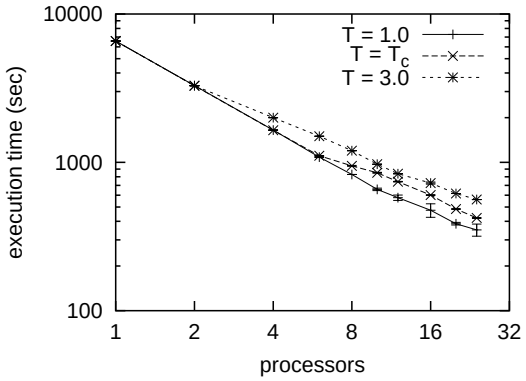


(b) Average parallelism versus number of processors for $L = 64$.

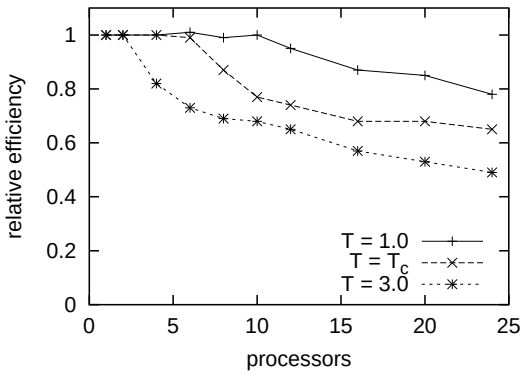
Figure 5.8: APSE average parallelism versus number of processors analysis.

Experiments

Figures 5.9 and 5.10 show the relation between execution time and the number of processors for fixed problem sizes 128×128 and 256×256. The number of Monte Carlo steps for each experiment is 12000 for lattice size 128×128, and 3000 steps for lattice size 256×256. In this way approximately the same number of events are executed for both system sizes. From these figures we can see that the parallel Ising spin simulation for $T = 1.0$ scales almost linearly up to 10 processors, but eventually drops to a relative efficiency of 0.78 for 24 processors with lattice size 128×128, and to a relative efficiency of 0.72 for 24 processors with lattice size 256×256. For temperature $T = 3.0$ the relative efficiency decreases gradually to 0.49 for 24 processors with lattice size 128×128, and to 0.62

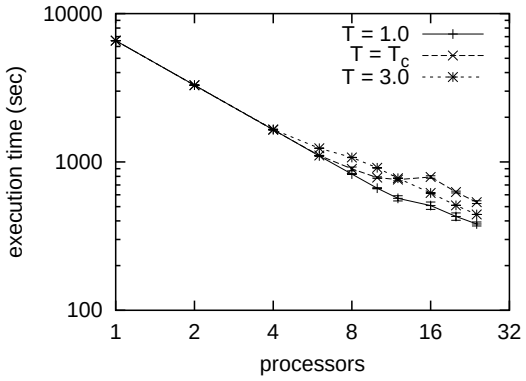


(a) Log-log plot of the execution time of parallel Ising spin for lattice size 128×128. The measure points and error boxes indicate the mean and standard deviation of 6 measurements.

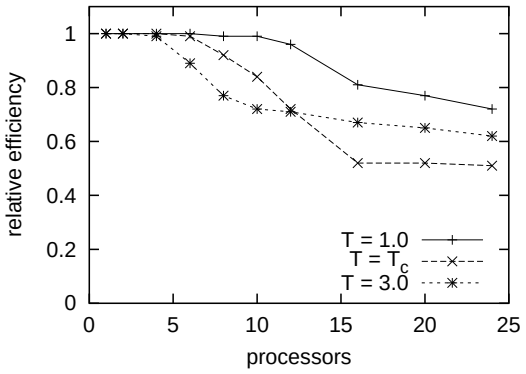


(b) Relative efficiency of parallel Ising spin for lattice size 128×128.

Figure 5.9: Scalability and relative performance of parallel Ising spin simulation.



(a) Log-log plot of the execution time of parallel Ising spin for lattice size 256×256. The measure points and error boxes indicate the mean and standard deviation of 6 measurements.



(b) Relative efficiency of parallel Ising spin for lattice size 256×256.

Figure 5.10: Scalability and relative performance of parallel Ising spin simulation.

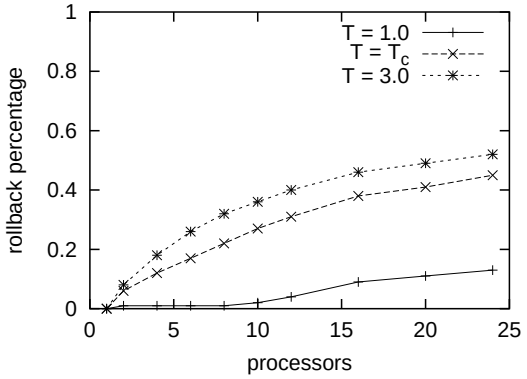
for 24 processors with lattice size 256×256. The decreasing efficiency is mainly due to the increased costs to synchronize the parallel processes. With the increase in the number of processors, the time period necessary to synchronize the parallel simulation processes also increases. A minimal memory manage-

ment requirement in optimistic simulation is the limitation of optimism, and hence the maximum rollback length (this is described further on in this section). For lattice size 256×256, the decrease in efficiency flattens at $P = 16$ for $T = 1.0$ and T_c, and at $P = 10$ for $T = 3.0$, since the rollback lengths in these regimes approach the maximum rollback length. Additional increase of the number of processors does not further increase the rollback lengths.

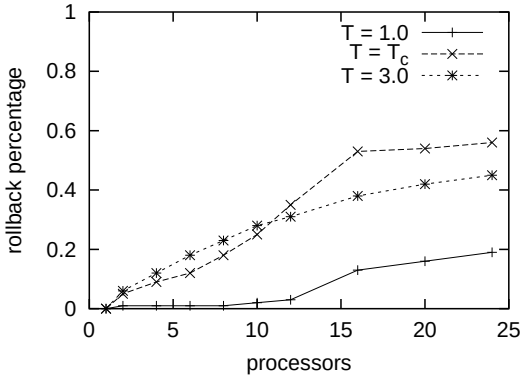
Temperature $T = T_c \approx 2.269$ is a special case. If we consider Fig. 5.9 and Fig. 5.10 separately, we see different scaling behavior for $T = T_c$ compared with $T = 1.0$ and $T = 3.0$. For lattice size 128×128 (Fig. 5.9), $T = T_c$ scales more or less within the bounds of $T = 1.0$ and $T = 3.0$, which is expected as synchronization costs increases with the temperature and the number of processors. However, for lattice size 256×256 (Fig. 5.10), the scalability behavior (i.e., the execution time and efficiency versus the number of processors) for $T = T_c$ is not bounded by $T = 1.0$ and $T = 3.0$. Up to 10 processors, the Ising spin scaling behavior can be explained by increased communication and synchronization costs due to the dynamics (or temperature) of the Ising spin system, but for 12 processors and more, another factor determines the execution time behavior—which is discussed further on. In Fig. 5.10(a) the execution time increases from 12 to 16 processors for $T = T_c$, resulting in execution times larger than for $T = 3.0$.

The influence of the temperature, the lattice size, and the number of processors on the execution behavior of the parallel simulation processes is further investigated. Figure 5.11 shows the rollback percentage of the total amount of executed events for the lattice size 128×128 and 256×256. The rollback percentage is an expression of the amount of synchronization errors due to optimistic execution of events. As such, it is a relative indication of the increased execution time due to event execution order dependencies and simulation protocol overhead. If we consider Fig. 5.11(a), the rollback percentages for lattice size 128×128, we find a correspondence between the increase of the rollback percentage and the decrease in relative performance as shown in Fig. 5.9(b). Moreover, in Fig. 5.11(b) we see the same anomalous rollback percentage behavior for $T = T_c$ as in the relative efficiency in Fig. 5.10(b). The increase in execution time for $T = T_c$ in the trajectory from 12 to 16 processors in Fig. 5.10(a) can also be found in the strong increase of rollback percentage for $T = T_c$ in the trajectory from 12 to 16 processors in Fig. 5.11(b). Hence, there is a strong relation between the rollback behavior and the execution time (and the derived relative performance).

To understand the anomalous performance behavior of the Ising spin simulation with lattice size 256×256 at temperature $T = T_c$, a detailed execution trace of the event rate is monitored. The event rate is the number of events that are committed per second, and in this respect a measure for progress. During normal operation, the Ising spin model simulation reaches an event rate of around 19 000 events per second (see Fig. 5.12(a)). If we look in more detail to Fig. 5.12(a), we can identify four serious glitches in the event rate, around execution time 20, 48, 58, and 88, which indicate periods of resynchronization of the parallel simulation. In these periods, the event rate drops to

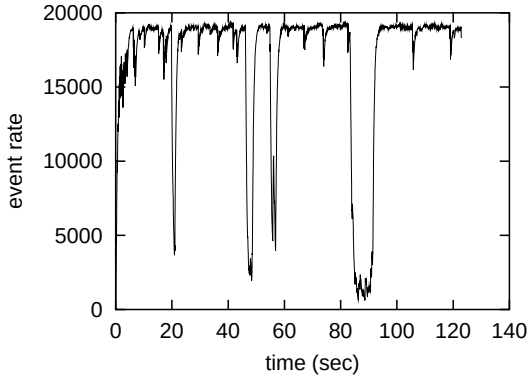


(a) Rollback percentages for lattice size 128×128.

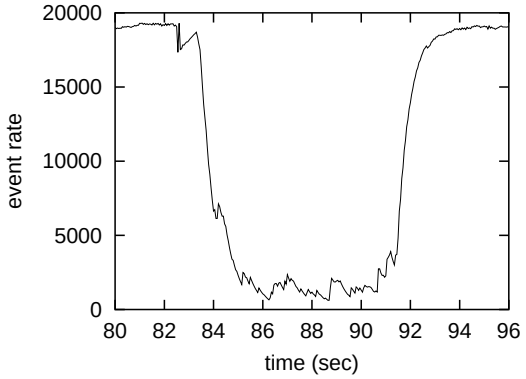


(b) Rollback percentages for lattice size 256×256.

Figure 5.11: Event rollback percentages for lattice sizes 128×128 and 256×256.



(a) Simulation progress (event rate) during execution. Parallel Ising spin with lattice size 256×256 on 6 processors.



(b) Simulation progress (event rate) during execution (detail of Fig. 5.12(a) between 80 and 96).

Figure 5.12: Simulation progress (event rate) during execution. The curve is smoothed by taking the exponential weighted moving average (EWMA), as the EWMA follows the dynamic behavior accurately and can be efficiently computed.

10% of the steady state performance (about 2000 events per second). In particular the period centered around 88 (see Fig. 5.12(b)) takes about 10 seconds to resynchronize and weights heavily upon the parallel performance.

The periods of resynchronization are a typical example of *thrashing*, where most of the time is spent on simulation rollback instead of forward simulation. While one simulation process rolls back, another process advances in simulation time. When the rollback is completed, the simulation process restarts

with event execution and as a result sends event messages to neighboring processes. These event messages arrive in the simulation past of the neighboring processes, and trigger a rollback, etc., etc., until the simulation processes are in synchrony. The thrashing behavior is a combination of a number of factors: number of processors, lattice size, event granularity, and temperature (synchronization frequency).

To shorten these periods of resynchronization, the optimism of the protocol must be throttled, that is, the simulation execution mechanism should not execute events that lie in the remote future as it is likely that these events have to be rolled back eventually. The progress of the individual simulation process should be bound to a limited simulation time window (see Section 2.4.5). In this way, the processes are forced to synchronize with each other in a short time frame, after which the simulation can continue as before. A key problem with Bounded or Moving Time Window optimism control is the determination of the appropriate size of the virtual time window. A narrow time window limits the rollbacks, but also the amount of parallelism. A time window that is too large, can potentially exploit more parallelism, but the rollbacks increase as well.

Another performance consideration in the determination of the appropriate virtual time window size is the sequential simulation performance and its relation to the GVT computation frequency (or progress rate). A narrow time window does not only limit the amount of parallelism, but can also limit the sequential event rate due to a slow GVT progress rate. As the sequential simulation process proceeds faster in simulation time (that is the progress of the LVT) than the progress of the GVT, the sequential simulation process will eventually reach the upper time window boundary, and will block until the next GVT progress update. The influence of virtual time window size on the sequential Ising spin simulation performance in APSIS is presented in Table 5.2 and Table 5.3. In the experiments, a new GVT update computation is started every 50 msec. The first columns of Tables 5.2 and 5.3 show the sequential simulation execution times with unbounded time window for the three temperatures $T = 1.0$, $T = T_c$, and $T = 3.0$. As one can see, the execution time of the sequential simulation increases with the temperature. This can be explained by the dynamics of the simulation: the higher the temperature, the higher the dynamics of the Ising spin system, and hence the higher the computational costs.

	VTW = ∞	VTW = 3000	VTW = 2000
T = 1.0	1146	1645 (1.44)	2463 (2.15)
T = 2.269	1232	1645 (1.34)	2464 (2)
T = 3.0	1340	1645 (1.23)	2464 (1.84)

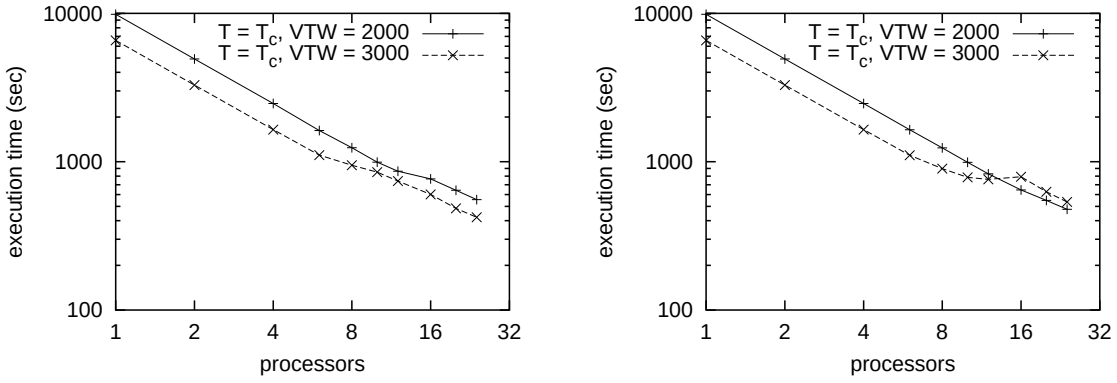
Table 5.2: Single processor execution time (in seconds) for Ising spin lattice size 128×128 and 3000 Monte Carlo steps. The slow down factor due to bounded virtual time window size is in parentheses.

	VTW = ∞	VTW = 3000	VTW = 2000
T = 1.0	4596	6578 (1.43)	9868 (2.15)
T = 2.269	4919	6577 (1.34)	9866 (2.01)
T = 3.0	5339	6579 (1.23)	9868 (1.85)

Table 5.3: Single processor execution time (in seconds) for Ising spin lattice size 256×256 and 3000 Monte Carlo steps. The slow down factor due to bounded virtual time window size is in parentheses.

The influence of the bounded virtual time window size, and hence the GVT update rate, is apparent from the second and third columns in Table 5.2 and Table 5.3. Ideally, for infinite GVT update rate, the time window size does not limit the sequential simulation performance as long as there is at least one pending event within the boundaries of the time window. In practice however, the GVT update rate is finite, and together with the time window size it limits the potential event execution rate, and hence determines the execution time. As one can see from Tables 5.2 and 5.3, the sequential execution time for virtual time window sizes VTW = 3000 and VTW = 2000 are determined by the time window size and *not* by the temperature of the Ising spin system (as with window size VTW = ∞). The interdependency between virtual time window size and GVT update rate adds another dimension to the determination of the appropriate time window size.

The effect of the bounded virtual time window on the simulation execution time can be clearly seen in Fig. 5.13. All the Ising spin experiments discussed before are performed with a virtual time window of 3000. In the APSIS environment, the virtual time window is also used as a memory management mechanism. If no virtual time window is used, arbitrary long rollbacks can



(a) Ising spin lattice size 128×128.

(b) Ising spin lattice size 256×256.

Figure 5.13: Log-log plot of the execution times of parallel Ising spin for different virtual time windows.

occur whose history (i.e., event queue, output queue, etc.) consumes all (virtual) memory. Hence, for memory management purposes the time window should be bounded, and experimentally a time window size of 3000 showed to be an appropriate starting value. (For a discussion on optimal virtual time window sizes, see Section 6.4.3.) The execution time figures for the virtual time window size VTW = 3000 in Fig. 5.13 are the same results as presented in Fig. 5.9(a) and Fig. 5.10(a) at temperature $T = T_c$. For virtual time window size VTW = 2000, the sequential simulation execution time (single processor execution) is about 3000 seconds longer than for VTW = 3000 (for both lattice sizes). The relative distance between the execution times for VTW = 2000 and VTW = 3000 remains constant for 2, 4, and 6 processors. In this region the execution time difference is predominantly determined by the virtual time window and its effect on the simulation event rate (see also previous discussion). However, where the execution times for VTW = 3000 start to deviate from linear scaling behavior (at 8, 10, 12, and 16 processors) due to excessive rollback behavior, the execution times for VTW = 2000 continues to scale linearly with the number of processors up to 12 processors for lattice size 128×128, and up to 16 processors for lattice size 256×256. The most prominent property of the figures is the crossover point in Fig. 5.13(b), between the execution times for VTW = 2000 and VTW = 3000. For 12 processors, the execution times for both virtual time window sizes are almost the same, but for 16 processors, the execution time for VTW = 2000 is significantly shorter.

The results presented in Fig. 5.13 show the potential of virtual time window management to control excessive rollback behavior, i.e., thrashing. However, the determination of an appropriate time window size is far from trivial, as it depends on various system parameters (i.e., number of processors) and application parameters (i.e., in Ising spin for example lattice size or temperature).

5.5.2 Absolute Parallel Performance and Scalability

In the second series of experiments, we study the *absolute efficiency* of the parallel Ising spin simulation compared to the best-known sequential Ising spin simulation for different temperatures, problem sizes, and event granularity (that is, the amount of work per event). The sequential continuous-time Ising spin simulation is basically a Monte Carlo simulation extended with a Poisson arrival process to incorporate time evolution into the model. The Monte Carlo simulation execution mechanism is a lightweight process compared to sequential discrete event simulation execution mechanism. With Monte Carlo simulation there is nearly no overhead involved in the execution of the spin flip trials: a random spin is selected and a trial is executed. With discrete event simulation, a trial is an event that must be scheduled for future execution, that is, inserted into the event list (in general a priority queue). Later, if the scheduled trial is the next pending event, the event is dequeued and the trial is executed. Parallel discrete event simulation includes, besides the event list management overhead, also the state saving and rollback overhead as described in the previous section. The absolute efficiency figures include all these

extra overhead costs compared to the sequential Monte Carlo simulation.

The experiments to assess the absolute performance, quantify the overhead induced by the APSIS parallel simulation protocol. An interesting perspective to approach the quantitative overhead evaluation is to relate this overhead with the so-called *event granularity*. The event granularity is defined as the amount of work per event (or trial in this discussion) and is in our study the amount of extra computational work in terms of a sinus and an exponential evaluation. The results for event granularity 0 are for the basic Ising spin system. The results for increasing event granularities give an indication how a similar problem scales as the amount of computational work to evaluate a trial (or state change) increases. Note that extra computational work is assigned to each trial, successful or not successful.

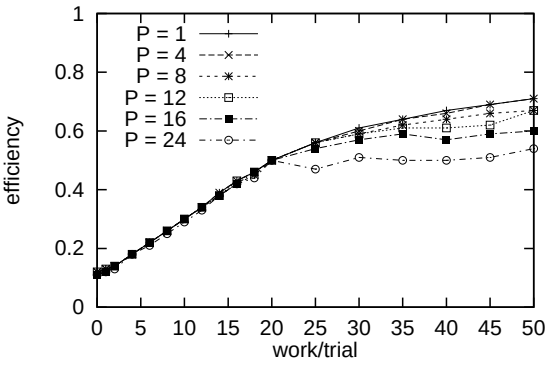
The absolute efficiency is defined as

$$E = \frac{T_s}{T_p(P) \cdot P},$$

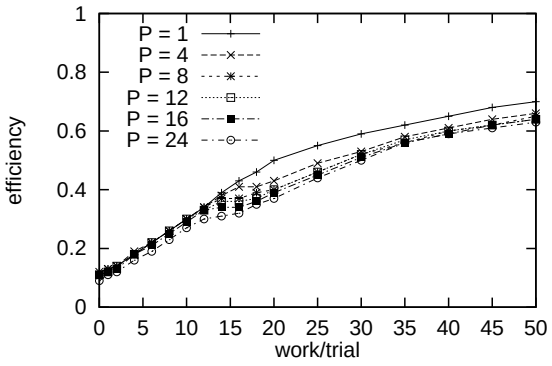
where T_s is the execution time of the sequential Monte Carlo Ising spin simulation, and $T_p(P)$ is the execution time of the parallel Ising spin simulation on P processors.

The absolute efficiency experiment results for Ising spin simulations with temperatures $T = 1.0$, $T = T_c$, and $T = 3.0$ are presented in Fig. 5.14. The absolute efficiency figures show the performance for different number of processors with scaled problem size, i.e., the lattice size is scaled with the number of processors such that the amount of local work at a processor remains constant. The lattice sizes for $P = 4, 8, 12, 16, 24$ are the square of $L = 91, 128, 158, 181, 222$ respectively. The performance figures for $P = 1$, that is the parallel simulation executed on one processor, is included as an upper boundary to the performance figures for other values of P. All results presented in the figures are the means of six experiments. The figures indicate that the parallel performance depends on the event granularity and Ising spin temperature. The event granularity determines the PDES protocol overhead ratio, apart from synchronization errors. The temperature T of the Ising spin system determines the computation/communication ratio: as the temperature increases, the behavior of the system becomes more dynamic and hence more communication occurs between the processors.

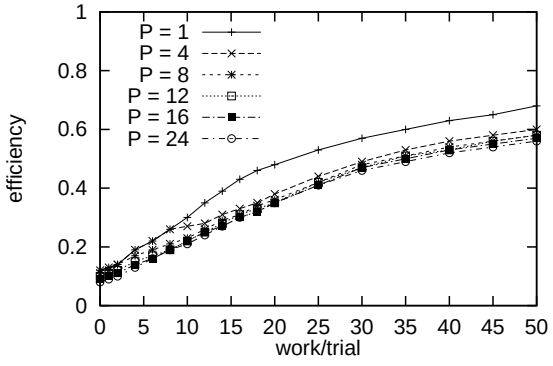
For low temperature $T = 1.0$, the absolute efficiency starts at 0.12 for work/-trial is 0. Around work/trial is 20, the absolute efficiency starts to diverge for the different number of processors, and eventually varies from 0.71 for $P = 4$ to 0.54 for $P = 24$ at work/trial is 50. For critical temperature $T = T_c$, we see in Fig. 5.14(b) that the point of divergence has moved from 20 to 15. However, at work/trial is 30, the performance figures per processor converge and end up in the range of $[0.63 - 0.66]$. Finally for high temperature $T = 3.0$, the point of divergence moved down to the range $[4 - 8]$, see Fig. 5.14(c). Also for high temperature, the performance figures converge and end up in the range of $[0.56 - 0.6]$.



(a) Temperature $T = 1.0$.



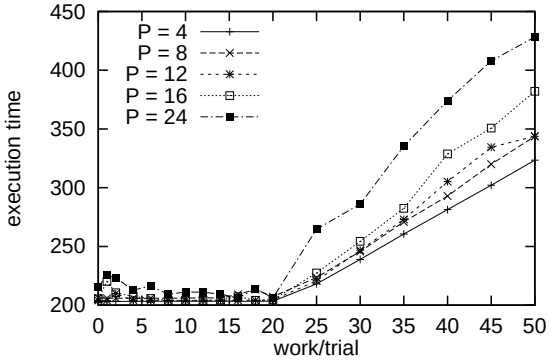
(b) Temperature $T = T_c$.



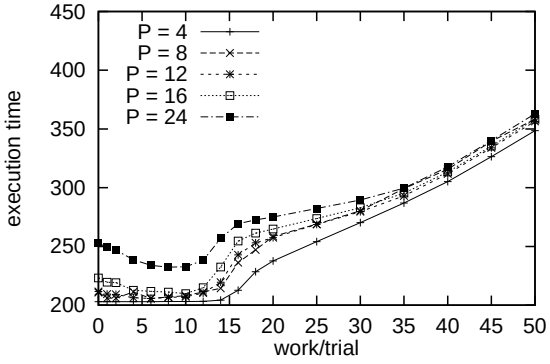
(c) Temperature $T = 3.0$.

Figure 5.14: Absolute efficiency versus event granularity (work/trial) for parallel Ising spin simulations with temperatures $T = 1.0$, T_c, and 3.0 on 1, 4, 8, 12, 16, and 24 processors, with scaled problem size.

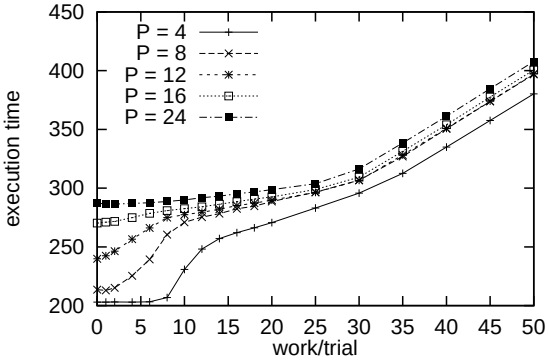
To understand Fig. 5.14, the relation between execution time, temperature and rollback behavior must become clear. Figure 5.15 shows the parallel Ising spin simulation execution times for the three temperatures. In Fig. 5.15(a), the execution times for low temperature $T = 1.0$ are almost constant up to a work/trial of 20, which is to be expected with a scaled problem size. However, after this point, the execution times for larger number of processors starts to increase faster than for smaller number of processors. This is due to increasing synchronization costs of the Time Warp protocol for increasing number of processors. For critical temperature $T = T_c$ and high temperature $T = 3.0$, we see an interesting transient execution time behavior in Fig. 5.15(b) and Fig. 5.15(c). At work/trial is 0, the execution times increase with the number of processors. For the critical and high temperature, there is more communication, and hence more synchronization overhead. As the work/trial increases,



(a) Temperature $T = 1.0$.



(b) Temperature $T = T_c$.

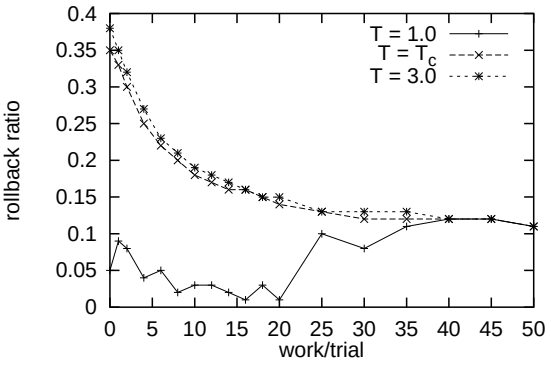


(c) Temperature $T = 3.0$.

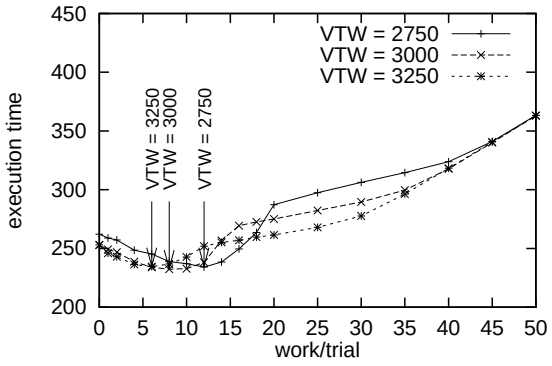
Figure 5.15: Parallel Ising spin simulation execution times for temperatures $T = 1.0$, T_c, and 3.0 on 1, 4, 8, 12, 16, and 24 processors, with scaled problem size.

the execution times converge to each other. For this magnitude of work/trial, the increased synchronization costs for larger number of processors are compensated by the larger event granularity.

The interesting transient execution time behavior is most prominent for intermediate temperatures around $T = T_c$, see Fig. 5.15(b). Here, the execution times for $P = 16$ and $P = 24$ even decrease with increasing work/trial up to 10. Apparently, two factors determine the execution time: one factor increases, and the other factor decreases for larger work/trial values. The factor that increases the execution time is of course the amount of work per trial, so we expect to see an increase in execution time for increasing work/trial. The second factor that decreases with work/trial is the rollback ratio, i.e., the ratio of events that are rolled back to the total number of executed events (rolled back or committed). In Fig. 5.16(a), the rollback ratio versus the work/trial are depicted for the three temperatures $T = 1.0$, T_c, and 3.0. For low temperature



(a) Event rollback ratio versus work/trial for temperatures 1.0, T_c, and 3.0.



(b) Execution time versus work/trial for the virtual time window sizes 2750, 3000, and 3250 with temperature T_c.

Figure 5.16: Influence of event rollback ratio and virtual time window on the execution time for 24 processors.

$T = 1.0$, the rollback ratio fluctuates significantly; the large variance can be explained by the infrequent synchronization which can actuate large cascaded rollbacks. For the intermediate and high temperatures, the rollback ratio falls off smoothly for increasing work/trial. Consequently, the synchronization overhead decreases. Thus, the increasing work per trial and the decreasing rollback ratio compete with each other, where the rollback ratio dominates for small values of work/trial, and the work per trial dominates for larger values of work/trial.

The crossover point, where the rollback ratio and work/trial are in balance, is partly determined by the virtual time window size. As the virtual time window size determines the amount of optimism, it also indirectly determines the rollback length and frequency. In Fig. 5.16(b), we see how the crossover point moves from work/trial is 12 to 6 for VTW = 2750 to 3250.

5.6 Summary and Discussion

An important subclass of dynamic complex systems, namely asynchronous cellular automata, has been used to rigorously evaluate the APSIS simulation environment. The specific asynchronous cellular automata used in our experiments is the continuous-time Ising spin model. The Ising spin model is a well-defined and understood problem, and shows a complex behavior that is essentially parameterized by the Ising spin temperature. The spatial decomposition of the Ising spin model over the parallel processors put severe memory constraints upon the APSIS environment, necessitating the use of incremental state saving.

The average parallelism analysis within the APSE framework exhibits the dependency of the average parallelism on the Ising spin temperature and the lattice size. Increasing temperatures results in (slowly) decreasing average parallelism, and increasing lattice sizes incorporate larger average parallelism. The APSE average parallelism analysis is consistent with our experiments, except around the critical temperature T_c. The parallel Ising spin simulation execution times around the critical temperature are larger than the execution times for temperature $T = 3.0$. Detailed study of the event rate showed that thrashing behavior of the parallel simulation occurs around the critical temperature, resulting in a simulation progress drop of 90%.

The absolute efficiency study compares the performance of the (parallel) discrete event simulation with the sequential Monte Carlo simulation implementation of the Ising spin model. The absolute efficiency is also a measure of the amount of overhead introduced by the (parallel) discrete event simulation compared to the relatively simple Monte Carlo simulation. In this respect, the event granularity is an important quantity as it determines the (parallel) discrete event simulation protocol overhead. The experimental results show a subtle interplay between the increased execution time for increasing event granularity, and decreasing rollback ratio and thus decreasing PDES overhead.

The application of optimistic parallel discrete event simulation methods such as Time Warp to asynchronous cellular automata is in potential a viable approach to parallelize the simulation. However, two essential extensions to the Time Warp method have to be included: incremental state saving and optimism control (throttling). The results show that given a fast communication network such as Myrinet, the Time Warp optimistic simulation method achieves good scalable performance. In particular, low communication latencies are essential to achieve performance, as the event messages are small.

The most promising approach to effective optimism control is the design and implementation of an adaptive mechanism. That is, the parallel simulation kernel determines the optimal virtual time window size using local state variables, such as event rate, rollback ratio, and communication statistics. A future research challenge is to devise a forecast method that exploits the local state variables for adaptive virtual time window control. The formulation of simple though applicable metrics to control the amount of optimism in the Time Warp method determines the success of the mechanism.

Chapter 6

Self-Organized Critical Behavior in Time Warp

> We call things we don't understand complex, but that means we haven't found a good way of thinking about them.
>
> —Tsutomu Shimomura

6.1 Self-Organized Criticality

Spatially extended dynamical systems, that is, systems with both temporal and spatial degrees of freedom, are common in physics, biology, and economics. The spatiotemporal behavior of these dynamic complex systems has been studied extensively, but there is still little understanding. In particular, two phenomena require some unifying underlying explanation, namely the temporal effect known as $1/f$ *noise*, and the emergence of spatial structures with *scale-invariant, self-similar (fractal) properties*.

Most of the time, equilibrium systems with short-range interactions, exhibit exponentially decaying correlations. Infinite correlations, i.e., scale invariance, can be achieved by fine-tuning some parameters (e.g., temperature) to a critical value. An example of such a system is the Ising spin model presented in Chapter 5.

Besides systems exhibiting critical behavior, a large class of non-equilibrium locally interacting, nonlinear systems spontaneously develop scale invariance. Such composite systems with many interacting degrees of freedom may evolve to a critical state in which minor events may trigger a chain reaction that can affect an arbitrarily large number of constituents of the system. This state is called *self-organized criticality* (SOC) (Bak et al. 1988). The probability of spontaneously generated structures or events, further called *avalanches*, of many different sizes s show a power-law distribution

$$P(s) \sim s^{-\tau} ,$$

where τ is a critical exponent and most other observables of the system have no intrinsic time or length scale. This implies that we expect to observe scale

invariance, or power-law scaling, in the system. The absence of intrinsic length scale is attributed to SOC, where avalanches of all sizes contribute to keep the system perpetually in a critical state. This critical state is robust with respect to any small change in the rules of the system. The size of an avalanche can be defined in different ways. It can be measured by the number of relaxation steps needed for the chain reaction to stop or the total number of sites involved in the avalanche.

Many naturally occurring systems exhibit this kind of scaling- or self-similar behavior, examples are earthquakes, stock markets, and ecosystems (Turcotte 1999). The concept of SOC is developed to explain the behavior of such systems. Self-organized critical behavior was first investigated for sandpile models (Bak et al. 1988). In this cellular automata model, a particle is dropped onto a randomly selected lattice point. When a lattice point accumulates four particles, they are redistributed to the four adjacent lattice points, or in case of edge lattice points they are lost from the grid. Redistributions can lead to further instabilities and avalanches of particles in which many particles may be lost from the edges of the lattice. The average number of particles per lattice point is the density that fluctuates about a quasi-equilibrium value. One measure of the avalanche size is given by the number of particles lost by the lattice during a sequence of redistributions, or alternatively can be given by the number of lattice points that participate in the redistribution. Figure 6.1 shows the distribution of the avalanche sizes, including the power-law fitted to the distribution.

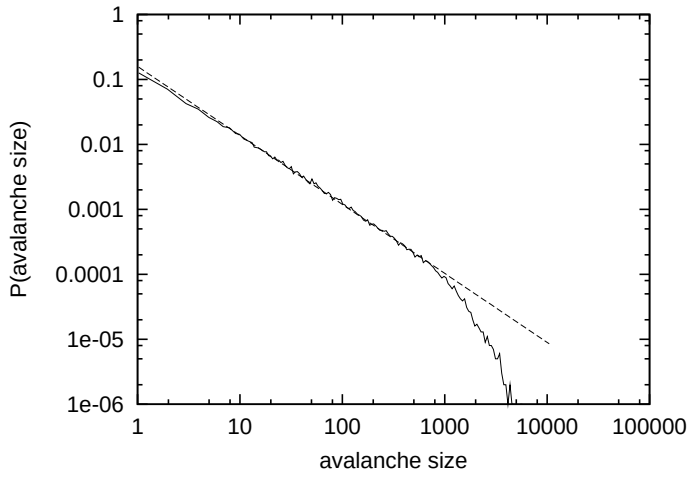


Figure 6.1: Avalanche size distribution in the sandpile model on a 100×100 lattice. The exponent of the fitted line is -1.06.

In this chapter a highly speculative conjecture is made, that the Time Warp dynamics can be characterized by self-organized criticality. The spatiotemporal behavior of the Time Warp method is investigated on the basis of the Ising spin model. The influence of temperature and various finite-size scaling effects such as lattice size, number of processors, and virtual time window size are

studied. If the conjecture is true, yet another hint is given that also problems in the field of parallel computing display behavior as is found in other complex systems (Macready et al. 1996; Schoneveld et al. 1997; Yuan et al. 2000).

6.2 Self-Organized Criticality in Time Warp Dynamics

6.2.1 Slowly Driven, Interaction-Dominated Threshold Systems

Self-organized critical behavior is found in *slowly driven*, *interaction-dominated threshold* systems (SDIDT); if an SDIDT system exhibits power-laws without any apparent tuning then it is said to exhibit self-organized criticality (Jensen 1998). Interesting behavior arises because many degrees of freedom are interacting. In addition, the dynamics of the system must be dominated by the mutual interaction between these degrees of freedom, rather than by the intrinsic dynamics of the individual degrees of freedom.

An important characteristic of systems exhibiting SOC behavior is a *separation of time scales*. As stated before, it is required that such systems are slowly driven, that is perturbations occur on a much larger time scale than the diffusion or relaxation dynamics. The critical state in SOC systems is furthermore characterized by a stationary state where the driving forces balance the cascades. For example in the sandpile model, adding sand causes on the one hand the pile to grow, on the other hand avalanches. The dynamically stationary state is obtained at the "critical point" where these two effects exactly balance.

A highly speculative analogy could be made with Time Warp: adding events causes on the one hand the event rate to grow, on the other hand rollbacks to occur. Our experiments show that in Time Warp, the event rate eventually reaches a kind of stationary state with superimposed rollback cascade effects (see Fig. 6.2). We define two time scales in Time Warp: *simulation time* and *protocol time*. The simulation time in Time Warp is updated by the rate at which the system is driven. This driving rate is determined by the dynamics of the simulation. The protocol time, needed to process a rollback, is determined by machine specific parameters. A difference between conventional SOC systems and Time Warp is that there is no explicit separation of time scales: asynchronous updates and rollbacks may intervene. Also, rollbacks take place on separate processors instead of an entire lattice of sites.

The Ising spin model, presented in Section 5.3.1, is a critical system, that is for a certain parameter regime the system exhibits scale invariant structures. This makes the Ising spin system an ideal simulation model to study the influence of spatial correlation on the dynamical behavior of the Time Warp protocol. The many degrees of freedom that are interacting show emergent behavior, e.g., magnetization, as the temperature of the Ising spin system approaches T_c^+ (i.e., approach T_c form high temperature T). This emergent behavior is the re-

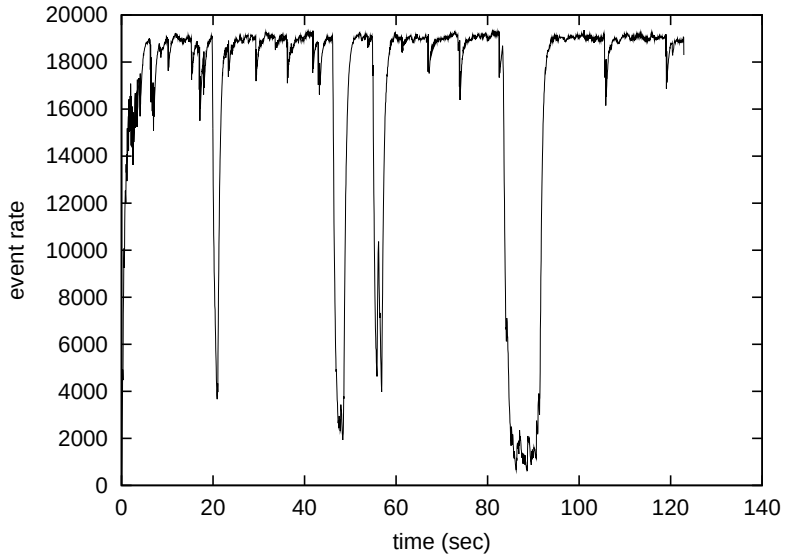


Figure 6.2: Event rate of one of the logical processes in a PDES Ising simulation, with number of processors $P = 6$ and Ising temperature $T = 1.0$.

sult of the correlation length in the system that diverges as the temperature approach T_c, and is even infinite at T_c.

To understand when "relaxation" in Time Warp occurs, we first have to explain how events are processed and can give rise to rollbacks in the Ising spin model. Events in the Ising spin model are *attempts* to flip a spin in the lattice. An attempted spin flip is accepted according to the Boltzmann probability distribution. Hence, not every event results in a spin flip, or equivalently in a state change. The parallel Ising spin simulation exploits the inherent parallelism by spatial domain decomposition. Each logical process in the parallel simulation represents a subdomain of the spin lattice. Events on the subdomain boundaries that result in a spin flip are communicated with the neighboring subdomain, i.e., logical process, by way of an event message.

The so-called relaxation in Time Warp occurs whenever the following three conditions are satisfied (threshold):

- if *accepted event* and;
- *boundary event* and;
- $\exists\, i \in \text{neighborhood}(local) : LVT_{local} < LVT_i$;

where LVT_{local} is the simulation time on the local processor and LVT_i is the simulation time on a neighboring processor i. For Ising spin simulations, simulation time and protocol time separate at low temperatures, when there are not that many spin flips, i.e., when the acceptance ratio of the Metropolis algorithm is low. For high temperatures many spin flips are accepted, and updates and rollbacks occur at comparable time scales.

6.2.2 Physical and Computational Critical Behavior

It is very important to note that, in fact, we are confronted with two kinds of critical behavior. The critical behavior of the first kind is a result of the Ising spin phase transition at the critical temperature T_c. At the Ising spin phase transition, long-range spin correlations occur, that might influence the Time Warp dynamics. We call this critical behavior of the first kind the *physical critical behavior*. The tendency to be correlated can be measured using the *correlation function*

$$C(r) = \langle s_0 s_r \rangle \ ,$$

where s_r is a spin that is located r lattice sites away from s_0. Some results for the correlation function are shown in Fig. 6.3, which shows $C(r)$ at several different temperatures. The important feature in the figure is not the average value of $C(r)$, but rather the amount that $C(r)$ increases above this average value as r becomes small. At low temperatures, $C(r)$ increases slightly at short distances, however the enhancement is very small. The correlation function at $T = T_c \approx 2.27$ differs from the low temperature behavior in two ways. First, the relative alignment at short distances is much larger than the value at large r. Second, the correlations are now long-range as $C(r)$ approaches the $r \to \infty$ limit very slowly as r is increased. As the temperature is increased further to temperatures above T_c, the correlations become smaller in magnitude and again extend over only a few lattice spacings.

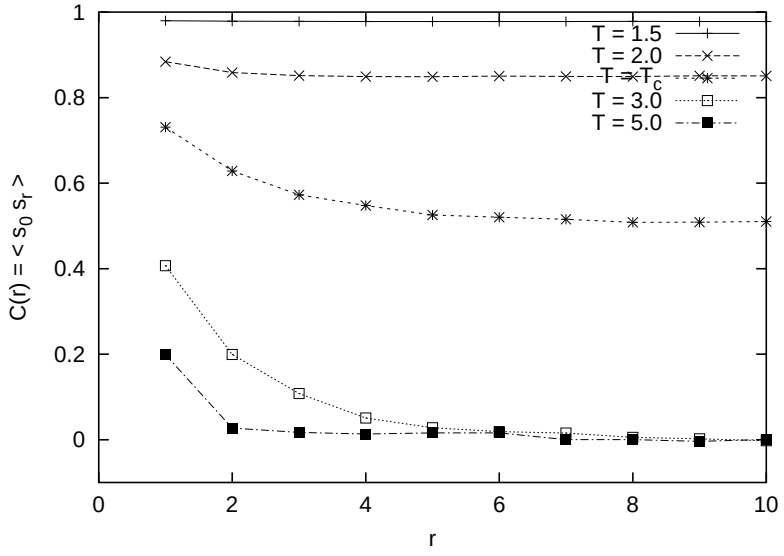


Figure 6.3: Ising spin correlation functions at several temperatures.

The critical behavior of the second kind is inferred from the dynamical behavior of the Time Warp protocol. We expect that in the low temperature Ising regime the Time Warp dynamics reaches a self-organized critical state, which we call *computational critical behavior*.

The average rollback length and rollback length distribution is studied at different temperatures in order to determine the influence of the Ising spin phase transition on the Time Warp protocol. It is expected that around the Ising spin phase transition, the long-range spin correlations increase the average rollback sizes. It is well known that these long-range correlations result in moving islands of actively flipping spins, located in a sea of inactive spins. This separation of activity can trigger very large rollbacks whenever an active island moves over a processor boundary.

We are interested in rollback length distributions in order to do a first order check of SOC in Time Warp dynamics. Remember that for SOC systems it is well known that many observables scale as power-laws.

6.3 A First Indication of Self-Organized Criticality in Time Warp

A series of experiments are executed using the APSIS parallel simulation environment (Chapter 3) on the Distributed ASCI Supercomputer (DAS), see Section 5.5 for a description of the DAS supercomputer. The Ising spin simulation model used in the experiments is described in Section 5.4. We experiment with two different grid decompositions for the parallel simulation of the Ising dynamics on a $L \times L$ square lattice: a one-dimensional "slice" decomposition and a two-dimensional "box" decomposition. Both decompositions are constructed to assure optimal load balance. For all parameter instances of the simulation experiments we measure the average rollback length and rollback length distributions.

For the first series of experiments in this section we have fixed the lattice size to $L = 220$, the number of processors to $P = 12$, and the virtual time window (VTW) to 3000, using a "sliced" 1D decomposition. In Fig. 6.4 the average rollback lengths are shown for the temperature range [0.1–2.7]. Ideally, rollback avalanches are measured instantaneously over the system, that is over all processors used by the parallel simulation. However, instantaneous measurement requires freezing the forward simulation, and allowing only rollbacks to occur. This would fundamentally alter the dynamics of the Time Warp protocol, and is therefore unacceptably intrusive. Hence, each processor records the local rollbacks for analysis. The rollback lengths are averaged over time for all processors. The results of three different runs are depicted in the figure. Close to the Ising phase transition ($T_c \approx 2.27$ for infinite lattices), the expected peak in the average rollback length can be observed. From this figure, three different regimes can be identified: the *physical sub-critical phase* ($< T_c$), the *physical critical phase* ($\approx T_c$) and the *physical super-critical phase* ($> T_c$).

The different phases influence the rollback length distributions. In the physical sub-critical temperature regime [0.1–1.4], power-law scaling is found (see Fig. 6.5), i.e., the Time Warp dynamics appear to be in a *computational critical regime*. As the temperature approaches the physical super-critical regime

a transition to exponential scaling can be observed (see Fig. 6.6). Close to the critical temperature, length distributions with "fat tails" (power-law distributions with exponential cutoff) develop due to the emergence of long-range spin correlations.

The scaling exponent α in the physical sub-critical regime seems to be uni-

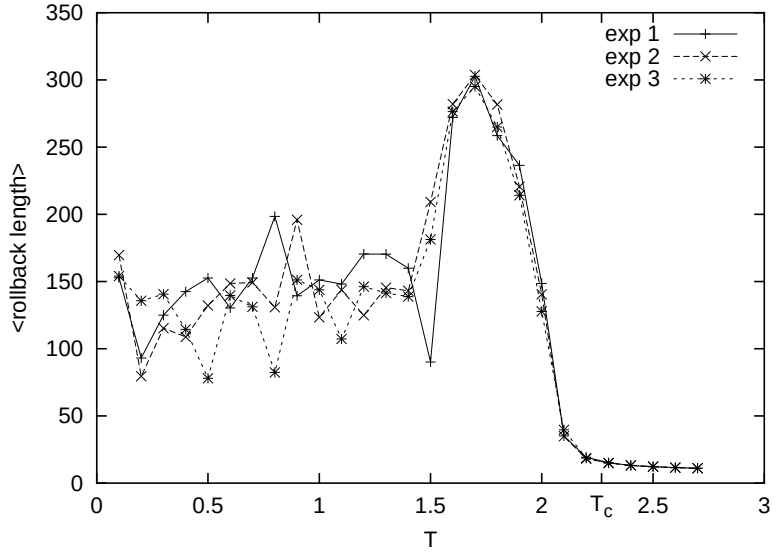


Figure 6.4: Average rollback length for different temperatures. For each temperature, the results of 3 experiments are shown. Using the simulation parameters $L = 220$, $P = 12$, and $VTW = 3000$, and a 1D decomposition.

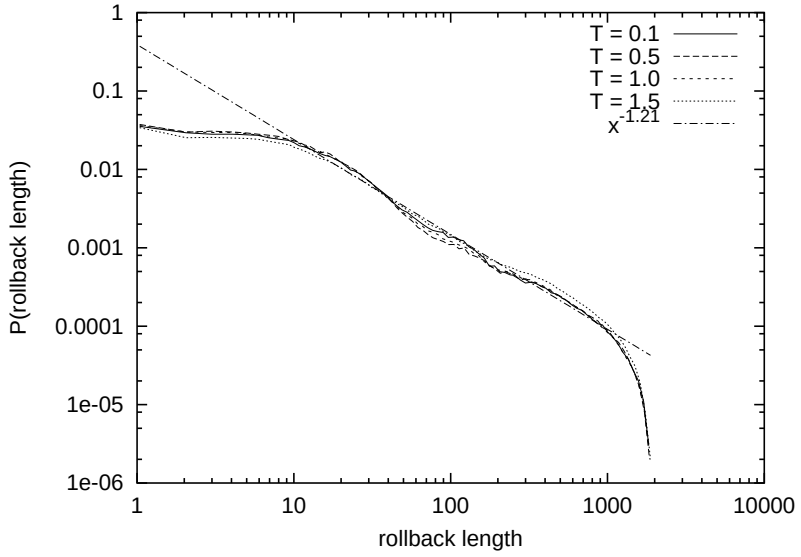


Figure 6.5: Rollback distribution for temperatures in the range 0.1–1.5, fitted exponent has value -1.21 (± 0.01). Using the parameters $L = 220$, $P = 12$, and $VTW = 3000$, and a 1D decomposition.

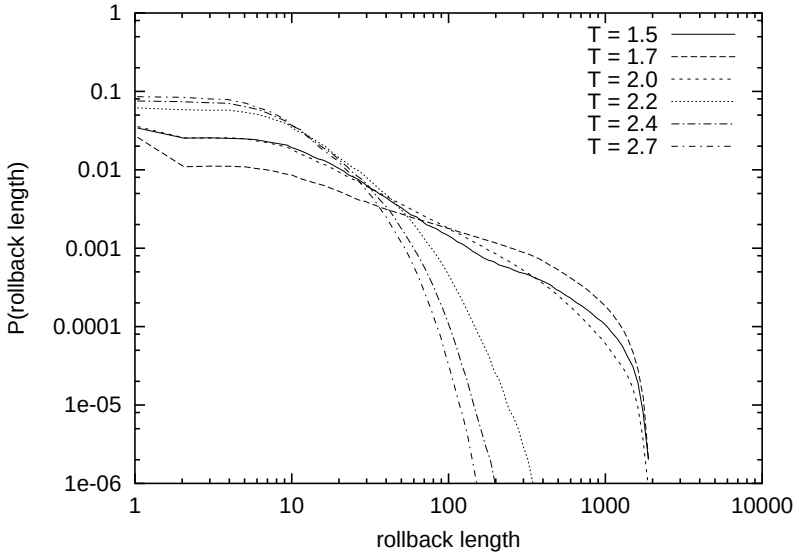


Figure 6.6: Rollback distribution for temperatures in the range 1.5–2.7. Using the parameters $L = 220$, $P = 12$, and $VTW = 3000$, and a 1D decomposition.

versal for all temperatures in this regime. From the experimental data a power-law with exponent $\alpha = -1.21$ is fitted (linear fit of the logarithmic values). Because the rollback length distribution obeys power-law scaling, we conjecture that the rollback dynamics are in a SOC regime.

The fact that the distributions in Fig. 6.5 begin to deviate from a power-law

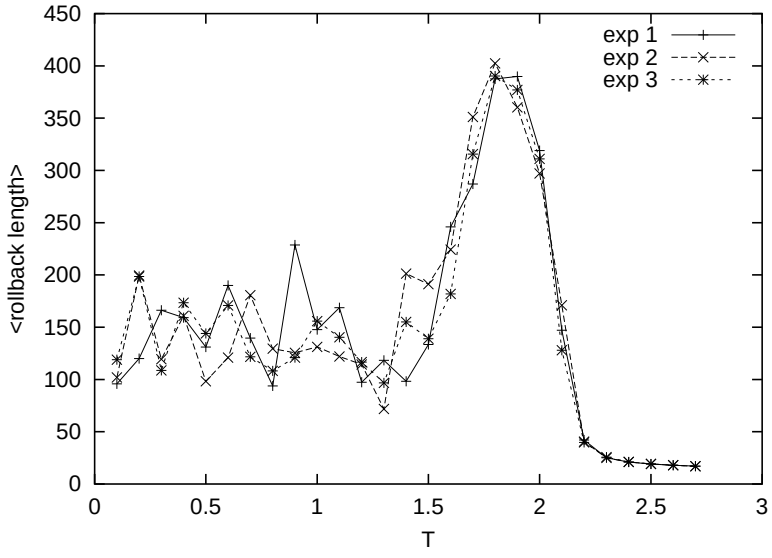


Figure 6.7: Average rollback length for different temperatures. For each temperature, the results of 3 experiments are shown. Using the parameters $L = 220$, $P = 12$, and $VTW = 3000$, and a 2D decomposition.

at large rollback lengths is a finite size effect, which is further investigated in Section 6.4. At small rollback lengths, the curve deviates from a straight line because discreteness effects of the rollback lengths come into play. As the rollback cascades approach the size of what is assumed to be the size of a real-world system's component particle, in our study an event, it becomes impossible for the fractal pattern to repeat at this scale (Bak et al. 1988; Brunk 2000; Frette et al. 1996).

For the 2D decomposition we repeat the same set of experiments as for the 1D case, again with $L = 220$ and $P = 12$. As for the 1D case, we observe a peak in the average rollback lengths around T_c (see Fig. 6.7). Again a transition from power-law scaling to exponential scaling is observed (see Figs. 6.8 and 6.9). In the physical sub-critical regime we find $\alpha = -1.25$, slightly larger than in the 1D case. This tendency is in accordance with the expectation that slightly shorter distances between processor partitions enable a faster propagation of cascading rollbacks.

In the next series of experiments, different parameters are varied. Note that two different processes intervene: the Ising simulation process and the Time Warp process. An important parameter for both processes is the lattice size. Due to finite size effects, increasing the lattice sizes causes the Ising spin phase transition point T_c to shift. For the Time Warp process, the probability to select a boundary cell decreases for increasing lattice sizes. If the number of processors is increased and the lattice size is kept fixed, the probability to select a boundary cell increases. Therefore we experiment with different numbers of processors. The virtual time window is a very important parameter that determines the maximum length of the rollbacks. This parameter is studied in

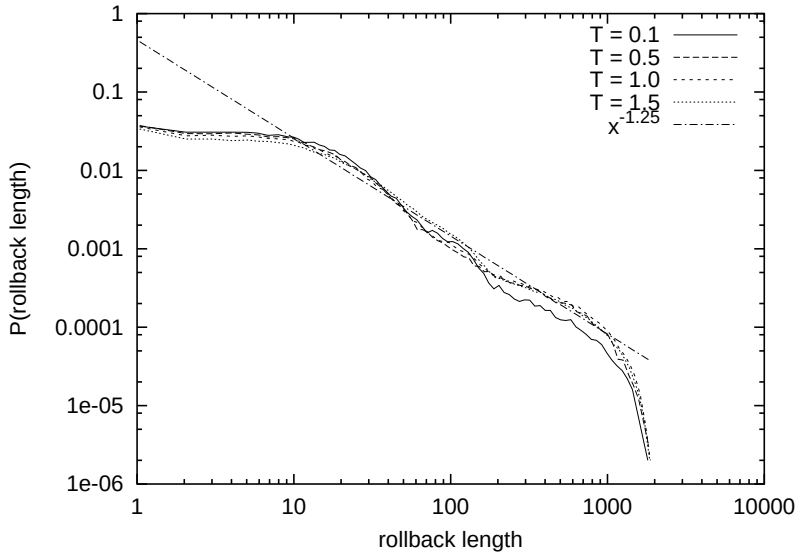


Figure 6.8: Rollback distribution for temperatures in the range 0.1–1.5, fitted exponent has value -1.25 (± 0.02). Using the parameters $L = 220$, $P = 12$, and $VTW = 3000$, and a 2D decomposition.

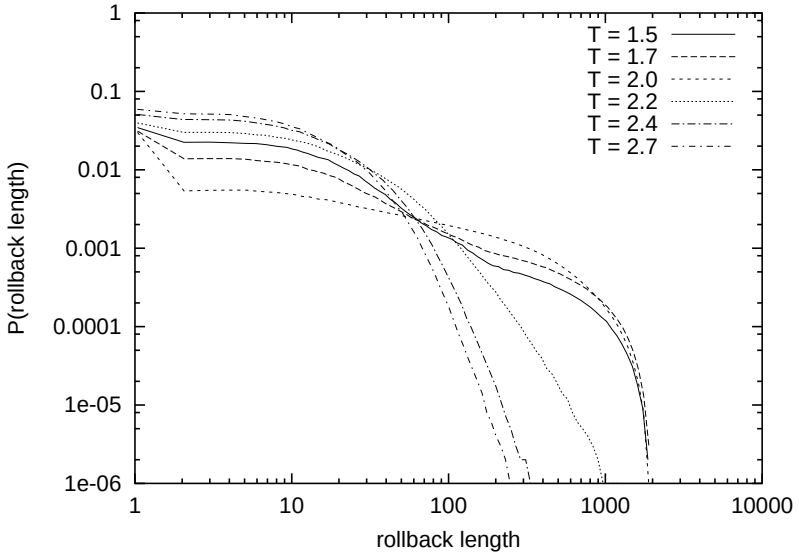


Figure 6.9: Rollback distribution for temperatures in the range 1.6–2.7. Using the parameters $L = 220$, $P = 12$, and $VTW = 3000$, and a 2D decomposition.

the last series of experiments.

6.4 Finite-Size Scaling Effects

6.4.1 Influence of lattice size

Because the peak in the average rollback appears close to T_c, we expect that this peak is related to the long-range correlations of the Ising phase transition. To support this hypothesis we have conducted a number of experiments with increasing lattice sizes in order to study the presence of finite size effects. For finite lattices, the critical temperature T_c shifts with increasing lattice sizes.

Due to limited computer and time resources we did not extract any critical exponents from the generated data; a more detailed study of this phenomenon is therefore necessary. To simulate an Ising spin system on one specific $T \approx T_c$, more than 2 days (see Fig. 6.11) of computing time on 12 Pentium II (at 200 MHz) nodes is needed; this gives an indication on the total amount of computer time required to run these experiments. Figure 6.10 shows the results for varying lattice size experiments (using $L = \{110, 220, 440, 880\}$ and $P = 12$). A shift towards T_c is observed for increasing lattice sizes.

Furthermore we find that for all lattice sizes in the SOC regime $\alpha \approx -1.21$ for 1D decompositions and $\alpha \approx -1.25$ for 2D decompositions. For 2D decomposition the rollback distributions are shown for $T = 1.0$ in Fig. 6.12.

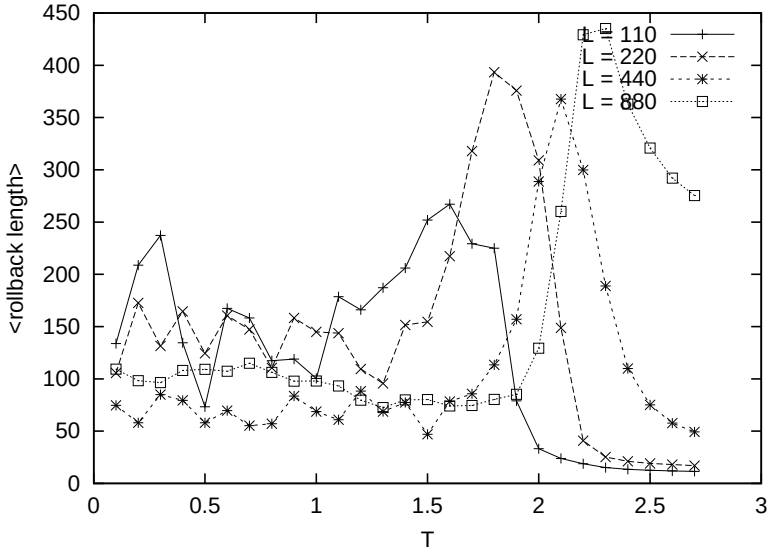


Figure 6.10: Average rollback lengths for $L = \{110, 220, 440, 880\}$ for varying T using the parameters $P = 12$, $VTW = 3000$, and a 2D decomposition.

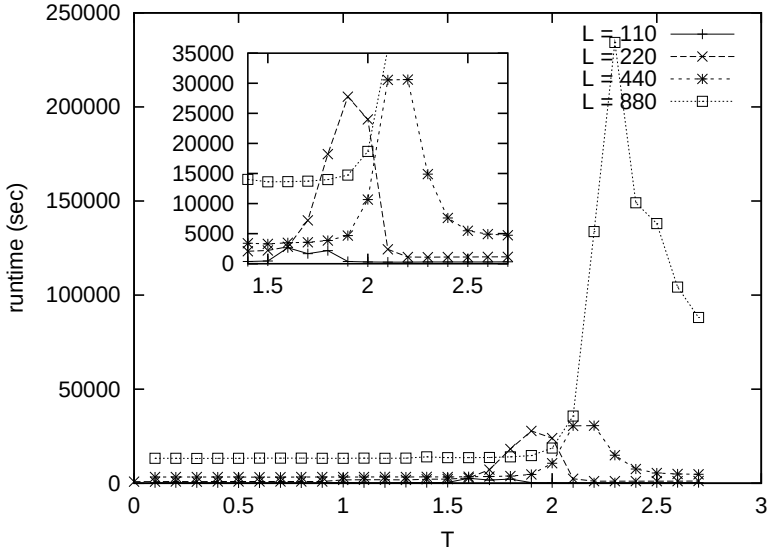


Figure 6.11: Run times for $L = \{110, 220, 440, 880\}$ for varying T using the parameters $P = 12$, $VTW = 3000$, and a 2D decomposition.

6.4.2 Varying the Number of Processors

To study the influence of the number of processors on the rollback length distribution in the SOC or computational critical regime, a series of experiments with $P = \{4, 8, 12, 24\}$ using a 2D decomposition has been performed. The lattice size has been fixed to $L = 220$. The rollback distributions of Ising spin simulations at $T = 1.0$ are shown in Fig. 6.13. Similar results are seen for

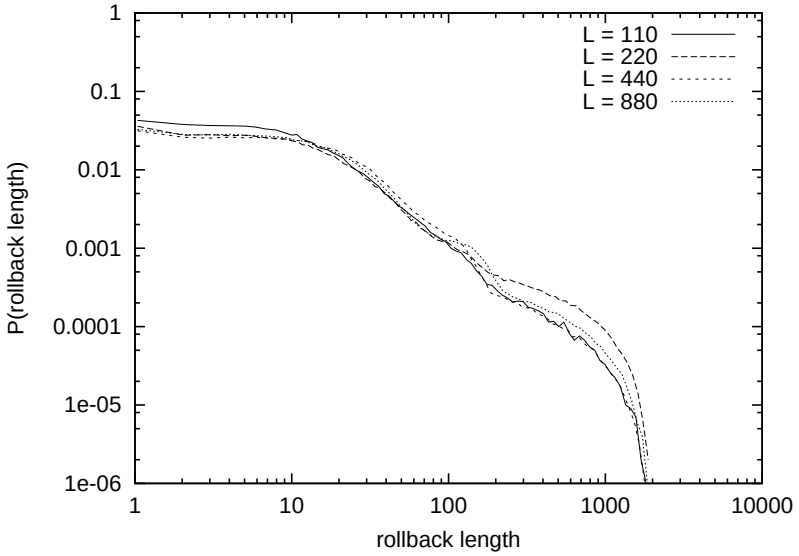


Figure 6.12: Rollback distributions for $L = \{110, 220, 440, 880\}$ at $T = 1.0$ using the parameters $P = 12$, $VTW = 3000$, and a 2D decomposition.

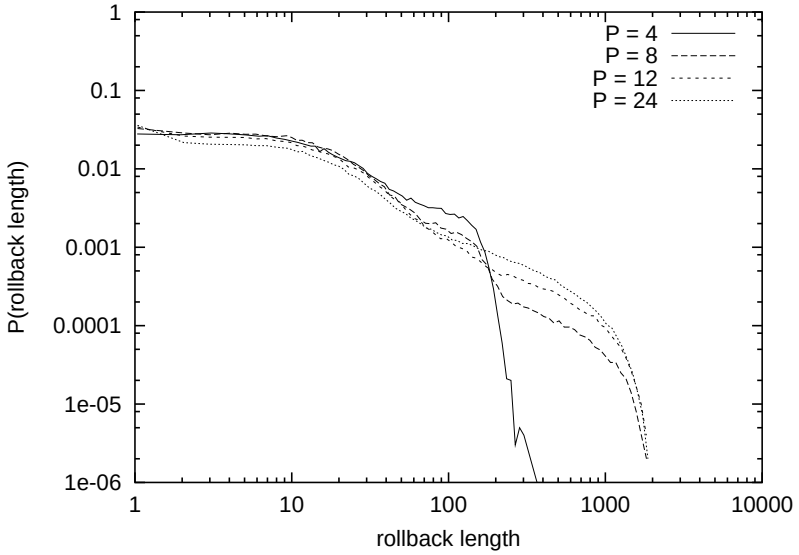


Figure 6.13: Rollback distributions for $P = \{4, 8, 12, 24\}$ at $T = 1.0$ using the parameters $L = 220$, $VTW = 3000$, and a 2D decomposition.

other temperatures T in the SOC regime. The results indicate that for increasing P the rollback length distributions converge. Again, the scaling exponent α is not influenced by increasing P.

The average rollback lengths for different processors in the range $T = [0.1 - 2.7]$ are shown in Fig. 6.15. For 8, 12 and 24 processors, again, a peak around T_c can be distinguished. For $P = 4$, this peak is not present. Apparently, the critical Ising spin dynamics does not reduce the performance of the Time Warp

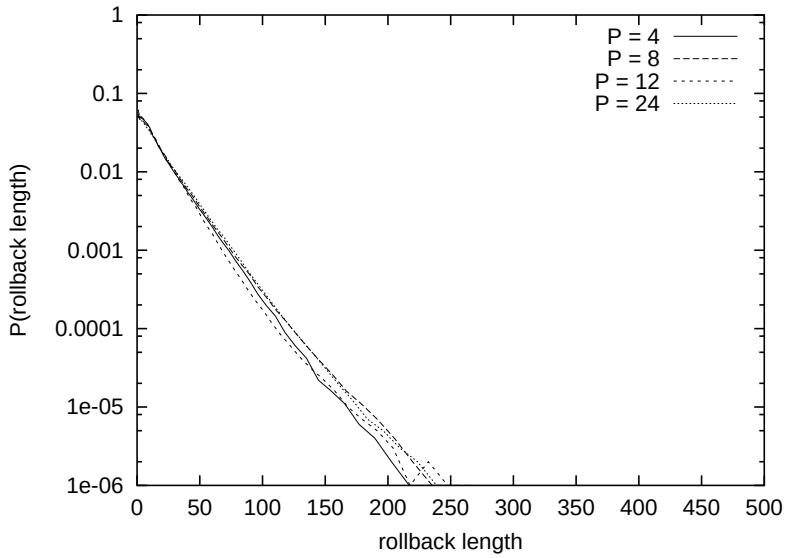


Figure 6.14: Rollback distributions for $P = \{4, 8, 12, 24\}$ at $T = 2.7$ using the parameters $L = 220$, $VTW = 3000$, and a 2D decomposition. Note that the horizontal axis has a linear scale.

protocol if only 4 processors are used.

In the low temperature regime the average rollback length increases with the number of processors. From Fig. 6.13 we observe that the rollback length cutoff size increases with P. Therefore it is expected that the average rollback lengths must increase with P in the low T regime (see Fig. 6.15).

This is not valid anymore in the high T regime, where the rollback length distributions approximately collapse (see Fig. 6.14) to the same exponentially decreasing distribution. As a direct consequence, the average rollback lengths will collapse (see Fig. 6.15). In the high T regime the rollback lengths are not influenced by P as in the low T regime. The processors are synchronized frequently in this regime, due to a high acceptance ratio of flipped spins. Therefore, there is hardly any real time to build large simulation time differences between the processors, resulting in only small rollback lengths.

It is interesting to compare the runtimes for different P in the same temperature range. The results are presented in Fig. 6.16. From the figure one can derive that in the low T regime, the runtime scales down if more processors are used. This is also valid for the high temperature regime. Around T_c, we find *non-trivial* scaling of the parallel runtime for different number of processors. Obviously, using only 4 processors gives the best result, which could be expected from the significantly lower average rollback length in this regime. Using 12 processors gives the worst results in this case.

For the high and low temperature regimes $P = 24$ gives the best performance results. Even though, in the low T regime, the average rollback length is maximal for $P = 24$, the extra overhead is beneficially applied to efficiently exploit the parallelism present in the simulation.

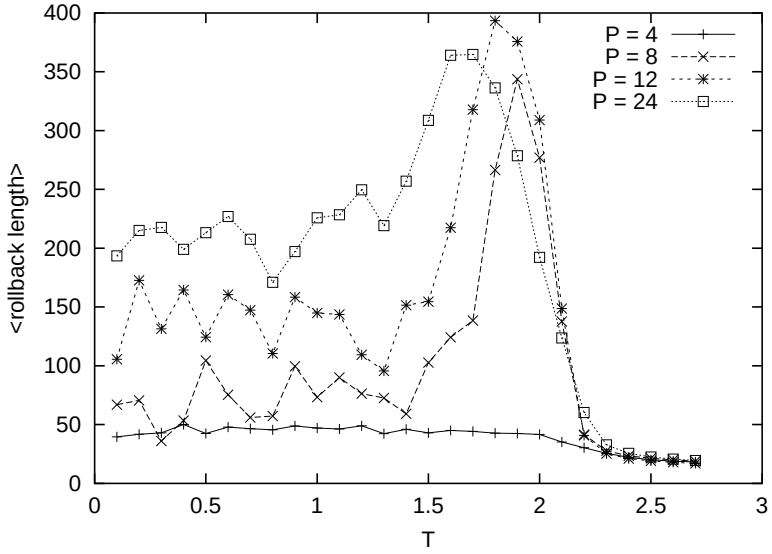


Figure 6.15: Average rollback lengths for $P = \{4, 8, 12, 24\}$ for varying T using the parameters $L = 220$, $VTW = 3000$, and a 2D decomposition.

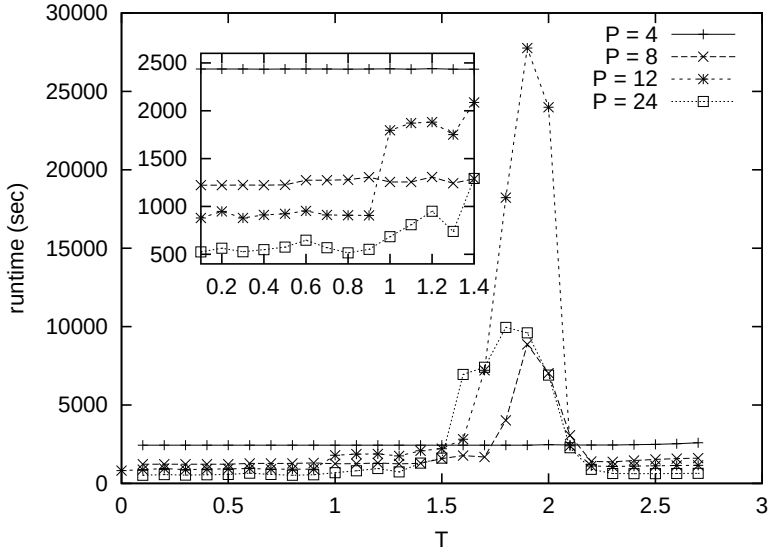


Figure 6.16: Run times for $P = \{4, 8, 12, 24\}$ for varying T using the parameters $L = 220$, $VTW = 3000$, and a 2D decomposition.

Although the average rollback lengths in the low T regime are much larger than the average rollback lengths in the high T regime, the execution times are comparable. This is a result of the frequency of rollback events. In the low T regime this frequency is much lower, due to the reduced acceptance probability of spin flips. It seems that, the average rollback lengths and the rollback frequency are balanced to approximately similar execution times for the low and high T regimes.

6.4.3 Different Virtual Time Window Sizes

An important parameter of the Time Warp protocol is the so-called *virtual time window* (VTW). This parameter controls the asynchronicity of the simulation. It specifies the maximum difference between the local virtual time and the global virtual time (the minimum of all local virtual times). It is expected that this parameter greatly influences the rollback dynamics. For the experiments presented in this section we have varied the VTW parameter while keeping all other parameters fixed ($P = 12$ and $L = 220$).

In Fig. 6.17 the rollback distributions are depicted for $T = 1.0$ for experiments with VTW parameters in the range $[750, 6000]$. Obviously, a small virtual time window decreases the maximum rollback length.

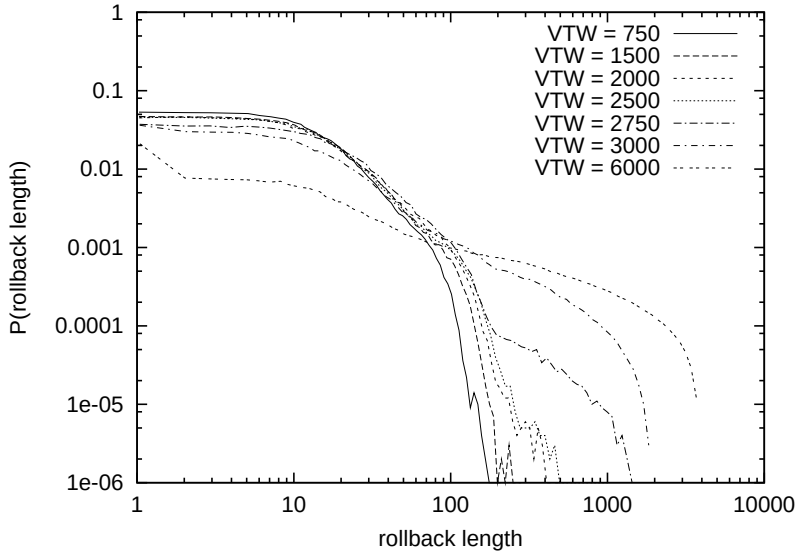


Figure 6.17: Rollback distributions for $VTW = \{750, 1500, 2000, 2250, 2500, 2750, 3000, 6000\}$ at $T = 1.0$ using the parameters $L = 220$, $P = 12$, and a 1D decomposition.

In Fig. 6.18 the rollback distributions for VTWs around 3000 are shown. There is a transition from $VTW = 2750$ to $VTW = 3000$. The VTW values $\{3000, 3250, 3500\}$ produce similar rollback length distributions, while $VTW = 4000$ deviates.

From Fig. 6.19 it can be observed that the peak of the average rollback length shifts and broadens for increasing VTW. This effect is caused by the Ising dynamics. Large virtual time windows effectively result in a more pronounced influence of the finite sub-lattices (decomposed over the 12 processors). Due to the increased asynchronicity for larger time windows the sub-lattices are effectively loosely coupled and act more like individual Ising spin lattices. It is a well-known fact in Ising spin simulations that decreasing the lattice size results in a broadening and shifting of the spin correlation peak around T_c.

The average rollback lengths roughly decrease for decreasing virtual time

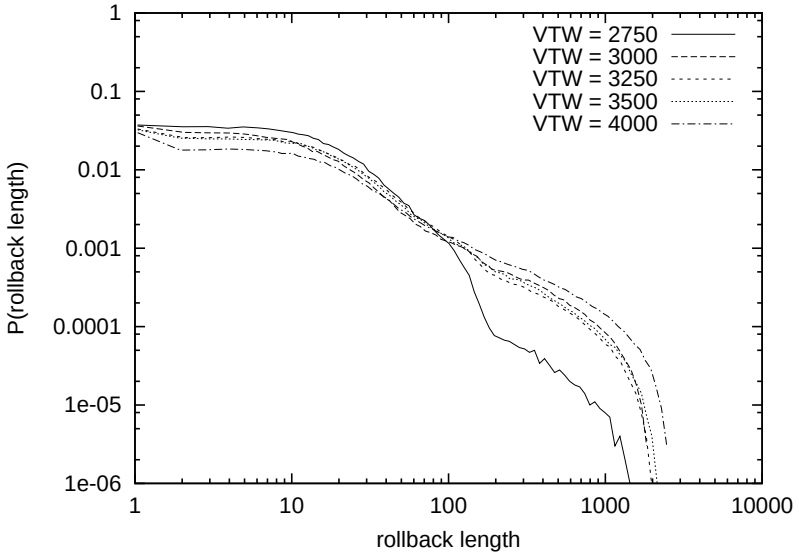


Figure 6.18: Rollback distributions for $VTW = \{2750, 3000, 3250, 3500, 4000\}$ at $T = 1.0$ using the parameters $L = 220$, $P = 12$, and a 1D decomposition.

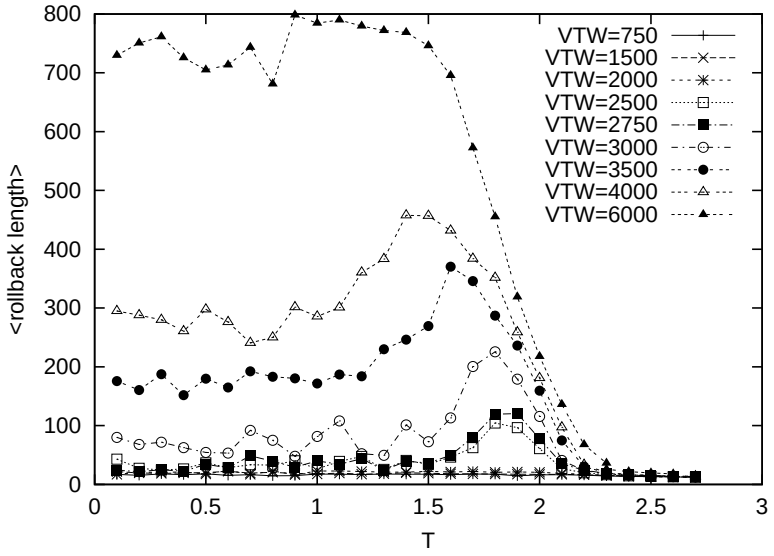


Figure 6.19: Average rollback length for $VTW = \{750, 1500, 2000, 2250, 2500, 2750, 3000, 3500, 4000, 6000\}$ for varying T using the parameters $L = 220$, $P = 12$, and a 1D decomposition.

window size (see Fig. 6.19). This is a result of the Time Warp dynamics. Small virtual time windows only allow for a small build up of local virtual time differences.

For smaller VTW values the average rollback lengths are comparable over the entire temperature range. For these values it can be concluded that the Time Warp dynamics are not constrained by the details of the Ising dynamics.

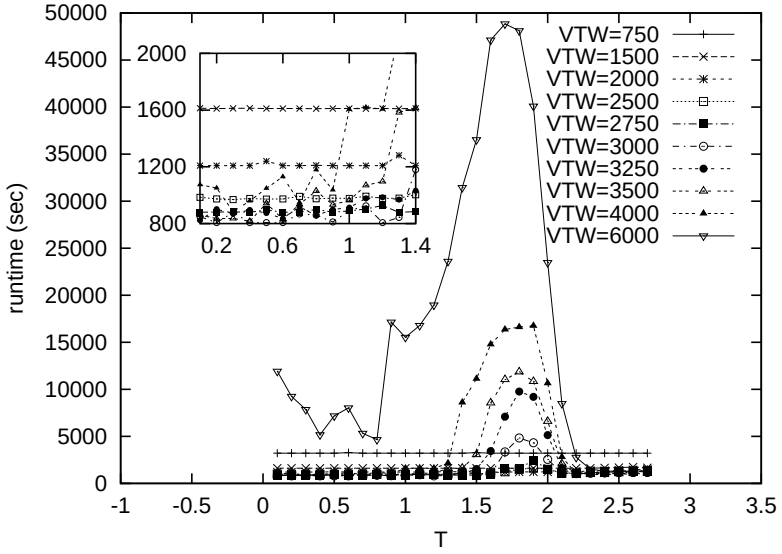


Figure 6.20: Runtimes for $VTW = \{750, 1500, 2000, 2250, 2500, 2750, 3000, 3250, 3500, 4000, 6000\}$ for varying T using the parameters $L = 220$, $P = 12$, and a 1D decomposition.

The increased synchronization frequency disables the build up of large time differences.

Somewhere there is a crossover point where increasing the maximum rollback lengths (by increasing VTW) does not improve the progress in simulation time due to the increased protocol overhead. For this specific simulation instance it seems that $VTW = 2750$ is optimal for the low and high temperature regime (see Fig. 6.20). For the regime around T_c it is almost optimal. Note that $VTW = 3000$ is comparable to $VTW = 2750$ in the low and high T regimes, while around T_c, $VTW = 3000$ produces significantly higher execution times. Hence the run times are highly susceptible for VTW around T_c, as a consequence of the critical Ising dynamics.

In contrast to the disappearance of the average rollback length peak in Fig. 6.19 for increasing VTW, a peak remains in the runtime curves. This can be explained from reduced rollback frequencies in lower temperature regimes. The increase of the runtimes around T_c with increasing VTW can be explained from the increased average rollback lengths (see Fig. 6.19) and the fat tail in the rollback size distribution around T_c for large virtual time windows (data not shown).

6.5 Summary and Discussion

In this chapter we have intensively studied the dynamical behavior of the Time Warp protocol for parallel discrete event simulations. As a simulation case we considered a model that supports tuning of the correlation length, namely the

Ising spin model, which is basically a cellular automata model. A property of the Time Warp protocol is the appearance of so-called rollbacks whenever a causality error occurs. This rollback mechanism can trigger a cascade of events that need to be undone. The local rollbacks are recorded by the simulation process, as the instantaneous global cascaded rollbacks cannot be administered without unacceptable intrusion into the Time Warp dynamics. It is known that so called slowly-driven, interaction-dominated threshold (SDIDT) systems can exhibit power laws without any apparent tuning. The specific feature of these dynamical systems is called self-organized criticality (SOC).

From the inset in Fig. 6.16 we can see that the optimistic simulation of the Ising spin system scales linearly with the number of processors for low temperatures $T = [0 - 1.5]$. However, it is found that the Ising spin phase transition influences the rollback behavior, and consequently the runtime. Around the critical temperature (physical critical behavior) T_c, the average rollback lengths increase dramatically, as well as the simulation runtimes, due to long-range spin correlations. The non-trivial scaling in runtimes around the critical temperature shows in Fig. 6.16 that the best performance is obtained with only 4 processors. For physical sub- and super-critical temperatures the simulation runtimes approximately coincide.

For the rollback dynamics three different phases can be distinguished: *physical sub-critical*, *physical critical*, and *physical super-critical* rollback length scaling behavior. In the sub-critical regime the scaling behavior appears to behave like a power-law, with exponents independent of the temperature. In this regime we conjecture that *computational critical* (SOC) behavior appears. Around the critical phase large rollback lengths become more abundant due the long-range spin correlations. Here the computational complexity and the physical complexity are entangled and contribute both to the runtime and rollback behavior in a non-linear way. In the physical super-critical phase a negative exponential distribution of the rollback lengths is observed.

Obviously a lot of work remains to be done in the study of physical- and computational critical behavior in Time Warp. The results presented in this chapter are, to our knowledge, the first series of experiments that have ever been conducted to study the influence and the appearance of critical behavior in Time Warp. The entanglement of the computational and physical complexity, and their non-trivial contribution to the runtime behavior might have consequences for other optimistic simulations.

Part II

Resource Management

Chapter 7

Dynamic Load Balancing:

Automatic Control of Execution Threads

> When one's ill or unhappy, one needs something outside oneself to hold one up. It is a good thing, I think, when one has been knocked out of one's balance, to have some external job or duty to hang on to.
>
> —Aldous Leonard Huxley

7.1 Introduction

The progressive use of event-driven simulation techniques as an essential approach to problem solving in, for example, science, engineering, and economics, has urged the need for robust performance. Efforts to parallelize the discrete event simulation execution mechanism resulted in two different parallel discrete event simulation (PDES) protocol classes: conservative and optimistic. In the preceding chapters of this thesis, we have extensively studied the performance and execution behavior of the optimistic scheduling protocol Time Warp. The performance of parallel programs, and in particular of optimistic simulations, is (negatively) influenced by the appearance of load imbalance over the processing nodes of the parallel or distributed computing platform. Due the very complex execution patterns in optimistic simulation, the load imbalance cannot be predicted before the execution of the parallel simulation. Hence, we must solve the load imbalance dynamically. To this end, we need an execution environment that allows for the dynamic migration of execution threads (processes, threads, or objects) from overloaded processing nodes to "underloaded" or less loaded processing nodes.

The Polder project is an experiment framework for wide-area resource management, which deals with both resource allocation and job placement, and dynamic resource management in local clusters. The main contribution of this chapter is the presentation of the design and implementation of, and experimentation with, dynamic resource management local to a cluster. The dynamic resource management environment is named Dynamite, and incorporates provisions for transparent process migration that allow for efficient load balancing

of parallel jobs over the processing nodes of the cluster.

The development of Dynamite is the first step towards dynamic load balancing of execution threads in PDES. Dynamic load balancing of parallel jobs is a complex task and a research topic that is extensively studied. Dynamic load balancing of PDES can be even more complex, as the notion of workload must be redefined, see for example thrashing behavior that incurs tremendous amounts of work but no progress (or useful work). Also, we have to consider which class of PDES applications can potentially benefit from dynamic load balancing over the processing nodes of a cluster. By the amount of event messages in PDES, the communication latency is a considerable factor in the determination of the performance. In this respect, *self-initiated* simulation applications, where the simulation processes schedule most of their events to themselves, seems to offer a good opportunity. The remainder of the introduction presents the general setting of wide-area resource management and dynamic load balancing local to a cluster. The challenges of dynamic load balancing of PDES are not considered in this chapter, but some ideas are discussed in Section 7.7.

The current developments in clusters of workstations, and on a larger scale wide-area distributed computing, or "grid" technologies (Foster and Kesselman 1998), indicate the importance of resource management to determine the efficacy of a distributed computing environment. In distributed environments the typical set of jobs consists of interactive and batch jobs, which in turn can be sequential or parallel execution runs. By the diversity of the jobs offered to the distributed environment—interactive users start sequential and parallel jobs, and batch jobs arrive with some arrival probability distribution function—both the demand for, and the availability of resources are highly dynamic.

Resource management in distributed environments spans a variety of activities such as job scheduling, I/O scheduling, load balancing, etc. In order to optimize performance of applications, or the utilization of resources, the resource management system should be able to react on changes in the distributed computing environment. As a consequence, several facilities have to be made available to the distributed computing environment in order to interact with resources and applications. The term "metacomputing" was introduced by Smarr and Catlett (1992), as a reference to such a set of widely different computing resources that presents itself to the user as a single computing environment.

A serious problem hampering the development of metacomputing environments is the lack of a sound theoretical basis for resource management strategies to build upon. In order to break the impasse, we developed an experimental environment that provides a framework for the development and evaluation of the various components making up the metacomputer (van Halderen et al. 1998). The experimental environment is essentially a metacomputer in its functionality and characteristics, but allows to study, for example, different policies for resource management or test designs and implementations of scalable I/O libraries, and the validation of theories.

In this chapter we focus on issues concerned with dynamic load balancing of parallel applications within a local cluster in the metacomputing environment. With the availability of high-speed networks, clusters of workstations achieve the same scalable parallelism as the current massively parallel processor (MPP) architectures. Hence, we currently witness a shift of emphasis in high-performance computing from expensive, special-purpose monolithic systems to the use of clusters of workstations or PCs. When using time-shared workstation clusters as high-performance computing servers, however, one has to cope with the dynamical behavior of the compute nodes, the network load and the application tasks. These can lead to local load imbalances, which hamper the application's execution and the overall system performance.

One way to deal with this dynamically changing resource requirement would be an adaptive system that supports the migration of processes from overloaded to under-loaded processors at runtime, without interference from the programmer. In addition, the resulting adaptive system should hide the complexity of the load balancing from the programmer/end-user. These observations resulted in the design and implementation of an experimental adaptive system called *Dynamite*.

The chapter is outlined as follows. Section 7.2 describes the current hardware and software trends in cluster and metacomputing. In Section 7.3 the Polder metacomputer framework is introduced. This section gives a global perspective of the research goals we are aiming for. The next sections, Section 7.4 and Section 7.5, present the main contribution of this chapter, namely the design and implementation of a local-area load balancing facility incorporated within a message passing library. The ideas and design of a local-area scheduler, i.e., process monitoring and migration decision, are presented briefly. The design and implementation of parallel process migration and restart are described in detail. A series of experiments and results are presented in Section 7.6. Finally, Section 7.7 discusses the results and observations of the Dynamite environment, and concludes with suggestions for future work.

7.2 Background and Design Aspects

The current developments in high performance cluster computing and metacomputing are moving along two axis: hardware and software. The hardware development of parallel supercomputing and modern networks/clusters of workstations are directing to the same point on the horizon. The compute nodes in the parallel supercomputer are the same processors found in workstations, and the performances of the distinguished proprietary interconnection networks are attained by independently available network interfaces such as Gigabit Ethernet, Fibre Channel, HIPPI, or Myrinet. Progress in wide-area networking, e.g., SONET and ATM, motivated the development of software infrastructures that smoothly integrate distant distributed resources into a metacomputer that enables the coordinated implementation of high performance applications.

7.2.1 Trends in Hardware

The development of high speed networks, both for local-area and wide-area networks, has triggered a refocus on the hardware used in high performance computing, and in particular a refocus on distributed memory architectures. For example, the massively parallel processors (MPPs) that are used to solve large computational problems, are distinct by their proprietary message passing networks, i.e., communication backplanes specifically designed for a family of MPPs. With the advent of fast network interfaces that are generally available, like (switched) Gigabit Ethernet, Fibre Channel, HIPPI*, and Myrinet, the same large computational problems can be solved effectively on clusters of workstations connected by a local-area network (LAN). In particular Myrinet is an outstanding example of how technology used for communication and switching in MPPs has evolved to a high speed LAN.

The availability of high speed LAN has initiated a number of research projects to build parallel supercomputers made of "commodity off the shelf" (COTS) components. Although the projects described below also cover software issues, their main focus is the implementation of a parallel supercomputer.

The Beowulf project (Warren et al. 1997) aims to develop a parallel computer architecture based upon Pentium Pro processors and switched Fast Ethernet communication links (i.e., switched Fast Ethernet is not used as a broadcast medium, but rather as a point-to-point interconnection fabric giving the full 100 Mbit/s bandwidth). In addition with the availability of powerful, free operating systems (Linux, FreeBSD) and message passing interfaces (MPI), the Beowulf project realized a low-cost commodity parallel computer. With a 16-node parallel computer a sustained performance of one Gflop/s has been obtained on scientific applications. A number of Beowulf offsprings have been build, among which the 140-node DEC Alpha cluster Avalon, the 276-node DEC Alpha cluster Jet (interconnected with Myrinet), and the 56-node (dual processor) Pentium II cluster SWARM. The Avalon cluster achieves 12.83 Gflop/s running a 64 million particle molecular dynamics simulation (SPaSM) on 70 nodes (Warren et al. 1998).

An interesting initiative that combines both high speed LAN and WAN interconnections in the implementation of a high performance computing platform is the Distributed ASCI Supercomputer (DAS)†. The DAS is a 200-node wide-area distributed system built out of four Myrinet-based Pentium Pro clusters. The four clusters are located at four universities: Free University Amsterdam, University of Amsterdam, Delft University of Technology, and University of Leiden.

Each node contains a Pentium Pro, 128 MB RAM, a 2.5 GB local disk, a Myrinet interface card, and a Fast Ethernet interface card. The nodes within a local cluster are connected by a Myrinet SAN network (SAN stands for system area network), which is used as a high speed interconnection, mapped in user-space. Fast Ethernet is used as the operating system network for NFS services,

*The new HIPPI-6400-PH ANSI standard provides 6.4 Gbit/s throughput.

†http://www.asci.tudelft.nl/das/das.shtml or http://www.cs.vu.nl/das/

etc. The four local clusters are connected by an ATM wide-area network, so the entire system can be used as a 200-node wide-area distributed cluster (see Fig. 7.1). The system runs the Linux operating system.

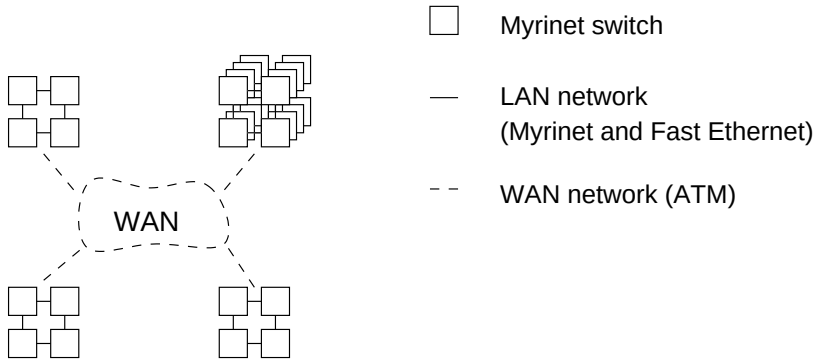


Figure 7.1: Overview of the DAS Architecture. Four local-area Myrinet clusters are connected by an ATM wide-area network.

The DAS distributed supercomputer, with its high speed local-area and wide-area interconnections, can be regarded as a prototypical metacomputer architecture for the near future. In this respect, the DAS architecture provides a unique experimental testbed for research in metacomputer software infrastructures.

7.2.2 Trends in Software

New technologies in wide-area networks have resulted in a new impetus to research directed to provide coordinated network services. The feasibility of wide-area high speed network technology (e.g., ATM, but also HIPPI and IP over SONET) has been demonstrated by the implementation of network testbeds including BERKOM, CASA, Abilene, and vBNS (Abilene and vBNS take part in the Internet2 consortium). The aggregation of distributed and high performance resources on high speed networks will change the perspective on distributed computing and have an impact on the development of scientific applications. In a similar way as parallel computing enabled scientists to solve computational problems that could not be obtained efficiently by sequential computing, aggregated distributed resources can engage larger computational power to a single application.

Although the hardware developments in high speed networks are impressive, the services provided to use the aggregated distributed resources in a coordinated manner are still in their infancy. To fully exploit the potential of distributed resources on coordinated networks, a software infrastructure must be developed to provide easy to use and transparent access to the resources. This software infrastructure, the *metacomputer* (Smarr and Catlett 1992), manages the complexity of the underlying physical system for the user. The key observation in metacomputing environments is that with the current conceptual

model, interacting autonomous hosts are stretched into a regime for which they were not designed. This has resulted in a collection of partial solutions without coherence and scalability. The challenge is to provide an integrated foundation that hides the underlying physical infrastructure from users and from the majority of programmers. By seamlessly integrating the diverse computational resources, the metacomputer provides a platform that fulfills the requirements of a new class of resource-intensive applications.

Two projects that are exemplary for the current trends in metacomputing research are Legion and Globus. A prototype of the Legion metacomputer and preliminary versions of Globus components have been demonstrated successfully as part of the I-WAY network experiment (DeFanti et al. 1996). Globus and Legion are currently used to provide the infrastructure for the National Technology Grid (Stevens et al. 1997; Foster and Kesselman 1998).

Legion is a metacomputer project designed to provide users with a transparent interface to the available resources, both at the programming interface level and at the user level (Grimshaw and Wulf 1996). Legion uses an object-oriented framework that enables a coherent solution to problems like access support, location, fault transparency, inter-operability, security, etc. The objects, written in either an object-oriented language or other languages such as C, will interact with other objects via well-defined interfaces. The use of objects allows for substantial flexibility in the semantics of user applications; a user is able to select both the kind and level of functionality, and make their own trade-offs between function and cost (e.g., the level of security in authentication).

The Globus (Foster and Kesselman 1997) project addresses the metacomputing challenge by a vertically integrated treatment of application, middleware, and network. In the Globus perspective, metacomputing can build on distributed and parallel software technologies, but also requires significant advances in mechanisms, techniques, and tools. The metacomputing software problem is approached from the bottom up, by developing basic mechanisms such as communication, authentication, network information and data access. These low-level components define a metacomputing abstract machine on which can be constructed a range of alternative infrastructures, higher-level services, and applications.

The long term goal of the Globus project is to construct an integrated set of higher-level services that enable applications to adapt to heterogeneous and dynamically changing metacomputer environments. The adaptive applications are able to configure themselves to fit the execution environment and optimize the performance.

Essential to the success of metacomputing is careful scheduling. Generally, there are two performance optimization objectives in wide-area systems: high performance computing (reducing turnaround time of jobs) and high throughput computing (e.g., maximize the aggregate amount of work per time period). Given one of these two goals, the scheduling process must decide where a job and its constituent tasks will run. The objectives and issues that must be addressed by a wide-area scheduling system are more complex than in local clus-

ter scheduling systems (Weissman and Grimshaw 1996; Chapin et al. 1999). For example, the wide-area scheduler should make use of the heterogeneity in the metacomputer by efficiently exploiting remote resources. However, in a metacomputing setting, resources are often managed by separate local schedulers (e.g., Condor, Codine, LSF) which are not coordinated. Consequently, the wide-area scheduler must make decisions in concert with the local site schedulers. The CCS system (Keller et al. 1999) is a typical example of a resource management system that was originally developed for high performance MPPs, and is adapted to modern workstation clusters. It provides allocation of exclusive and non-exclusive resources, scheduling of interactive and batch jobs, and has an open, extensible interface to other resource management systems.

The delicate interplay of the wide-area scheduler with the local site schedulers is one of the research interests in the Polder metacomputer project, which is presented in the next section.

7.3 The Polder Metacomputer Experimental Framework

The Polder metacomputer initiative (Overeinder and Sloot 1997; van Halderen et al. 1998) is an ambitions project that aims to provide an experimental framework for metacomputer design tradeoffs and gradually build a metacomputer environment that organizes heterogeneous distributed resources into one single computing environment with a uniform access. By its distributed nature, the resources are administered by local authorizing resource managers. Therefore, the Polder metacomputer must incorporate existing management software concerning resource control, access control, accounting and monitoring while supporting the multitude of hardware platforms present within the distributed system.

In the Polder metacomputer experiment different ways of use of metacomputing are addressed: high performance computing, high throughput computing, multi-site computing and automatic task balancing for dynamic resources. Each of these different usages of the metacomputing environment has its own requirements with respect to the services provided by the metacomputer. The underlying mechanisms should be flexible and generic in order to efficiently support these different requirements in services. To tackle these problems, a number of subprojects have been initiated to deal with issues like metacomputer access and job submission, wide-area and local scheduling, load balancing, and scalable I/O. These subprojects are performed by the different participants in the Polder initiative, among which the University of Amsterdam, NIKHEF (Amsterdam), Delft University of Technology, University of Wisconsin–Madison, and Paderborn Center for Parallel Computing.

Some of the issues concerning metacomputer access and job submission, wide-area scheduling, and local-area load balancing are discussed in the next section. Within the MOL partner project, the PLUS lightweight communica-

tion interface (Brune et al. 1997b) addresses inter-operability between heterogeneous platforms and different message passing layers. PLUS encapsulates message passing specific communication primitives (e.g., `MPI_Send`, `pvm_send`) and enables inter-operability between MPI and PVM applications.

7.3.1 Resource Management in the Polder Metacomputer

The efficiency of a metacomputing system can be viewed in two different ways. For high performance computing (HPC) a parallel job perspective is taken. The system performance is defined in terms of the turnaround time of highly demanding parallel jobs. In the view taken by high throughput computing (HTC), the performance of the system is mainly defined in terms of the number of jobs that are processed within a certain period of time.

The global resource management structure of the Polder metacomputer model is depicted in Fig. 7.2. The structure determines how the heterogeneous distributed resources are presented to the metacomputer user or application. On the base-level there are resources (e.g., workstations, MPPs, or I/O devices) administered by a local resource manager (e.g., Condor, Codine, or LSF). The aggregated local resources (that is, at the base-level the resources administered by the local resource manager) are represented by self-describing active agents. These agents (in Fig. 7.2 the entities in the shaded area) describe the type of resources, amount of memory, disk space, connectivity, etc.—the agent is essentially not limited in its descriptive plurality. The agents can be aggregated into a new agent, and hence represent a larger set of distributed resources. The aggregation of agents and the information advertised by the agents can reflect local authorization decisions. Although the organization of the agents is hierarchical, the perspective to resources is one-dimensional; that is, a unified view to the heterogeneous distributed resources on a coordinated network.

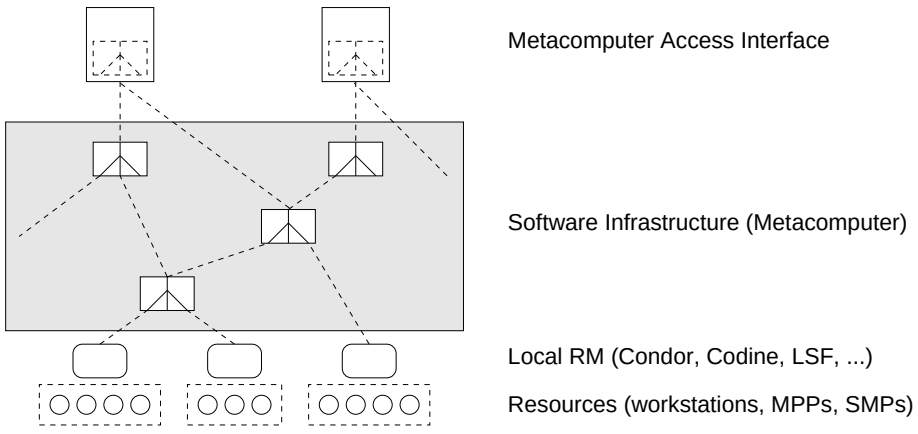


Figure 7.2: The Polder metacomputer global resource management model. The software infrastructure (active agents) organizes the distributed resources to a metacomputer.

The Polder metacomputer access interface is distributed and WWW-based to allow for a scalable, flexible and generic interface that interacts with the resource agents. The wide-area resource management actually takes place at the metacomputer access interface. Upon job submission via the access interface—with the job requirements being specified—the agents start with bidding on the job. In accordance with the wide-area scheduling policy, one of the agents offers the best fit on the job requirements. The job and its constituent tasks are allocated to the resources in coordination with the local resource manager. In this top-down approach, the wide-area scheduler determines the resources assigned to a job, and direct the local resource managers to actually allocate these resources.

Wide-area scheduling is a complex problem and subject of various research projects. Within the Polder metacomputer project a simulation model of the resource management infrastructure has been developed to allow for rapid prototyping and evaluation of scheduling strategies (Santoso et al. 2000). Experiments with scheduling strategies under strict conditions can be instrumented on top of the resource management simulation model, which is essential for validation with theoretical models. After a scheduling algorithm has been thoroughly evaluated, it can be integrated within the metacomputing environment.

The previous discussion did not mention the scheduler for dynamic load balancing in a local-area cluster. This subject is presented in Section 7.5.1, where the Dynamite local-area scheduler is introduced.

7.3.2 The Curse of Dynamics

In general the resources in the metacomputing environment are not exclusively allocated to one user or application, that is, resources are often shared among users and applications. Consequently, changes in the distributed system such as variation in demand of processor power, variation in number of available resources, or dynamic changes in the runtime behavior of the application, hamper the efficient use of the metacomputing environment.

Consider, for example, an application that after a straightforward domain decomposition, is mapped onto the processors of a parallel architecture. If the hardware system is homogeneous and allocated to only one application program, then the execution will run balanced until completion: we have mapped a static resource problem to a static resource system. However, if the underlying hardware system is a cluster of multi-user workstations we run into problems because the available processing capacity per node may change: in this case the static resource problem is mapped to a system with dynamic resources, resulting in a potentially unbalanced execution. Things can get even more complicated if we consider the execution of an application with a dynamic runtime behavior on a metacomputer environment, i.e., the mapping of a dynamic resource problem onto a dynamic resource machine. The notion of redundant decomposition has been posed by de Ronde et al. (1996) to introduce sufficient *richness* in parallel tasks to make a balanced workload in such a dynamic resource machine possible.

One way of dealing with this dynamically changing resource requirement would be to dynamically rebalance a job and its (parallel) constituents by migration of processes from overloaded to under-loaded resources at runtime. If the dynamic load balancing occurs locally, the wide-area scheduler does not participate. However, the local resource manager might request the wide-area scheduler that a job be re-scheduled elsewhere. The next section describes the design and implementation of these functionalities that are needed for dynamic load balancing, i.e., process migration of running (parallel) jobs.

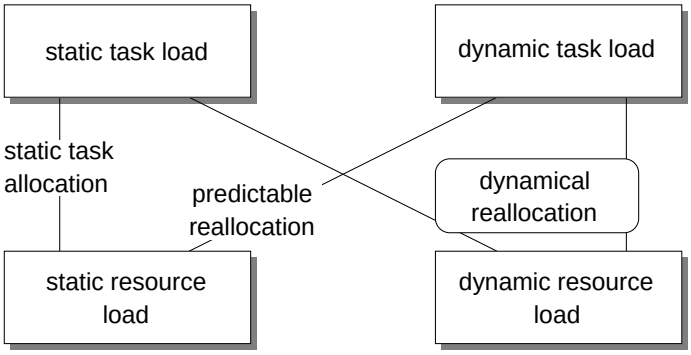


Figure 7.3: Task allocation of static and dynamic applications and resources.

7.4 Dynamite: Process Migration in Message Passing Environments

Process migration support can be incorporated at two operation levels: *operating system level* and *user level*. In operating system level implementations the resource management facilities are supported by the OS kernel. Examples of such systems are Mach (Milojicic et al. 1993), Sprite (Douglis and Ousterhout 1991), and MOSIX (Barak and La'aden 1998). User level designs and implementations of adaptive systems include dynamic resource management facilities by providing their own dynamic load balancing runtime support. Examples of user level designs are Condor (Litzkow et al. 1988) for sequential, and MPVM (Casas et al. 1995) for parallel application systems.

In our project we have the following design constraints for the process migration facility:

- since we assume that the major computational resource is a scalable cluster environment, the application programming model must be based on message passing;
- it is essential we support a platform independent operating system, therefore the operating system should be Unix;
- by hiding the complexity in libraries, the dynamic load balance runtime support system must be incorporated at user level.

Furthermore, the design of a self-contained experimental environment for dynamic load balancing of parallel application systems should include at least the following three components: (i) parallel programming environment, (ii) parallel runtime support system, and (iii) checkpointing/migration facility. The parallel programming environment enables the programmer to decompose the application problem into parallel subtasks. The parallel runtime support system allows for the parallel execution of the parallel application system; and the checkpoint/migration facility extends the runtime support system with functionality necessary for dynamic load balancing.

The first two facilities are provided by the PVM system (Sunderam et al. 1994). The PVM system includes an application programming interface for parallel program development and a runtime support system to allow for parallel execution of the application. The task checkpoint/migration functionality extension must be integrated with the PVM runtime support. The choice to use PVM as the basic parallel programming environment is motivated by the free availability of the source code and the extendibility of the runtime support. The application programming interface incorporates the dynamic addition and deletion of hosts (resources) and processes.

These design constraints have motivated the development of Dynamic PVM or DPVM for short (Dikken et al. 1994; Overeinder et al. 1996). The development of DPVM is now continued in the Dynamite project (van Albada et al. 1999; Iskra et al. 2000). Dynamite is an acronym for Dynamic Task Migration Environment, and currently supports PVM-based programs only. However, the principles of Dynamite should be easily portable to MPI (MPI Forum 1998). Both message passing environments are generally available on many different platforms and allow for the extension of process migration into their libraries. Although the checkpoint/migration design considerations are equal for PVM and MPI, there are some differences in the implementation. PVM (as basis for DPVM/Dynamite) is a message passing environment that also includes process creation and termination, and other resource management functionalities such as primitives for the allocation and deallocation of resources. The MPI-1.1 definition however, does not include any hooks for resource management functionalities required with process migration. This has to be included in the MPI runtime support system, but must be transparent to the application programmer. The MPI-2 definition includes process management, but no resource control as in PVM. MPI-2 assumes that resource control is provided externally—probably by computer vendors, in the case of tightly coupled systems, or by a third party software package when the environment is a cluster of workstations.

In the following discussion we briefly outline aspects of the PVM system and present the design issues to incorporate checkpoint/migration facilities in Dynamite.

7.4.1 The PVM System

The PVM (Parallel Virtual Machine) system presents an integrated environment for heterogeneous concurrent computing on a network of workstations. The computational model is process-based, that is, the unit of parallelism in PVM is an independent sequential thread of control, called a task. A collection of tasks constituting the parallel application, cooperate by explicitly sending and receiving messages to one another. The support for heterogeneity permits the exchange of any data type between machines having different data representations.

The PVM system consists of two parts: a daemon, called *pvmd*, and a library of PVM interface routines, the *pvmlib*. The PVM daemon and library enable a uniform view of the network of workstations, called hosts in PVM, as a parallel virtual machine.

Each host in the virtual machine is represented by a daemon that takes care of task creation and dynamic (re-)configuration of the parallel virtual machine. PVM tasks are assigned to the available hosts using a round-robin allocation scheme. Once a task is started, it runs on the assigned host until completion, i.e., the task is statically allocated.

The PVM library implements the application programming interface that includes primitives for process creation and termination, host addition and deletion, coordinating tasks, and message-passing primitives. The underlying communication model can be classified as asynchronous message-passing, where the messages are buffered at the receiving end. An important aspect of the communication model is that the message order from each sender to each receiver in the system is preserved. The PVM message-passing interface supplies both point-to-point communication primitives and global communication

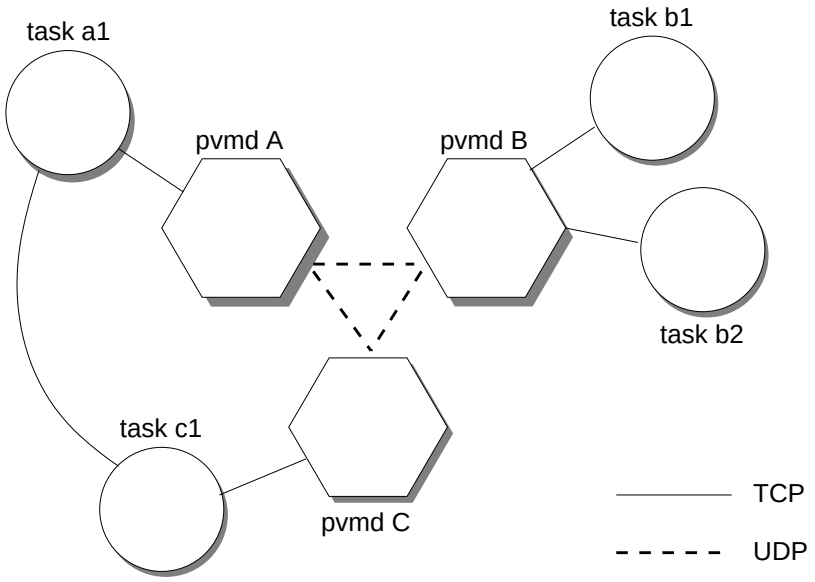


Figure 7.4: The PVM system composed of daemons and tasks.

primitives based on dynamic process groups. To enable the use of heterogeneous host pools, messages can be encoded using an external data representation (see XDR (Sun Microsystems, Inc. 1987)).

A relevant issue in the context of the forthcoming discussion, is message routing. PVM supports two routing mechanism for messages, namely *indirect* and *direct* routing. By default, the messages exchanged between tasks are indirectly routed via the PVM daemon. With indirect routing, a task sends the messages first to the local PVM daemon. The local daemon determines the host on which the destination task resides, and sends the message over the User Datagram Protocol (UDP) transport-layer to the responsible daemon. This daemon eventually delivers the message to the destination task. For example in Fig. 7.4, an indirect path from task a1 to b2 goes via pvmd A and pvmd B. Direct message routing allows a task to send messages to another task directly over a Transmission Control Protocol (TCP) link, without interference of the PVM daemons and thereby enhancing communication performance (see for example the TCP connection between tasks a1 and c1 in Fig. 7.4).

7.4.2 Design Aspects of Process Migration in Dynamite

Process *migration* (operating system level and user level) is realized by the movement of an active process from one machine to another in a parallel or distributed computing system. The process is suspended and detached from its environment, its state and data (the *checkpoint*) transfered to the destination host, where it is restarted and attached to the destination environment. The major requirement for providing a migration facility is *transparency*: the execution of a process should proceed as if the migration never took place. In parallel application systems like PVM applications, this transparency should hold also for the migrated process's communication partners. The application programs then do not have to take into account for possible complications of checkpointing and migration.

From the requirements defined above, it follows that Dynamite must incorporate a checkpoint/migration facility and location independent task identifiers, in order to support transparent process migration. The checkpoint/-migration functionality in Dynamite is based on the ideas of the facility provided by the Condor system. Dynamite extended the checkpoint protocol to safely checkpoint communicating parallel tasks without loss of messages. The location independent task identifiers, or virtual task identifiers, guarantee a unique name space for tasks independent of their location. Thus the same task can be addressed with the same task identifier after migration. Compare the virtual task identifiers with virtual memory addresses: the virtual memory address can be mapped to different physical addresses during the execution of a program.

7.5 Implementation Aspects of the Dynamite Environment

This section describes the extensions to PVM that are necessary to support dynamic load balancing within the runtime support system. In order to implement task migration, see Section 7.5.3, functionalities in the PVM daemon *pvmd* and library *pvmlib* need to be enhanced with checkpoint/migration mechanisms.

It is essential to note that the intertask communication, viz., message routing by the *pvmd*, is strongly affected by the added functionality of task migration. Therefore, we need to develop a methodology to guarantee the transparency and correctness of this intertask communication.

The extensions to the *pvmd* and *pvmlib* must not change the PVM programming interface and semantics, such that source code portability is guaranteed. The packet routing by the *pvmd* ensures migration transparency. With this approach, any standard PVM application can be linked and executed with the Dynamite system without a modification to the source code of this application, thus hiding the complexity for the end-user.

7.5.1 The Scheduler

Although the scheduler is not considered an integral part of Dynamite, its role and interface is mentioned here. In line with the top-down perspective of the wide-area and local site scheduler, the Dynamite scheduler resides beneath the local site scheduler. The local site resource manager is the authority that allocates the resources for the Dynamite cluster. The Dynamite scheduler acts as a resource manager within this Dynamite cluster, that is, it decides when to migrate a task and to which host it is moved. In addition, the Dynamite scheduler can request or relinquish resources in interaction with the the local resource manager.

In this scenario, the Dynamite scheduler largely determines the efficacy of the Dynamite system in its aim for load balancing. The development of good algorithms or heuristics for load balancing is a study in itself and is beyond the scope of this thesis. The current scheduler decides on (re-)allocation of processors for tasks, based on gathered load information of the workstation pool. The scheduler was developed in collaboration with the University of Paderborn within the Dynamite project (van Albada et al. 1999).

The scheduler consists of two functional components: *resource monitoring* and a *migration decider*. The resource monitors and migration decider are implemented as normal PVM tasks. This approach makes the incorporation of new scheduling strategies flexible and provides for a flexible experimental platform for studying the effectivity of the different load balancing disciplines. A consequence of implementing the scheduling components as PVM tasks, is that an additional interface must be provided to enable the migration decider to interact with the Dynamite system, in particular with the PVM daemons.

To this end, the *pvmlib* is extended with an interface routine, `pvm_move(tid, host)`, that initiates the migration of task `tid` to the specified host `host`.

Resource Monitoring

The monitoring subsystem keeps track of the load on the hosts and the communication between the tasks. In order to make migration decisions, the following information is administered:

- nominal capacity on each node (CPU, memory, disk space);
- current load on each node;
- required capacity for each task;
- network capacity for each task;
- communication pattern for each task.

Each of these items can be measured at execution time by the monitoring subsystem, but we assume that node capacity and network properties are sufficiently stable that they can best be specified beforehand by the system administrator (see Brune et al. (1997a) for further details).

Because of the assumed dynamic behavior of the application and the system load, the other items need to be obtained by the monitoring subsystem. Information about load and capacity must be collected from all nodes of the cluster, also those where currently no task of the parallel application is running. This is accomplished by running a small monitoring slave on each node (Fig 7.5).

The statistics obtained by the monitor slaves are sent to the monitor master process that is not only responsible for maintaining the whole cluster statistics, but also has to make migration decisions. The information on communication patterns is obtained directly from the DPVM environment. Therefore, DPVM has been enhanced by a message monitoring thread. This thread keeps track of each message sent and received. These communication statistics are also sent to the monitor master process.

Migration Decider

The migration decider is the main part of the scheduler thread that is executed periodically by the monitor master process. Based on the monitored data, the migration decider has to judge where and when to migrate a task from an overloaded node. Additionally, the task to be moved causes some constraints on the migration decision. Therefore, the master load monitor has to supply some normalized values about the attributes CPU, memory, and disk swap space of each node and additionally the available network capacity.

The increasing interest in distributed computing has lead to intensive scientific research in load balancing schemes for distributed memory systems, see for example Decker et al. (1998). Because not every load balancing scheme is

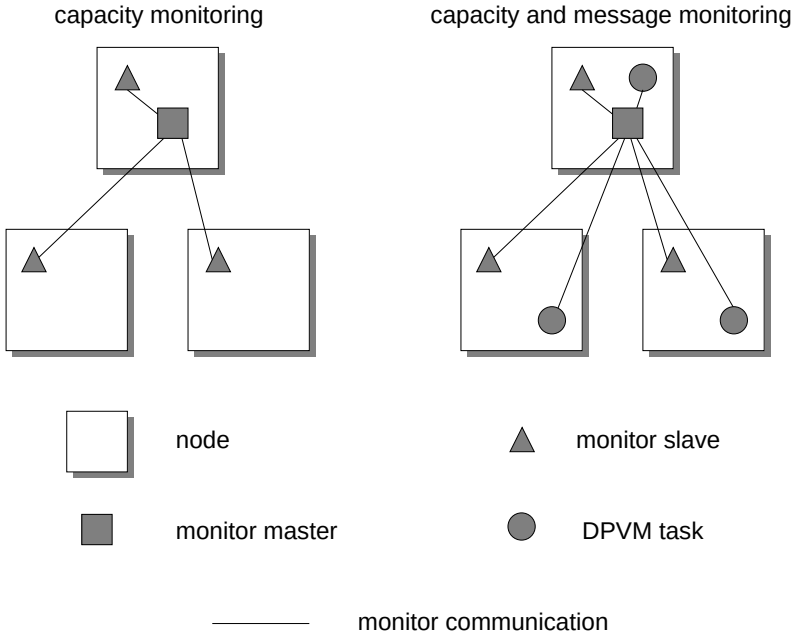


Figure 7.5: Capacity and message monitoring.

applicable to every application, the migration decider has been designed in a flexible manner to support a broad range of applications. For the first prototype we have implemented a straightforward solution with a greedy-like algorithm and a constraints list. The details of the algorithm are beyond the scope of the thesis, and can be found in van Albada et al. (1999).

7.5.2 Consistent Checkpointing Through Critical Sections

To implement dynamic load balancing by task migration, the runtime support system must be able to create an image of the running process, the so-called *checkpoint*. A checkpoint of an active process consists of the state and data of the process, together with some additional information to recreate the process. To incorporate file I/O migration, the state vector also includes information about open files together with their modes, file descriptors, etc.

A complication with checkpointing communicating PVM tasks, is that the state of the process also includes the communication status of the socket connections. Thus, to save the state of the process, the interprocess communication must also be in a well-defined state. Since suspension of the related communicating task is not desirable, the task should not be communicating with another task at the moment a checkpoint is created. To prohibit the creation of process checkpoints during communication, we apply the notion of *critical sections* and embed all interprocess communication operations in such sections. Checkpointing can only take place outside a critical section. When a checkpoint signal arrives during the execution of a critical section, the checkpointing is deferred.

The checkpointing functionality is implemented in the dynamic loader, to which the following changes have been made:

- it can handle a checkpoint signal (`SIGUSR1`);
- it can treat a checkpoint file just like any other executable;
- it *wraps* certain system and library calls:
  - for open files (a.o., `open`, `write`, `creat`),
  - for memory allocation (`mmap`, `munmap`, and `mremap`[‡]);
- cross-checkpoint data is stored separately.

When a checkpoint signal is sent to the process, control is passed to the checkpoint handler. First of all the current signal status and the contents of the processor registers are saved. Next, the name and location of the checkpoint file is determined. The checkpoint file is placed in a directory which must be accessible from all the nodes in the cluster. After saving the signal mask and the status of open files, the checkpoint itself is created. Basically, the checkpoint handler saves the address space of the process: the text segment, the data segment, the stack, the dynamically allocated pages, and the shared libraries used.

Restarting the checkpoint is realized by running the checkpoint binary. When the binary is run, the dynamic loader is executed first. As soon as the dynamic loader is finished, control is passed to the actual program. The Dynamite dynamic loader has some extra functionality included. One of the first things this loader tries to locate is the special section containing the name of the checkpoint file. If such a section is present, it knows that it is restoring form a checkpoint, and specialized subroutines take care of a proper handling of the process' segments. The signal status and processor registers are restored, and the process resumes its execution at the point where it was checkpointed.

The wrapped system calls enables the checkpointing/restoring facility to deal with open files. Basically, these wrapper routines invoke the original C-library calls, doing some extra administration, which allow the open file connections to be restored properly. The system call `mmap` is wrapped as the memory allocated by this system call must be restored too. This implies that all the memory allocations done by `mmap` have to be monitored as well. The cross-checkpoint storage is used to preserve data structures across a checkpoint/restart, such as the mapping of the shared libraries used by the process or the status of the open files.

7.5.3 The Migration Protocol

The main objective of the Dynamite migration facility is transparency of the migration protocol, i.e., to allow for the movement of tasks without affecting

[‡]Linux specific.

the operation of other tasks in the system. With respect to the individual task selected for migration this implies transparent suspension and resumption of execution: the task has no notion that it is migrated to another host, and the communication can be delayed without failure, triggered by migration of one of the tasks.

In the task migration protocol we distinguish four phases:

1. create new process context at destination host;
2. the new routing information is broadcasted;
3. disconnect task from its local *pvmd* and checkpoint task;
4. move task to its new host, and restart and reconnect the task to its new *pvmd*.

The first step in the migration protocol is the creation of a new process context at the destination host by sending a message to the *pvmd* representing that host. A new PVM task context is created, so that the PVM daemon can accept any messages addressed to the migrating task and temporarily store them.

Next, all the PVM daemons but the source and the destination one are notified that a migration is about to take place. The daemons update their routing information, so that messages sent via the daemons to the migrating task are sent to the destination node, see also Section 7.5.4.

The checkpoint phase (the third step) is executed on the source node, i.e., the node the task runs on before the migration takes place. First, routing information is updated, so that any messages sent to the migrating task via the PVM daemon are forwarded to the destination node instead of being delivered locally. Finally, the task finds out that it is to be migrated. A `SIGUSR1` signal is sent to the task by the PVM daemon. Control is passed to the checkpoint signal handler in the Dynamite dynamic loader. However, before the actual checkpointing takes place, the communication between this task and its PVM daemon and the other tasks has to be flushed. The signal handler invokes a DPVM function that reads all the available data from all the connections, closes the task connections and sends the final `TM_MIG` migration message to the local PVM daemon. Subsequently, the checkpoint handler creates the checkpoint file and terminates the process.

In the final restart stage, executed on the destination node, the task is restarted at the new location using the `spawn_task` function. In the process of restarting the task from the checkpoint file, the dynamic loader invokes a DPVM function that reconnects the restored task to the PVM daemon on the destination node. Control is passed back to the application code, and the PVM daemon can finally deliver all the messages addressed to the migrating task which it had to store during the migration.

7.5.4 Packet Routing and Direct Connections

In message passing environments like PVM, the process identifier or task identifier, *task id* for short, is a unique identifier which serves as the task's address and therefore may be distributed to other PVM tasks for communication purposes. For this reason the *task id* must remain unchanged during the lifetime of a task, even when the task is migrated.

Indirect Connections

By default, PVM tasks use indirect connections to communicate with each other. In this mode, messages between tasks are routed through two PVM daemons, local to the source and destination tasks. As a consequence, PVM application tasks do not have any remote network connections open, their only communication channel is with the daemon.

As the *task ids* must remain unchanged over migrations, this has implications for the packet routing of messages. The *task id* contains the host identifier at which the task is enrolled and a task sequence number (local to the host). This information is used by the PVM daemon to route packets to their destination, i.e., to the appropriate PVM daemon and task. When a task is migrated to another host, this routing information is not correct anymore. Therefore, an additional routing functionality must be incorporated in the PVM daemon routing software in order to support the migration of tasks.

An important design constraint is that the routing facility must be highly efficient and should not impose additional limitations on the scalability. This is accomplished by maintaining in the PVM daemons the routing tables for

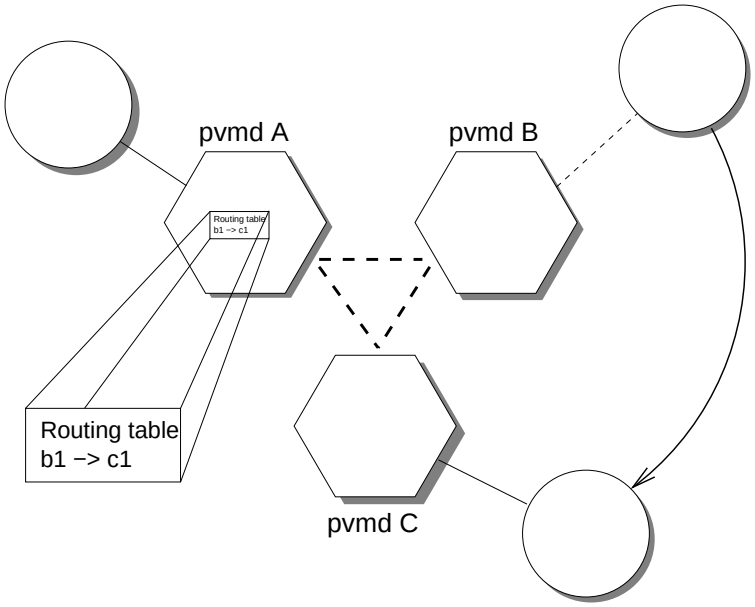


Figure 7.6: Routing tables keep track of the migrated tasks.

migrated tasks, which contains the current locations of migrated tasks (see Fig. 7.6). These routing tables are consulted for all inter-task communication. Upon migration of a task, the routing table of the PVM daemons are updated to reflect the change in location of the migrated task. How this is accomplished was described in the previous section. Figure 7.6 depicts the migration of a task attached to pvmd B and the subsequent routing table update.

Direct Connections

To improve efficiency, an alternative direct communication mode is available on application request. In this mode, tasks that wish to communicate with each other can establish a direct TCP/IP network connection between themselves.

Special care must be taken when migrating a task that has direct connections with other tasks, or else messages that are being processed or are cached in the kernel buffers will be lost during the migration.

In the first two stages of the migration protocol, along with updating the routing information, DPVM notifies all the PVM tasks that a migration is about to take place. Because it is important that the tasks reply in a timely manner, PVM daemons also send SIGURG signal along with the migration notification messages. It is the responsibility of the asynchronously invoked signal handler function to handle this message.

In the checkpoint stage, the checkpoint signal handler of the migrating task sends an end-of-connection (TC_EOC) message via all the open direct connections. The remote tasks read all the data from the connection until they receive TC_EOC, at which point they send the TC_EOC message back. The migrating task reads all the data on its side of the connection, and closes the connection upon receipt of TC_EOC. As a result of the close on the migrating side, the remote tasks receive EOF at this point, and can close the connection on their side.

Any messages that were only partially sent by the migrating task are fully resent after the task is restarted. Any messages that were partially sent by the remote tasks are fully resent via PVM daemons, i.e., indirectly. The direct connection is reestablished as soon as the migrating task restarts and there are new messages to be sent.

7.6 Performance Evaluation

Originally, DPVM was developed on a network of IBM AIX/32 machines (Dikken et al. 1994). Further development of DPVM and later Dynamite has been accomplished on Sun workstations operating under SunOS4 and Solaris (Vesseur et al. 1995), and PC's running Linux. Dynamite currently supports applications written for PVM 3.3.x, running under Solaris/UltraSPARC 2.5.1 and 2.6 and Linux/i386 2.0 and 2.2 (Iskra et al. 2000). From the user's perspective, all that is needed is to relink the application with Dynamite's version of the PVM libraries and with the Dynamite dynamic loader.

The stability of the Dynamite environment has been assessed by a series of tests under Solaris and Linux. Dynamite has been used to make over 2500 successful migrations of large processes (over 20 MB of memory image size) of a commercial PVM application PAM-CRASH (Clinckemaillie et al. 1997) using direct connections, after which the application finished normally.

The performance evaluation of Dynamite is accomplished by four different experiments: (i) communication performance evaluation, (ii) migration overhead evaluation, (iii) evaluation by the NAS Parallel Benchmark (NPB) suite, and (iv) the GRAIL simulation application. The focus of the first two experiments is on the increased overhead of Dynamite compared to standard PVM. The third series of experiments explores the efficacy of the Dynamite environment (including scheduler) for a number of parallel benchmark kernels with different computation and communication behavior. The last experiment measures the performance of Dynamite for a large scientific finite-element model simulation, called GRAIL (de Ronde et al. 1997b).

7.6.1 Measuring DPVM Communication Overhead

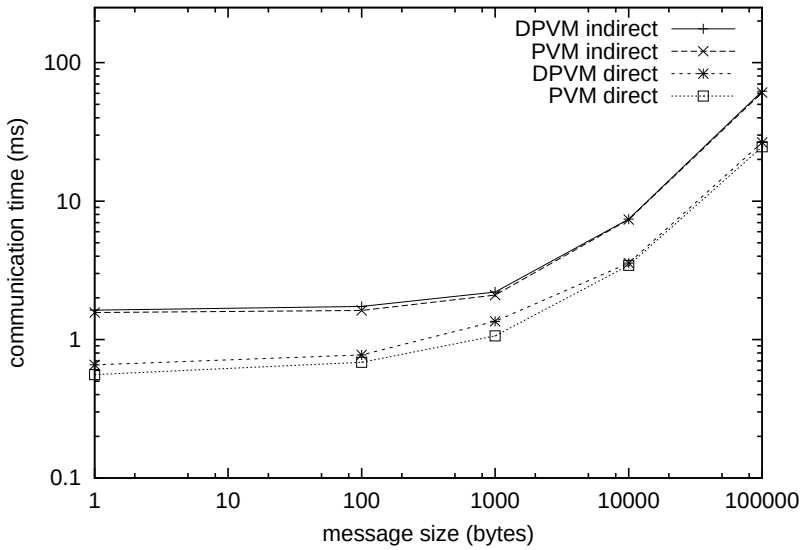
The basic communication properties of a message-passing system, such as *latency* time and *throughput* bandwidth, can be measured by the well-known *ping-pong* experiment. With the ping-pong experiment, series of messages of different sizes are sent between two tasks. The initiating task sends a message to the second task, the second task receives the message into a buffer, and immediately returns it to the initiating task. Half the time of this message ping-pong is recorded as the time t to send a message of length n.

The ping-pong experiments are performed for both the standard PVM implementation as well as the DPVM implementation, with message size ranging from 1 byte to 100 KB. The experimental results obtained for Solaris and Linux are presented in Fig. 7.7 and Table 7.1. The Solaris network consists of Sun Ultra 5/10 workstations interconnected by switched 100 Mb/s Fast Ethernet. The Linux cluster is equipped with Pentium Pro 200 MHz nodes and is also interconnected by switched 100 Mb/s Fast Ethernet.

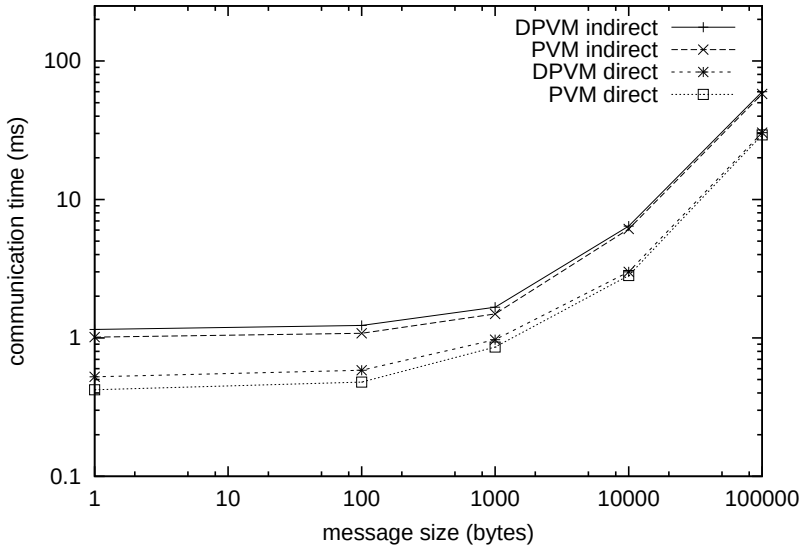
From Fig. 7.7, we can see that in all cases DPVM has (little) increased communication costs. This stems from two factors:

- signal blocking/unblocking on entry and exit from PVM functions (function call overhead);
- extra header in message fragments (communication overhead).

The first factor adds a fixed amount of time for every PVM communication call, whereas the second one increases the communication time by a constant percentage. For small message sizes, the signal blocking/unblocking factor dominates over the extra header information factor, since there is little communication and the message is not fragmented into multiple packets. The DPVM overhead for 1 byte messages ranges from 25% for direct communication under Linux to 4% for indirect communication under Solaris. Although the DPVM



(a) Performance results for Solaris.



(b) Performance results for Linux.

Figure 7.7: Communication performance of PVM and DPVM for Solaris and Linux.

direct communication overhead is significant, we must point out that it represents the worst-case scenario, as the relatively fast direct communication is hurdled by a fixed signal blocking/unblocking overhead, resulting in a large overhead percentage.

As the message size increases, the overhead of extra message header information in message fragments becomes more dominant. The DPVM overhead

	PVM		DVPM	
	latency (msec)	throughput (MB/s)	latency (msec)	throughput (MB/s)
Solaris				
indirect	1.57	1.64	1.63	1.61
direct	0.59	4.01	0.66	3.77
Linux				
indirect	1.01	1.73	1.15	1.67
direct	0.42	3.42	0.52	3.30

Table 7.1: Latency and throughput performance of PVM and DPVM for Solaris and Linux.

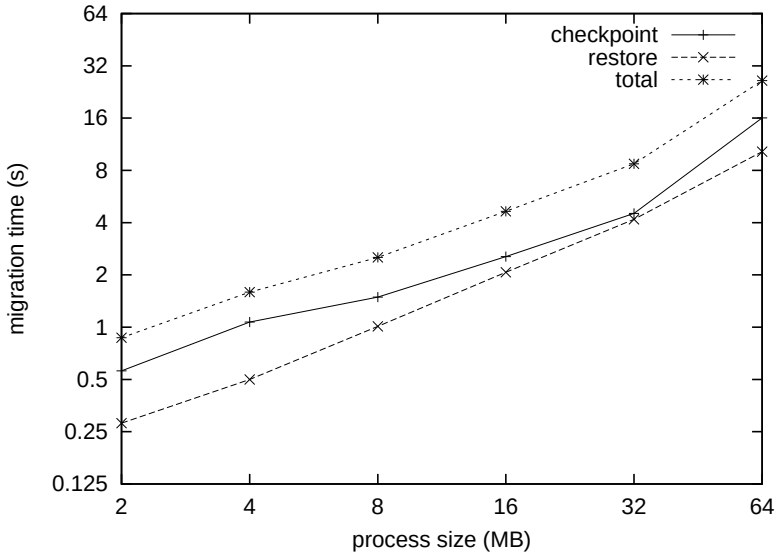
for 100 KB messages eventually becomes 2% for indirect communication and 8% for direct communication under Solaris, and is 4% for both direct and indirect communication under Linux. The extra message header overhead can be tuned by changing the PVM/DPVM message fragment size. By increasing the packet size, a smaller number of fragments are necessary per message sent. However, the overhead for small messages increases, as the packets sent between the tasks will be largely empty.

7.6.2 Checkpoint and Migration Overhead

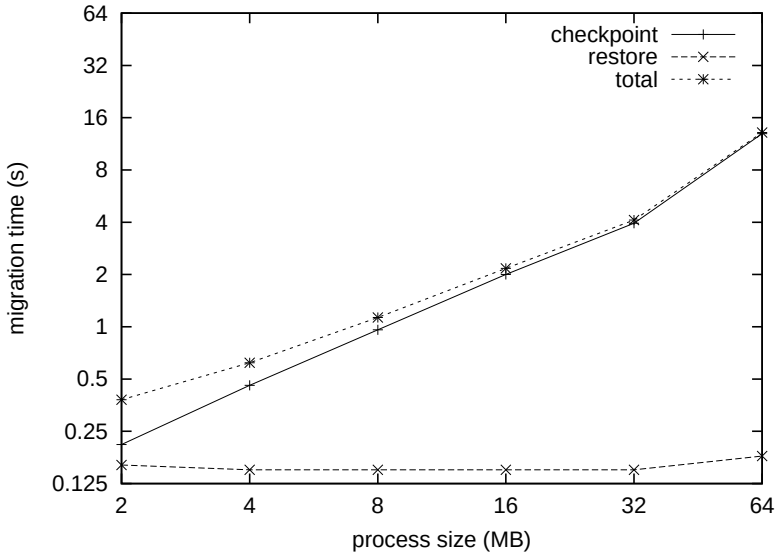
The checkpoint and migration experiments measure the DPVM load balance overhead for various process sizes. A simple ping-pong type program communicating once a few seconds via direct connection was used, process size was set with a single large `malloc` call. The experiments measure the executing time of each of the four migration stages (see Section 7.5.3). The average results over five experiments for Solaris and Linux are shown in Fig. 7.8. The standard deviation σ is smaller than 10%.

The results from the experiments show that the major part of the load balancing overhead is due to checkpointing and restarting. The migration protocol and connection flushing amount together to approximately 0.01–0.03 seconds, and are not depicted in Fig. 7.8. The checkpoint and restart times are limited by the speed of the shared file system. On the two platforms used, the Solaris network and Linux cluster, the bandwidth of the NFS system is 4–5 MB/s over the 100 Mb/s network. From the figure we can see that the checkpoint times under Solaris and Linux approximately increase linearly with process size. However, the restoring phase in Linux takes more or less a constant amount of time, while it grows with the process size under Solaris.

The difference in restore times between Solaris and Linux is due to differences in the implemented memory allocation strategy in `malloc`. For large memory allocations, Linux creates a new memory segment (separate from the heap) using `mmap`, whereas Solaris always allocates from the heap with `sbrk`. When restoring, the head and stack are restored with `read` (see Section 7.5.3),



(a) Checkpoint and migration performance for Solaris.



(b) Checkpoint and migration performance for Linux.

Figure 7.8: Checkpoint and migration performance of DPVM.

which forces an immediate data transfer. However, for the other segments the Linux implementation takes advantage of `mmap`, which uses a more advanced *page on demand* technique, delaying network transfer until the data is actually needed. Since the allocated memory region is not needed to reconnect to the PVM daemon, the time it takes to restart the task is constant under Linux. Clearly, delays may be incurred later, when the `mmap`ped memory is accessed and loaded.

7.6.3 NAS Parallel Benchmarks

The NAS Parallel Benchmarks (NPB)[§] is a suite of applications used by the Numerical Aerodynamic Simulation (NAS) Program at NASA for the performance analysis of parallel computers. The benchmark suite consists of five "kernels" and three simulated applications which mimic the computational behavior of large scale computational fluid dynamics applications. A unique property of the NPB is that the applications are specified algorithmically. The implementation of the NPB kernels used in the experiments with DPVM are described in White et al. (1995).

The characteristic behavior of the NBP kernels used in the performance analysis of PVM and DPVM are:

CG The communication patterns in the conjugate gradient kernel are long-distance and unstructured.

EP The embarrassingly parallel kernel is based on a trivial partitionable problem requiring little or no communication between processors.

FT In the 3-D Fast Fourier Transformation, the communication patters are structured and long distance.

IS Integer sort performs rankings of equally distributed integer keys; the communication is frequent and relatively low-volume, and the pattern of communication is a fully connected graph.

MG The 3-D multigrid solver is characterized by highly structured short- and long-distance communication patterns.

The NPB experiments are performed on eight nodes of the DAS Linux cluster (Pentium Pro 200 MHz and 100 MB/s Fast Ethernet). The nodes are exclusively reserved for the experiments. The NPB kernels are configured to use four computation tasks each, running for approximately 30 minutes in the optimal situation without background load. The number of available nodes exceeds the number of tasks of the NPB kernel. Thus, during the execution of the benchmarks some of the nodes are idle, which allows the Dynamite scheduler to migrate PVM tasks from overloaded nodes to idle nodes.

The NPB kernels are executed and timed for three situations: (i) no background load, (ii) with background load, and (iii) with background load and Dynamite scheduling and migration. Without background load, the compute nodes are fully dedicated to the NPB kernels, and the timings are used as a reference performance. The external background load is generated by running a single computationally intensive process for five minutes on each node used by the benchmark kernel. In this way, one node at a time is overloaded using a round-robin schedule. The performance results of the NPB kernels for the three situations are shown in Table 7.2.

[§]http://www.nas.nasa.gov/NAS/NPB/

	no load	load		ckpt. size	migrations
		no Dynamite	Dynamite		
CG	1795	3352 (+87%)	2226 (+24%)	19 MB	8
EP	1620	1919 (+18%)	1773 (+9%)	14 MB	6
FT	1859	2693 (+45%)	2237 (+20%)	31 MB	8
IS	1511	1688 (+12%)	1758 (+16%)	41 MB	9
MG	1756	2466 (+40%)	1863 (+6%)	17 MB	6

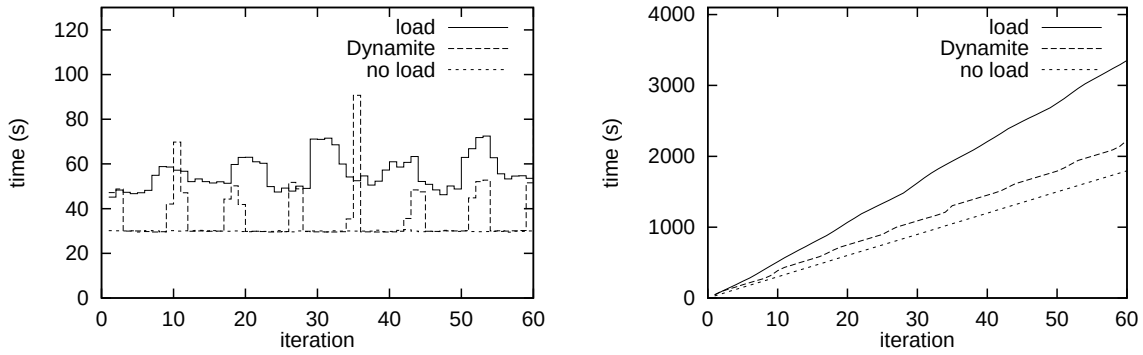
Table 7.2: Execution times of the NAS Parallel Benchmarks (in seconds).

The performance results in Table 7.2 show that the execution times with background load increase for both situations, whether with or without Dynamite. However, the performance results with Dynamite scheduling and migration significantly improve with respect to the results without Dynamite, reducing the percentage of slowdown by a factor two to six. One notable exception is the IS kernel, for which Dynamite scheduling and migration diminishes the performance results compared to the results without Dynamite. The relative difference in slowdown percentages of the NPB kernels depends on the computation and communication characteristics of the respective benchmark kernel.

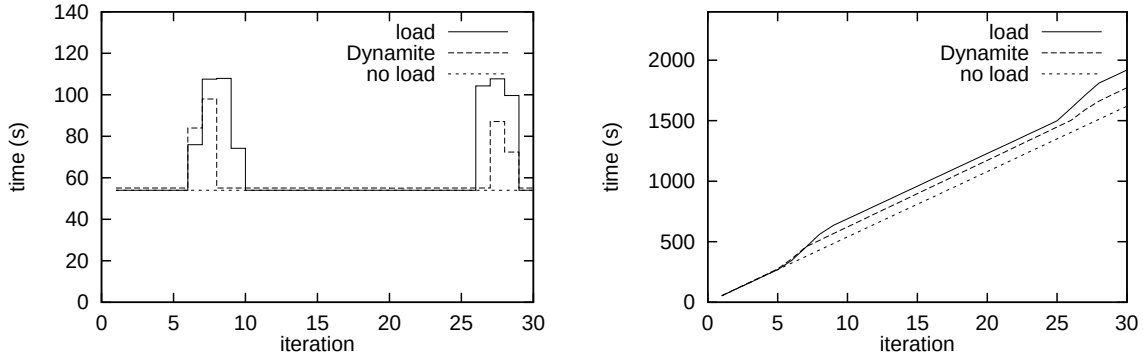
Figure 7.9 presents the execution progress of three NPB kernels: conjugate gradient, embarrassing parallel, and integer sort. These three benchmarks each show different computation and communication characteristics, resulting in significant different performance slowdown figures. The figures show the data for one of the tasks of the parallel benchmark. The left column presents the time spent on executing each individual step (ideally, this should be constant), whereas the right column presents the accumulative execution time.

The results for the CG kernel are shown in Fig 7.9(a). The execution of the CG benchmark slows down 87% when subjected to external background load. The considerable slowdown indicates that a substantial amount of execution time must be spent on computation, thus the CG kernel and the external background load compete for computing resources (CPU time). Furthermore, the communication pattern of the benchmark (loosely synchronous, global communication) forces all the other process to wait for the one lagging behind. The computational intensiveness and the loosely synchronous global communication pattern of the CG kernel makes that the overall performance of the benchmark severely drops. The scheduling capability in Dynamite is able to migrate the CG task from an overloaded to an idle processor; see also the short periods of increased execution time in the left figure of Fig 7.9(a), while the results without Dynamite show constantly increased execution time per step.

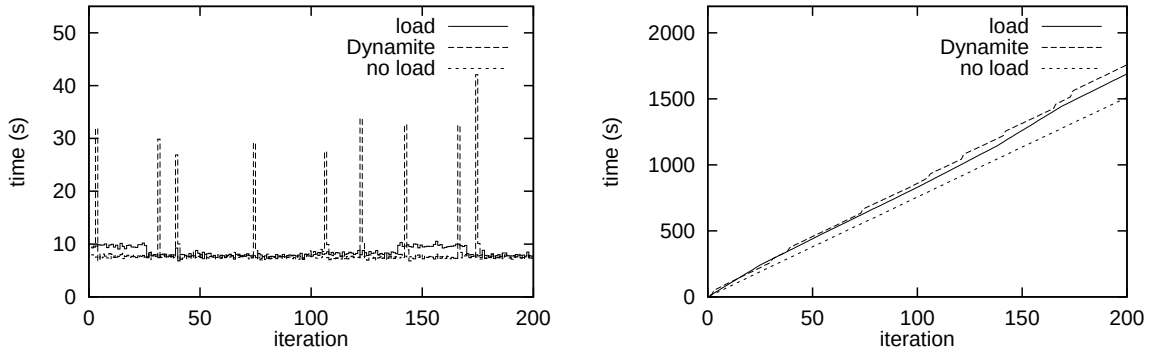
The performance results for the EP kernel in Fig 7.9(b) show a different picture. The “embarrassingly parallel” benchmark is based on trivial partitionability of the problem, while incurring no data or functional dependencies, and requiring little or no communication between processors. The external background load significantly hampers the performance of the affected task,



(a) Conjugate gradient (CG) kernel.



(b) Embarrassing parallel (EP) kernel.



(c) Integer sort (IS) kernel.

Figure 7.9: Execution progress of NAS parallel benchmarks: the time to execute one step (left) and the accumulative time (right).

but has almost no influence on the other tasks (the execution times per step coincidence for the “no load” and “load” figures in Fig. 7.9(b) where other tasks of

the benchmark are affected by the external background load). The gain of Dynamite scheduling is limited, as the overall performance of the EP benchmark is only partially determined by the individual tasks.

The IS benchmark is in particular a communication intensive application. In Fig. 7.9(c), we see that the IS performance is only slightly affected by the external background load, as most of the execution time is spent on communication: frequent and in large-volume. The pattern of communication is a fully connected graph. The performance of Dynamite is slightly worse than the results without any scheduling and migration. Although the migration decisions of Dynamite's scheduler are not unreasonable, there is little performance gain that fails to exceed the migration costs, which is rather high in the IS benchmark because of large process size (41 MB).

The large process size of the FT kernel (31 MB) also limits the attainable performance of Dynamite over no scheduling and migration. The slowdown reduction from 45% to 20% would be significant larger if the processes to be migrated were smaller.

7.6.4 The GRAIL Finite-Element Model Simulation

The GRAIL simulation application is a finite element model of a gravitational radiation antenna (de Ronde et al. 1997a). The finite element model is parallelized by decomposition of a finite-element mesh. The parallel finite-element simulation is time-driven, using loosely synchronous communication to exchange the subdomain boundaries between the neighbors. The computation/communication behavior of the GRAIL simulation kernel is characterized as computational intensive with regular communication patterns of large messages.

A series of GRAIL simulation experiments is executed on the DAS cluster to evaluate the effectiveness of Dynamite for a real-world large scientific simulation application. The series of experiments is carried out *without* external background load for PVM, DPVM, and DPVM with scheduler; and *with* external background load for DPVM and DPVM with scheduler. The parallel simulation consists of 3 tasks running on 4 nodes. The checkpoint size of the GRAIL tasks is 8.5 MB. The results of the experiments are presented in Table 7.3. The individual results are the average over three experiments, with standard deviation σ smaller than 10%.

	parallel environment	exec. time
1	PVM	1854
2	DPVM	1880
3	DPVM + sched.	1914
4	DPVM + load	3286
5	DPVM + sched. + load	2564

Table 7.3: Execution time of the GRAIL application (in seconds).

The first three rows in Table 7.3 show the increased costs of DPVM and DPVM with scheduler over PVM, if there is no external background load. The execution time increased with 1.5% for DPVM, and is accounted to the critical section locking and the extra communication overhead that is necessary for transparent message routing to migrated tasks. On top of this overhead, DPVM with scheduler and monitoring, i.e, the complete Dynamite environment, accounts for another 2% overhead. As there is no migration of the tasks, this 2% overhead is interpreted as the monitoring overhead.

The fourth and fifth set of experiments include an artificial, external background load. The external background load is a single, CPU-intensive process that runs for 600 seconds on each node in turn, using a round-robin schedule. In the fourth set, the monitoring and scheduling subsystem is not running, and DPVM does not migrate tasks from overloaded nodes. A considerable slowdown of 75% over DPVM without background load is observed. The computational intensive GRAIL simulation is fairly sensitive to the presence of external background load. Furthermore, the loosely synchronous communication between the simulation tasks results in an overall performance of the GRAIL simulation that suffers from the background load, and results in the significant slowdown. The fifth set of experiments combines the external background load with monitoring and scheduling; thus the complete Dynamite environment, including the presence of external background load. The results in Table 7.3 show that Dynamite manages to reduce the slowdown percentage from 75% to 34%. This remaining 34% slowdown is contributed to the following factors:

- the delay before the monitor notices increase in load on the node, and to make the migration decision;
- the non-zero costs of the migration; and
- the master task, which is started directly from the shell, cannot be migrated; when the round-robin schedule of the external background load skips the node with the master task, the slowdown decreases further by 10%.

Figure 7.10 depicts the execution progress of the GRAIL simulation for three sets of experiments: (i) PVM, (ii) DPVM with background load, and (iii) DPVM with monitor and scheduler, and background load. The figure shows the progress in iterations of the GRAIL simulation versus the execution time. The performance figure for PVM is a straight line with the largest angle. The other two lines are the performance figures for DPVM with background load, with and without scheduling, and show some discontinuities due to background load and task migration. Initially, the progress of both experiments is slower than the PVM experiment—as the load is initially applied to the node with the master task, no migrations take place. After approximately 600 seconds, the background load moves to another node. Subsequently, in the case where the monitoring and scheduling subsystem runs, the scheduler migrates the application task from the overloaded node, and the progress improves significantly,

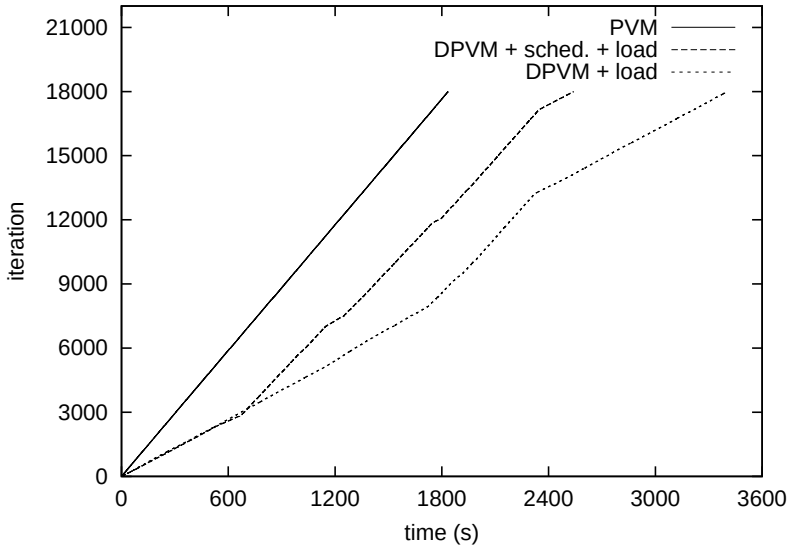


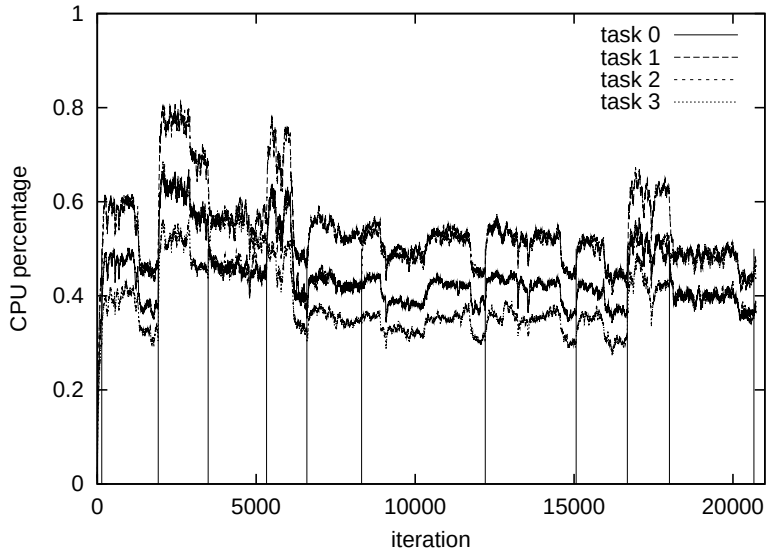
Figure 7.10: Execution progress of GRAIL for three cases. Note that the plain PVM run was made without an external background load, whereas both DPVM runs were done with such a background load.

coming close to the performance of PVM without any background load (performance line of DPVM with scheduling shows same angle). For the experiment with DPVM without the monitoring and scheduling subsystem running, there is no observable change in the performance at this point. However, DPVM without scheduler does improve between 1800 and 2400 seconds from the start, that is when the idle node is overloaded. After 2400 seconds from the start, the node with the master task is overloaded again, so the performance deteriorates in both DPVM cases.

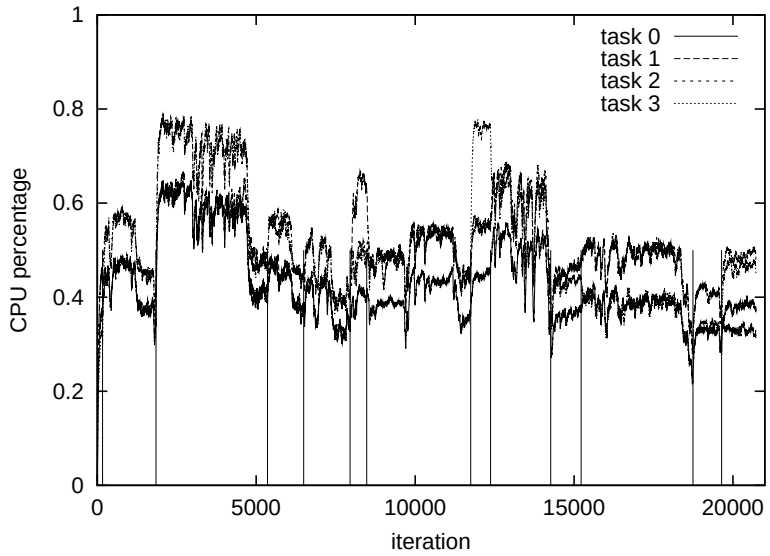
Figure 7.11 shows two typical Dynamite execution runs of the GRAIL simulation in a heterogeneous environment. The execution time of both simulation runs is approximately 1:15 hours. The two figures depict the CPU percentage (the CPU time divided by the elapsed time) of the DPVM tasks during the GRAIL simulation. The results are taken from simulation runs with four tasks on a network of workstations (Solaris/UltraSPARC). The host pool consists of ten workstations with different relative speed[¶], varying from 500, 900, 1000, to 1150. One can see from the figures that for most of the time, the four tasks are assigned different CPU percentages. As the GRAIL simulation has a loosely synchronous communication pattern, the tasks on the fastest workstation experience longer idle times than tasks on the slower workstations. This translates to lower CPU percentages for the tasks on the faster workstations, and vice versa.

The external background load consists of two components. First, the regular background load generated by other users working at their workstations. Sec-

[¶]The relative speed is determined by running a small computation kernel.



(a)



(b)

Figure 7.11: Two typical DPVM execution runs of the GRAIL simulation with four tasks and a host pool of ten processors. The relative speed of the processors in the host pool varies from 500, 900, 1000, to 1150. The migrations are denoted by the vertical lines.

ond, there is an artificial background load imposed on each workstation in turn for 300 seconds, in a round-robin schedule. These external background load factors bring about the large fluctuations in the CPU percentage in Fig. 7.11. But even in the ideal situation without background load, we observe fluctuations

in the GRAIL simulation progress due to the heterogeneity of the workstation pool. By this heterogeneity, it is difficult to interpret the figures directly. The scheduler relates the relative speed of the processor with the measured CPU percentage. In this way the scheduler is successful in making sensible migration decisions. The migrations are denoted by vertical lines in the figures. During the GRAIL simulation run in Fig 7.11(a), there were 11 migrations. The first migration was almost at the start of the execution. The second migration around iteration 1900 in Fig 7.11(a) is more prominent. It demarcates the migration of task 2 from the overloaded node to an lightly loaded node, resulting in a significant overall performance improvement. The concerning node was overloaded two minutes before the scheduler made the decision to migrate task 2 (see the period of reduced performance in Fig 7.11(a). A similar situation appears just before the third migration, where task 3 is migrated. Here however, task 3 is moved from a relative fast, overloaded workstation to a moderate fast, slightly loaded workstation, so there is less global performance impact. Other migrations with a global performance impact are migrations five, seven, eight, nine, and eleven. Figure 7.11(b) gives a similar impression of the GRAIL simulation execution. Although twelve migration are shown in the figure, actually fifteen migrations are made. Three of the migrations were double migrations, where two tasks were migrated at the same time (namely migrations one, five, and twelve).

7.7 Summary and Discussion

The Dynamite environment provides a robust framework for load balancing, where the runtime support system migrates tasks from a parallel program when necessary. The overhead incorporated with dynamic load balancing is small compared to the possible costs of load imbalance. Experiments show a slight performance penalty in a well-balanced system (less than 5%), but significant performance gains can be obtained for task migration in an unbalanced system. The communication and checkpoint overhead experiments in Sections 7.6.1 and 7.6.2 indicate that the Dynamite system provides efficient task migration support. The experiments in a controlled highly dynamic cluster environment (see Sections 7.6.3 and 7.6.4) exposes the ability of Dynamite to react to changes in the cluster environment and reduce the turnaround time of the applications. The eventual success of Dynamite depends on the scheduling strategy.

The concept of implementing the checkpoint/restart facilities in the dynamic loader and using it to migrate PVM tasks has been proven to work in practice. The architecture of Dynamite is modular so that it can easily be adapted to specific application requirements. For example, it is possible to use just the dynamic loader of Dynamite and get checkpoint/restart facilities for sequential jobs that do not use PVM. Also, in the development phase this modularity is used for experimentation with various migration policies. It is even not required to use the Dynamite monitor/scheduler: the user can migrate tasks

manually from the PVM console (using the new `move` command) or from custom programs (using the new `pvm_move` function call).

The future development of Dynamite aims to provide a complete integrated solution for dynamic load balancing of parallel jobs on networks of workstations or clusters. To realize such an environment, a number of enhancements have to be included, such as support for MPI and generic support for the migration of TCP/IP sockets. The need for MPI support is motivated by the large user base, whereas support for migration of TCP/IP sockets is motivated by the potential large application base. With support for the migration of TCP/IP sockets, a large number of parallel applications, either developed with PVM, MPI, or another parallel programming environment based on TCP/IP communication, can benefit from the virtues of dynamic load balancing by task migration. Besides message-passing parallel programs, support for shared-memory parallel programs is projected. The definition of the OpenMP standard (Dagum and Menon 1998) for shared-memory parallel programs allows for an excellent opportunity to include support for dynamic task migration of shared-memory parallel programs in Dynamite.

Another research line is the integration of Dynamite into the Polder metacomputer. The key issue in the integration is the interplay of the Dynamite scheduler with the local resource manager. The design and implementation of more advanced scheduling strategies will be directed by experimental validation of the strategies in the resource management simulation model, which is developed within the Polder metacomputer framework (Santoso et al. 2000). The DAS distributed supercomputer provides an excellent experimental platform to implement and validate different the designs of components in the Polder metacomputer. The DAS architecture with high speed local-area and wide-area network, grasps the characteristics of the prototypical metacomputer of the near future. In this respect, the various issues in wide-area scheduling and local site scheduling have to be included in the DAS resource management.

Finally, the application of Dynamite as a runtime support system for the APSIS Time Warp simulation environment is a challenging research topic. The dynamic computational requirements of the optimistic Time Warp simulation protocol seems to perfectly match the ability of Dynamite to dynamically rebalance the computational load over the processors. However, there are some different considerations for optimistic Time Warp simulations than for parallel applications in general. First, the notion of progress or useful work in optimistic simulation is not easily translated to CPU percentage. Consider for example a simulation process that inhibits trashing behavior, i.e., spends more time on protocol overhead than on forward simulation. Such a process can have a high CPU percentage, while the amount of progress is almost zero. Thus the monitor/scheduler subsystem must be provided with the internal Time Warp kernel statistics such as event rate, rollback rate, and committed event rate. Another consideration is the simulation class that can benefit from dynamic load balancing on a network of workstations. The potentially advantageous simulation application has irregular internal workloads and is an instance of a

class of applications that are *self-initiated* (Nicol 1991). In this self-initiated application class, the simulation processes typically schedule most of their events to themselves, which leads to relatively few remote messages making this class of applications well suited for networks of workstations, which are known to have high communication overheads.

Chapter 8

Summary and Conclusions

> The White Rabbit put on his spectacles. "Where shall I begin, please your Majesty?" he asked. "Begin at the beginning," the King said gravely, "and go on till you come to the end: then stop."
>
> —Lewis Carroll

The increasing understanding of the complexity of the world around us, both in natural sciences and computer science, necessitates realistic models reflecting the complexity of systems under study. The realistic models, and consequently more complex and larger simulations, require vast amounts of execution time. We are particular interested in real-world and theoretical systems that are characterized by heterogeneous spatial and temporal behavior. Systems with this behavior are very common in, for example, population dynamics, immunology, statistical physics, and in computer science. The heterogeneous spatial and temporal behavior is most exactly mapped to asynchronous models.

Experimentation with complex simulations and interpretation of vast amounts of generated data is a challenging endeavor. In this respect, virtual environments that integrate the visualization of data and the interaction with the simulation are becoming more and more important. Consider for example a virtual laboratory where scientists interact with a simulation in a virtual environment (such as a CAVE Automatic Virtual Environment). As the scientist interactively makes changes to the model or to the parameters of the model, the running simulation must be stopped or even rolled back to a previous state, such that the changes can be sustained. Furthermore, support for combined continuous simulation and discrete event simulation is necessary, as applications used in virtual environments can be of both types.

The combination of complex simulations, asynchronous execution mechanisms, and virtual environments, cumulates in large applications that require sufficient computational power to allow for interactive experimentation with the simulation model. Such high performance computing demands cannot be easily fulfilled by a single supercomputer (also not by the distributed nature of the virtual laboratory application). Distributed and parallel computing techniques are a viable solution to the virtual laboratory application, where the

aggregated computing resources are bundled to provide a high performance computing platform.

In our endeavor to distributed and parallel computing, we exploit the locality of data and processing by event scheduling methods from parallel discrete event simulation. The overall balance of workload over the distributed and parallel processing nodes is accomplished by the Dynamite dynamic process migration environment. The combined parallel scheduling of simulation events and the load balance of computational work over the processing nodes is a very complex task. In this thesis, we have approached the two (sub-) problems independently, but generically as both components must be integrated in one environment.

The design and implementation of a simulation environment for parallel discrete event simulation resulted in the APSIS system. The core of APSIS is the Time Warp optimistic simulation kernel. Central issues in the design were the efficient support for large, data-intensive scientific simulations with dynamic behavior. Data-intensive simulations put efficient memory management requirements to the Time Warp simulation kernel. The Time Warp method must regularly checkpoint the simulation state in order to recover from erroneous, optimistic processing of simulation events. We proposed, implemented, and validated an incremental state saving mechanism that *only* saves the changes to the state, instead of the complete state vector. The dynamic behavior of the system asks for the scheduling *and* retraction of simulation events. Event retraction is originally not included in the Time Warp method, but *is* included in the APSIS environment.

The APSIS environment is complemented with the APSE parallelism analysis methodology. The effectiveness and dynamic behavior of the Time Warp method is extensively evaluated using a prototypical application from statistical physics, namely the Ising spin system. The APSE parallelism analysis and the performance experiments with the Ising spin simulation show that the APSIS environment can efficiently schedule the events over the parallel processors.

The experiments with the Ising spin simulation showed also the need for optimism control. In general, Time Warp is quite robust, i.e., performs fairly well without specific adaptations to the simulation application. However, unexpected long turnaround times are observed near the phase transition (or critical phase) in the Ising spin system due to an increase in length and frequency of cascaded rollbacks. The length and frequency of cascaded rollbacks can be bounded by limiting the optimism in the Time Warp method, which is shown by the experiments. The non-trivial interference between application and Time Warp method is conjectured to show self-organized critical behavior. Here the computational complexity of the Time Warp method and the physical complexity of the application are entangled and contribute both to turnaround time and rollback behavior in a non-linear way.

The Dynamite environment incorporates our ideas on dynamic load balancing of computational work over the distributed and parallel processing nodes. We have realized a runtime support system for dynamic load balancing of par-

allel programs by supporting task migration in the PVM system. To allow for task migration of parallel running tasks, a number of issues had to be taken care of: consistent checkpointing, a migration protocol, and packet routing. To implement dynamic load balancing by task migration, the runtime support system must be able to create an image of the running process, the so-called checkpoint. A complicating factor with checkpointing communicating PVM tasks, is that the state of the process also includes the communication status. To prohibit the creation of process checkpoints during communication, we apply the notion of critical sections and embed all interprocess communication operations in such sections. A transparent migration protocol allows for the movement of tasks without affecting the operation of other tasks in the system. To provide transparent and correct message routing with migrating tasks, the task identifiers must be made location independent. This imposes additional requirements on the runtime support system in order to route the messages to their correct destination.

The dynamic load balancing facilities of the Dynamite environment are validated on a cluster of workstations. The ability of the Dynamite environment to adapt to dynamically changing environments is assessed by a number of NAS Parallel Benchmark kernels and two large finite element simulation runs (typically time-driven simulations). One of the finite element simulation models is GRAIL: a large antenna for the detection of gravitational waves. Other finite element simulation experiments are realized with PAMCRASH, a finite element package to evaluate safety issues in car crashes. The results from the experiments showed the potential of Dynamite to dynamically balance the computational load over the parallel or distributed processors.

Future research directions for the APSIS environment include further development of efficient memory management strategies, such as hybrid state saving where copy state saving and incremental state saving techniques are combined and adaptively engaged according to the behavior of the application. Another important feature that is prominent on our wish list is adaptive optimism control. The strategy to approach adaptive optimism control is still an open research question. The difference in predictive quality of computational intensive and complex statistical methods, and fast and simple statistical methods to control the optimism in Time Warp, does not give a marked off advantage to complex methods. Also the influence of self-organized critical behavior in optimistic simulation has to be studied in relation to optimism control, and might lead to new insights.

One of the future developments in Dynamite is the support of task migration for MPI. This work will be a collaboration with Mississippi State University and incorporates their work on Hector. Furthermore, global resource management strategy research will be conducted in context with the Polder metacomputer.

Finally, and maybe in the context of this thesis most important, the Dynamite environment will be used as a runtime support system for the APSIS simulation environment. The main research issue for Dynamite with APSIS is

the study of load balancing strategies for optimistic parallel simulations. In optimistic parallel simulation there is a difference between computational work or load, and the notion of progress. For example, event rollback is computational work but there is no progress of the simulation. The essential question is to find a compact set of system and Time Warp parameters, and an effective strategy to balance the parallel simulation.

Bibliography

Agre, J. R. and P. A. Tinker (1991, January). Useful extensions to a Time Warp simulation system. In *Proceedings of the SCS Multiconference on Advances in Parallel and Distributed Simulation*, Anaheim, CA, pp. 78–85.

al Mourabit, M. (2000, September). Time Warp performance. Master's thesis, Department of Computer Science, University of Amsterdam, Amsterdam, The Netherlands.

Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the AFIPS Spring Joint Computer Conference*, Volume 30, Atlantic City, NJ, pp. 483–485.

Auriche, L. R. G., F. Quaglia, and B. Cicianiy (1998, July). Run-time selection of the checkpoint interval in Time Warp based simulations. *Simulation Practice and Theory 6*(5), 461–478.

Baezner, D., G. Lomow, and B. Unger (1994). A parallel simulation environment based on Time Warp. *International Journal in Computer Simulation 4*(2), 183–207.

Bagrodia, R. L. (1996, December). Perils and pitfalls of parallel discrete-event simulation. In *Proceedings of the 1996 Winter Simulation Conference*, Coronado, CA, pp. 136–143.

Bagrodia, R. L. (1998, April–June). Parallel languages for discrete-event simulation models. *IEEE Computational Science & Engineering 5*(2), 27–38.

Bagrodia, R. L., R. Meyer, M. Takai, Y. an Chen, X. Zeng, J. Martin, and H. Y. Song (1998, October). Parsec: A parallel simulation environment for complex systems. *Computer 31*(10), 77–85.

Bak, P., C. Tang, and K. Wiesenfeld (1988, July). Self-organized criticality. *Physical Review A 38*(1), 364–374.

Balakrishnan, V., P. Frey, N. B. Abu-Ghazaleh, and P. A. Wilsey (1997, December). A framework for performance analysis of parallel discrete event simulators. In *Proceedings of the 1997 Winter Simulation Conference*, Atlanta, Georgia, pp. 429–436.

Ball, D. and S. Hoyt (1990, January). The Adaptive Time-Warp concurrency control algorithm. In *Proceedings of the SCS Multiconference on Distributed Simulation*, San Diego, CA, pp. 174–177.

Banks, J., J. S. Carson, II, and B. L. Nelson (1999). *Discrete-Event System Simulation* (2nd ed.). Upper Saddle River, NJ: Prentice Hall.

Barak, A. and O. La'aden (1998, March). The MOSIX multicomputer operating system for high performance cluster computing. *Future Generation Computer Systems 13*(4-5), 361–372.

Bauer, H. and C. Sporrer (1992, January). Distributed logic simulation and an approach to asynchronous GVT-calculation. In *Proceedings of the Workshop on Parallel and Distributed Simulation*, Newport Beach, CA, pp. 205–209.

Bauer, H. and C. Sporrer (1993, March). Reducing rollback overhead in Time-Warp based distributed simulaiton with optimized incremental state saving. In *Proceedings of the 26th Annual Simulation Symposium*, Washington, DC, pp. 12–20.

Bauer, H., C. Sporrer, and T. Krodel (1991, August). On distributed logic simulation using Time Warp. In *Proceedings of the IFIP International Conference on Very Large Scale Integration (VLSI)*, Edinburgh, Scotland, pp. 127–136.

Bellenot, S. (1990, January). Global virtual time algorithms. In *Proceedings of the SCS Multiconference on Distributed Simulation*, San Diego, CA, pp. 122–127.

Berry, O. and D. Jefferson (1985, jan). Critical path analysis of distributed simulation. In *Proceedings of the 1985 SCS Multiconference on Distributed Simulation*, San Diego, CA, pp. 57–60.

Bersini, H. and V. Detours (1994, July). Asynchrony induces stability in cellular automata based models. In *Proceedings of the IVth Conference on Artificial Life*, Cambridge, MA, pp. 382–387.

Bhatt, S., R. Fujimoto, A. Ogielski, and K. Perumalla (1998, August). Parallel simulation techniques for large-scale networks. *IEEE Communications Magazine 36*(8), 42–47.

Binder, K. and D. W. Heermann (1992). *Monte Carlo Simulation in Statistical Physics*. New York: Springer-Verlag.

Błażewicz, J., K. H. Ecker, and T. Yang (2000, July). New trends on scheduling in parallel and distributed systems. *Parallel Computing 26*(9), 1061–1063.

Boden, N. J., D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su (1995, February). Myrinet: A gigabit-per-second local-area network. *IEEE Micro 15*(1), 29–36.

Bratley, P., B. L. Fox, and L. E. Schrage (1987). *A Guide to Simulation* (2nd ed.). New York: Springer-Verlag.

Brown, R. (1988, October). Calendar queues: A fast O(1) priority queue implementation for the simulation event set problem. *Communications of the ACM 31*(10), 1220–1227.

Bruce, D. (1995, June). The treatment of state in optimistic systems. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, Lake Placid, NY, pp. 40–49.

Brune, M., J. Gehring, and A. Reinefeld (1997a). Heterogeneous message passing and a link to resource management. *Journal on Supercomputing 11*(4), 355–369.

Brune, M., J. Gehring, and A. Reinefeld (1997b, May). A lightweight communication interface for parallel programming environments. In *High-Performance Computing and Networking (HPCN Europe '97)*, Volume 1225 of *LNCS*, pp. 503–513. Springer-Verlag.

Brunk, G. (2000, January/February). Understanding self-organized criticality as a statistical process. *Complexity 5*(3), 26–33.

Brush, S. G. (1967). History of the Lenz-Ising model. *Reviews of Modern Physics 39*, 883–893.

Bryant, R. E. (1977, November). Simulation of packet communication architecture computer systems. Technical Report LCS-TR-188, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA.

Cai, W. and S. J. Turner (1990, January). An algorithm for distributed discrete-event simulation – the "carrier null messag" approach. In *Proceedings of the 1990 SCS Multiconference on Distributed Simulation*, San Diego, CA, pp. 3–8.

Casas, J., D. L. Clark, P. S. Galbiati, R. Konuru, S. W. Otto, R. M. Prouty, and J. Walpole (1995, May). MIST: PVM with transparent migration and checkpointing. In *Third Annual PVM Users' Group Meeting*, Pittsburgh, PA.

Ceperley, D., M. Mascagni, and A. Srinivasan (1999). *SPRNG: Scalable Parallel Random Number Generators*. Champaign, IL: University of Illinois at Urbana-Champaign. `http://www.ncsa.uiuc.edu/Apps/SPRNG/`.

Chandy, K. M. and L. Lamport (1985, February). Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems 3*(1), 63–75.

Chandy, K. M. and J. Misra (1979, September). Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering SE-5*(5), 440–452.

Chandy, K. M. and J. Misra (1981, April). Asynchronous distributed simulation via a sequence of parallel computations. *Communications of the ACM 24*(11), 198–206.

Chandy, K. M., J. Misra, and L. M. Haas (1983, May). Distributed deadlock detection. *ACM Transactions on Computer Systems 1*(2), 144–156.

Chandy, K. M. and R. Sherman (1989, March). Space-time and simulation. In *Proceedings of the 1989 SCS Multiconference on Distributed Simulation*, Tampa, Florida, pp. 53–57.

Chapin, S. J., D. Katramatos, J. Karpovich, and A. Grimshaw (1999, October). Resource management in Legion. *Future Generation Computer Systems 15*(5-6), 583–594.

Cherkassky, B. V., A. V. Goldberg, and C. Silverstein (1996, June). Buckets, heaps, lists, and monotone priority queues. Technical Report NECI TR 96-070, NEC Research Institute, Princeton, NJ.

Choi, E. (1998, April). Virtual time window for balancing progress on parallel optimistic protocol and its effect on computation complexity. In *Proceedings of the 31st Annual Simulation Symposium*, Boston, MA, pp. 9–16.

Clark, D. (1998, April–June). ASCI Pathforward: to 30 Tflops and beyond. *IEEE Concurrency 6*(2), 13–15.

Clark, D. (2000, January–March). Blue Gene and the race toward petaflops capacity. *IEEE Concurrency 8*(1), 5–9.

Cleary, J., F. Gomes, B. Unger, X. Zhonge, and R. Thudt (1994, July). Cost of state saving and rollback. In D. K. Arvind, R. Bagrodia, and J. Y.-B. Lin (Eds.), *Proceedings of the 8th Workshop on Parallel and Distributed Simulation*, Edinburgh, U.K., pp. 94–101.

Clinckemaillie, J., B. Elsner, G. Lonsdale, S. Meliciani, S. Vlachoutsis, F. de Bruyne, and M. Holzner (1997, Spring). Performance issues of the parallel PAM-CRASH code. *The International Journal of Supercomputing Applications and High-Performance Computing 11*(1), 3–11.

Coffman, E. G. (1976). Introduction to deterministic scheduling theory. In E. G. Coffman (Ed.), *Computer & Job/Shop Scheduling Theory*, pp. 1–50. John Wiley & Sons.

Concepcion, A. I. and S. G. Kelly (1991, January). Computing global virtual time using the multi-level token passing algorithm. In *Proceedings of the SCS Multiconference on Advances in Parallel and Distributed Simulation*, Anaheim, CA, pp. 63–68.

Dagum, L. and R. Menon (1998, January–March). OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science & Engineering 5*(1), 46–55.

Dahmann, J. S. (1999, May). The High Level Architecture and beyond: Technology challenges. In *Proceedings of the 13th Workshop on Parallel and Distributed Simulation*, Atlanta, GA, pp. 64–70.

Das, S., R. Fujimoto, K. Panesar, D. Allison, and M. Hybinette (1994, December). GTW: A Time Warp system for shared memory multiprocessors. In *Proceedings of the 1994 Winter Simulation Conference*, Orlando, FL, pp. 1332–1339.

Das, S. R. and R. M. Fujimoto (1997, April). Adaptive memory management and optimism control in Time Warp. *ACM Transactions on Modeling and Computer Simulation 7*(2), 239–271.

de Ronde, J. F., A. Schoneveld, P. M. A. Sloot, N. Floros, and J. Reeve (1996, April). Load balancing by redundant decomposition and mapping. In *High Performance Computing and Networking (HPCN Europe '96)*, Volume 1067 of *LNCS*, pp. 555–561. Springer.

de Ronde, J. F., G. D. van Albada, and P. M. A. Sloot (1997a, April). High performance simulation for resonant-mass gravitational radiation antennas. In *High Performance Computing and Networking (HPCN Europe '97)*, Volume 1225 of *LNCS*, pp. 200–212. Springer.

de Ronde, J. F., G. D. van Albada, and P. M. A. Sloot (1997b, September). Simulation of gravitational wave detectors. *Computers in Physics 11*(5), 484–497.

de Vries, R. C. (1990, January). Reducing null messages in Misra's distributed discrete event simulation method. *IEEE Transactions on Software Engineering 16*(1), 82–91.

Decker, T., M. Fischer, R. Lüling, and S. Tschöke (1998, July). A distributed load balancing algorithm for heterogeneous parallel computing systems. In *Proceedings of the 1998 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'98)*, Volume II, Las Vegas, NV, pp. 933–940.

Deelman, E., B. K. Szymanski, and T. Caraco (1996, December). Simulating Lyme disease using parallel discrete event simulation. In *Proceedings of the 1996 Winter Simulation Conference*, Coronado, CA, pp. 1191–1198.

DeFanti, T. A., I. Foster, M. E. Papka, R. Stevens, and T. Kuhfuss (1996). Overview of the I-WAY: Wide area visual supercomputing. *International Journal of Supercomputer Applications and High Performance Computing 10*(2/3), 123–130.

Defense Modeling and Simulation Office (1999). High Level Architecture technical specifications. `http://hla.dmso.mil/`.

Dickens, P. M., D. M. Nicol, P. F. Reynolds, Jr., and J. M. Duva (1996, October). Analysis of bounded Time Warp and comparison with YAWNS. *ACM Transactions on Modeling and Computer Simulation 6*(4), 297–320.

Dikken, L., F. van der Linden, J. Vesseur, and P. Sloot (1994). DPVM: Dynamic load balancing on parallel systems. In *High Performance Computing and Networking*, pp. 273–277.

Douglis, F. and J. Ousterhout (1991, August). Transparent process migration: Design alternatives and the Sprite implementation. *Software—Practice & Experience 21*(8), 757–785.

D'Souza, L. M., X. Fan, and P. A. Wilsey (1994, July). pGVT: An algorithm for accurate GVT estimation. In D. K. Arvind, R. Bagrodia, and Y.-B. Lin (Eds.), *Proceedings of the 8th Workshop on Parallel and Distributed Simulation*, Edinburgh, Scotland, U.K., pp. 102–109.

D'Souza, L. M., X. Fan, and P. A. Wilsey (1997, September). Modifications to the pGVT algorithm to eliminate acknowledgement messages and improve the GVT broadcast frequency. In *Proceedings of the World Congress on Systems Simulation: Conference on Parallel & Distributed Simulation*, Singapore, pp. 288–292.

Dyadkin, I. G. and K. G. Hamilton (2000, March). A study of 128-bit multipliers for computer pseudorandom number generators. *Computer Physics Communications 125*(1-3), 221–238.

Eager, D. L., J. Zahorjan, and E. D. Lazowska (1989, March). Speedup versus efficiency in parallel systems. *IEEE Transactions on Computers 38*(3), 408–423.

Entacher, K., A. Uhl, and S. Wegenkittl (1998, May). Linear and inversive pseudorandom numbers for parallel and distributed simulation. In *Proceedings of the 12th Workshop on Parallel and Distributed Simulation*, Banff, Canada, pp. 90–97.

Ferscha, A. (1995, June). Probabilistic adaptive direct optimism control in Time Warp. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, Lake Placid, NY, pp. 120–129.

Ferscha, A. (1999, March/April). Adaptive Time Warp simulation of timed Petri nets. *IEEE Transactions on Software Engineering 25*(2), 237–257.

Ferscha, A. and J. Johnson (1996, December). A testbed for parallel simulation performance prediction. In *Proceedings of the 1996 Winter Simulation Conference*, Coronado, CA, pp. 637–644.

Ferscha, A. and S. K. Tripathi (1994, August). Parallel and distributed simulation of discrete event systems. Technical Report CS-TR-3336, Department of Computer Science, University of Maryland, College Park, MD.

Fishwick, P. A. (1997, December). Web-based simulation. In *Proceedings of the 1997 Winter Simulation Conference*, Atlanta, GA, pp. 100–102.

Fleischmann, J. and P. A. Wilsey (1995, June). Comparative analysis of periodic state saving techniques in Time Warp simulators. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, Lake Placid, NY, pp. 50–58.

Foster, I. and C. Kesselman (1997). Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications and High Performance Computing 11*(2), 115–128.

Foster, I. and C. Kesselman (1998). *The Grid: Blueprint for a Future Computing Infrastructures*. San Francisco, CA: Morgan Kaufmann Publishers.

Franks, S., F. Gomes, B. Unger, and J. Cleary (1997, June). State saving for interactive optimistic simulation. In *Proceedings of the 11th Workshop on Parallel and Distributed Simulation*, Lockenhaus, Austria, pp. 72–79.

Frette, V., K. Christensen, A. Malthe-Sørensen, J. Feder, T. Jøssang, and P. Meakin (1996, January). Avalance dynamics in a pile of rice. *Nature 379*(6560), 49–52.

Fujimoto, R. M. (1989a, April). Performance measurements of distributed simulation strategies. *Transactions of the Society for Computer Simulation 6*(2), 89–132.

Fujimoto, R. M. (1989b, July). Time Warp on a shared memory multiprocessor. *Transactions of the Society for Computer Simulation 6*(3), 211–239.

Fujimoto, R. M. (1990a, October). Parallel discrete event simulation. *Communications of the ACM 33*(10), 30–53.

Fujimoto, R. M. (1990b, January). Performance of Time Warp under synthetic workloads. In *Proceedings of the SCS Multiconference on Distributed Simulation*, San Diego, CA, pp. 23–28.

Fujimoto, R. M. (1998, December). Time management in the High Level Architecture. *Simulation 71*(6), 388–400.

Fujimoto, R. M. (2000). *Parallel and Distributed Simulation Systems*. Wiley Series on Parallel and Distributed Computing. New York: Wiley.

Fujimoto, R. M. and M. Hybinette (1997, October). Computing global virtual time in shared-memory multiprocessors. *ACM Transactions on Modeling and Computer Simulation 7*(4), 425–446.

Gafni, A. (1988, July). Rollback mechanisms for optimistic distributed simulation systems. In *Proceedings of the SCS Multiconference on Distributed Simulation*, San Diego, CA, pp. 61–67.

Glauber, R. J. (1963, February). Time-dependent statistics of the Ising model. *Journal of Mathematical Physics 4*(2), 294–307.

Gomes, F., S. Franks, B. Unger, Z. Xiao, J. Cleary, and A. Covington (1995, December). SimKit: A high performance logical process simulation class library in C++. In *Proceedings of the 1995 Winter Simulation Conference*, San Diego, CA, pp. 706–713.

Gomes, F., B. Unger, and J. Cleary (1996, December). Language based state saving extensions for optimistic parallel simulation. In *Proceedings of the 1996 Winter Simulation Conference*, Coronado, CA, pp. 794–800.

Grimshaw, A. S. and W. A. Wulf (1996, August). Legion—A view from 50,000 feet. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, Syracuse, NY, pp. 89–99.

Groselj, B. and C. Tropper (1989, March). A deadlock resolution scheme for distributed simulation. In *Proceedings of the 1989 SCS Multiconference on Distributed Simulation*, Tampa, FL, pp. 108–112.

Gustafson, J. L. (1988, May). Reevaluating Amdahl's law. *Communications of the ACM 31*(5), 532–533.

Hamnes, D. O. and A. Tripathi (1994, July). Investigations in adaptive distributed simulation. In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation*, Edinburgh, UK, pp. 20–23.

Hellekalek, P. (1998, May). Don't trust parallel Monte Carlo! In *Proceedings of the 12th Workshop on Parallel and Distributed Simulation*, Banff, Canada, pp. 82–89.

Hollingsworth, J. K. (1998, October). Critical path profiling of message passing and shared-memory programs. *IEEE Transactions on Parallel and Distributed Systems 9*(10), 1029–1040.

Hooper, J. W. (1986, April). Strategy-related characteristics of discrete-event languages and models. *Simulation 46*(4), 153–159.

Hwang, K. (1993). *Advanced Computer Architecture: Parallelism, Scalability, Programmability*. New York: McGraw-Hill, Inc.

Ingerson, T. E. and R. L. Buvel (1984, January). Structure in asynchronous cellular automata. *Physica 10D*(1/2), 59–68.

Iskra, K. A., F. van der Linden, Z. W. Hendrikse, B. J. Overeinder, G. D. van Albada, and P. M. A. Sloot (2000, July). The implementation of Dynamite: An environment for migrating PVM tasks. *ACM Operating Systems Review 34*(3), 40–55.

Jefferson, D. and P. Reiher (1991, April). Supercritical speedup. In *Proceedings of the 24th Annual Simulation Symposium*, New Orleans, Lousiana, pp. 159–168.

Jefferson, D. R. (1985, July). Virtual time. *ACM Transactions on Programming Languages and Systems 7*(3), 404–425.

Jefferson, D. R. et al. (1987, November). Distributed simulation and the Time Warp Operating System. *ACM Operating Systems 21*(5), 77–93.

Jefferson, D. R. and A. Motro (1986, February). The Time Warp mechanism for database concurrency control. In *Proceedings of the International Conference on Data Engineering*, Los Angeles, CA, pp. 474–481.

Jefferson, D. R. and H. Sowizral (1982, December). Fast concurrent simulation using the Time Warp mechanism, Part I: Local control. Technical Report N-1906-AF, RAND Corporation, Santa Monica, CA.

Jensen, H. (1998). *Self-Organized Criticality*. Cambridge University Press.

Jha, V. and R. Bagrodia (1996, May). A performance evaluation methodology for parallel simulation protocols. In *Proceedings of the 10th Workshop on Parallel and Distributed Simulation*, Philadelphia, PN, pp. 180–185.

Johnson, S. C. (1979). Yacc: Yet another compiler-compiler. In *UNIX Programmer's Manual*, Volume 2. Murray Hill, NJ: Bell Telephone Laboratories.

Joshi, A., T. Drashansky, J. Rice, S. Weerawarana, and E. Houstis (1997, May). Multiagent simulation of complex heterogeneous models in scientific computing. *Mathematics and Computers in Simulation 44*(1), 43–59.

Keller, A., M. Brune, and A. Reinefeld (1999, April). Resource management for high-performance PC clusters. In *High Perfromance Computing and Networking (HPCN Europe 1999)*, Volume 1593 of *LNCS*, pp. 270–280. Springer.

Lamport, L. (1978, July). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21*(7), 558–565.

Larzelere II, A. R. (1998, January–March). Creating simulation capabilities. *IEEE Computational Science & Engineering 5*(1), 27–35.

Leivent, J. I. and R. J. Watro (1993, November). Mathematical foundations for Time Warp systems. *ACM Transactions on Programming Languages and Systems 15*(5), 771–794.

Lesk, M. E. and E. Schmidt (1979). Lex – a lexical analyzer generator. In *UNIX Programmer's Manual*, Volume 2. Murray Hill, NJ: Bell Telephone Laboratories.

Li, X., J. Cleary, and B. Unger (1992, April). Virtual time and virtual space. *International Journal of Parallel Programming 21*(2), 123–150.

Lim, C.-C., Y.-H. Low, B.-P. Gan, S. Jain, W. Cai, W. J. Hsu, and S. Y. Huang (1999, May). Performance prediction tools for parallel discrete-event simulation. In *Proceedings of the 13th Workshop on Parallel and Distributed Simulation*, Atlanta, GA, pp. 148–155.

Lin, Y.-B. (1992, July). Parallelism analyzer for parallel discrete event simulation. *ACM Transactions on Modeling and Computer Simulation 2*(3), 239–264.

Lin, Y.-B. and E. D. Lazowska (1990a, August). Determining the global virtual time in a distributed simulation. In *Proceedings of the 1990 International Conference on Parallel Processing*, Volume III, Saint Charles, IL, pp. 201–209.

Lin, Y.-B. and E. D. Lazowska (1990b, October). Exploiting lookahead in parallel simulation. *IEEE Transactions on Parallel and Distributed Systems 1*(4), 457–469.

Lin, Y.-B., B. R. Preiss, W. M. Loucks, and E. D. Lazowska (1993, May). Selecting the checkpoint interval in Time Warp simulation. In R. Bagrodia and D. Jefferson (Eds.), *Proceedings of the 7th Workshop on Parallel and Distributed Simulation*, San Diego, CA, pp. 3–10.

Lipton, R. J. and D. W. Mizell (1990, January). Time Warp vs. Chandy–Misra: A worst-case comparison. In *Proceedings of the SCS Multiconference on Distributed Simulation*, San Diego, CA, pp. 137–143.

Litzkow, M., M. Livny, and M. W. Mutka (1988). Condor—a hunter of idle workstations. In *8th IEEE International Conference on Distributed Computing Systems*, pp. 104–111.

Liu, J., D. Nicol, B. Premore, and A. Poplawski (1999, May). Performance prediction of a parallel simulator. In *Proceedings of the 13th Workshop on Parallel and Distributed Simulation*, Atlanta, GA, pp. 156–164.

Liu, L. Z. and C. Tropper (1990, January). Local deadlock detection in distributed simulation. In *Proceedings of the 1990 SCS Multiconference on Distributed Simulation*, San Diego, CA, pp. 64–69.

Livny, M. (1985, January). A study of parallelism in distributed simulation. In *Proceedings of the 1985 SCS Multiconference on Distributed Simulation*, San Diego, CA, pp. 94–98.

Lomow, G., J. Cleary, B. Unger, and D. West (1988, February). A performance study of Time Warp. In *Proceedings of the SCS Multiconference on Distributed Simulation*, San Diego, CA, pp. 50–55.

Lomow, G., S. R. Das, and R. M. Fujimoto (1991, July). Mechanisms for user-invoked retraction of events in Time Warp. *ACM Transactions on Modeling and Computer Simulation 1*(3), 219–243.

Loucks, W. M. and B. R. Preiss (1990, January). The role of knowledge in distributed simulation. In *Proceedings of the 1990 SCS Multiconference on Distributed Simulation*, San Diego, CA, pp. 9–16.

Low, Y.-H., C.-C. Lim, W. Cai, S.-Y. Huang, W.-J. Hsu, S. Jain, and S. J. Turner (1999, March). Survey of languages and runtime libraries for parallel discrete-event simulation. *Simulation 72*(3), 170–186.

Lubachevsky, B. D. (1987, December). Efficient parallel simulation of asynchronous cellular arrays. *Complex Systems 1*(6), 1099–1123.

Lubachevsky, B. D. (1988, March). Efficient parallel simulations of dynamic Ising spin systems. *Journal of Computational Physics 75*(1), 103–122.

Lumer, E. D. and G. Nicolis (1994, March). Synchronous versus asynchronous dynamics in spatially distributed systems. *Physica D 71*(4), 440–452.

Macready, W. G., A. G. Siapas, and S. A. Kauffman (1996, jan). Criticality and parallelism in combinatorial optimization. *Science 271*(5245), 56–59.

Madisetti, V., J. Walrand, and D. Messerschmitt (1990, January). Synchronization in message-passing computers—Models, algorithms, and analysis. In *Proceedings of the 1990 SCS Multiconference on Distributed Simulation*, San Diego, CA, pp. 35–48.

Madisetti, V. K., D. A. Hardaker, and R. M. Fujimoto (1993, August). The MIMDIX environment for parallel simulation. *International Journal of Parallel and Distributed Computing 18*(4), 473–483.

Mascarenhas, E., F. Knop, and V. Rego (1995, December). ParaSol: A multithreaded system for parallel simulation based on mobile threads. In *Proceedings of the 1995 Winter Simulation Conference*, San Diego, CA, pp. 690–697.

Matsumoto, M. and T. Nishimura (1998, January). Mersene twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation 8*(1), 3–30.

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

McCauley, E., W. G. Wilson, and A. M. de Roos (1993, September). Dynamics of age-structured and spatially structured predator-prey interactions: Individual-based models and population-level formulations. *The American Naturalist 143*(3), 412–442.

McKerrow, P. (1988). *Performance Measurements of Computer Systems*. International Computer Science Series. Sidney: Addison-Wesley.

Mehl, H. (1992, January). A deterministic tie-breaking scheme for sequential and distributed simulation. In M. Abrams and P. F. Reynolds, Jr. (Eds.), *Proceedings of the 6th Workshop on Parallel and Distributed Simulation*, Newport Beach, CA, pp. 199–200.

Mellott, L. E., M. W. Berry, E. J. Comiskey, and L. J. Gross (1999, March). The design and implementation of an individual-based predator-prey model for a distributed computing environment. *Simulation Practice and Theory 7*(1), 47–70.

Miller, B. P., M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall (1995, November). The Paradyn parallel performance measurement tool. *Computer 28*(11), 37–46.

Milojicic, D. S., W. Zint, A. Dangel, and P. Giese (1993, April). Task migration on the top of the Mach microkernel. In *MACH III Symposium Proceedings*, Santa Fe, New Mexico, pp. 19–21.

Mink, A., R. J. Carpenter, G. G. Nacht, and J. W. Roberts (1990, September). Multiprocessor performance measurement instrumentation. *Computer 23*(9), 63–75.

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

Mohr, B. (1990, September). Performance evaluation of parallel programs in parallel and distributed systems. In *Proceedings of CONPAR 90 / VAPP IV, Joint International Conference on Vector and Parallel Processing*, Zurich, Switzerland, pp. 176–187.

MPI Forum (1998). MPI2: A message-passing interface standard. *The International Journal of High Performance Computing Applications 12*(1-2), 1–299.

Nagel, W. E. and R. Williams (1998, November). Metacomputing: From ideas to real implementations. *Parallel Computing 24*(12-13), 1709–1711.

Nance, R. E. (1993, March). A history of discrete event simulation programming languages. *ACM SIGPLAN Notices 28*(3), 149–175.

Nicol, D. and X. Lui (1997, June). The dark side of risk (What your mother never told you about Time Warp). In C. Tropper and R. Ayani (Eds.), *Proceedings of the 11th Workshop on Parallel and Distributed Simulation*, Lockenhaus, Austria, pp. 188–195.

Nicol, D. M. (1991, January). Performance bounds on parallel self-initiating discrete-event simulations. *ACM Transactions on Modeling and Computer Simulations 1*(1), 24–50.

Nicol, D. M. (1993, April). The cost of conservative synchronization in parallel discrete event simulations. *Journal of the ACM 40*(2), 304–333.

Nicol, D. M. and R. M. Fujimoto (1994, December). Parallel simulation today. *Annals of Operations Research 53*, 249–286.

Nygaard, K. and O.-J. Dahl (1978, August). The development of the Simula languages. *ACM SIGPLAN Notices 13*(8), 245–272.

Overeinder, B. J., L. O. Hertzberger, and P. M. A. Sloot (1991, May). Parallel discrete event simulation. In W. J. Withagen (Ed.), *Proceedings of the Third Workshop Computer Systems*, Eindhoven, The Netherlands, pp. 19–30.

Overeinder, B. J. and P. M. A. Sloot (1993, October). Application of Time Warp to parallel simulations with asynchronous cellular automata. In A. Verbraeck and E. J. H. Kerckhoffs (Eds.), *Proceedings of the 1993 European Simulation Symposium*, Delft, The Netherlands, pp. 397–402.

Overeinder, B. J. and P. M. A. Sloot (1995, May). Parallel performance evaluation through critical path analysis. In B. Hertzberger and G. Serazzi (Eds.), *High-Performance Computing and Networking (HPCN Europe '95)*, Volume 919 of *LNCS*, pp. 634–639. Springer-Verlag.

Overeinder, B. J. and P. M. A. Sloot (1997). Breaking the curse of dynamics by task migration: Pilot experiments in the polder metacomputer. In M. Bubak, J. Dongarra, and J. Waśniewsky (Eds.), *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Volume 1332 of *Lecture Notes in Computer Science*, Berlin, pp. 194–207. Springer-Verlag.

Overeinder, B. J., P. M. A. Sloot, R. N. Heederik, and L. O. Hertzberger (1996, May). A dynamic load balancing system for parallel cluster computing. *Future Generation Computer Systems 12*(1), 101–115.

Overeinder, B. J., P. M. A. Sloot, and L. O. Hertzberger (1992, September). Time Warp on a Transputer platform: Pilot study with asynchronous cellular automata. In M. Valero et al. (Eds.), *Parallel Computing and Transputer Applications*, Barcelona, Spain, pp. 1303–1312.

Overeinder, B. J., J. J. J. Vesseur, F. v/d Linden, and P. M. A. Sloot (1995, May). A communication kernel for parallel programming support on a massively parallel processor system. In *Proceedings of the Workshop on Parallel Programming and Computation (ZEUS'95) and the 4th Nordic Transputer Conference (NTUG'95)*, Linkøping, Sweden, pp. 259–266.

Palaniswamy, A. C. and P. A. Wilsey (1996, September). Parameterized Time Warp (PTW): An integrated adaptive solution to optimistic PDES. *Journal of Parallel and Distributed Computing 37*(2), 134–145.

Pham, C. D. and R. L. Bagrodia (1999, October). HLA support in a discrete event simulation language. In *Proceedings of the 3rd International Workshop on Distributed Interactive Simulation and Real-Time Applications (DIS-RT '99)*, College Park, Maryland, pp. 93–100.

Prakash, A. and C. V. Ramamoorthy (1988, June). Hierarchical distributed simulations. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, San Jose, CA, pp. 341–347.

Prakash, A. and R. Subramanian (1991, April). Filter: An algorithm for reducing cascading rollbacks in optimistic distributed simulation. In *Proceedings of the 24th Annual Simulation Symposium*, New Orleans, Louisiana, pp. 123–132.

Preiss, B. R. (1989, March). The Yaddes distributed discrete event simulation specification language and execution environment. In *Proceedings of the SCS Multiconference on Distributed Simulation*, Tampa, FL, pp. 139–144.

Preiss, B. R., W. M. Loucks, and I. D. Macintyre (1994, July). Effects of the checkpoint interval on time and space in Time Warp. *ACM Transactions on Modeling and Computer Simulation 4*(3), 223–253.

Preiss, B. R., W. M. Loucks, I. D. MacIntyre, and J. Field (1991, January). Null message cancellation in conservative distributed simulation. In *Proceedings of the 1991 SCS Multiconference on Advances in Parallel and Distributed Simulation*, Anaheim, CA, pp. 33–38.

Preiss, B. R. and K. W. C. Wan (1999, January). The Parsimony project: A distributed simulation testbed in Java. In *Proceedings of the 1999 International Conference On Web-Based Modelling & Simulation*, San Francisco, CA, pp. 89–94.

Priese, L. (1978, October). A note on asynchronous cellular automata. *Journal of Computer and System Sciences 17*(2), 237–252.

Radhakrishnan, R., T. J. McBrayer, K. Subramani, M. Chetlur, V. Balakrishnan, and P. A. Wilsey (1996, April). A comparative analysis of various Time Warp algorithms implemented in the WARPED simulation kernel. In *Proceedings of the 29th Annual Simulation Symposium*, New Orleans, LA, pp. 107–116.

Rajan, R. and P. A. Wilsey (1995, April). Dynamically switching between lazy and aggressive cancellation in a Time Warp parallel simulator. In *Proceedings of the 28th Annual Simulation Symposium*, Phoenix, AZ, pp. 22–30.

Reed, D. A. (1994, May). Experimental analysis of parallel systems: Techniques and open problems. In G. Haring and G. Kotsis (Eds.), *Computer Performance Evaluation*, Volume 794 of *LNCS*, pp. 25–51. Berlin: Springer-Verlag.

Reed, D. A., R. A. Aydt, L. DeRose, C. L. Mendes, R. L. Ribler, E. Shaffer, H. Simitci, J. S. Vetter, D. R. Wells, S. Whitmore, and Y. Zhang (1998, June). Performance analysis of parallel systems: Approaches and open problems. In *Proceeding of the Joint Symposium on Parallel Processing (JSPP)*, Nagoya, Japan, pp. 239–256.

Reed, D. A., R. A. Aydt, R. J. Noe, P. C. Roth, K. A. Shields, B. Schwartz, and L. F. Tavera (1993, October). Scalable performance analysis: The Pablo performance analysis environment. In *Proceedings of the Scalable Parallel Libraries Conference*, Starkville, MS, pp. 104–113.

Reed, D. A., R. D. Olson, R. A. Aydt, T. M. Madhyastha, T. Brikett, D. W. Jensen, B. A. A. Nazief, and B. K. Totty (1991). Scalable performance environments for parallel systems. In *Proceedings of the Sixth Distributed Memory Computing Conference*, Portland, OR, pp. 562–569.

Reif, F. (1965). *Fundamentals of Statistical and Thermal Physics*. New York: McGraw-Hill.

Reiher, P., R. Fujimoto, S. Bellenot, and D. Jefferson (1990, January). Cancellation strategies in optimistic execution systems. In *Proceedings of the SCS Multiconference on Distributed Simulation*, San Diego, CA, pp. 112–121.

Reiher, P. L. (1990, December). Parallel simulation using the Time Warp operating system. In *Proceedings of the 1990 Winter Simulation Conference*, New Orleans, LA, pp. 38–45.

Reynolds, Jr., P. F. and P. M. Dickens (1989, March). SPECTRUM: A parallel simulation testbed. In *Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers, and Applications*, Monterey, CA, pp. 865–870.

Rich, D. O. and R. E. Michelsen (1991, December). An assessment of the MODSIM/TWOS parallel simulation environment. In *Proceedings of the 1991 Winter Simulation Conference*, Phoenix, AZ, pp. 509–518.

Rönngren, R. and R. Ayani (1997, April). A comparative study of parallel and sequential priority queue algorithms. *ACM Transactions on Modeling and Computer Simulation 7*(2), 157–209.

Rönngren, R., M. Liljenstam, R. Ayani, and J. Montagnat (1996, May). Transparent incremental state saving in Time Warp parallel discrete event simulation. In *Proceedings of the 10th Workshop on Parallel and Distributed Simulation*, Philadelphia, PN, pp. 70–77.

Ruhl, T., H. E. Bal, K. G. L. R. A. F. Bhoedjang, and G. Benson (1996, August). Experience with a portability layer for implementing parallel programming systems. In *Proceedings of the 1996 International Conference on Parallel and Distributed Processing Techniques and Applications*, Sunnyvale, CA, pp. 1477–1488.

Salmi, M., J. Harju, and J. Porras (1994, October). Computing the parallel potential of event-oriented simulation applications. In *Proceedings of the 6th European Simulation Symposium (ESS'94)*, Volume 1, Istanbul, Turkey, pp. 153–157.

Samadi, B. (1985). *Distributed Simulation, Algorithms and Performance Analysis*. Ph. D. thesis, Computer Science Department, University of California, Los Angeles, CA.

Santoso, J., G. D. van Albada, B. A. A. Nazief, and P. M. A. Sloot (2000, May). Simulating job scheduling for clusters of workstations. In *High Performance Computing and Networking (HPCN Europe 2000)*, Volume 1823 of *LNCS*, pp. 395–406. Springer.

Schoneveld, A., J. F. de Ronde, and P. M. A. Sloot (1997, November/December). On the complexity of task allocation. *Complexity 3*(2), 52–60.

Sevcik, K. C. (1989, May). Characterizations of parallelism in applications and their use in scheduling. *Performance Evaluation Review 17*(1), 171–180.

Sköld, S. and R. Rönngren (1996, December). Event sensitive state saving in Time Warp parallel discrete event simulations. In *Proceedings of the 1996 Winter Simulation Conference*, Coronado, CA, pp. 653–660.

Sloot, P. M. A. and B. J. Overeinder (1999, September). Time-Warped automata: Parallel discrete event simulation of asynchronous CA's. In *Proceedings of the Third International Conference on Parallel Processing and Applied Mathematics*, Kazimierz Dolny, Poland, pp. 43–62.

Sloot, P. M. A., A. Schoneveld, J. F. de Ronde, and J. A. Kaandorp (1997, July). Large scale simulations of complex systems Part I: Conceptual framework. SFI Working Paper 97-07-070, Santa Fe Instute, Sante Fe, NM.

Smarr, L. and C. E. Catlett (1992, June). Metacomputing. *Communications of the ACM 35*(6), 44–52.

Smith, R. D. (1998, December). Essential techniques for military modeling and simulation. In *Proceedings of the 1998 Winter Simulation Conference*, Washington, DC, pp. 805–812.

Sokol, L. M., B. K. Stucky, and V. S. Hwang (1989, August). MTW: A control mechanism for parallel discrete simulation. In *Proceedings of the 1989 International Conference on Parallel Processing*, Volume III, pp. 250–254.

Soliman, H. M. and A. S. Elmaghraby (1998, October). An analytical model for hybrid checkpointing in Time Warp distributed simulation. *IEEE Transactions on Parallel and Distributed Systems 9*(10), 947–951.

Srinivasan, S. and P. F. Reynolds, Jr. (1993, December). Non-interfering GVT computations via asynchronous global reductions. In *Proceedings of the 1993 Winter Simulation Conference*, Los Angeles, CA, pp. 740–749.

Srinivasan, S. and P. F. Reynolds, Jr. (1995, June). Super-criticality revisited. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, Lake Placid, NY, pp. 130–136.

Srinivasan, S. and P. F. Reynolds, Jr. (1998, April). Elastic time. *ACM Transactions on Modeling and Computer Simulation 8*(2), 103–139.

Steinman, J. S. (1992). SPEEDES: A multiple-synchronization environment for parallel discrete-event simulation. *International Journal in Computer Simulation 2*(3), 251–286.

Steinman, J. S. (1993a, May). Breathing Time Warp. In *Proceedings of the 7th Workshop on Parallel and Distributed Simulation*, San Diego, CA, pp. 109–118.

Steinman, J. S. (1993b, December). Incremental state saving in SPEEDES using C++. In *Proceedings of the 1993 Winter Simulation Conference*, Los Angeles, CA, pp. 687–696.

Steinman, J. S., C. A. Lee, L. F. Wilson, and D. M. Nicol (1995, June). Global virtual time and distributed synchronization. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, Lake Placid, NY, pp. 139–148.

Stevens, R., P. Woodward, T. DeFanti, and C. Catlett (1997, November). Form the I-WAY to the National Technology Grid. *Communications of the ACM 40*(11), 50–60.

Stevenson, D. E. (1999, May/June). A critical look at quality in large-scale simulations. *Computing in Science & Engineering 1*(3), 53–63.

Su, W. K. and C. L. Seitz (1989, March). Variant of the Chandy-Misra-Bryant distributed discrete-event simulation algorithm. In *Proceedings of the 1989 SCS Multiconference on Distributed Simulation*, Tampa, FL, pp. 38–43.

Sun, X. H. and L. M. Ni (1993, September). Scalable problems and memory-bounded speedup. *Journal of Parallel and Distributed Computing 19*(1), 27–37.

Sun Microsystems, Inc. (1987). *XDR: External Data Representation Standard*. Sun Microsystems, Inc.

Sunderam, V. S., G. A. Geist, J. Dongarra, and R. Manchek (1994, April). The PVM concurrent computing system: Evolution, experiences, and trends. *Parallel Computing 20*(4), 531–545.

Talia, D. and P. M. A. Sloot (1999, December). Cellular automata: Promise and prospects in computational science. *Future Generation Computer Systems 16*(2-3), v–vii.

Tay, S. C., Y. M. Teo, and S. T. Kong (1997, June). Speculative parallel simulation with an adaptive throttle scheme. In *Proceedings of the 11th Workshop on Parallel and Distributed Simulation*, Lockenhaus, Austria, pp. 116–123.

Tsitsiklis, J. N. (1989, May). On the use of random numbers in asynchronous simulation via rollback. *Information Processing Letters 31*(3), 139–144.

Turcotte, D. L. (1999, October). Self-organized criticality. *Reports on Progress in Physics 62*(10), 1377–1429.

Turner, S. J. and M. Q. Xu (1992, January). Performance evaluation of the Bounded Time Warp algorithm. In *Proceedings of the 6th Workshop on Parallel and Distributed Simulation*, Newport Beach, CA, pp. 117–126.

Ulam, S. (1970). Some mathematical problems connected with patterns of growth figures. In A. W. Burks (Ed.), *Essays on Cellular Automata*. Champaign, IL: University of Illinois Press.

Unger, B. W. and J. G. Cleary (1993). Practical parallel discrete event simulation. *ORSA Journal on Computing 5*(3), 242–244.

Unger, B. W., J. G. Cleary, A. Convington, and D. West (1993, December). An external state management system for optimistic parallel simulation. In *Proceedings of the 1993 Winter Simulation Conference*, Los Angeles, CA, pp. 750–755.

van Albada, G. D., J. Clinckemaillie, A. H. L. Emmen, J. Gehring, O. Heinz, F. van der Linden, B. J. Overeinder, and P. M. A. Sloot (1999, April). Dynamite—Blasting obstacles to parallel cluster computing. In *High Perfromance Computing and Networking (HPCN Europe 1999)*, Volume 1593 of *LNCS*, pp. 300–310. Springer.

van den Brink, J. (1997, May). Performance metrics in parallel simulation. Master's thesis, Department of Computer Science, University of Amsterdam, Amsterdam, The Netherlands.

van Halderen, B. A. W. and B. J. Overeinder (1998, September–November). Fornax: Web-based distributed discrete event simulation in Java. *Concurrency: Practice & Experience 10*(11-13), 957–970.

van Halderen, B. A. W., B. J. Overeinder, and P. M. A. Sloot (1998, June). Using Java for distributed discrete event simulation. In B. M. ter Haar Romeny, D. H. J. Epema, J. F. M. Tonino, and A. A. Wolters (Eds.), *Proceedings of the Fourth Annual Conference of the Advanced School for Computing and Imaging*, Lommel, Belgium, pp. 66–72.

van Halderen, B. A. W., B. J. Overeinder, P. M. A. Sloot, R. van Dantzig, D. H. J. Epema, and M. Livny (1998, November). Hierarchical resource management in the Polder metacomputing initiative. *Parallel Computing 24*(12/13), 1807–1825.

Veen, A. H. (1986, December). Dataflow machine architectures. *ACM Computing Surveys 18*(4), 365–396.

Vesseur, J. J. J., R. N. Heederik, B. J. Overeinder, and P. M. A. Sloot (1995). Experiments in dynamic load balancing for parallel cluster computing. In P. Fritzon and L. Finmo (Eds.), *Proceedings of the Workshop on Parallel Programming and Computation (ZEUS'95) and the 4th Nordic Transputer Conference (NTUG'95)*, Linköping, Sweden, pp. 189–194.

Vetter, R. J. (1995, February). ATM concepts, architecture, and protocols. *Communications of the ACM 38*(5), 31–38.

von Neumann, J. (1966). *Theory of Self-Reproducing Automata*. Champaign, IL: University of Illinois Press.

Warren, M. S., T. C. Germann, P. S. Lomdahl, D. M. Beazley, and J. K. Salmon (1998, November). Avalon: An Alpha/Linux cluster achieves 10 Gflops for $150k. In *Proceedings of Supercomputing '98: High Performance Networking and Computing Conference*, Orlando, FL.

Warren, M. S., M. P. Goda, D. J. Becker, J. K. Salmon, and T. Sterling (1997, June). Parallel supercomputing with commodity components. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97)*, Las Vegas, NV, pp. 1372–1381.

Weissman, J. B. and A. S. Grimshaw (1996, August). A federated model for scheduling in wide-area systems. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, Syracuse, NY, pp. 542–550.

West, D. and K. Panesar (1996, May). Automatic incremental state saving. In *Proceedings of the 10th Workshop on Parallel and Distributed Simulation*, Philadelphia, PN, pp. 78–85.

White, S., A. Ålund, and V. S. Sunderam (1995, April). Performance of the NAS parallel benchmarks on PVM-based networks. *Journal of Parallel and Distributed Computing 26*(1), 61–71.

Wieland, F. (1997, June). The threshold of event simultaneity. In *Proceedings of the 11th Workshop on Parallel and Distributed Simulation*, Lockenhaus, Austria, pp. 56–59.

Wieland, F. (1998, December). Parallel simulation for aviation applications. In *Proceedings of the 1998 Winter Simulation Conference*, Washington, DC, pp. 1191–1198.

Wong, Y.-C., S.-Y. Hwang, and J. Y.-B. Lin (1995, June). A parallelism analyzer for conservative parallel simulation. *IEEE Transactions on Parallel and Distributed Systems 6*(6), 628–638.

Wonnacott, P. and D. Bruce (1996, May). The APOSTLE simulation language: Granularity control and performance data. In *Proceedings of the 10th Workshop on Parallel and Distributed Simulation*, Philadelphia, PN, pp. 114– 123.

Yen, S. H. C., D. H. C. Du, and S. Ghanta (1989, June). Efficient algorithms for extracting the k most critical paths in timing analysis. In *The 26th ACM/IEEE Design Automation Conference*, 649-654.

Yuan, J., Y. Ren, and X. Shan (2000, February). Self-organized criticality in a computer network model. *Physical Review E 61*(2), 1067–1071.

Zeigler, B. P., H. Praehofer, and T. G. Kim (2000). *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems* (2nd ed.). San Diego, CA: Academic Press.

Publications

Overeinder, B. J., L. O. Hertzberger, and P. M. A. Sloot (1991, May). Parallel discrete event simulation. In *Proceedings of the Third Workshop Computer Systems*, Eindhoven, The Netherlands, pp. 19–30.

Overeinder, B. J., P. M. A. Sloot, and L. O. Hertzberger (1992, September). Time Warp on a Transputer platform: Pilot study with asynchronous cellular automata. In *Parallel Computing and Transputer Applications*, Barcelona, Spain, pp. 1303–1312.

Overeinder, B. J. and P. M. A. Sloot (1993, October). Application of Time Warp to parallel simulations with asynchronous cellular automata. In *Proceedings of the 1993 European Simulation Symposium*, Delft, The Netherlands, pp. 397–402.

Overeinder, B. J., P. M. A. Sloot, and J. Petersen (1994, October). Finalization report: Homogeneous PVM/PARIX. Technical Report CAMAS-TR-2.3.4, University of Amsterdam, Amsterdam, The Netherlands.

Overeinder, B. J., J. J. J. Vesseur, F. v/d Linden, and P. M. A. Sloot (1995, May). A communication kernel for parallel programming support on a massively parallel processor system. In *Proceedings of the Workshop on Parallel Programming and Computation (ZEUS'95) and the 4th Nordic Transputer Conference (NTUG'95)*, Linkøping, Sweden, pp. 259–266.

Vesseur, J. J. J., R. N. Heederik, B. J. Overeinder, and P. M. A. Sloot (1995, May). Experiments in dynamic load balancing for parallel cluster computing. In *Proceedings of the Workshop on Parallel Programming and Computation (ZEUS'95) and the 4th Nordic Transputer Conference (NTUG'95)*, Linkøping, Sweden, pp. 189–194.

Overeinder, B. J. and P. M. A. Sloot (1995, May). Parallel performance evaluation through critical path analysis. In *High-Performance Computing and Networking (HPCN Europe '95)*, Number 919 in LNCS, pp. 634–639. Springer-Verlag.

Overeinder, B. J., P. M. A. Sloot, R. N. Heederik, and L. O. Hertzberger (1996, May). A dynamic load balancing system for parallel cluster computing. *Future Generation Computer Systems 12*(1), 101–115.

Overeinder, B. J. and A. G. Hoekstra (1997, June). Performance measurements of a light scattering code on the Parsytec CC: Comparison with the Parsytec PowerXplorer. In *Proceedings of the Third Annual Conference of the Advanced School for Computing and Imaging*, Heijen, The Netherlands, pp. 116–119.

Overeinder, B. J. and P. M. A. Sloot (1997). Breaking the curse of dynamics by task migration: Pilot experiments in the polder metacomputer. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Volume 1332 of *Lecture Notes in Computer Science*, Berlin, pp. 194–207. Springer-Verlag.

van Halderen, B. A. W., B. J. Overeinder, and P. M. A. Sloot (1998, June). Using Java for distributed discrete event simulation. In *Proceedings of the Fourth Annual Conference of the Advanced School for Computing and Imaging*, Lommel, Belgium, pp. 66–72.

van Halderen, B. A. W. and B. J. Overeinder (1998, September–November). Fornax: Web-based distributed discrete event simulation in Java. *Concurrency: Practice & Experience 10*(11–13), 957–970.

van Halderen, B. A. W., B. J. Overeinder, P. M. A. Sloot, R. van Dantzig, D. H. J. Epema, and M. Livny (1998, November). Hierarchical resource management in the Polder metacomputing initiative. *Parallel Computing 24*(12/13), 1807–1825.

Overeinder, B. J. and P. M. A. Sloot (1999, April). Extensions to Time Warp parallel simulation for spatially decomposed applications. In *Proceedings of the Fourth United Kingdom Simulation Society Conference (UKSim 99)*, Cambridge, UK, pp. 67–73.

van Albada, G. D., J. Clinckemaillie, A. H. L. Emmen, J. Gehring, O. Heinz, F. van der Linden, B. J. Overeinder, and P. M. A. Sloot (1999, April). Dynamite—Blasting obstacles to parallel cluster computing. In *Proceedings of the 7th International Conference on High-Perfromance Computing and Networking Europe, HPCN Europe 1999*, Amsterdam, The Netherlands, pp. 300–310.

Sloot, P. M. A. and B. J. Overeinder (1999, September). Time-warped automata: Parallel discrete event simulation of asynchronous CA's. In *Proceedings of the Third International Conference on Parallel Processing and Applied Mathematics*, Kazimierz Dolny, Poland, pp. 43–62.

Sloot, P. M. A., J. A. Kaandorp, A. G. Hoekstra, and B. J. Overeinder (1999, November). Distributed simulation with cellular automata: Architecture and applications. In *Proceedings of the 26th Conference on Current Trends in Theory and Practice of Informatics (SOFSEM'99)*, Milovy, Czech Republic, pp. 203–249.

Iskra, K., Z. W. Hendrikse, G. D. van Albada, B. J. Overeinder, and P. M. A. Sloot (2000a, April). Experiments with migration of PVM tasks. In *Proceeding of ISThmus 2000 Conference: Research and Development for the Information Society*, Poznan, Poland, pp. 295–304.

Iskra, K. A., F. van der Linden, Z. W. Hendrikse, B. J. Overeinder, G. D. van Albada, and P. M. A. Sloot (2000, July). The implementation of Dynamite: An environment for migrating PVM tasks. *Operating Systems Review 34*(3), 40–55.

Iskra, K. A., Z. W. Hendrikse, G. D. van Albada, B. J. Overeinder, and P. M. A. Sloot (2000b, September). Performance measurements on Dynamite/DPVM. In *Recent Advances in PVM and MPI: Proceedings of the 7th European PVM/MPI User's Group Meeting*, Balatonfüred, Hungary, pp. 27–38.

Iskra, K. A., Z. W. Hendrikse, G. D. van Albada, B. J. Overeinder, P. M. A. Sloot, and J. Gehring (2000, December). Experiments with migration of message-passing tasks. In *Proceedings of GRID'2000: International Workshop on Grid Computing*, Bangalore, India. In press.

Sloot, P. M. A., J. A. Kaandorp, A. G. Hoekstra, and B. J. Overeinder (2000). Distributed cellular automata: Large scale simulation of natural phenomena. In A. Zomaya (Ed.), *Solutions to Parallel and Distributed Computing Problems: Lessons from Biological Sciences*. New York: Wiley. In press.

Overeinder, B. J., A. Schoneveld, and P. M. A. Sloot (2000). Self-organized criticality in optimistic simulation of correlated systems. *Journal of Parallel and Distributed Computing*. Submitted.

Dutch Summary
Nederlandse Samenvatting

Simuleren is een alledaagse activiteit die de laatste decennia niet meer weg te denken is in onze maatschappij. Simuleren omvat een scala van activiteiten dat gericht is op het verkrijgen van meer inzicht in het systeem dat onderwerp van studie is. Het bestudeerde systeem kan variëren van de aerodynamische eigenschappen van een vliegtuig in ontwikkeling, tot de eigenaardigheden van de dynamica van een proteïne die zich opvouwt in een complexe, drie-dimensionale structuur. Of het bestudeerde systeem is een organisatie of gemeenschap, zoals bijvoorbeeld efficiënt voorraadbeheer van een bedrijf, of de verspreiding van HIV en hepatitis C bij injecterende drugsgebruikers.

Alhoewel de onderzoekdisciplines verschillen, blijft de aanpak van een simulatiestudie min of meer identiek. Een model van een bestaand of theoretisch fysisch systeem wordt ontworpen, experimenten met het model worden uitgevoerd, en de resultaten van de experimenten worden geanalyseerd. Modellen van een fysisch systeem komen voor in alle mogelijke vormen: het meest exacte model is het fysisch systeem zelf, een in grootte geschaald model van het fysisch systeem, of een abstract model beschreven door middel van wiskunde of een formele logische taal. In het algemeen, worden abstracte modellen gerealiseerd in computerprogramma's, ook wel computersimulaties genoemd, zodat de experimenten met het abstracte model uitgevoerd kunnen worden op computers. Het onderwerp van dit proefschrift is computersimulatie en in het bijzonder de efficiënte computersimulatie van systemen, die gekarakteriseerd worden door een heterogeen spatiëel en temporeel gedrag, hetgeen betekent dat veranderingen in het systeem op wisselende plaatsen en op variërende tijden plaats vinden. Systemen met dit gedrag komen algemeen voor in, bijvoorbeeld, populatie dynamica, immunologie, statistische natuurkunde, en informatica. Systemen met heterogeen spatiëel en temporeel gedrag worden het meest nauwkeurig op zogenaamde asynchrone modellen afgebeeld.

Experimenteren met complexe simulaties en de interpretatie van grote hoeveelheden gegenereerde data is een uitdagende onderneming. In dit opzicht worden virtuele omgevingen, die de visualisatie van data en de interactie met de computersimulatie integreren, steeds belangrijker. Een goed voorbeeld is een virtueel laboratorium, waar wetenschappers interactief met een simulatie experimenteren. Als de wetenschappers een wijziging in het simulatiemodel aanbrengen, moet de computersimulatie stop gezet worden, of zelfs terugge-

bracht worden naar een situatie in het (simulatie-) verleden, zodat de wijzigingen doorgevoerd kunnen worden.

De combinatie van complexe simulaties, asynchrone modellen en virtuele omgevingen resulteert in grote applicaties, die voldoende computercapaciteit eisen om interactieve experimentatie mogelijk te maken. Dergelijke vereisten zijn zelfs met de snelste supercomputers niet eenvoudig te realiseren (ook niet door het gedistribueerde karakter van de virtuele laboratorium toepassing). Gedistribueerde en parallelle oplossingsmethoden zijn een passend alternatief voor de virtuele laboratorium toepassing, waar de verzamelde computercapaciteiten gebundeld worden om de benodigde prestatie te leveren.

In onze studie naar gedistribueerde en parallelle oplossingsmethoden, exploiteren we de lokaliteit van data en verwerking van gebeurtenissen met behulp van technieken uit parallelle discrete event simulatie. De verdeling van werk over de gedistribueerde computers wordt gecoördineerd door het Dynamite executiesysteem. De gecombineerde parallelle verwerking van simulatiegebeurtenissen en de verdeling van werk over de gedistribueerde computers is een zeer complexe taak. Daarom zijn de twee (sub-) problemen onafhankelijk van elkaar bestudeerd, maar wel zodanig dat beide componenten geïntegreerd kunnen worden in een omgeving.

Het ontwerp en implementatie van een simulatie-omgeving voor parallelle discrete event simulatie heeft geresulteerd in het APSIS systeem, beschreven in Hoofstuk 3. De kern van APSIS is het Time Warp optimistische simulatiemechanisme. De belangrijkste verantwoordelijkheid van een simulatiemechanisme is de handhaving van causaliteit, d.i. oorzaak en gevolg. Op een sequentiële computer is dit redelijk eenvoudig te realiseren door de gebeurtenissen in de simulatie op toenemende simulatietijd te sorteren en steeds de gebeurtenis met de eerste simulatietijd te selecteren (de voorste in de rij van gesorteerde gebeurtenissen). Met de parallelle executie van asynchrone simulatiemodellen ligt dit anders. Hier worden gelijktijdig meerdere gebeurtenissen verwerkt door verschillende computers. Het bijhouden van één gesorteerde lijst met gebeurtenissen is niet effectief voor de klasse van asynchrone simulatiemodellen: de parallelle processoren zouden het merendeel van de tijd op elkaar moeten wachten eer zij een gebeurtenis kunnen verwerken. Bij de parallelle executie van asynchrone simulatiemodellen wordt de lijst met simulatiegebeurtenissen over de computers verspreid om zo de lokaliteit van data en verwerking van gebeurtenissen zoveel mogelijk uit te buiten. Maar omdat iedere computer zijn eigen gesorteerde lijst met gebeurtenissen heeft, kan het voorkomen dat de causaliteit in de simulatie geschonden wordt. Bijvoorbeeld: als computer A tijdens de simulatie een nieuwe gebeurtenis met simulatietijd 15 bij computer B wil plaatsen, kan het voorkomen dat computer B al een gebeurtenis met simulatietijd 20 aan het verwerken is. Als een dergelijke causaliteitsfout optreedt, wordt het zogenaamde “rollback and recovery” mechanisme in werking gezet. In het voorbeeld zou computer B zijn simulatietijd terugzetten naar de laatste gebeurtenis direct voor de nieuwe gebeurtenis met simulatietijd 15 (“rollback”) en herstelt vervolgens alle veranderingen in de simulatie ten gevolge van de voorbarige verwerking van gebeurtenissen tussen simulatietijd 15 en 20 (“re-

covery").

De Time Warp methode moet regelmatig de status van de simulatie bewaren om zich te kunnen herstellen van causaliteitsfouten. Bij grote systemen kan de volledige status van de simulaties zeer groot worden en het volledig geheugen in beslag nemen. Wij hebben een meer efficiënte methode geïntroduceerd die alleen de wijzigingen in de status bewaard. Deze incrementele methode is efficiënter in geheugen gebruik, maar tijdens de "recovery" fase moet de oude status van de simulatie stap voor stap gereconstrueerd worden. Verder wordt de APSIS omgeving gecompleteerd door de APSE parallellisme analyse methodiek, zie Hoofdstuk 4. De APSE parallellisme analyse is een waardevolle aanvulling op de APSIS simulatie omgeving, omdat het inzicht geeft in het intrinsieke parallellisme van het simulatiemodel en het parallelisme dat gerealiseerd is door APSIS. Daarbij kan door middel van kritieke pad analyse, de obstakels geïdentificeerd worden die het parallellelisme in de simulatie limiteren.

De toepasbaarheid van de Time Warp simulatie-omgeving is geëvalueerd in Hoofdstuk 5. De applicatie is een Ising spin systeem, waarmee de magnetisatie van ferro-metalen bestudeerd kan worden. Ising spin systemen zijn een prototype voor een klasse van applicaties in statistische fysica. De experimenten en de APSE parallellisme analyse geven aan dat de APSIS omgeving in staat is de simulatiegebeurtenissen efficiënt te verwerken over de parallelle processoren. De experimenten tonen ook aan dat er een mate van optimisme controle nodig is. In Time Warp kunnen twee processoren een willekeurige simulatietijd uit elkaar lopen, bijvoorbeeld: simulatietijd 10 op processor A en simulatietijd 2040 op processor B. Als nu een nieuwe gebeurtenis met simulatietijd 15 door processor B verwerkt moet worden, moet processor B een "rollback en recovery" actie van simulatietijd 2040 naar simulatietijd 15 maken. Dergelijke lange rollbacks vereisen substantiële computerrekentijden en kunnen zich herhaaldelijk voordoen. Dit kan voorkomen worden door het beperken van het optimisme in de simulatie, bijvoorbeeld door een simulatietijdsinterval vast te stellen waarbinnen gebeurtenissen verwerkt mogen worden. In ons vorig voorbeeld met simulatietijd 10 op processor A en een simulatietijdsinterval van 100, zou processor B zijn gebeurtenissen niet verder dan simulatietijd 110 mogen verwerken.

De effectiviteit van het optimisme controle mechanisme wordt aangetoond in Hoofdstuk 6. In de Ising spin systeem simulatie treden (onverwacht) lange rollback series op rondom de fasetransitie van magneet naar paramagneet. De lengte en frequentie van de rollback series kunnen binnen grenzen gehouden worden door het optimisme controle mechanisme, resulterend in kortere responsietijden. De niet-triviale interferentie tussen Ising spin systeem applicatie en Time Warp methode is mogelijk zelforganiserend kritisch gedrag, waarbij de complexiteit van de berekingen en de complexiteit van de fysica op een niet-lineaire wijze met elkaar vermengd zijn.

De Dynamite omgeving is een experimenteel platform voor onze ideeën betreffende dynamische werkverdeling van rekentaken (processen) over de gedistribueerde of parallelle processoren. In Hoofdstuk 7 beschrijven we het onder-

liggende principe van proces migratie, waarbij de executie van een programma gestopt wordt, verplaatst naar een andere processor en weer opgestart wordt. Een complicerende factor is communicatie tussen parallelle processen. De proces migratie moet transparant zijn, zodat de andere parallelle processen geen notie hebben van de gewijzigde situatie. De transparante proces migratie in Dynamite is geëvalueerd aan de hand van een aantal experimenten met verschillende test applicaties en twee simulatie applicaties. De resultaten van de experimenten tonen de effectiviteit van dynamische proces migratie in Dynamite aan.

In de toekomst worden de APSIS en Dynamite omgeving met elkaar geïntegreerd. Het belangrijkste onderzoeksonderwerp bij de integratie van APSIS met de Dynamite omgeving, is de werkverdelings strategie voor optimistische simulaties over de parallelle processoren. In optimistische simulatie is er een verschil tussen rekenwerk en de het begrip voortgang. Bijvoorbeeld, “rollback en recovery” acties worden als rekenwerk aangemerkt, maar dragen niet bij aan de voortgang van de simulatie. Een van de belangrijkste onderzoeksvragen zal zijn: het vinden van een compacte beschrijving van het executiegedrag van optimistische simulaties en daarop gebasseeerd, een effectieve strategie om werk te verdelen over de parallelle processoren.

Nawoord

Promotieonderzoek doe je alleen. In principe dan. Met een begeleider natuurlijk. Maar zonder al mijn collega's zou dit proefschrift niet in zijn huidige vorm verschenen zijn. De veelheid aan onderwerpen die ik bestudeerd heb, is zeker een afspiegeling van de verschillende onderzoeksgebieden van dezelfde collega's. En verder heb ik het natuurlijk ook getroffen met mijn collega's in, laten we zeggen, sociaal opzicht.

Ten eerste wil ik Peter Sloot bedanken voor zijn inzet en begeleiding en voor het wegwijs maken van mij in de "wetenschap." Wat wist ik als net afgestudeerde tenslotte van de wetenschappelijke methode. Het enthousiasme van Peter over zijn grootste hobby, namelijk het bedrijven van wetenschap, werkt aanstekelijk. Onze gesprekken daarover, boven een pot bier, zijn mij zeer waardevol.

Next, I would like to thank Miron Livny for his involvement in my PhD. research. Although Miron is a strong advocate for high throughput computing and my research is mainly about high performance computing, he promptly agreed to become my co-promoter. I will remember Miron saying: "Forget about high performance, it gives you factors. High throughput gives you orders more computing power."

Gedurende de laatste periode van mijn onderzoek heb ik met veel plezier samengewerkt met Arjen Schoneveld. Samen hebben we ideeën uit onze onderzoeksgebieden gecombineerd wat heel leuke resultaten heeft opgeleverd. Verder ben ik ook Zeger Hendrikse, Dick van Albada en vooral Kamil Iskra dank verschuldigd. De samenwerking heeft geresulteerd in de mooie Dynamite resultaten in mijn proefschrift.

Andere helden uit het verleden zijn de drie musketiers Joep Vesseur, Frank van der Linden en Berry van Halderen. Samen met Joep en Frank heb ik menig doorwaakte nacht aan onze "commerciële" producten .....PVM en .....MPI gewerkt. (Frank heeft daarbij ook nog aan DPVM bijgedragen.) Met Berry heb ik samengewerkt aan ondermeer het Polder metacomputer project en aan het succesvolle Fornax artikel. Maar even belangrijk is hun inzet geweest om na werktijd met z'n allen ergens een biertje te pakken.

Ook wil ik de doctoraalstudenten bedanken die met hun afstudeerproject een bijdrage hebben geleverd aan mijn onderzoek. Robbert Heederik voor een van de eerste versies van DPVM, Joost van den Brink voor zijn werk aan APSE en Mostapha al Mourabit voor de APSIS/MPI communicatielaag en performance analyse.

Bob Hertzberger mag hier niet onvermeld blijven. Ik ben hem veel verschuldigd. Indertijd heeft Bob mij hier, toen nog Faculteit Wiskunde en Informatica, een promotieplaats aangeboden. Halverwege mijn promotietraject heeft Bob er ook voor gezorgd dat ik bij de faculteit kon blijven toen ik op de hielen werd gezeten door de dienstplicht. In eigen persoon heeft Bob de luitenant-kolonel laten weten waar het op stond als ik die volgende maandag zou moeten aantreden bij de genietroepen. Niet lang daarna is de dienstplicht afgeschaft...

De andere collega's waar ik niet zozeer intensief mee heb samengewerkt zijn wellicht net zo belangrijk voor mij geweest. Jan de Ronde, jarenlang kamergenoot, met een groot rechtvaardigheidsgevoel en een groot Ajax-hart. Martin Bergman met een groots gevoel voor humor. Drona Kandhai, onze junior, met een grote levenswijsheid. Diederik Burer met een groot inzicht. En Rob Belleman, huidig kamergenoot en deejay, met een groot gevoel voor muziek en gezelligheid.

Andere oudgedienden van het eerste uur zijn Alfons en Jaap. Op onze reis naar Indonesië hebben Alfons en ik in één maand een reputatie opgebouwd in de snookerlokalen van Bandung, waarachtig; en misschien ook op het ITB. En Jaap, de man die alle 1500 plantjes tijdens onze Ardennen Avonturen weet te determineren, en die 's ochtends (de morning-after na onze Ardennen Avonduren) alle sardientjes in tomatensaus verorbert en daarna fluitend rondloopt met een rugzak van 30 kg (verrekijker, planten- en dierengidsjes, blikken sardientjes in tomatensaus en verder de halve inhoud van Peter z'n rugzak). Verder wil ik Piero bedanken voor onze bioscoopavondjes, David D. als liefhebber van het betere Nederlandse levenslied en Roeland voor de gezellige thee onderbrekingen tijdens het werk. Andere collega's waarmee ik veel lol heb gehad tijdens conferenties, workshops en Ardennen Avonturen zijn Jeroen, Andy, Marcel B. en Michiel M.

Ook de systeemgroep ben ik dank verschuldigd, met name Jan W. en Gert. Ons netwerk met computers wordt perfect beheerd. Het secretariaat: Virginie, Erik, Saskia, Jacqueline, Monique en Hugo, wil ik bedanken voor hun medewerking en ondersteuning. Ook wil ik hier Laura en Ina vermelden.

Een bijzondere plaats neemt Oscar Meezen in. Hij is jarenlang mijn hardwaredealer geweest, die me PC systemen van allerlei signatuur toegeschoven heeft. Terwijl Nederland en masse overstapte naar Pentium systemen, omdat het anders ontstoken zou blijven van de geneugten van Window$95, maakte mijn 50 MHz 486DX overuren. Draaide perfect onder Linux, geen probleem. Vervolgens werd m'n systeempje opgewaardeerd met extra geheugen tot 32 MB en later met een 100 MHz Pentium. Nu staat er een tweede PC naast met een 133 MHz Cyrix 166+ processor. Alhoewel de GigaHz machines nu over de toonbank geschoven worden, voldoet mijn PC'tje perfect aan de wensen van een wetenschapper die iets met computers doet. Oscar bedankt!

Tot slot wil ik mijn ouders bedanken voor alles. Dat woordje "alles" is makkelijk opgeschreven, maar het is teveel om hier een opsomming te geven en het is zelfs onmogelijk om volledig te zijn. Hun steun en vertrouwen zijn erg belangrijk voor mij geweest.

En dan Daantje: bedankt voor je steun en liefde.

Index