Time Warp
from Cluster to Grid

Time Warp from Cluster to Grid

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam
op gezag van de Rector Magnificus
prof. mr. P. F. van der Heijden
ten overstaan van een door het college voor promoties ingestelde
commissie, in het openbaar te verdedigen in de Aula der Universiteit
op woensdag 8 juni 2005, te 10:00 uur

door

Kamil Antoni Iskra

geboren te Krakau, Polen

Promotiecommissie:

Promotor:	prof. dr. P. M. A. Sloot
Co-promotor:	dr. G. D. van Albada
Overige leden:	prof. drs. M. Boasson dr. ir. M. Bubak prof. dr. R. J. Meijer dr. B. J. Overeinder dr. S. J. Turner
Faculteit:	Faculteit der Natuurwetenschappen, Wiskunde en Informatica

Section Computational Science

Universiteit van Amsterdam

The work described in this thesis has been carried out in the Section Computational Science of the University of Amsterdam, with financial support of:

- University of Amsterdam,
- *Virtual Laboratory for e-Science* project,
- Netherlands Organisation for Scientific Research – *Massively Parallel Programming* programme (NWO–MPR),
- European Union – ESPRIT project 23499 *Dynamite*.

This work has been carried out in the ASCI graduate school.
ASCI dissertation series number 114.

Advanced School for Computing and Imaging

ISBN 90-5776-139-4

Typeset with $\LaTeX 2_\varepsilon$.
Printed by Ponsen & Looijen, Wageningen.

Author contact: `kamil@iskra.name`

Adm. Kirk: 'Start your computations… for time warp.'

Star Trek IV: The Voyage Home

Contents

1 Introduction

1.1 Modelling

Modelling is a key scientific approach towards a better understanding of reality. A model is a selective reflection of reality. Different kinds of models exist, but in the case of exact sciences, mathematical models are commonly used.

A mathematical model is typically a set of analytical equations providing information on the relevant aspects of a system. For example, the $M/M/1/\infty/\infty$ model, that can represent a queuing system such as a shop with a single shop assistant, provides equations that allow one to calculate the average number of customers in the shop or the average time spent in the shop by a customer.

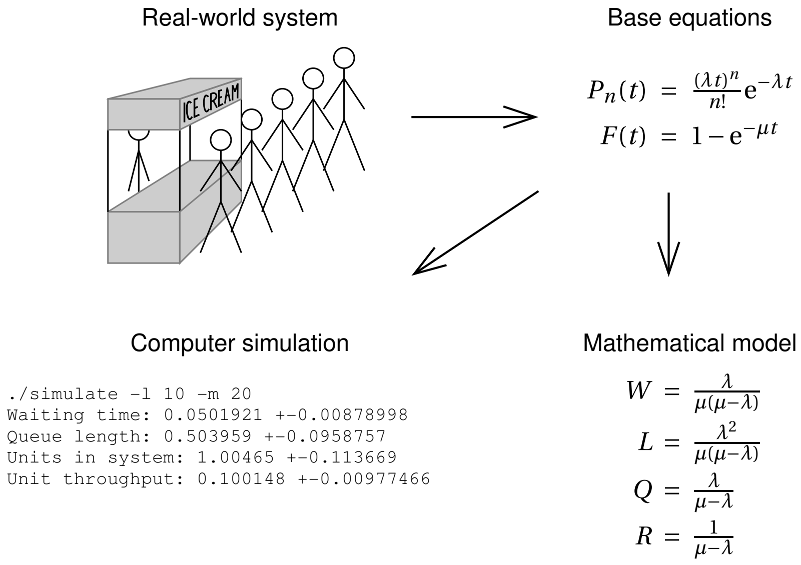
Processes taking place in the real world tend to be immensely complex. The underlying mechanisms are often quite simple, but their nature might be covered by a thick shroud of largely irrelevant detail. A mathematical model only includes the characteristics of the system that are most relevant for the current study, ignoring everything else. The approximation of the real world produced in this way is rather crude in many areas, yet the correlation between the results of the model and the real world can be amazingly good in the few areas that the model was built for. A model must be a simplification of the reality, otherwise it would be too complex to be helpful.

The number of components in many real-world systems presents a problem. Human intuition scales rather badly, and so we quickly lose track of what is going on in even a relatively small system. The phenomenon of *emergent behaviour* does not make things any easier: even if the behaviour of any individual component is well understood, once a large number of them interact, a qualitatively new behaviour can appear, one that is "greater than the sum of all the components" (see e.g. Bak [12] or Wolfram [193]). Human intelligence itself is often given as an example of this phenomenon.

Mathematics sometimes suffers from some of the same problems as human intuition. Gödel's Incompleteness Theorem [80] is the ultimate proof that mathematics has limits, but one does not need to go that far. It is enough to consider the *n-body problem* [172], i.e. the problem of finding the motions of n bodies in their own gravitational field given their initial configuration. Already for as few as three bodies there is no analytical solution; it simply cannot exist. Even if an analytical solution for some problem does exist, finding it might be a daunting task, or the solution found can be so complex that its practical usefulness is limited.

1.2 Simulation

An alternative approach to mathematical modelling is to simulate a system on a computer. The procedure is shown in Fig. 1.1. First, just like with the mathematical mod-



$$P_n(t) = \frac{(\lambda t)^n}{n!}\mathrm{e}^{-\lambda t}$$
$$F(t) = 1 - \mathrm{e}^{-\mu t}$$

```
./simulate -l 10 -m 20
Waiting time: 0.0501921 +-0.00878998
Queue length: 0.503959 +-0.0958757
Units in system: 1.00465 +-0.113669
Unit throughput: 0.100148 +-0.00977466
```

$$W = \frac{\lambda}{\mu(\mu-\lambda)}$$
$$L = \frac{\lambda^2}{\mu(\mu-\lambda)}$$
$$Q = \frac{\lambda}{\mu-\lambda}$$
$$R = \frac{1}{\mu-\lambda}$$

Figure 1.1: From a real-world system to its mathematical model or computer simulation. *Guessing what the weather was like when we drew this picture is left as an exercise for the reader.*

elling, we need to identify the underlying mechanisms controlling the system and represent them in a form convenient for a computer – using mathematical equations. With simulation, the next step is different. When building a mathematical model, we subsequently sit down with a sheet of paper (usually more than one) and analytically transform the base equations into the equations describing the quantities we are really interested in. This is not necessary with a computer simulation, as it uses the base equations to emulate, step by step, the progress of the real system in simulated time, and the quantities we are interested in are obtained through *instrumentation*, i.e. by adding extra code to measure the values of the quantities.

Compared to an analytical model, a simulation has some practical drawbacks. A simulation must be allowed to run for a sufficient amount of time to make sure that the quantities measured, which are usually averages, are statistically valid. Consequently, for larger simulations the computational requirements are often rather high. If a user wants to analyse the behaviour of the system for different sets of input parameters, which is usually the case, these expensive runs must be repeated for each parameter set. Still, given the steady speed improvements in the computer world, these disadvantages become less relevant.

Processes taking place in real-world systems are experienced as being continuous. Computers, on the other hand, are inherently discrete, so continuous behaviour is usually only approximated. A continuous quantity such as time can be simulated using discrete increments – the smaller, the higher the accuracy. This is known as *discrete time simulation*.

However, for many applications the continuous nature of real-world systems is irrelevant, and can be represented by a short series of atomic* *events*. For example, a simple operation of movement can be simulated by just two events: the beginning and end of the movement. The simulation can thus optimise its operation by skipping over the time when the movement takes place, since "nothing interesting" happens then. This is known as *discrete event simulation*.

In practice, a discrete event simulation consists of a number of autonomous components, known as *simulation entities*, that interact with each other by scheduling events. An event contains an identifier of the simulation entity it is addressed to, the (simulated) time when it should be handled by the destination (the *schedule time*), as well as event-specific data. It is possible, and not uncommon, for a simulation entity to schedule an event addressed to itself. Scheduled events are put on a single *event list*, in the *timestamp order*, i.e. the event with the lowest schedule time first. The simulation makes progress by taking the first event off the list, increasing the simulation clock to match the event's timestamp, and handing it over to the destination simulation entity for processing. The entity can perform arbitrary actions, including the scheduling of new events. Once it is done, the next event is taken from the event list and the operation repeats. The simulation terminates when there are no more events on the event list, or when a user-specified condition occurs, such as when a certain number of events have been processed or when a certain value is reached on the simulation clock.

Any system where progress can be represented using a series of atomic operations can be simulated using the discrete event simulation approach, the aforementioned shopping queue included. Other examples of every-day applications are road networks, see Merrifield et al. [125], or air routes, see Wieland [192], typically simulated to test their behaviour under various levels of traffic. Communication networks, in particular mobile networks, are frequently simulated with the help of discrete event simulation, see e.g. Carothers et al. [37], Lin and Fishwick [107], or Yeung et al. [195]. In the case of mobile networks, one can for example test if the density of the base stations, as well as their capacity, is sufficient to support the growing number of customers with a high quality

*atomic: (*jargon*) indivisible, cannot be split up (The Free On-line Dictionary of Computing [85]).

of service. Simulation is frequently used in high-technology research and design, from the digital logic simulation of VLSI circuits, see Bailey et al. [11], through the simulation of computer systems, see e.g. Chandrasekaran and Hill [40], to the simulation of wide area computer networks, including the simulations of the whole Internet by Cowie et al. [47] or Szymanski et al. [178]. In scientific computing, discrete event simulation can be used in various biological or physical systems, for example the colliding pucks problem of Hontalas et al. [84], models of magnetisation studied by Lubachevsky [118], more generally in asynchronous cellular automata, see e.g. Overeinder et al. [136, 167], or prey–predator models such as the sharks world of Conklin et al. [46].

1.3 Parallel discrete event simulation

1.3.1 Parallelisation

Simulation systems mentioned above can be very large, consisting of many thousands of simulation entities. The significant computing resources required to execute them might not be readily available on a single computer system.

Parallelisation is normally used to address such problems. Nowadays, the Single Program Multiple Data (SPMD) programming paradigm is most commonly used, running on top of a Multiple Instruction Multiple Data (MIMD in Flynn's Taxonomy [62]) parallel architecture. An application is split into a number of autonomous components, that are subsequently mapped onto separate processors and run in parallel. Provided that *spatial decomposition* is used, each of the components will be smaller than the original monolithic application, so it can more easily fit in the memory available on the individual processor. More importantly, the amount of work per processor also decreases, and given that the processes all run in parallel, a *speedup* can be achieved. The speedup, defined as a ratio between the run times of sequential and parallel versions of the application, can in principle be as high as the number of processors (or even higher under some particularly favourable circumstances). In practice, the speedup is limited by the fact that the components running on separate processors must cooperate with each other in order to synchronise their actions and to achieve a common result. Various interprocess communication techniques can be used to implement this cooperation, and the exact amount of cooperation required is application dependent. More formally, Amdahl's Law [6] states that every algorithm has a non-parallelisable component that, assuming a fixed problem size, becomes dominant with a sufficiently large number of processes and limits the speedup. Applications that parallelise best (that are most *scalable*) are known as *embarrassingly parallel*.

Unfortunately, discrete event simulation is not one of them. Decomposing the simulation problem is easy: it is naturally divided into the simulation entities, which can be mapped onto separate processes, known in parallel simulation as *logical processes* (LPs). However, splitting the underlying infrastructure, the so-called *simulation kernel*, is hard. In a simulation, the events are normally processed in the timestamp order – that implies a centralised event list. But this is impractical: even at its most efficient – on a shared memory parallel machine – such a list would quickly become a bottleneck, since, as a *critical section* modified frequently by multiple processes, it would have to be protected with exclusive locks. The scheme would be totally unrealistic on a distributed memory machine, where the processes can only communicate via messages. Yet this is the platform that we are primarily interested in. Using a shared event queue, the whole system would essentially become serialised, with little hope of achieving any speedup. Therefore, the event list must be distributed. Also the simulation time clock is distributed, and each logical process maintains its own copy. Under these circumstances, how can we achieve the same result as a sequential simulation, and with a good performance? This is known as the *synchronisation problem.* We must ensure that the *causality constraint* is maintained: an event must not be processed before any events that could influence it have been processed (more accurately, the event must not be *committed,* see Sec. 1.3.3).

Figure 1.2 (*left*) presents an example of a *causality error,* in the form of a snapshot

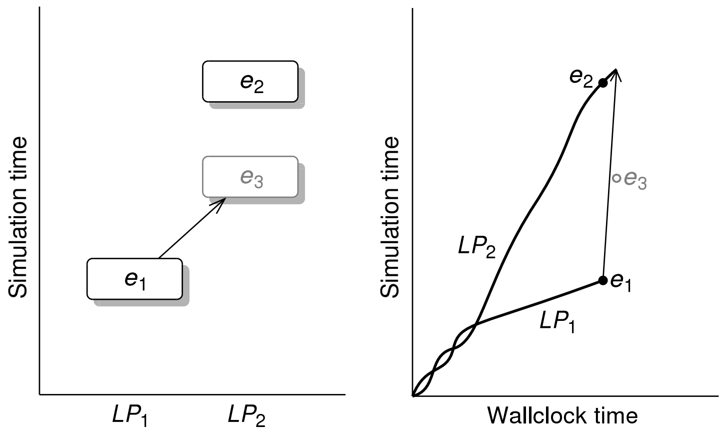


Figure 1.2: A potential causality error: a snapshot at a particular wallclock time (*left*) and a diagram showing the simulation's evolution towards this state (*right*).

taken at a particular real time (often referred to as *wallclock* time). At that time, process LP_2 processing event e_2 is more advanced in the simulation time than process LP_1 processing event e_1. Processing event e_1 results in the scheduling of event e_3 on process LP_2. However, the timestamp of the new event is lower than that of event e_2 currently being processed on LP_2. Thus, e_3 should have been processed before e_2, but it has not – this is a causality error. Figure 1.2 (*right*) presents the situation in a different form: here, the evolution of the simulation in the wallclock time can be observed. At the beginning, the two processes progress at a similar pace, but then LP_2 accelerates, while LP_1 slows down. Eventually, LP_1 schedules event e_3 on LP_2 (the vertical arrow); the timestamp of the new event is marked with the hollow circle. We will be using this form of presenting the progress of simulation processes in many places further in this thesis.

The causality constraint will be satisfied if total timestamp processing order is enforced. However, a weaker, partial order can suffice as well. If two events are independent, they can be processed in any order, irrespective of their timestamps, so if they are on two different logical processes, they can be processed in parallel without any further synchronisation steps. The more such events there are, the higher the *degree of parallelism* in the system, and consequently the higher the potential speedup.

The literature on the subject of parallelising discrete event simulation kernels is extensive. Comprehensive summaries include the work of Fujimoto [72, 74], Nicol and Fujimoto [132], Lin and Fishwick [107], and Ferscha [60]. As Reynolds [156] pointed out, there exists a whole spectrum of options for parallel simulations. Only a part of it has been researched, and in practice parallel simulation algorithms are divided into two main approaches: *conservative* and *optimistic.*

1.3.2 Conservative approach

The *conservative approach* to Parallel Discrete Event Simulation (PDES), proposed independently by Chandy and Misra [41] and Bryant [28], works according to the principle signalled above. All the pairs of processes that might need to exchange events must be declared at the outset, creating a graph of communication channels used. If there is no channel between two logical processes, then there is no need to consider any event dependencies between those processes. If a channel from process A to process B exists, then an event on process B with a timestamp t is assumed to be dependent on all events scheduled by process A on process B with a timestamp lower than t. Every logical process must process events strictly in the time stamp order. This is known as the *local causality constraint*, since it applies to each logical process separately, only enforcing order on the events addressed to that process. To satisfy this constraint, the simulation kernel may only process an event with timestamp t if it can prove that no event with a lower timestamp can arrive from another logical process. An assumption commonly made is that the events arriving via any given communication channel have a non-decreasing timestamp (this does limit the application domain to some extent). Thus, if events with a timestamp greater than or equal to t have been received via all the incoming channels, the event with timestamp t is *safe* and can be processed immediately. Otherwise, the logical process must wait, because an event with a lower timestamp than t could still arrive, and it would have to be processed first. It is because of this extreme care in ensuring that the events are processed in the right order, that the algorithm is called “conservative”. This conservative idle waiting obviously decreases the speedup, but it can also lead to a deadlock. Namely, a cycle of logical processes can spontaneously develop, where each process has at least one “unsafe” input channel and must thus wait for an event from its partner, which is waiting as well. The deadlock can be avoided using the null message algorithm: after processing an event, a logical process sends a null message (a message not accompanied by an event) to all its partners. The message contains the minimum timestamp of the next event that the process could send to its parters. This timestamp is the sum of the current local simulation time at the source and of a quantity called *lookahead.* The lookahead is the minimum increase in the simulation time for newly scheduled events – its value is simulation dependent, but given the finite speed of the spread of information in the real-world systems, it is normally greater than zero. Thanks to the null messages, the processes are up to date with the state of their partners, and at least one safe event can always be identified, thus avoiding the deadlock. However, deadlock will not be avoided if there exists a cycle of processes with a zero lookahead. Generally, a high lookahead value is a prerequisite for obtaining good performance with conservative simulations, as pointed out by Fujimoto [70] or Lin and Lazowska [109]. Further, the algorithm is rather expensive, as it necessitates sending large numbers of empty messages. This can be alleviated to a certain extent by using Misra's algorithm [128] that only sends the null messages if a process is about to start waiting. Schemes exist that do not make use of null messages at all, instead relying on deadlock detection and recovery, see e.g. the work of Chandy and Misra [42] or Liu and Tropper [117].

1.3.3 Optimistic approach

The *optimistic approach* presents an entirely different attitude to the causality problem. The best known algorithm of this sort is the *Time Warp* protocol proposed by Jefferson [94]. Logical processes process events disregarding the possibility of a causality error: so long as there are events on the event list, they are processed. The underlying idea is that meticulously ensuring that the causality constraint is met is very expensive, but often not necessary, so it is better not to do it – this is the "optimistic" nature of the algorithm. Time Warp essentially relies on a more general technique known as *speculative computation*, see e.g. Govindan and Franklin [81] or Hybinette and Fujimoto [86]. The algorithm does not prevent causality errors, but, in order to produce a correct result, it cannot ignore them when they happen. The errors are easy to detect: if a newly received message has a lower timestamp than that of the event currently being processed, then we have a causality error. To rectify it, the logical process must perform a *rollback*: it needs to move back in the simulation time (known as the *local virtual time* (LVT)) to the point preceding the timestamp of the newly received message. Only then the newly received message (known as a *straggler*) can be processed, and the logical process can restart the event processing from that point. Performing a rollback is a non-trivial task: we must not only decrease the LVT value, but also undo all the over-optimistically performed operations, that is, the operations executed at a higher simulation time than the straggler's timestamp. There are two main sorts of operations performed during event processing. First, the simulation state can be altered: one or more variable describing the state of the system can be changed. To be able to undo these changes, a *state queue* must be maintained, where simulation state is saved during the ordinary event processing, so that it can later be restored, should that be needed during a rollback. Second, processing an event can cause a new event to be scheduled – on the same or on another logical process. In either case, the new event's copy (more precisely, its *anti-copy*) is stored in the *output queue*. This makes it possible to find out what events have been scheduled and where. Locally scheduled events (i.e. those where the destination is the same as the source) are simply removed from the event list, known in Time Warp as the *input queue*. Remote events are sent to the destination process, which must cooperate in their undoing, a procedure also known as *cancellation*. The logical process that is rolling back sends to the destination process an *anti-message* from the output queue, that uniquely identifies the previously scheduled event. The event (to distinguish it from the anti-message, often referred to as a *positive event*) is dropped (*annihilated*) from the input queue. Should it be the case that the event has already been processed, a secondary rollback must take place at the destination, resulting in a *rollback cascade*.

Time Warp tends to use resources extremely intensively. Maintaining the three queues requires large amounts of memory. To be able to quickly cancel erroneous computations, the communication infrastructure needs to deliver large numbers of anti-messages in a very short time. Because of the optimistic nature of the algorithm, the processor is occupied almost constantly, processing more and more events.

Compared to a conservative algorithm, the overhead induced by the Time Warp is twofold. First, during an ordinary, forward processing, the Time Warp kernel must constantly update the state queue and the output queue, in anticipation of a rollback. This includes expensive operations such as dynamic memory allocations and memory copies. To prevent queues from growing indefinitely, the entries that are no longer needed must be purged periodically. An entry is no longer needed if its timestamp is lower than the minimum timestamp of all currently processed events and all messages in transit. A *global virtual time* (GVT) algorithm is responsible for periodically recalculating this minimum and disseminating it to all the logical processes, so that the *garbage collection* can take place. Processed events older than GVT are *committed*: they can no longer be undone. The second source of the overhead is the rollback operation itself. Consulting long event and state queues can be rather expensive, and certainly so is the sending of large numbers of anti-messages. However, we should point out that the processes that roll back are the most advanced ones in the simulation progress: a rollback does not take place on the *critical path*. With conservative simulation these overheads are not present, but logical processes can waste a lot of time doing nothing. An optimistic simulation will thus outperform a conservative one provided that its loss due to the forward overhead and the rollback cost is lower than its gain attributed to a more aggressive processing of events. Whether this is possible depends on the simulation model and the configuration of the simulation kernel. An interesting study has been performed by Lipton and Mizell in [114], where, under a number of simplifying assumptions, it has been proved that for the worst-case simulation model, an optimistic simulation can arbitrarily outperform a conservative one, but the opposite is not true (a conservative simulation can only outperform an optimistic one by a constant factor).

Apart from its primary use for the garbage collection, the GVT algorithm can have many other non-standard applications. At this point, we will only mention its function in the commitment of irreversible operations. Operations such as I/O (writing data to a file) or memory management (releasing dynamically allocated memory) cannot easily be undone. If they are requested while processing an event, such a request needs to be logged and the operation itself must be deferred, otherwise we would have a problem if the event requesting the operations were to be rolled back. The operations can safely be carried out only when it is guaranteed that they will not need to be rolled back. This condition is met as soon as the GVT value passes the timestamp associated with the deferred operations.

1.4 Modern computing platforms

1.4.1 Cluster computing

Cluster computing, made popular thanks to the spectacular success of the *Beowulf* project in 1994, has a longer history than that. Even in the 1980s, companies and research laboratories were experimenting with the concept, but the projects of that period could not be commercially successful due to the lack of inexpensive, sufficiently high quality hardware and software.

A related concept of a *network of workstations* [7] was more viable at the time: organisations had many networked workstations that were only used for a part of the day. During the out-of-office hours and in the weekends, the workstations were unoccupied, so their computing power could be used for processing batch jobs. Time sharing was usually used, i.e. multiple jobs could run simultaneously on the same resources. Consequently, there were no guarantees on the performance. Systems such as Condor [116] provided (and still provide) necessary management services, including workstation monitoring, job startup, migration (should the workstation be needed again) and termination services. The focus was on *high throughput* computing: the ability to successfully execute large numbers of mostly small jobs. If the only difference between those jobs was in their input parameter set, then systems such as Nimrod [2] could be used to manage the experiment, automatically performing a parameter sweep through the whole parameter space (there is now also Nimrod/G [1] – a Grid-enabled version). Parallel jobs could also be run, making use of the network interconnecting the workstations for inter-process communication, and typically using a parallel programming library such as PVM [78, 177] or later MPI [82, 126]. In such cases, co-scheduling processor and network resources is an advantage, see e.g. Basney and Livny [18].

A breakthrough came with the aforementioned Beowulf project [176]. Instead of using ordinary workstations doubling as batch nodes, a *dedicated cluster* was built. This has significantly improved the users' perception of the platform (they were no longer "cycle scavengers"), in addition to simplifying its maintenance. With a dedicated, centralised computing resource, making use of space sharing, a far more predictable level of availability and performance can be guaranteed. The success of Beowulf was a function of the availability of the needed components. Cheap, commodity PC computers had then reached a performance level where they became interesting for scientific computing. Also the growing popularity of the Internet made computer network equipment ubiquitous, driving the prices down and resulting in a bandwidth increase making it possible for clusters to be used for a much broader domain of scientific problems. Finally, the emergence of free operating systems such as GNU/Linux and the various BSD offspring made it cost efficient to use even old hardware, since there was no per-processor licence fee to pay. All in all, the performance/price ratio was so much better than that of a traditional supercomputer, that the clusters could no longer be ignored. Nowadays, many high performance computing problems are solved on clusters.

A scientific Beowulf-like cluster usually consists of a file-server (often doubling as the interactive node used for compiling, etc) and a number of batch nodes. Each node runs its own copy of the operating system – there is no single system image like in more expensive parallel machines. Users' home directories are normally exported to the batch nodes via the *network file system* (NFS). Cluster management software ensures an orderly access to the batch nodes. Commonly, submitting a job results in an exclusive allocation of the needed number of nodes for a specified time period, although other schemes are certainly possible.

Figure 1.3 shows an example of such a computer cluster: the DAS-2 cluster at the Vrije

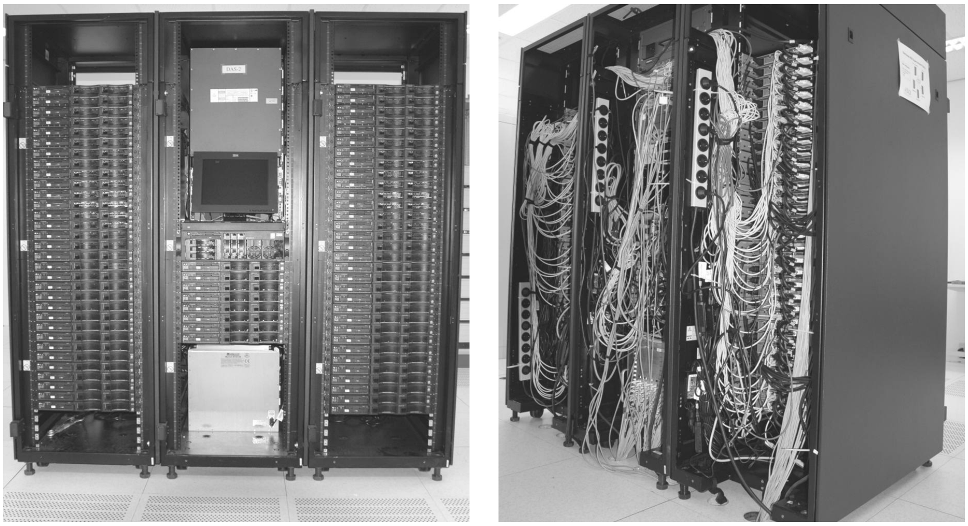


Figure 1.3: The DAS-2 cluster at the Vrije Universiteit Amsterdam, from the front (*left*) and from the back (*right*). *Photos courtesy of Kees Verstoep, VU.*

Universiteit Amsterdam (we used almost identical clusters to conduct the majority of the experiments presented further in this thesis). This is not a typical Beowulf cluster, for at least two reasons. First, as can be seen in Fig. 1.3 (*left*), it is very compact: because ultra-slim, rack-mountable cases are used, the 72 batch nodes easily fit in three racks, which would be out of the question should ordinary workstation cases be used. Second, an ultra-fast fiber-based Myrinet local area network [25] is used, and given its catalogue price of over US$1000 per port, which is easily 10 times more than that of standard solutions, it can hardly be described as a commodity item. Figure 1.3 (*right*), showing a seemingly incomprehensible tangle of cables at the back, can serve as an example of a more general problem with clusters: maintenance. With that many nodes, it is non-trivial to set it all up properly in the first place, and so is keeping it up and running. It is a problem of scale, further exacerbated by the sometimes insufficient quality of the commodity components. The DAS-2 cluster had its share of problems with hard drives and a more general problem of inadequate heat dissipation, in spite of the fact that it was built by a top brand manufacturer, with a matching price tag attached.

1.4.2 Grid computing

The term *Grid computing* was born around 1997, and made popular with the publication of a book by Foster and Kesselman [66]. According to the current definition by Foster [64], Grid *coordinates resources that are not subject to centralized control, using standard, open, general-purpose protocols and interfaces to deliver nontrivial qualities of service.* It builds on a previous research in *metacomputing*, see e.g. Smarr and Catlett [170], where researchers had been attempting to interconnect distributed computing resources to be able to solve larger problems. Grid computing is more ambitious. Taking advantage of the steadily improving interconnection infrastructure, it ultimately aims to create one vast computational resource, supported by a world-wide network of computers. Needless to say, this aim has not yet been reached, assuming it ever will.

Even forgetting the obstacles of a political or personal nature, this aim is very difficult to achieve for technical reasons. The resources are vast, and managing such a large pool is non-trivial. They belong to different administrative domains, with different access policies. The resources are heterogeneous, so care must be taken when mixing them. In spite of all that, a uniform means must exist for describing the resources' availability and enabling access. A global method of authentication (establishing a user's identity) and authorisation (granting access to the resources) is also needed. A certain degree of predictability regarding the performance of the interconnecting network would also be very useful, see e.g. Wolski et al. [194].

Grid computing is still very much a work in progress, coordinated by the Global Grid Forum, and far-reaching changes regularly take place. The influential *Anatomy of the Grid* by Foster et al. [68] defined the Grid using the concepts familiar to metacomputing researchers. Only a year later *Physiology of the Grid* [67], by pretty much the same authors, defined the *Open Grid Services Architecture* (OGSA), advocating the use of web services and other internet standards such as WSDL and SOAP, as well as an extensive use of Java – a small revolution in comparison to the previous work. OGSA-compliant implementations did not even get a chance to stabilise, before another, admittedly smaller, evolution of the infrastructure, in the form of WS-Resource Framework, was announced two years later by Czajkowski et al. [49].

Currently, instead of a single, global Grid, there exist a number of smaller scale experimental grids, each managing its own share of resources. The DAS-2 machine could be considered a very small example of this. It is a multi-cluster – as shown in Fig. 1.4, it consists of five separate clusters, distributed across five Dutch universities: the Vrije Universiteit Amsterdam (VU, the central and largest of the clusters, discussed earlier in Sec. 1.4.1), the University of Amsterdam (UvA), the Leiden University, the Utrecht University and the Delft University of Technology. Each of the five sub-clusters is running a copy of the Globus toolkit [65] that provides the necessary Grid infrastructure. As a result, it is quite simple to execute jobs with some of the processes running on one cluster, and some on one or more others. We have performed the majority of the experiments presented later in this thesis using the UvA and Leiden clusters of the DAS-2 machine, making extensive use of their cross-cluster capability.

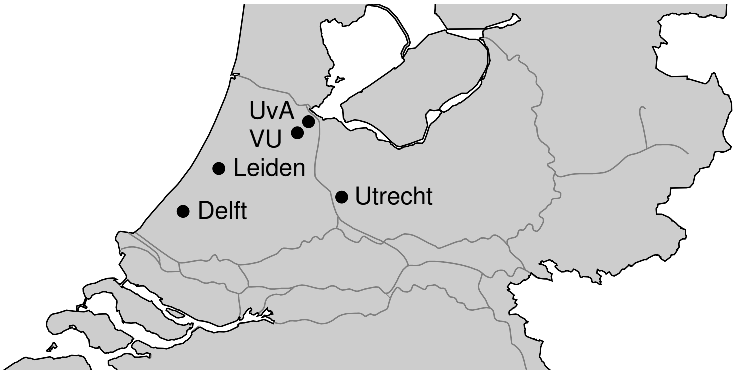


Figure 1.4: Geographic distribution of the DAS-2 multi-cluster across five locations in the central provinces of the Netherlands. *Map obtained from* Kartografie in Nederland, *http://www.kartografie.nl/*.

1.5 Objectives

The main objective of our work is to move an optimistic Time Warp parallel discrete event simulation kernel, as an example of a latency-sensitive application, beyond clusters, to a Grid environment.

The challenge lies in the fact that *latency* of the communication channels, i.e. the travel time of a single short message, is bound to be relatively high on wide area distributed Grid resources due to the fundamental limits of physics. In PDES, many small messages are exchanged, and a low-latency communication infrastructure helps to keep the logical processes well synchronised, so the rollbacks are less frequent and shorter. That is why such latency-sensitive problems work best on machines with very efficient communication infrastructure.

We are motivated by the rapidly growing popularity of the Grid as a computing resource, coupled with a slow decline of tightly coupled multi-processor supercomputers. Given these trends, an investigation of the viability and constraints of moving to the Grid a latency-sensitive high performance application such as a Time Warp kernel is of high relevance. It is important to evaluate the fitness of the existing Grid components for high performance computing problems, and, if it turns out to be necessary, to improve this fitness.

Efficient utilisation of distributed resources is important not only in the future, assuming that the Grid becomes a dominant computing platform, but also here and now. Many problems are too large to run efficiently on a single parallel machine. Larger computing and memory power, while generally available, tends to be distributed.

We expect that the project's conclusions will extend beyond its domain, both application- and hardware-wise. Namely, we hope that the conclusions will be applicable to many other parallel applications, and not only when running on the Grid, but also on other (future) architectures featuring hierarchical communication latency infrastructures.

To the best of our knowledge, we are the first to move a PDES kernel to the Grid. While other researchers have been porting Time Warp to architectures such as networks of workstations that do share some characteristics with the Grid, in particular the distributed memory, those architectures are significantly more centralised. Resource management is local (and thus far simpler), and communication is not that much of a problem. Below, we present an outline of this thesis:

The current chapter provides a very general introduction to the thesis domain and states the objectives of this work.

In Ch. 2, we will provide a more specific introduction to our work by introducing all the infrastructural components. We will discuss the parallel simulation kernel used, the simulation models that we will run on top of it to conduct the experiments, and the computer infrastructure we will be using for running the kernel.

We will need to ensure that we have a good understanding of how our Time Warp kernel works. This will be the focus of Ch. 3, where we will build analytical models and computer simulations of an optimistic PDES kernel, and attempt to validate them against a real parallel simulation kernel.

Once this process is successfully completed, we will be ready to conduct the first experiments on the Grid. These efforts will be described in Ch. 4. The purpose of these experiments will be to investigate how running a PDES kernel on the Grid influences the performance of standard Time Warp protocols. We will also have an opportunity to evaluate the performance of our infrastructure.

Based on these findings, in Ch. 5 we will propose new mechanisms designed to alleviate the problems observed when running our Time Warp kernel on the Grid. We will implement these mechanisms in the kernel and verify that they work according to our expectations.

In Ch. 6, we will briefly discuss another Grid-related project that we have been working on: the Polder Metacomputing Environment.

Finally, Ch. 7 will summarise the work performed within the scope of our research, discuss its implications and present possible future directions.

Additionally, App. A will summarise all the major development work on our parallel simulation kernel.

2

APSIS Simulation Kernel

2.1 Introduction

The purpose of a parallel simulation kernel is to provide a level of abstraction shielding the user (a simulation designer), who might not be an expert in issues such as parallel programming, synchronisation, scheduling, etc, from the low-level implementation details. To achieve this goal, an optimistic PDES kernel should handle at least the following functionality:

- initialisation of the parallel environment,
- maintaining the local event list,
- saving simulation state,
- scheduling external events – this includes encapsulating them in a message and forwarding to the communication layer for delivery to the destination process,
- receiving external messages – this includes obtaining them from the communication layer, modifying the input queue, checking for causality errors and, should one be detected, performing rollbacks and sending out the anti-messages,
- computing GVT and collecting garbage,
- termination detection and a graceful termination of the parallel environment.

Some of this functionality, like computing the GVT estimate or performing rollbacks, is performed in the background by the simulation kernel, requiring no actions from the user. Other features need to be made directly available via a suitable *application programming interface* (API). This applies in particular to the event scheduling, that is, to event sending, receiving and, if the simulation kernel supports it, *retracting* (an "unschedule" operation for events that the user decides should not be processed).

Expanding on the above items:

The initialisation of the parallel environment is typically delegated by the simulation kernel to the communication library used or to the operating system.

While the maintenance of the event list is relatively straightforward, the choice of the data structure to use is not necessarily so. A simple double-linked list will often be sufficient, but sometimes more elaborate data structures are better, as shown by Rönngren et al. [158]. This is dependent on the target simulation: if it tends to have very few events scheduled for the future at any particular time, or if new events almost always have the highest timestamp, the overhead of using a high performance data structure will usually not pay off.

There are many different GVT algorithms, more information about them will be provided in Sec. 5.2. Garbage collection is a straightforward operation, involving the release of obsolete (i.e. older than GVT) items from the event lists and the state list.

Termination detection is most easily performed as a part of the GVT calculation: if a newly calculated estimate is infinity, then there are no more events in the system and the simulation is thus over. The termination itself is then typically left to the communication library or the operating system.

2.2 State of the art

State saving

Multiple options are available for the state saving. As originally defined by Jefferson [94], *copy* state saving was used. In this scheme, the user informs the kernel at the beginning of the simulation about the location of the simulation state (that must often be organised in one, continuous block), and the kernel copies the whole state before processing each event. This can be prohibitively expensive so far as the memory requirements go, and also very inefficient if the state is large and only part of it changes frequently. *Periodic* state saving, discussed for example by Lin et al. [113], has been devised to overcome these problems: the state is only copied once every n events are processed. This complicates the rollback operation: because an earlier state than required will usually be restored, the right state needs to be recreated by allowing the simulation to progress forward under controlled conditions (this is known as *coasting forward*). Another solution is to use *incremental* state saving, discussed by Bauer and Sporrer [20]. In this scheme, the user must explicitly request the kernel to save the state of a variable before it can be modified. This ensures that only the needed part of the state is saved, but it also complicates the rollback procedure: because only partial states are saved, all of them that have a timestamp higher than that of the straggler need to be restored, which increases the computational complexity from constant to linear. Naturally, various combinations of the above state saving schemes can be used, in particular the periodic and incremental state saving can easily be made to work together, as shown by Franks et al. [69], eliminating the need for coasting forward and limiting the cost of a rollback. Other schemes, such as probabilistic checkpointing by Quaglia [145] or reverse computation by Carothers and Fujimoto [34] are also possible.

Stability

A Time Warp kernel performs best if all logical processes progress with the same, or a very similar, pace, because this reduces the probability of large rollbacks taking place. One way to achieve this is to have all logical processes perform (stochastically) the same operations, and to run them on homogeneous computing resources – this should ensure a reasonably balanced progress most of the time. However, some simulations might be inherently difficult to split into homogeneous logical processes, or homogeneous computing resources to run them on might simply not be available. In some cases, *dynamic load balancing* [35,79,163], in the form of redistributing work at run time, might be necessary to alleviate the problem. If the imbalance is not extreme, a simpler scheme, known as *optimism control*, may suffice. The general idea is to provide some means of limiting the progress of the most quickly performing processes, since their optimism will usually be unfounded (i.e. they will have to roll back) if the rest of the simulation is lagging behind. Over the years, a number of such schemes have been devised that differ from each other in how they deal with two fundamental issues: how do we decide that a process is "too" optimistic, and how do we deal with that? Regarding the first issue, a process is clearly over-optimistic if it is running out of memory, or at least if its memory requirements cross an earlier established threshold. Alternatively, the LVT value can act as an indication: the further the process progresses from the most recently calculated GVT value, the higher the probability of a rollback. A "safety valve" threshold on the virtual time can thus be established. Once we have established that a process is overly optimistic, we need to slow it down, an operation commonly known as *throttling*. The simplest thing to do is to suspend the processing of events, typically until a new GVT value is calculated, which will result in the release of some of the no longer needed memory and will move the virtual time threshold forward (event processing will also be resumed if a rollback takes place). The most straightforward of such schemes is the Moving Time Window proposed by Sokol and Stucky [171], where a window of fixed width, moving forward in the simulation time, is defined, with the most recent GVT value as its lower limit. In this scheme, the logical processes can only process events with timestamps within this window. Optionally, a global synchronisation might take place once all the processes reach the top of the window, as is the case with Bounded Time Warp, evaluated by Turner and Xu [184]. Some Time Warp kernels, under low memory conditions, might attempt to reclaim the memory by *sending back* the messages to their sources (this was already suggested by Jefferson [94]) or performing *artificial rollbacks*, as shown by Lin and Preiss [112] – both actions will synchronise the simulation processes and essentially throttle them. Preiss and Loucks [143] point out that if periodic state saving is used, some entries from the state queue can simply be pruned.

Time Warp kernels

Below, we will present a short survey of the Time Warp kernels described in the literature.

The first well-known optimistic Time Warp kernel was **TWOS**[*]: the Time Warp Oper-

[*] pronounced like *tw-os*, as one word.

ating System [96, 153], developed at the Jet Propulsion Laboratory. As the kernel's name implies, it was designed as a separate, special purpose operating system. Such a low-level implementation made it possible to perform optimal processor scheduling and message queuing. Processor scheduling was not time-sliced, but it was pre-emptive, with the logical process with the lowest LVT value being executed first. Messages were not served on the first come first serve basis, instead those with the lowest timestamp got a preferential treatment. However, later on most implementations of TWOS were simply running on top of an existing host system. Optimism control was penalty based: logical processes experiencing too many rollbacks were scheduled to run less often. TWOS featured both lazy and aggressive cancellation (see Sec. 4.2), and many dynamic features, such as dynamic load management, dynamic object creation and dynamic memory allocation. It has been used to run many sorts of simulations, including neural networks, Jupiter's atmosphere, military battlefields, and computer networks. Originally designed for the Caltech/JPL Mark II hypercube, it was also ported to the Mark III, BBN Buttefly GP1000, Intel Delta, Cray T3D and the Inmos Transputer, as well as to a network of Sun workstations.

GTW, the Georgia Tech Time Warp [55, 74], is another influential Time Warp kernel. Originally developed with cache-coherent shared-memory multiprocessors in mind, it was later extended to support more distributed platforms such as the networks of workstations. On a shared memory machine, GTW implements a number of interesting optimisations to minimise the amount of event processing overhead, such as direct cancellation (event lists are stored in shared memory and can be read or modified by other processes, so no anti-messages are required, see Fujimoto [71]) or a specialised GVT algorithm that requires no "GVT messages". GTW supports both copy and incremental state saving. User-level dynamic memory allocation and I/O operations (I/O events) are also supported. Optimism control is provided globally in the form of application-defined simulated time barrier, or locally using blocking I/O events (a process will block until GVT advances past the event time). Thanks to its high performance and a relatively liberal licensing, it has been used by many researchers. Its authors have used it primarily for telecommunication network simulations and battlefield management.

Another relatively well known Time Warp kernel is **WARPED** [122, 149]. Its popularity probably stems from the fact that it is quite modern (it is very modular and is written in C++, programming a simulation can be done in object-oriented manner using inheritance), and that it is in public domain. It supports multiple state saving and memory management algorithms, lazy and aggressive cancellation, and two different GVT algorithms. Optimism control using time windows is provided. It can use either light-weight thread-based messaging for SMP machines or MPI on machines with distributed memory. It has been used to model computer systems, such as a RAID–5 disk array and a shared memory SMP machine. A VHDL (Very High Speed Integrated Circuit Hardware Description Language) simulation kernel running on top of it has also been developed.

HLA, the High Level Architecture [50], has generated much interest lately. HLA is much more than a Time Warp kernel, it is an architecture designed to facilitate simulation interoperability and reuse. Developed with US military simulations in mind, it was later commercialised and standardised by the IEEE as a common architecture applicable across all classes of simulations. HLA provides mechanisms for building simulation systems consisting of multiple interacting modular components with well defined interfaces. Multiple interacting simulations, called *federates* in HLA, form *federations*. *Rules* govern how federates and federations are constructed. An *Interface Specification* defines how they interact with the *Runtime Infrastructure* (RTI), a software layer responsible for the management of simulations, including the time management, data distribution and communication. Finally, an *Object Model Template* provides a method for documenting key information about simulations and federations. HLA can be used for all sorts of simulations, including the optimistic PDES, as shown for example by Wang et al. [189].

Other parallel simulation kernels that have been described in the literature include BSP Time Warp [121], CCL [147], CTW [8], ParaSol [123], Parsec [10], ROSS [33] and SPEEDES [173]. For the sake of brevity, we will not elaborate on these environments here.

Benchmarks

In this chapter, we are also going to introduce the simulation models we have been using. Many other potentially suitable simulation models have been described in the literature.

Focusing on the more generic benchmarks, Rönngren et al. [159] propose an incremental benchmark suite based on ping models: self-ping, ping-pong and ping. In the simplest version, they are free from causality errors. These are very minimalistic benchmarks, so-called *loops*, that allow to test a single, specific area of a PDES kernel, such as the maximum local and non-local event rate or the scalability overhead. They are thus best suited as a help tool for a PDES kernel developer, not as a comprehensive benchmark for comparing various PDES implementations.

Balakrishnan et al. [15] introduce an infrastructure called *performance and scalability analysis framework* (PSAF). With the help of the *workload specification language* (WSL), the properties of a parallel simulation can be represented in a platform-independent manner, using parameters such as the connectivity of each object, event size and granularity, state size, the parameters to model the progress of the simulation time and the scheduling of new events, etc. A *synthetic workload generator* can be used to create generic workloads in WSL that focus on specific performance-related properties. These generic workloads include the shark's world [46], colliding pucks [84] and the ping models [159] discussed above. Translation backends for different simulation environments can be implemented that automatically translate WSL into a ready-to-run code targeted at a particular parallel kernel.

On the other hand, there also appears to exist a demand for freely available, complex benchmarks. Chamberlain and Discher [39] propose to collect a representative set of large benchmarks for logic simulation, written in industry standard languages such as VHDL. Such benchmarks could help in the development and performance evaluation of new algorithmic ideas, and help convince the actual circuits designers of the advantages of using the parallel simulation techniques.

2.3 Kernel architecture

All of the development work and experiments we will describe in this thesis has been conducted using **APSIS**: the **A**msterdam **P**arallel **Si**mulation **S**ystem [134–136, 167]. AP-

SIS is an optimistic Time Warp parallel discrete event simulation kernel, developed in our research group at the University of Amsterdam by B. J. Overeinder. It is fairly small, which makes it well suited for our experimental work. Below, we will describe the most essential properties of the kernel, based on its state at the beginning of our research. In the mean time, we have made many improvements and added a number of extensions, but the basic features described here have remained unchanged. The changes made will be described throughout the thesis, a summary can also be found in App. A.

The architecture of APSIS is relatively straightforward. The kernel was designed with distributed memory machines in mind, so all the inter-process communication is explicitly handled via messages. Because shared memory is not used (at least, not explicitly by the kernel), the access to various variables does not need to be protected. This helps to avoid one of the notorious problems of parallel programming: the unsynchronised access.

The architecture of the kernel assumes a 1:1 mapping between the logical processes and the available processors. Running multiple logical processes on one processor, if allowed by the communication library, is in principle possible, but is generally not recommended, since the coarse-grain process scheduling performed by the operating system is unlikely to result in a good performance. Within each logical process, there is a single thread of control, so the *event scheduling world view* is the most natural one to use. In this scheme, the focus is on events and how they affect the simulation state: each event type has an event handler, invoked when the event of this type is to be processed. The handler can modify the simulation state and schedule new events. Basically, this is the lowest-level scheme, and other, more elaborate ones such as *process interaction* can be built on top of it.

APSIS has been designed for complex systems simulations, such as asynchronous cellular automata. Because such simulations typically have a relatively large and slowly changing simulation state, the kernel features the incremental state saving algorithm.

A simple GVT algorithm is used, based on the work of Bauer and Sporrer [19]. More information about it will be provided in Sec. 5.2 and 5.5.2, for now suffice it to say that it is based on the principle of counting messages at both ends of a channel, and that it recomputes a new GVT value 20 times per second. The Moving Time Window scheme is used to keep the optimism within bounds, the size of the virtual time window (VTW) is tunable.

2.3.1 Interfaces

APSIS is written in C. It compiles into a library that must be linked with the user simulation code to form an executable. Layer-wise (see Fig. 2.1), at the bottom the kernel communicates with the inter-process communication library via what is called an `async_comm` interface. At the top, it provides an API for the user simulation code.

The interface between the kernel and the inter-process communication layer is quite simple. Apart from some initialisation and termination routines, it consists of one message sending routine and two message receiving ones (a blocking and a non-blocking one). For the best performance, Carothers et al. [36] recommend that these routines should be asynchronous. A number of communication modules have been implemen-

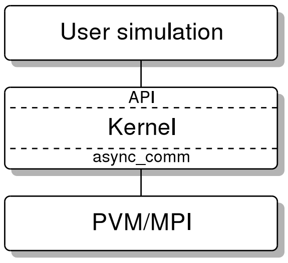


Figure 2.1: A layered structure of the APSIS simulation kernel.

ted over the years, but all the recent work has been focusing on the versions for PVM and MPI. Both communication libraries guarantee the *First-in First-out* (FIFO) property of the communication channels. The Time Warp protocol, in principle, does not require the ordered delivery of messages (as was mentioned in Sec. 2.2, the TWOS kernel actually strives *not* to deliver them this way). However, given that the common programming libraries provide the FIFO property "for free", APSIS does take advantage of that fact, as it simplifies some operations. For example, without the FIFO property, a Time Warp kernel must function properly if an anti-message arrives before the matching positive event is received, even though the latter is always sent first. Also, the GVT algorithm needs to be more complex.

The application programming interface for the user simulation code consists of a number of function calls, accessible from the C language or other languages capable of calling C functions, such as C++. The functions can be grouped as follows:

- support routines: initialisation, termination, parameter tuning, etc,
- topology routines,
- query routines: obtaining the LP identifier, the current values of LVT, GVT, wallclock time, etc,
- event routines: sending, retracting, receiving,
- state saving.

Normally, a user simulation initialises the environment, schedules the first events, and enters a loop, where it receives events and processes them. If the event receive routine returns without an event, then the simulation phase has finished. The processes should then output their results and terminate. Right before the termination, the kernel will also output some statistical information providing an insight into its run-time behaviour.

2.3.2 Internal data structures

Figure 2.2 shows the interconnections between the most important APSIS data structures: the event and state queues. The queues are implemented as double-linked lists (more precisely, double-linked rings). To simplify entry insertion and removal, each queue contains a placeholder entry (marked black in Fig. 2.2) that is always present, so there is no need to test for NULL pointers.

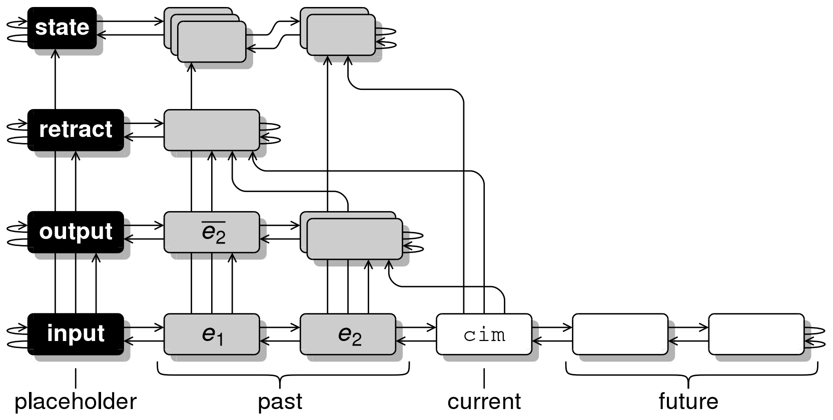


Figure 2.2: Interconnections between the most important data structures in APSIS: the input-, output-, retract- and state- queues.

The input queue is the main queue, and it is sorted by the schedule time of events. A variable `cim` points to the currently processed event: all prior events (marked grey) have already been processed. Each entry in the input queue contains an embedded event, as well as pointers to the output, state and retract queues. Each of these pointers points to the last entry in the respective queue added there when processing the event from the input queue. If no entry was added, the pointer will be the same as that of the previous event, as can be seen in Fig. 2.2 for `cim` or for the pointer to the retract events of event e_2. This setup makes it easy to determine the set of operations performed during the processing of each event.

The output queue is sorted by the creation time of the entries. Each entry contains an embedded anti-copy of a new event generated when processing the event from the input queue. To remind the reader, when rolling back, this anti-copy, combined with the positive event, results in the annihilation of both.

The structure of the retract queue (used for user-requested event retractions) is identical to that of the output queue. It stores anti-copies of the retraction requests, but because retraction requests act very similarly to anti-messages, their anti-copies are in fact ordinary positive events, that, if sent out during a rollback, cause the retracted events to be re-scheduled.[†]

Like the previous two queues, the state queue is sorted by the creation time of the entries. Each entry contains the address and the size of the saved state variable, as well as the saved memory itself. With this information, the previous simulation state can easily be restored when rolling back.

As can be seen in Fig. 2.2, events from the input queue that have not been processed yet do not have any associated entries in the other queues. The entries in the other queues have a timestamp at most equal to LVT (that, in turn, equals the timestamp of the `cim` event), and they are all potential candidates for garbage collection. If a rollback occurs, the entries from the top of the output-, retract- and state- queues are used to

[†]Currently, the implementation in APSIS is limited to locally scheduled events.

restore the previous state of the simulation, and are subsequently released (the pointers from the input queue entries pointing to them are reset). The events in the input queue are also affected by the rollback, although indirectly. If event e_2 was scheduled locally by another event e_1 that is rolled back, then there is an anti-copy $\overline{e_2}$ in the output queue (see Fig. 2.2), and event e_2 will be annihilated (since it is causally dependent on an erroneous computation). An event will not be annihilated even if it is in the rolled back region, provided that it was scheduled by another process, or that it was scheduled locally by an event with a timestamp earlier than the timestamp to which the process is rolled back.

2.4 Simulation models

Below, we will provide a description of two simulation models we have been using with APSIS to conduct the experiments discussed in this thesis.

2.4.1 Ising spin

This is our main simulation model. It is convenient to use, because it is easy to adjust to an arbitrary problem size and an arbitrary number of processes; the amount of communication can also fairly easily be adjusted. We have been using this model exclusively for the experiments in this chapter, as well as in Ch. 3, 4, and partly in Ch. 5. This simulation application is an asynchronous cellular automaton implementation of the *Ising spin model* [87].

Ising spin is a popular model in statistical physics, originally proposed for modelling of magnetism. The model defines a grid of interacting particles (a *lattice*). Every particle has a *spin*, with a value of either 1 or −1. At random, *spin flip attempts* take place: randomly chosen particles get a chance to change their spin to the opposite value. The probability of the change being accepted depends on the energy difference between the new and the old state. The energy of the system is:

$$E = -J \sum_{\langle ij \rangle} s_i s_j - \mu_0 H \sum_i s_i \tag{2.1}$$

The first sum in (2.1) covers the interaction between the particles: it goes over all the pairs of the nearest neighbours, as only adjacent particles are assumed to interact with each other. The constant J is normally positive, and as a result the particles have a tendency to align themselves in the same direction as their neighbours. The second sum in (2.1) covers the influence of an external magnetic field H.

The *Metropolis algorithm* [127] is used to simulate the dynamics: a flip attempt is accepted according to the Boltzmann probability distribution:

$$p = \min(1, \theta) \qquad \theta = \exp(-\Delta E / kT) \tag{2.2}$$

Therefore, if ΔE is less than zero, a spin flip attempt always succeeds ($p = 1$). If ΔE is greater than zero, the flip only occurs with a certain probability. In (2.2), T is the model *temperature*: if it is lower than the *critical* temperature T_c, the neighbouring sites are strongly correlated, the state changes are infrequent and a spontaneous non-zero magnetisation occurs – the model exhibits a ferromagnetic behaviour. If the temperature is above T_c, many more spin flip attempts succeed, the individual sites are in disorder, and the model resembles a paramagnetic material.

The simulation is implemented as follows: it starts from a random state of the lattice. During the execution, a fixed number of spin flip attempts is performed, typically in the range of several thousands per statistical site. The state of the lattice can be dumped at the end, primarily in order to verify that it is identical to the previous runs performed with the same parameters. Figure 2.3 presents a sample progress of the simulation, for a sub-critical temperature $T = 1.0$.

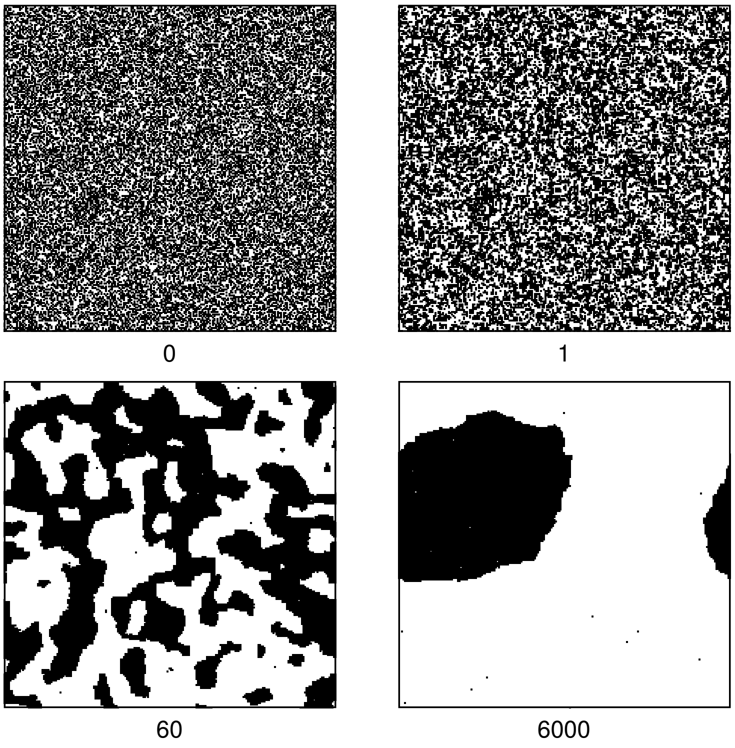


Figure 2.3: Evolution of the Ising spin simulation lattice for a sub-critical temperature $T = 1.0$. Progress measured in the average number of spin flip attempts per site.

The parallelisation is performed as follows: the cellular automaton lattice is spatially decomposed and mapped onto the simulation processes arranged in a torus topology. The borders of the individual sub-lattices are duplicated at the neighbours (see Fig. 2.4). When the state of a site at the border changes, the copy at the neighbour is updated by scheduling an external update event, with the timestamp equal to the current LVT of the sender. Apart from these external events, each process executes a self-sustaining stream of locally scheduled events, with random inter-arrival times taken from a negative

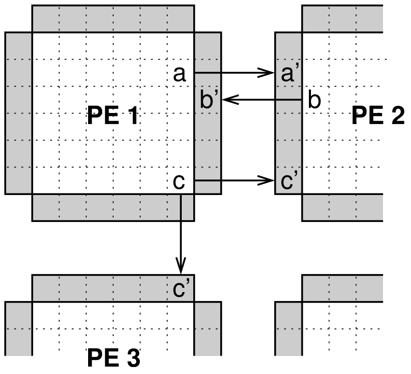


Figure 2.4: Spatial decomposition of the lattice in the Ising spin simulation. *Picture courtesy of Dr. Benno Overeinder, UvA (now at VU).*

exponential distribution. Every local event is a flip attempt.

There exist more optimal algorithms for parallel implementations of the Ising spin model, notably by Lubachevsky and Weiss [118, 119]. Our implementation is not meant to be of a production quality, it is intended more as a testing tool for our parallel simulation kernel APSIS. We often use it as a black-box benchmark, as the problem size and the simulation length can be easily adjusted as needed, and the simulation can be decomposed onto any practical number of logical processes. Further, the run-time behaviour of the simulation can be significantly altered with the help of just one parameter: the temperature of the model, related to the correlation in the system. The higher the temperature, the more spin flip attempts are successful, including those at the borders, that result in events being scheduled on other logical processes. Therefore, increasing the temperature increases the amount of inter-process communication in the system. For example, between the temperatures of 1.0 and 3.5 (that are, respectively, in sub-critical and super-critical regions) there is a difference of two orders of magnitude in the communication volume (see Sec. 4.3).

When measuring the performance of the simulation, we will usually show either the run time (in seconds), or the speedup. To calculate the latter, we need to know the run time of a sequential simulation. We have used the same parallel implementation of the Ising spin model for this purpose, configured to run using a single process only. While this is not optimal, the simulation kernel has been optimised to skip the operations causing most overhead (maintaining output and state queues, GVT calculation and garbage collection of past events, communication layer polls). Given the small computational grain of Ising spin events, this overhead significantly affects the performance for the parallel runs.

2.4.2 PHOLD

This is the second simulation model we have been using. Its run-time behaviour is significantly different from that of the Ising spin simulation, which should help us ensure that our findings are generic. The experiments conducted with this model are described in Ch. 5.

PHOLD (Parallel Hold) is a well known parallel simulation benchmark defined by Fujimoto [73]. The underlying model is relatively straightforward: there is a fixed number of events in the system, and each event, when processed, schedules exactly one new event on a randomly chosen destination process, typically with a timestamp increment taken from an exponential distribution. The events thus form a fixed number of *event threads*, as shown in Fig. 2.5. The event population size and the number of logical processes are tunable. Most implementations of PHOLD will also allow the user to influence the choice of destination logical processes for a new event, and to set the event granularity by specifying a computational grain (an extra computation performed when processing an event).

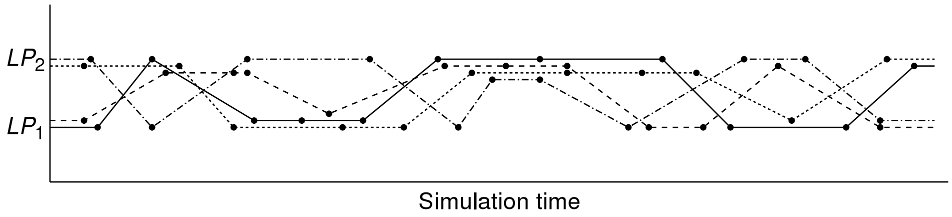


Figure 2.5: Sample event trace of a PHOLD simulation with two LPs and four event threads.

The implementation we have used is based on the work of M. al Mourabit [5]. Our modifications included mostly the adjustments needed to match the current programming interface of the APSIS kernel and some cleanup.

PHOLD can be extremely communication intensive. In its most "aggressive" configuration, when the destination process for a new event is chosen completely randomly, making the simulation run stably has proved to be difficult even on a single cluster. To keep the communication within limits, in the experiments presented here the freedom of choice of the destination process for a new event has been restricted. A new event can only be scheduled on the current process or on one of the four direct neighbours (in the 2D torus topology). The majority of new events are scheduled on the current process. Because the number of event threads is kept constant across all experiments, to make the scalability studies more realistic, with the increase in the number of processes also the probability of scheduling an event on the neighbours increases. For 32 logical processes, this probability has been chosen arbitrarily to be 10% (the current process is thus chosen with a 90% probability). The probability increases with a square root of the increase factor in the number of processes, to mimic the behaviour typical for spatially decomposed matrices (assuming that only the sites situated directly at the border need to exchange messages with other processes). Therefore, for 128 processes the probability of scheduling an event on one of the neighbours equals 20% and for two processes it is just 2.5%.

The communication pattern thus becomes similar to that of the Ising spin simulation, but some important differences remain. Ising spin is a self-sustaining model, i.e.

the external events have only an auxiliary function compared to the locally scheduled events. PHOLD, on the other hand, is message-initiated: there is no constant stream of local events, instead, the external events have a key role in defining the dynamics of the model. This distinction has a number of practical consequences. While with the Ising spin model, lazy cancellation (see Sec. 4.2.2) results in a large improvement (see Sec. 4.3), with PHOLD it tends to be a loss, because the events are too strongly correlated. Our PHOLD implementation is also far less sensitive to the properties of the GVT algorithm (to be discussed in Sec. 5.5), since it features a much smaller simulation state (mostly the state of a few random number generators) than our Ising spin simulation. Yet another difference is that PHOLD schedules external events into the future, whereas the Ising spin simulation schedules them for the current simulation time. The lookahead of the two models is thus different, and the number of rollbacks observed could be affected, although exact predictions are difficult given the fundamental differences between the models.

In all the experiments with PHOLD we will show, the number of event threads has been set to 1024. We have set it so high so that there is enough parallelism in the system even if we run on 128 processes. A side effect is that the performance for small numbers of processes (in particular one process, used as a reference to calculate the speedup) is not optimal, since the simple, double linked list implementation of the input event queue (see Sec. 2.3.2) is not best suited for that many events.

2.5 Distributed ASCI Supercomputer 2

The majority of the experiments presented in this thesis have been obtained on the DAS-2 machine (see App. A for a complete list, also including exceptions). We have mentioned this machine before, in Sec. 1.4, when talking about the cluster and grid computing. Still, given the importance of DAS-2 for our work, a proper introduction with focus on the performance is in order.

DAS-2 is a wide area distributed computer consisting of 200 nodes. The nodes are organised into five clusters, located at different universities in the Netherlands. One location has 72 nodes, the remaining ones have 32 nodes each. Every node contains two Pentium-III 1 GHz CPUs. The nodes are interconnected using switched Fast Ethernet and low-latency Myrinet 2000. The clusters are mutually interconnected by SURFnet5, the Dutch education and research gigabit backbone (the wide area link introduces a high latency, but normally does not limit the bandwidth, since the bottleneck is the 100 Mbit/s Fast Ethernet within each cluster). The nodes run Linux kernel 2.4.*x*. The resources can be globally managed using Globus [65] 2.2.4 (recently, an update to Globus 3.2 took place). A description of DAS, the predecessor of the current machine, was provided by Bal et al. [14].

MPICH 1.2.5-1a (later updated to 1.2.6) is used for the inter-process communication. Multiple communication options are available, as can be seen in Tab. 2.1. The table presents the message travel times for a message of 156 bytes (a standard message size in the APSIS kernel), measured using a dedicated ping-pong benchmark. We have used

Table 2.1: Message travel times in μs using different communication layers.

MPICH flavour	Intra-node	Inter-node	Inter-cluster
GM	2.18	16.14	—
G2	62.01	702.00	1406
G2 over GM (orig.)	9.49	23.55	1413
G2 over GM (mod.)	4.11	18.36	1413

a non-blocking receive, because it is more suitable for optimistic PDES, in spite of the fact that it has a slightly worse performance. We measured the travel times between two processes running on a single node (please remember that each node is a two-processor SMP machine), running on different nodes of one cluster and running on two different clusters. MPICH-GM [130] communicates using the low-latency Myrinet network, and thus achieves the best performance. However, it is incapable of a wide area (inter-cluster) communication, necessary on the Grid. MPICH-G2 [99], a Grid-enabled implementation of MPI, addresses this problem. Using Globus services, it provides transparent resource co-allocation, wide area communication, topology discovery extensions and many others. However, the performance of MPICH-G2 within one cluster is orders of magnitude worse than that of MPICH-GM. This is because MPICH-G2 communicates over Fast Ethernet, using TCP/IP, which has a much higher latency than the Myrinet solution. However, it is possible to make MPICH-G2 work on top of MPICH-GM, thus only using TCP/IP for the inter-cluster communication (this is called "a configuration with vendor MPI", but we will call it *G2 over GM*). While the improvement over an ordinary MPICH-G2 is clear (see Tab. 2.1 (*G2 over GM (orig.)*)), there is still a significant performance gap compared to MPICH-GM. This comes because, under certain circumstances, the TCP/IP sockets are still queried, even if all the processes run on a single cluster. This is, of course, not necessary, so we have modified MPICH-G2 over GM to prevent it from querying the TCP/IP sockets if all the processes of a communicator are on one cluster.[‡] This has decreased the overhead to far more acceptable values (see Tab. 2.1 (*G2 over GM (mod.)*)), although it is still not negligible. Also, no amount of optimisation can overcome the inherent performance gap between the Myrinet and the wide area network present between the clusters: a difference of two orders of magnitude in the latency can be observed if one process runs at the DAS-2 cluster of the University of Amsterdam and the other runs in Leiden.

2.6 Process topology

When we took over the development of APSIS, it had several limitations in its implementation of the Time Warp protocol. These limitations were mostly irrelevant for the stochastic, asynchronous cellular automata simulations used with APSIS at the time, but they made it difficult to run other sorts of applications or to verify the correctness of the kernel. Therefore, one of our initial efforts with APSIS was to clean it up and at the

[‡]We have submitted the patch to MPICH-G2 developers for integration.

same time to lift some of its restrictions. These efforts will be described in the rest of this chapter.

Cellular automata are *localised*: the future state of a cell is only dependent on the current state of itself and its neighbours (in a two-dimensional lattice, the four adjacent neighbours are usually considered). If the lattice is spatially decomposed for a parallel simulation, the inter-process communication pattern will follow that of the underlying cellular automaton: only the logical processes that contain the adjacent cells will need to communicate with each other. Provided that the cellular automata lattice is arranged in a torus (to avoid the problem of border conditions), a topology of the communication channels as in Fig. 2.6 (*left*) will be created.

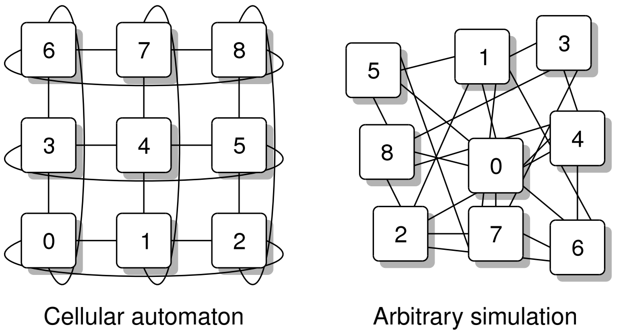


Figure 2.6: Process topologies in APSIS.

APSIS provides support for setting up this sort of topology. However, a torus topology was the only one allowed. This simplified the GVT algorithm, as during the GVT calculation the kernel needed to consider a smaller number of channels, known in advance.

A kernel compliant to the Jeffersons's [96] Time Warp protocol is normally assumed to be able to support any sort of communication topology. Therefore, we have extended the GVT algorithm of APSIS to support arbitrary topologies, such as the one in Fig. 2.6 (*right*). We have made sure that if a communication channel is never used, it only minimally contributes to the overhead of the GVT calculation, so that the still most commonly run cellular automata simulations do not suffer a significant performance degradation.

2.7 Execution determinism

The property of determinism is extremely useful when debugging a simulation model or a simulation kernel. Not every aspect of the execution can be deterministic, for example the run time in general is not. When we talk about the execution determinism here, we actually mean a deterministic result: during subsequent runs of a particular simulation, an identical result should be produced every time.

According to Jefferson et al. [96], *a process, when rolled back and restarted in an earlier state with the same input messages as before, should generate exactly the same output* *messages.* A process that obeys this constraint is said to be *rigidly deterministic in its input-output behaviour*.

This property is required for some Time Warp optimisations such as lazy cancellation (see Sec. 4.2.2) to work. However, it is not immediately obvious whether the rigid determinism is needed or useful for stochastic simulations. Normally, only some statistical data are extracted from such simulation runs, so as long as a simulation adheres to its stochastic model, the results should be fine, irrespective of whether the runs were deterministic or not. There is some precedent to such claims, e.g. the *temporal uncertainty* described by Beraldi and Nigro [24]. In their approach, events are not assigned exact timestamps, but time intervals. Since a simulated model is normally only an approximation of reality, such a relaxation should not have negative consequences for the validity of the model. On the other hand, it can reduce the number of rollbacks, thus improving the run-time performance.

Still, if a user wanted a deterministic execution, APSIS lacked the necessary support for it, so we felt obliged to implement it.

2.7.1 Random number generation

The primary source of non-determinism in Time Warp is the inter-process communication, and in particular rollbacks. They are asynchronous events that can take place at any time, triggered by the arrivals of messages from other logical processes. For all practical purposes, rollbacks occur randomly.

Stochastic simulations make intensive use of random number generators (RNG). For a simulation to be deterministic, these generators must reproduce exactly the same stream of numbers after the rollback as before it, as shown in Fig. 2.7 (*left*). This is possible, provided that a pseudo-random number generator is used, i.e. one where the numbers

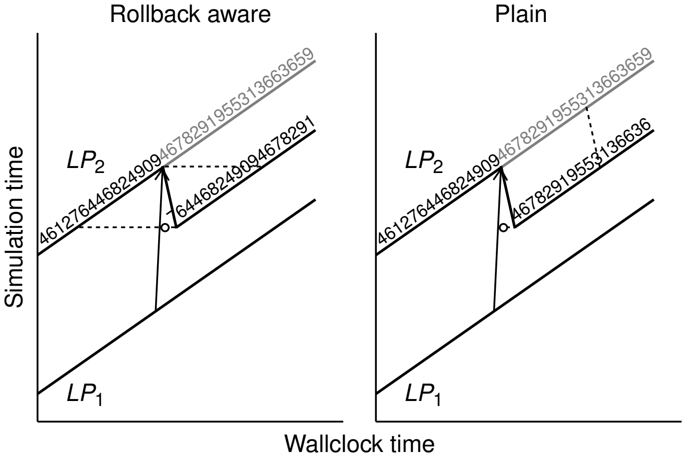


Figure 2.7: Possible sequences of pseudo-random numbers generated across a rollback with different generators. Grey indicates a hypothetical sequence if no rollback had taken place.

generated are only dependent on the internal state of the generator. If this internal state is saved together with the rest of the simulation state and restored during the rollback, the generator will reproduce the same pseudo-random stream. Otherwise, a continuation of the stream since before the rollback will be generated, as shown in Fig. 2.7 (*right*).

APSIS did not provide any support for the generation of random numbers. Simulations were usually using standard C library functions such as `drand48`, typically without bothering to save the generator's state before drawing a new pseudo-random number. While saving the state is simple, doing it optimally is not necessarily so. If a random number is drawn multiple times per event, only the state prior to the first drawing needs to be saved. This happens automatically with the copy state saving, but extra effort is necessary with the incremental state saving.

That is why we have decided to provide a built-in support for random number generation. The user can request multiple generators per each logical process, and the kernel makes an effort to initialise each of them with a unique seed. The state of the generators is saved automatically by the kernel with the optimal frequency. Simple `drand48`-style generators are used, because they feature a small, six bytes state.

2.7.2 Deterministic event ordering

Until now, we have been talking about the "timestamp order" of events in the input queue. But what if two events have an identical timestamp? Such a situation is actually quite likely in APSIS, because by default it uses an integer variable to represent the simulation time (although it can be configured to use a floating point type instead). If we allow such "simultaneous" events to be processed in any order, then two subsequent runs of a simulation might produce two different results – a situation that we are trying to prevent.

If the two events with an identical timestamp come from the same source, the problem is easily solvable: the FIFO property of the communication channels in APSIS guarantees a deterministic arrival order, and the event that arrives as second will be put in the queue after the first one.

However, the arrival order of events from two different sources is non-deterministic. A secondary sorting key must thus be established to predictably order such events. In APSIS, we have decided to use the *sender* field of an event for this purpose. Each logical process has a unique integer associated with it, and that integer is put in the *sender* field as an identifier of the source. Now, if two events have the same timestamp, the one with a lower *sender* will be placed in the input queue first. The tie is broken.

Unfortunately, as is often the case in such situations, solving one problem causes another one to appear.

An example of a problematic case is depicted in Fig. 2.8. Process LP_2 executes a locally scheduled event e_0 at the simulation time t_0 (the second black box from the left in Fig. 2.8). As a result, a new event e_1 is scheduled on process LP_1, with the schedule time equal to the send time (t_0). Processing this event on LP_1 results in the scheduling of yet another event, e_2, on process LP_2. This event, again, has the schedule time equal to its send time (that equals t_0). We thus have two events at LP_2, e_0 scheduled locally and e_2 scheduled by LP_1, both with the schedule time t_0. Now, because the unique identifier

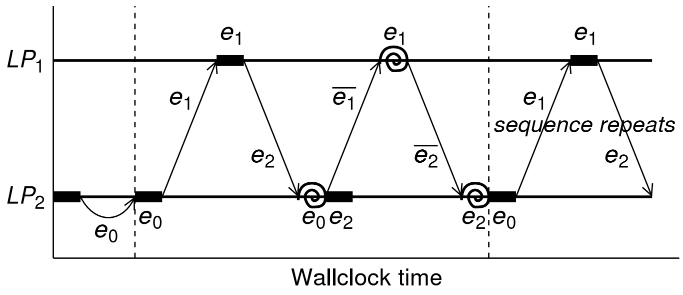


Figure 2.8: A paradoxical case of an event rolling back its causal predecessor. All events take place at the simulation time t_0. Black boxes denote the forward processing of events, swirls denote rollbacks.

of LP_1 is lower than that of LP_2, e_2 needs to be processed first. We have a paradox: an effect (e_2) precedes the cause (e_0)! e_0, that has already been processed, will have to be rolled back (the left-most swirl in Fig. 2.8), resulting in the anti-message $\overline{e_1}$ being sent. When annihilating e_1, LP_1 will send $\overline{e_2}$ to annihilate e_2. e_2, the cause of this mess, has just ceased to exist! We can now process e_0 again, causing the same message exchange to repeat, *ad infinitum*. We have an endless loop. The reason is the inability of the simulation kernel to track the dependencies between the events: it can only compare their timestamps, and if they are equal, it can make a wrong choice. This is the case if a *zero lookahead loop* exists between LP_1 and LP_2: if in both directions events can be scheduled with the schedule time equal to the send time (*zero lookahead* thus), and one of these events can cause the other, then this problem is bound to occur. The solution is to write the simulation in such a way that these loops do not occur. Alternatively, the timestamps could be extended: if Lamport's vector clocks [105] were used, the problem would not occur. Unfortunately, the overhead of vector clocks would be too high for a large number of processes.

In the end, we have decided not to fix this problem at the simulation kernel level. Instead, we made sure that our simulation models cannot trigger it. With the Ising spin simulation, externally scheduled events do not result in any new events being scheduled, which prevents any loops from occurring. However, such danger does exist with PHOLD, but the problem could easily be solved by ensuring that new events are always scheduled into the future.

The problems with the ordering of simultaneous events have been known for years, and different solutions have been proposed. For example, as discussed by Reiher et al. in [155], TWOS used the whole message bodies to break the ties. Good summaries on the subject are provided by Rönngren and Liljenstam [160] or Jha and Bagrodia [97].

2.7.3 Influence on run-time behaviour

We have discussed our efforts to obtain deterministic simulation runs, but how do the modifications made affect the run-time behaviour of the kernel? Figure 2.9 presents a simple performance comparison of the Ising spin simulation running with either a rollback aware or a plain random number generator. On the x axis, we have the probability

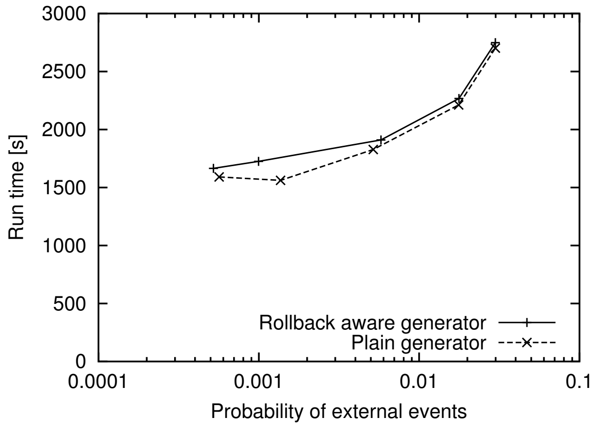


Figure 2.9: Performance comparison of the Ising spin simulation running with a rollback aware or plain pseudo-random number generator. Note the logarithmic scale on the *x* axis.

of encountering an externally scheduled event (the simulation model allows to tune the number of these events quite easily), on the *y* axis we have the run time in seconds (thus, the lower the better). Not surprisingly, the runs with the plain generator perform better, since such a generator introduces a lower overhead. The difference is largest for low external event probabilities, because with little inter-process communication, local execution overheads become more pronounced.

A far more interesting phenomenon can be observed when comparing the rollback length histograms obtained for the same simulation running with the two random number generators, as shown in Fig. 2.10. Here, on the *x* axis we have the rollback length expressed in the number of rolled back events, and on the *y* axis the probability of a rollback having that length. We will discuss such histograms at length in Sec. 3.3, for now

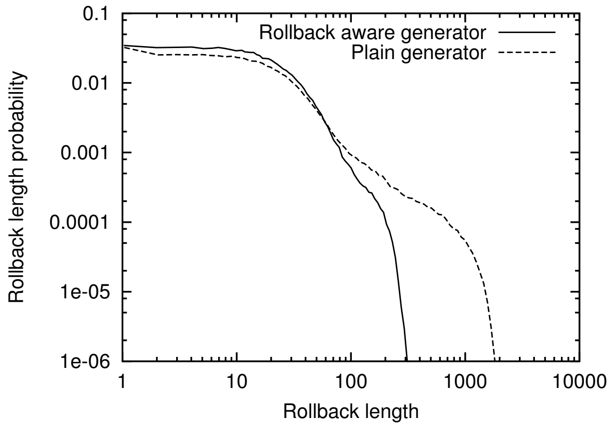


Figure 2.10: Rollback length histogram obtained for a simulation running with a rollback aware or plain pseudo-random number generator. Note the logarithmic scale on both axes.

our focus is exclusively on the maximum rollback length. As can be observed, this maximum is over five times larger for the plain generator than for the rollback aware one (please note that a logarithmic scale is used). For such a relatively minor modification to the kernel, this is an amazing change in the behaviour! Even more significantly, research into the rollback behaviour of Time Warp has previously been performed in our group by Overeinder et al. [134, 135, 164, 168], and some interesting conclusions depended on the presence of the long rollbacks. A further study and an explanation of the observed phenomenon will be presented in Sec. 3.4.

2.8 Conclusion

In this chapter, we have introduced the parallel simulation kernel APSIS, that we have used for our development work and the experiments described in this thesis. It features event scheduling world view, distributed memory model, incremental state saving, optimism control via moving virtual time window, and can run on top of popular communication libraries such as PVM or MPI. Compared to the better known Time Warp kernels, it lacks some of the more advanced features, such as support for the user-allocated dynamic memory or for the I/O events. This is not because of some fundamental limitations of its architecture, but rather due to a lack of need in the simulation models that have been run on it. Because APSIS is not packed with features, its codebase is simple, so it does not take too much effort to extend it when it is necessary to do so, a characteristic that is most useful when trying out new, experimental ideas.

We have introduced the two simulation models used. Ising spin is a model of magnetism, our implementation is a spatially decomposed asynchronous cellular automaton. PHOLD is a synthetic workload, featuring a fixed number of events travelling through the system, forming event threads. We will be using two models to ensure that our findings are generally applicable. PHOLD is typically far more communication intensive than the Ising spin simulation, and, unlike the Ising spin, it is message-initiated, so it is more difficult to optimise.

We have also described the machine used for the majority of the experiments: the wide area distributed DAS-2 cluster.

We have extended APSIS in two areas. First, we generalised it to support arbitrary process topologies, not only a torus one. Second, we added support for deterministic execution of a simulation, by extending the kernel with rollback aware random number generators and by providing a deterministic processing order of simultaneous events. These two extensions make it a lot easier to port other parallel simulations or to implement some advanced optimisations.

While testing the influence of the new extensions on the run-time behaviour of APSIS, we have observed an interesting phenomenon. If a rollback aware random number generator is used, the longest rollbacks observed are very significantly shorter than if a plain generator is used. Because of the possible implications to research on rollback behaviour previously carried out in our group, this phenomenon should be studied further. This will be one of the topics of the next chapter.

3

Modelling a Parallel Simulation Kernel*

3.1 Introduction

In Sec. 2.7.3, we have discussed the influence of the random number generator used on the run-time behaviour of the APSIS kernel. Some of those results clearly indicate that our understanding of this behaviour is incomplete.

Analysing the behaviour of a parallel simulation kernel is difficult. The popular technique of logging the necessary information while the program is running and analysing such a trace file *post-mortem* would be impractical in this case. Instead of one, consistent trace file, we will get a number of them, separately for each logical process. If any non-trivial level of detail is required, the volume of data will be huge, given that LPs handle tens of thousands of events per second. Not only is the subsequent processing of such data cumbersome, collecting them in the first place incurs a significant overhead. The non-deterministic run-time behaviour of the parallel simulation kernel can be significantly affected by this overhead, seriously limiting the usefulness of the obtained trace file. This is sometimes referred to as the *Heisenberg principle* of performance monitoring.

Therefore, another approach would be preferable. As we have stated in Sec. 1.1, *modelling is a key scientific approach towards a better understanding of reality.* This statement applies to our own research as well – we should try to build a model of an optimistic PDES kernel.

Our aim is to build a model that would explain the run-time behaviour observed in Sec. 2.7.3. How accurate does such a model need to be? Our system is highly complex, as it consists of three interacting layers:

*This chapter is partly based on: K. A. Iskra, G. D. van Albada, and P. M. A. Sloot. Rollbacks in Time Warp – Analysis and Modelling. In *Proceedings of the 8th Annual Conference of the Advanced School for Computing and Imaging*, pp. 88–94, Lochem, The Netherlands, June 2002.

- target simulation,
- parallel simulation kernel,
- hardware platform.

If we were to build a model capable of accurately predicting the performance, each of these layers would have to be modelled at the right level of abstraction to achieve a good result. Finding this level could prove challenging, given that a Time Warp kernel typically utilises resources very intensively, pushing the machine "to the limits".

An accurate model of a Time Warp kernel would allow us to answer a number of other questions. Fundamental questions such as: what drives the dynamic behaviour of a Time Warp simulation kernel? Application characteristics, Time Warp extensions and implementations, or hardware parameters? From an engineering perspective, a model can give answers to performance and efficacy effects of the different protocol design parameters.

However, we are primarily interested in a qualitative description of the rollback behaviour, not in a quantitative accuracy. We would be very interested in extracting from the model to be built e.g. a rollback length distribution, so that we could compare at least its shape to the actual results from the parallel simulation kernel. This should give us an insight necessary to understand the results from Sec. 2.7.3. Hopefully, a simple model will be sufficient to satisfy this goal.

We will begin our modelling using an analytical approach. Later in the chapter we will also describe our efforts to build a computer simulation of a parallel kernel. From a microscopic simulator modelling the processing of single events, it must be possible to extract the lengths of the individual rollbacks, and hence, over a longer simulation time period, also the rollback length distribution.

3.2 State of the art

Analytical modelling

Analytical modelling of parallel discrete event simulation kernels has been done before. A summary of many early efforts was provided by Nicol and Fujimoto [132]. The primary application of these models is in the area of performance prediction. Most of them are based on very simple assumptions, such as an instantaneous communication or zero-overhead rollbacks, and are often limited to just two logical processes.

An example of this can be found in the work of Felderman and Kleinrock [59], where the Markov chain approach is used, assuming geometrical distributions of the event processing time and the simulation time increment. The processes are *self-sustaining* (more about that later) and schedule external events for the current simulation time of the sender. The focus is on the speedup, in particular when a cost is added to the state saving. A formula is derived that allows one to determine if any speedup can be achieved, depending on the cost of the state saving and the amount of interaction between the processes. The authors have subsequently extended the model in [58] to handle multiple processes, assuming a uniform communication between the processes. The cost of processing an event follows an exponential distribution, and the simulation time increment is fixed. The model tracks the progress of GVT by analysing the slowest performing process(es), as these processes determine the progress rate of the entire system. Bounds on the speedup are derived, as functions of the number of processes and the amount of interaction between them. Interestingly, the lower bound is found to increase linearly with the growing number of processes, suggesting a good scalability, although the model does not include the costs of communication and rollbacks. A similar model was proposed by Nicol [131], with the exception that the event processing characteristics are reversed (fixed processing cost, exponential simulation time increment), and that the model takes into account the costs of state saving and rollbacks (the latter simplified). With these assumptions, the derived performance bounds diverge significantly from those by Felderman and Kleinrock [58]: the upper bound on speedup in [131] lies below the lower bound in [58], showing the sensitivity of such models to the assumptions made.

The above models assume a self-sustaining (or self-initiating) target simulation, that is, one where each logical process schedules its own stream of local events to keep itself occupied, and the events scheduled on other logical processes play a secondary role (or, at least, that is what the models assume). Gupta et al. [83] analyse the case of a *message-initiated* model, one where there is no stream of local events; instead, the future events scheduled are uniformly distributed among all the logical processes. Both the event processing time and the simulation time increase are assumed to be exponentially distributed. Like in most other models, the costs of communication, state saving and rollback are assumed to be negligible, but the model does take cascading rollbacks into account. The numerically approximated results obtained from the model are validated against the measurements from a real Time Warp kernel. While the divergence in quantities such as the *efficiency* (the percentage of processed events being committed) is rather large, the approximation of speedup is pretty good, and quite in line with the expectations (the analytical results exceed the experimental ones, but the model does not consider many of the overheads).

The issue of the optimality of Time Warp is discussed by Lin and Lazowska in [110]. Different Time Warp cancellation protocols are compared with each other and with conservative simulation schemes, using the critical path analysis. With no operational overheads, and provided that correct computations are never rolled back, Time Warp can be proved to perform optimally. These conditions are of course unrealistic, but it is proved that with realistic assumptions in place, applying a lazy cancellation scheme (see Sec. 4.2.2) allows to reach the optimal performance, or even exceed it (a super-linear speedup). This is possible by processing the events from the critical path out of order: if the results eventually turn out to be correct, the critical path is effectively shortened. Also, Time Warp is shown to be able to always outperform conservative schemes in *feed-forward* network simulations (i.e. networks described by a directed acyclic graph). Comparison between the Time Warp and the conservative schemes can also be found in the work of Lipton and Mizell [114], that we have already discussed in Sec. 1.3.3. In short, it proves that an optimistic simulation can arbitrarily outperform a conservative one, but a conservative simulation can only outperform an optimistic one by a constant factor.

An analysis focusing on the rollback behaviour was performed by Lubachevsky et al. in [120]. Every rollback is modelled as a node of a tree, the system is analysed using a model called *branching random walk with a barrier*. Several different patterns of the rollback behaviour are identified, as are the conditions necessary for a stable and efficient run. The authors identify three sorts of rollback cascades (echo, gushing and wildfire), distinguished by whether the subsequent rollbacks grow or subside and whether the rollbacks spread through the system or only affect a fixed group of processes. Sudden phase transitions can occur between stable and unstable systems, triggered by a growing number of processes or even taking place when the simulation is already running.

The overhead of the state saving, most of the time ignored by others, is the focus of Palaniswamy and Wilsey [138]. Periodic and incremental methods (see Sec. 2.2) are compared, and conditions are derived that show when either of them performs better. Another often neglected limitation is analysed by Akyildiz et al. in [4], where a model is presented that takes into account the limited amount of memory available in real systems. The model analyses the *cancelback* protocol of Jefferson [95], used to recover free memory by discarding some objects if the kernel executing on a shared memory machine runs out of space.

Looking at more recent efforts, Tay et al. [179] present a quite complete model, that, using a probabilistic approach, supports multiple (homogeneous) processes, takes into account the communication overheads (both buffer access and message transmission times), state saving costs, rollback cascades, and even an optimism throttling scheme, although the cost of performing a rollback is not considered. The simulated elapsed run time is validated against a real Time Warp kernel, and a good match can be observed. Another model was proposed by Quaglia et al. [146], that, unlike most other models, takes into account the effects of aggregation of multiple LPs on one physical processor. The overhead of checkpointing and rolling back is taken into account, but the communication is assumed to be instantaneous. The model provides the upper bound on the completion time of a Time Warp simulation; an additional algorithm is presented that makes it easy to choose between Time Warp and a sequential simulation, and, in the former case, to pick the optimal values for the number of processors and the state saving interval.

An interesting analytical approach can be found in the publication of Kolakowska et al. [102]. Scalability modelling of a simulation with nearest neighbours only communication is performed. Using the concepts of *non-equlibrium surface growth* and *kinetic roughening* from statistical physics, the *time horizon* (the set of all local simulation times at a particular wallclock time) is modelled. A basic conservative scheme is found to be asymptotically scalable, i.e. in the limit of an infinite number of LPs, a conservative simulation has a non-zero rate of progress. However, the simulated time horizon roughens with a growing number of LPs, making practical measurements difficult. This can be prevented by introducing a moving time window, or by adding weak random interactions among LPs, as described by Korniss et al. [104]. While the above models were stochastic, an explicit connection between the utilisation and the microscopic structure of the simulated time horizon is made by Kolakowska et al. in [103], where a theoretical mean utilisation and speedup as a function of the system size are derived. Based on this work, Shchur and Novotny [165] have developed the *freeze-and-shift* synchronisation algorithm suitable for distributed spatially decomposed simulations of models such as the Ising spin. The execution is divided into cycles. Within each cycle, only computations are performed, no communication between processing elements takes place. Each processing element runs a number of threads performing the functions of logical processes, each responsible for a single lattice site. The timestamps of the threads responsible for the boundaries are frozen. The rest of the threads can progress, but they eventually stop once they run out of safe events (a conservative synchronisation algorithm is used). The domain is then shifted between the processing elements by half a phase, so that the most advanced threads become new boundaries and the simulation can continue. Communication thus only takes place in short bursts, between the long computation phases, so the algorithm should be suitable for running on distributed resources such as the Grid. The idea actually bears some resemblence to the work of others, e.g. the Breathing Time Buckets of Steinman [173], where the execution is also divided in cycles and the communication only takes place between them. However, that algorithm is optimistic and the communication is limited to prevent rollbacks from spreading through the system.

Still, the focus of almost all of these publications is on the prediction of the performance bounds or on the identification of parameters responsible for stable simulation runs, etc. On the other hand, as already discussed, we are more interested in predicting the rollback behaviour, for example in the form of obtaining a rollback length distribution.

Simulating PDES

While the literature on the analytical models of the PDES kernels is extensive, it appears that very little has been done in the area of simulating the parallel kernels on a computer. As a matter of fact, we were only able to identify one publication with goals similar to ours:

Ferscha et al. [61] present a performance data mining approach based on the simulated execution of skeletal implementations of parallel simulation protocols, performed with the help of a performance prediction tool N-MAP. The properties of the simulated kernel are highly configurable: it can be made either conservative or optimistic, and for either of these approaches a number of options are available (GVT reduction and null-message sending initiation for conservative protocols; cancellation, state saving and optimism control for optimistic ones). The target simulation is modelled using a directed graph of simulation objects, expressing the event causalities and communication frequencies. The platform attributes such as the number of processors to use are also tunable. An automated performance prediction testbed can then conduct the analysis, based on the incremental code development and simulation. A full Cartesian product of all the parameters of interest is generated, and each combination is separately explored via experiments. Numerical output can then be put into a data mining tool, where, with the help of statistical reasoning methods, conclusions can be drawn.

3.3 Self-organised critical behaviour

In an experimental study of the dynamic run-time behaviour of an optimistic Time Warp kernel running an Ising spin simulation (see Sec. 2.4.1) conducted in our research group by Overeinder et al. [134, 135, 164, 168], it has been shown (see Fig. 3.1 (*left*)) that the distribution of the rollback length probability as a function of the number of rolled back events exhibits the power-law behaviour over a wide range of parameters:

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

This is rather unusual, as most systems with short-range interactions exhibit exponentially decaying correlations (see Fig. 3.1 (*right*)). Only with a careful tuning can a *critical* state of a system be achieved, i.e. one where correlations are potentially infinite, in other words, where *scale invariance* can be observed.

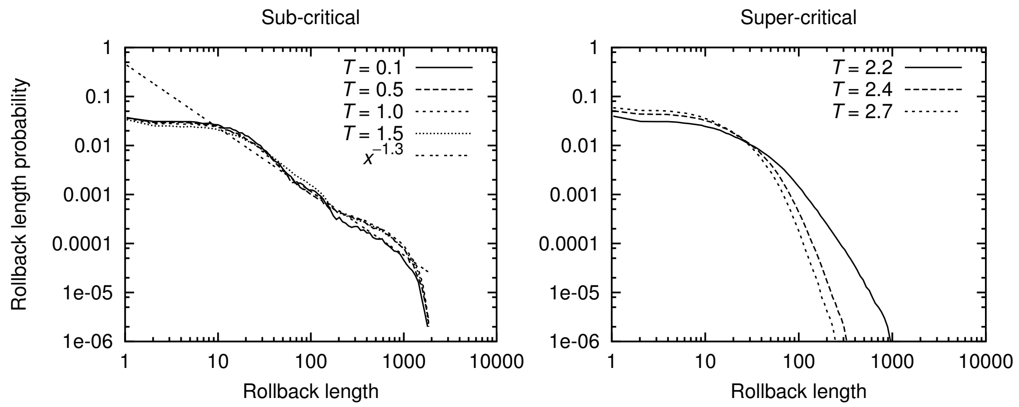


Figure 3.1: Rollback length histograms with the Ising spin simulation: power-law distribution for sub-critical temperatures, exponentially decaying distribution for super-critical temperatures. *Plots courtesy of Dr. Benno Overeinder, UvA (now at VU).*

Therefore, a speculative conjecture has been made, that the Time Warp dynamics can be characterised as a *self-organised critical system* (SOC, see e.g. Bak et al. [13]). Such a system spontaneously evolves towards a critical state, and minor disturbances can trigger a reaction of an arbitrary size, potentially affecting the state of the whole system. The presence of scale invariance, i.e. of the power law being observed for many measurables, is a strong indication of the SOC behaviour.

In case of a Time Warp kernel, we focused on the rollback length expressed in the number of rolled back events, as shown in Fig. 3.1. The characteristic power-law like distribution (with $\tau = 1.3$) can be observed for any sub-critical model temperature. This would imply that rollbacks are scale-invariant, i.e. that small and large rollbacks are different manifestations of the same phenomenon. In the super-critical temperature range, the more usual exponentially decaying distribution is found.

The explanation for this phenomenon, as presented by Overeinder et al. in [134, 135, 164, 168] is as follows: when the simulation processes move forward, their unsynchronised simulation times diverge from each other, slowly but steadily building the tension in the system. Eventually, the state of a site at the border will change, forcing inter-process communication that in some cases results in a rollback. A rollback is an almost instantaneous relaxation operation that synchronises the simulation times, thus reducing the tension. This behaviour is analogous to that of sandpiles and earthquakes, two well-known examples of self-organised critical systems (see e.g. Turcotte [183]). The power law is not observed in Time Warp for super-critical temperatures of the Ising spin model, as the high rate of inter-process communication prevents any significant tension from building in the system. Besides, at high temperatures the correlation between the neighbouring sites is low, while it has been observed that correlation is an important factor in the emergence of the power-law behaviour at low temperatures.

3.3.1 Critical behaviour revisited

The explanation presented above, while relatively clear, left some issues open, especially in the light of experiments performed later.

First, we have determined that the rollback behaviour changes in time, especially for the sub-critical model temperatures, as can be seen in Fig. 3.2. Because the simulation

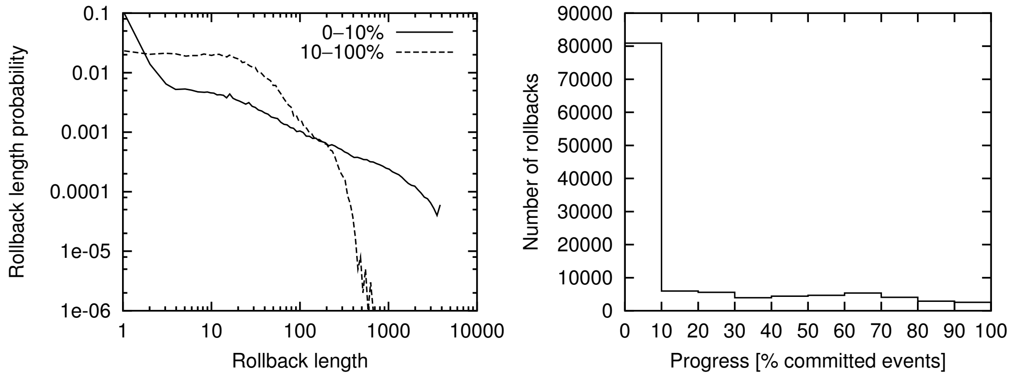


Figure 3.2: Changes in the rollback behaviour as the simulation progresses, for the Ising spin simulation with $T = 1.0$: rollback length histogram (*left*), number of rollbacks (*right*).

starts from a randomly initialised cellular automaton lattice, major rearrangements take place at the beginning, forced by the strong correlation between the neighbouring sites. This transient phase can be characterised as a period of instability. As it turns out (see Fig. 3.2 (*left*)), it is this short phase, responsible for no more than 10% of the simulation's progress (expressed as a percentage of the committed events), that exhibits a power-law like behaviour; the rollbacks taking place afterwards follow an exponentially decaying distribution. Thus, even though the initial phase dominates the simulation so far as the number of rollbacks is concerned (67% of rollbacks take place then, see Fig. 3.2 (*right*)), it is only a transient phenomenon.

Another problem, that we have already signalled in Sec. 2.7.3, is the disappearance of the power-law behaviour once a rollback aware random number generator has been installed (see Fig. 2.10 on p. 33). As it turns out, the power-law behaviour reappears as soon as the VTW size is increased, see Fig. 3.3. While changing the random number generator or the VTW size might appear insignificant at first, in fact it has a potential to significantly

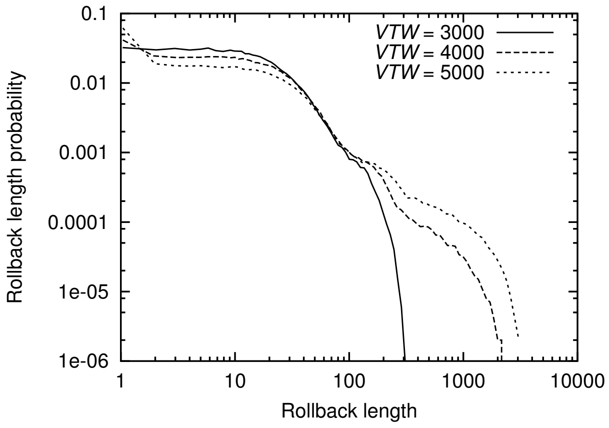


Figure 3.3: Rollback length histograms for the Ising spin simulation with $T = 1.0$ and a rollback aware random number generator, for different sizes of the virtual time window.

affect the run-time behaviour of the kernel. We will present an explanation for this below.

3.4 Random number generators and optimism

We have thus determined that if a rollback aware random number generator is used, the power-law behaviour, initially absent, reappears as soon as the VTW size is increased. We will now present an interpretation of this phenomenon. Figure 3.4 shows two alternative initial scenarios to be considered. Three logical processes (LP_1–LP_3) of a running simulation are being analysed. The local virtual time of each of them is different, but they generally progress at the same rate. The solid lines denote already executed operations, whereas the dashed parts present speculation of how the simulation could progress. At some point, the processes will want to schedule events on each other. The cases un-

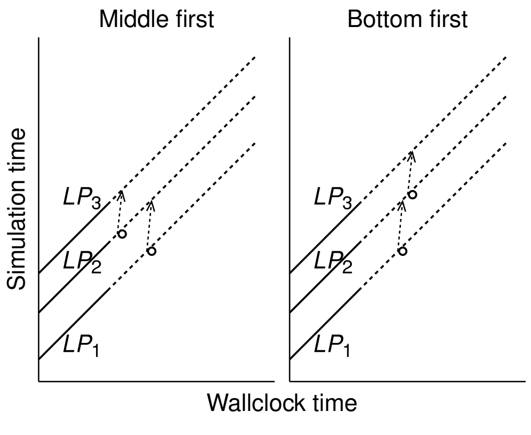


Figure 3.4: Initial states of two possible rollback scenarios.

der consideration are the ones where the processes with a lower LVT schedule events on the processes with a higher LVT. Because in the Ising spin model simulation used, the schedule time of external events always equals the sender's current LVT (as indicated with the small circles in the figures), scheduling in such cases is guaranteed to result in a rollback. We will analyse these rollbacks in greater detail. Figure 3.5 expands on the

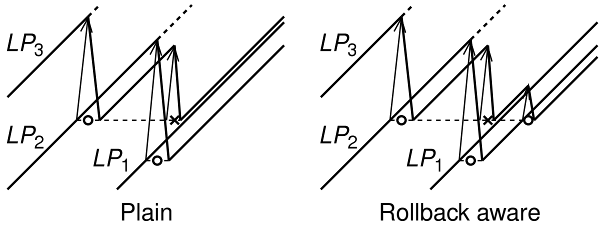


Figure 3.5: Rollback behaviour for the *middle first* scenario with different random number generators.

left case of Fig. 3.4 by presenting the behaviour of a Time Warp kernel with a plain and a rollback aware random number generator. Let us focus on the case with a plain RNG (Fig. 3.5 (*left*)) first. The middle process LP_2 schedules an event on the top process LP_3 (the thin arrow pointing upwards). In order to process the event, which is in its simulation time past, process LP_3 must roll back to that point in time (the steep thick line going down and to the right). Once it has rolled back, LP_3 can take the externally scheduled event into account and restart the forward processing. Soon afterwards, the bottom process LP_1 schedules an external event on the middle process LP_2, that causes the latter to roll back. While rolling back, process LP_2 notices that it has over-optimistically scheduled an external event on LP_3, so it sends an anti-message to LP_3 (to distinguish them from positive events, the timestamps of anti-messages are marked with a small × in the figures). The anti-message causes another rollback on LP_3 before it can be taken into account. Once this is over, all three processes can move forward in unison again. Using a rollback aware RNG (Fig. 3.5 (*right*)) can add an extra step at the end. Because RNG rolls back with the simulation, when process LP_2 restarts the forward processing after it was rolled back by LP_1, it gets the same stream of random numbers and consequently schedules the external event on LP_3 again, that could cause one more small rollback on LP_3. So, in total, three middle-sized rollbacks can be observed and, in the case of the rollback aware RNG, one small one. Let us now focus on Fig. 3.6, that expands on the right case of Fig. 3.4. In this case, the differences between left and right side of the figure are more significant. Let us again focus on the case with a plain RNG (Fig. 3.6 (*left*)) first. Process LP_1 schedules an external event on process LP_2, that causes the latter to roll back. It is known that if process LP_2 had not rolled back, it would have scheduled an external event on process LP_3 soon afterwards. And, indeed, that is what happens soon after the rollback is over: LP_2 schedules an external event on LP_3, that triggers a very large rollback. Moving on to the case with a rollback aware RNG (Fig. 3.6 (*right*)), the scheduling of the external event by LP_2 on LP_3 does not happen so soon. First LP_2 must go again through the rolled back sequence of random numbers, and only then is an event scheduled on LP_3. With either generator, there is a middle-sized rollback followed by a very large one. So, stat-

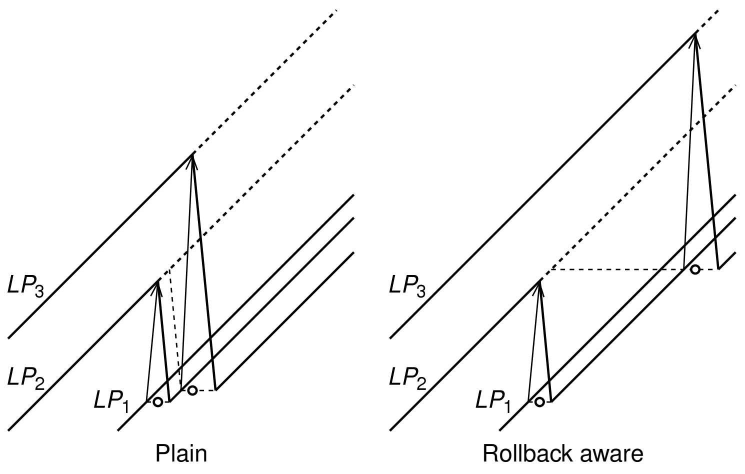


Figure 3.6: Rollback behaviour for the *bottom first* scenario with different random number generators.

istically, there should not be any difference. However, until now, one important factor was not taken into account, namely the virtual time window. In Fig. 3.6 (*right*), process LP_3 must be able to significantly advance forward in order for the large rollback to occur. Figure 3.7 shows the same situation as before, but with a virtual time window. As we can see, for a particular range of values, it is possible that the window does not influence the case with a plain RNG, but it might limit the excessive optimism in the case of a rollback aware RNG. This in turn results in a significantly reduced maximum rollback length.

This explanation is by no means complete. It does not handle all the possible cases and does not consider the probabilities of particular cases taking place. Nevertheless, it

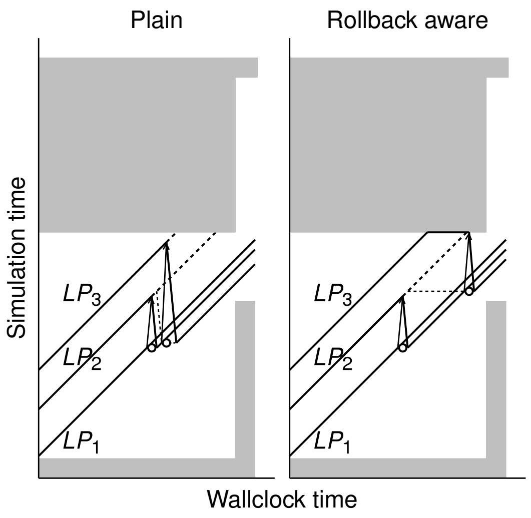


Figure 3.7: Rollback behaviour for the *bottom first* scenario with a virtual time window and with different random number generators.

shows on some specific examples the mechanism of how changing the properties of the random number generator can affect the simulation's rollback behaviour in the presence of a VTW throttling, confirming the previously obtained experimental results.

3.5 Analytical rollback modelling

To remind the reader, we are interested in obtaining, through whatever means, information on what a rollback length histogram *should* look like, so that we can make an informed comparison with what we actually get from the parallel simulation kernel.

Below, we will attempt to solve the problem analytically. The model we are about to build will simulate the run-time behaviour of the simulated application, i.e. the Ising spin model, and from there we will derive the rollback length distribution. This should be possible to do, thanks to a very simple structure of the Ising spin model.

Two approaches are presented below, differing in how the rollback length is defined.

3.5.1 Rollback length expressed in simulation time

The simulation time in the Ising spin model is increased by random intervals from a negative exponential distribution with the mean of $1/\lambda = 1$. The probability density function for the exponential distribution is as follows:

$$f(t) = \lambda \mathrm{e}^{-\lambda t} \tag{3.1}$$

Equation (3.1) can be interpreted as the probability that a process has increased its simulation time in a single step by t. From that, we can calculate:

$$f_n(t) = \frac{t^{n-1}}{(n-1)!}\mathrm{e}^{-t} \tag{3.2}$$

Equation (3.2) is the probability distribution of the time advance t after n random intervals from a negative exponential distribution with the mean of $1/\lambda = 1$ (it is the Erlang distribution). We can now attempt to calculate the probability distribution of the distance in simulation time between two processes after n steps (we assume that the event processing time is constant):

$$h_n(d) = \int_0^\infty f_n(t) f_n(t+d)\,dt \qquad \text{if} \qquad d \geq 0 \tag{3.3}$$

$$h_n(d) = \int_{-d}^\infty f_n(t) f_n(t+d)\,dt \qquad \text{if} \qquad d < 0 \tag{3.4}$$

What (3.3) provides is a function that specifies the probability that, after n steps, the two processes have diverged from each other by the time difference d. Therefore, if the lagging process scheduled at that point on the other process an event for its current simulation time (as the Ising spin simulation does), the rollback would have the length of d (in time units) with the probability given by (3.3). Equation (3.3) is only defined for the positive values of d, (3.4) covers the negative values, but it is actually redundant given that the distribution obviously must be symmetrical. Further calculations lead to:

$$h_n(d) = \frac{e^{-|d|}}{[(n-1)!]^2} \int_0^{\infty} [t(t+|d|)]^{n-1} e^{-2t} dt = \frac{e^{-|d|}}{(n-1)!} \sum_{i=0}^{n-1} \frac{(n-1+i)!\,|d|^{n-1-i}}{(n-1-i)!\,i!\,2^{n+i}} \tag{3.5}$$

Therefore, an exact, analytical solution does exist. However, it is not always practical, particularly for large (in the range of thousands) values of n, for which the factorials are not representable in typical floating-point computer arithmetic. Another approach would therefore be preferable.

$$h_{n+1}(d) = \int_{-\infty}^{+\infty} h_n(t)\, h_\Delta(d-t)\, dt \tag{3.6}$$

Equation (3.6) provides an alternative definition of $h_n(d)$, one that considers the evolution of the time difference d since the previous step. The evolution function is simply:

$$h_\Delta \equiv h_1 \tag{3.7}$$

Applying (3.6) recursively and substituting (3.7), we get:

$$h_n \equiv \underbrace{h_1 \otimes \ldots \otimes h_1}_{n}$$

h_n is thus an n-convolution of h_1, so we can easily express it in the Fourier domain. From (3.5) we also know the analytical form of $h_1(d)$:

$$h_1(d) = \frac{1}{2} e^{-|d|} \qquad \Longleftrightarrow \qquad H_1(s) = \frac{1}{1+(2\pi s)^2}$$

$$H_n(s) = H_1(s)^n = \frac{1}{\left[1+(2\pi s)^2\right]^n}$$

$$h_n(d) = \int_{-\infty}^{+\infty} \frac{1}{\left[1+(2\pi s)^2\right]^n} e^{2\pi i d s} ds \tag{3.8}$$

Equation (3.8) can easily be solved numerically using the Fast Fourier Transform. Figure 3.8 presents some results. Next to the numerical results from FFT, the results from a rollback simulator are presented, for the number of positive events ranging between 2 and 500. This rollback simulator is a trivial program that simulates the progress of two processes using (3.1), after a predefined number of steps calculates the distance in simulation time between the processes, and finally, after a large number of trials, generates a histogram.

If an extra assumption is made that the rollbacks take no time to perform, then after the rollback the two processes become fully synchronised, so (3.8) can be applied again, taking the time of the rollback as the new start time. This makes it possible to derive the final rollback length distribution, independent of the number of processed events:

$$l(d) = \sum_{n=1}^{\infty} \left[h_n(d)(1-P_{\text{rb}})^{n-1} P_{\text{rb}}\right] \tag{3.9}$$

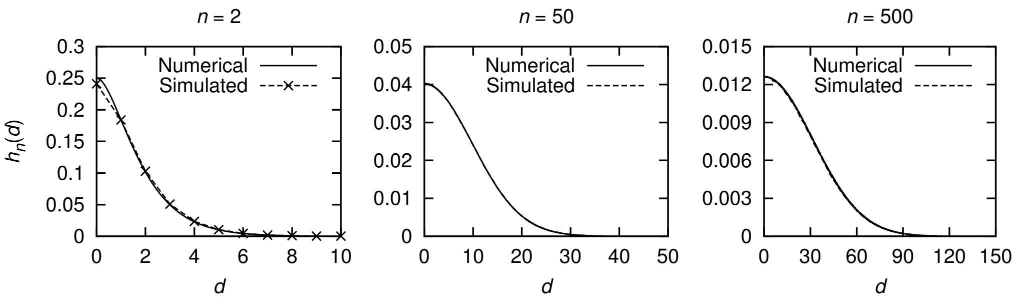


Figure 3.8: Probability distribution function $h_n(d)$ of the rollback length d expressed in simulation time after n forward steps. Comparison of numerical results from FFT (3.8) and results from a rollback simulator based on (3.1).

P_{rb} is the probability of a rollback taking place, which is assumed to be constant for every step; the probabilities in different steps are assumed to be independent. This is in fact not fully correct: a simulation model can specify a constant probability of an external event being scheduled, but this does not translate directly into the rollback probability. Here we assume that when an externally scheduled event arrives, it is equally probable that its timestamp is in the future as that it is in the past. The latter will result in a rollback; consequently, $P_{\text{rb}} = 0.5 P_{\text{ext}}$.

3.5.2 Rollback length expressed in events

In the approach presented above, the rollback length was expressed in the units of simulation time. However, in the rollback histograms generated by the Time Warp kernel (see Fig. 3.1), the rollback length is measured in the number of rolled back events.

Consider two simulation processes, LP_1 and LP_2, that have processed n positive simulation events each. As before, processing every positive event increases the simulation time by a value drawn from a random number generator with a negative exponential distribution, and the time needed to process an event is constant. What is the probability that, should a rollback occur at this point, it will have the length of one, i.e. the minimum possible? This can only occur if the sum of all n time advances of one process, say LP_1, denoted as $t_{1,n}$, is greater than the sum of $n-1$ time advances in LP_2 ($t_{2,n-1}$), but is lower than the sum of all n advances in LP_2 ($t_{2,n}$). This is presented in Fig. 3.9 (*left*). In this case, when process LP_1 schedules an external event on process LP_2 after n events, process LP_2 will need to undo just one event to handle the externally scheduled event.

And what is the probability that the rollback will have length n, i.e. the maximum possible? This can take place only if one of the processes, say LP_1 again, made less progress in all n events added together ($t_{1,n}$) than process LP_2 in just the first event ($t_{2,1}$). Figure 3.9 (*right*) presents this situation.

These probabilities can be described analytically as follows:

$$h_n(k) = 2 \cdot P\left(\sum_{i=1}^{n-k} b_i < \sum_{i=1}^{n} a_i < \sum_{i=1}^{n-k} b_i + c \right) \tag{3.10}$$

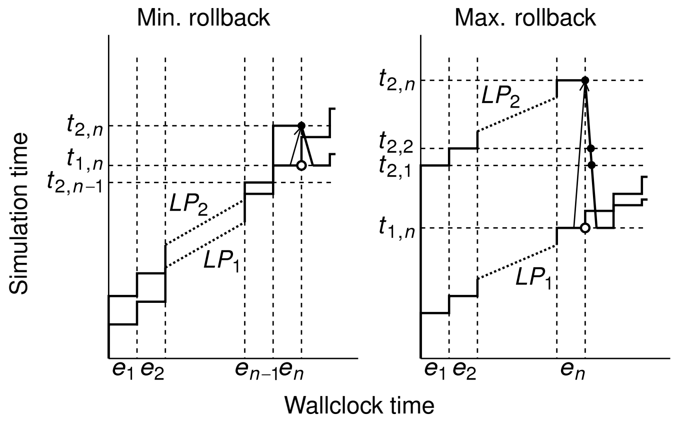


Figure 3.9: Situations needed to get particular rollback lengths: minimum rollback length one (*left*) and maximum length n (*right*).

Equation (3.10) specifies the probability that if a rollback occurs after two processes made n positive steps forward each, then the rollback will have a length of k events, $k = 1,\dots,n$. Basically, it is a probability that the sum of all n time advances of process LP_1 (each advance denoted as a_i), is greater than $n-k$ time advances of process LP_2 (each advance denoted as b_i), but is lower than $n-k+1$ time advances, where the last advance is denoted as c. This probability needs to be multiplied by 2 in order to take into account the symmetrical situation of process LP_2 lagging behind process LP_1.

For negative exponential distributions of a_i and b_i, the sums in (3.10) follow the Erlang distribution, and the equation can be rewritten as:

$$\begin{aligned} h_n(k) &= 2\cdot\int_{c=0}^{\infty}\mathrm{e}^{-c}\int_{\beta=0}^{\infty}\frac{\beta^{n-k-1}}{(n-k-1)!}\mathrm{e}^{-\beta}\int_{\alpha=\beta}^{\beta+c}\frac{\alpha^{n-1}}{(n-1)!}\mathrm{e}^{-\alpha}d\alpha\, d\beta\, dc \\ h_n(n) &= 2\cdot\int_{c=0}^{\infty}\mathrm{e}^{-c}\int_{\alpha=0}^{c}\frac{\alpha^{n-1}}{(n-1)!}\mathrm{e}^{-\alpha}d\alpha\, dc \end{aligned} \tag{3.11}$$

The top version of (3.11) is meant for $k = 1,\dots,n-1$, the bottom for $k = n$. Both versions are solvable analytically, the results are presented below:

$$\begin{aligned} h_n(k) &= \frac{\sum\limits_{i=1}^{n} 2^i\left(\frac{(2n-k-1-i)!}{(n-i)!} - \sum\limits_{j=0}^{n-i-1}\frac{(n-k-1+j)!}{j!}\right)}{(n-k-1)!\,2^{2n-k}} \\ h_n(n) &= \frac{1}{2^{n-1}} \end{aligned}$$

Several sample results for small values of the parameters can be found in Tab. 3.1. Not surprisingly, we get a probability of one for a rollback length of one after one step forward. This is because the time advances are floating point values drawn from a random number generator, so the probability of both processes drawing the same value (which would result in no rollback) is essentially zero. The probabilities of maximum rollbacks

Table 3.1: Probability distribution function $h_n(k)$ of the rollback length k expressed in the number of events, after n forward steps.

	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$
$n=1$	1				
$n=2$	$\frac{1}{2}$	$\frac{1}{2}$			
$n=3$	$\frac{3}{8}$	$\frac{3}{8}$	$\frac{1}{4}$		
$n=4$	$\frac{5}{16}$	$\frac{5}{16}$	$\frac{1}{4}$	$\frac{1}{8}$	
$n=5$	$\frac{35}{128}$	$\frac{35}{128}$	$\frac{15}{64}$	$\frac{5}{32}$	$\frac{1}{16}$

$(h_n(n))$ decrease geometrically by a factor of two. The probabilities of shorter rollbacks do not follow such a simple pattern. Interestingly enough however, the probabilities of rollbacks of length one and two turn out to be the same for all values of $n > 1$. We do not have an intuitive explanation for this phenomenon. We have validated the above probabilities against those obtained from the rollback simulator for 100 000 000 simulated rollbacks, and we have observed an excellent match (correlation coefficient > 99%).

While (3.11) denotes a probability and hence has values in the range of $[0, 1]$, calculating these values for n and k in the range of thousands involves factorials which are not representable even in 128-bit `long double` arithmetic. Maximum rollback lengths observed "in nature" were around 2000 (see Fig. 3.1), while the arithmetic used currently limits the model to around 1500. Not only the range, but also the precision of the arithmetic is a problem, since we need to compute sums of elements differing by many orders of magnitude, that in the end mostly cancel each other out. We have signalled this problem earlier (see the discussion of (3.5) above), where it could be overcome by switching to the Fourier domain. This does not appear to be an option here. Possible solutions include switching to an even higher precision software floating-point library or not using the final analytical solution, but trying to solve the simpler intermediate equations with the help of numerical integration.

Even with the $h_n(k)$ currently limited to $n = 1500$, we can attempt to calculate the final rollback length distribution for the length expressed in the number of events to roll back:

$$l(k) = \sum_{n=1}^{\infty} \left[h_n(k)(1 - P_{\text{rb}})^{n-1} P_{\text{rb}} \right] \tag{3.12}$$

This is equivalent to (3.9).

3.5.3 Comparison of the two approaches

In both approaches, we have presented methods to calculate the probability distribution function h_n of the rollback length after n steps, as well as the final probability distribution function l. How similar are the results?

Figure 3.10 presents a comparison of $h_n(d)$ and $h_n(k)$ for two values of n: 50 and 500. We should point out that the two functions operate on different domains: $h_n(d)$ on a continuous domain of the simulation time length and $h_n(k)$ on a discrete domain of

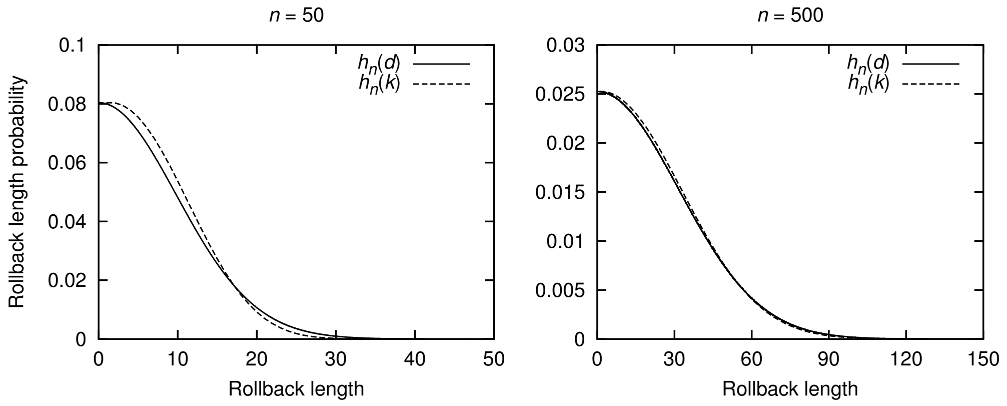


Figure 3.10: Comparison of the probability distribution functions $h_n(d)$ and $h_n(k)$ for different values of n. x axis is the rollback length expressed in the simulation time d or in the number of events k.

the number of events. While direct comparisons should be avoided, some general trends can still be observed. We can see that the two curves are quite similar, and that this similarity grows with the increasing value of n. There is an important difference however. Because the simulation time increments are taken from an exponential distribution, the domain of $h_n(d)$ extends to infinity. $h_n(k)$ however is limited from the top by the number of steps performed: n. This makes $h_n(k)$ go to zero a lot faster than $h_n(d)$, as can clearly be seen for the case of $n = 50$ (Fig. 3.10 (*left*)) and somewhat less clearly for $n = 500$ (Fig. 3.10 (*right*)).

Given the similarity of $h_n(d)$ and $h_n(k)$, it should not be surprising that $l(d)$ and $l(k)$ are also quite similar, as we can see in Fig. 3.11, where the approximations have been shown for the sums of the first 1400 elements (see (3.9) and (3.12)).

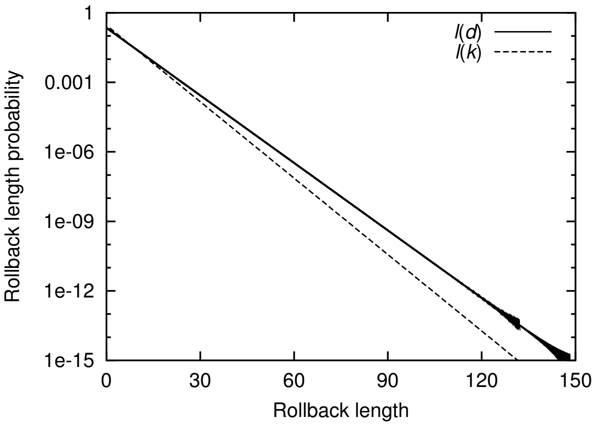


Figure 3.11: Comparison of the final rollback length distributions $l(d)$ and $l(k)$ for $P_{\text{rb}} = 0.05$ and n limited to 1400. x axis is the rollback length expressed in the simulation time d or in the number of events k.

Comparisons aside, the important thing to notice at this point is that both curves clearly decay exponentially, like in Fig. 3.1 (*right*) (the shape in Fig. 3.11 is different, because a linear scale on the x axis has been used). No power-law behaviour can be observed, so this result can be considered as an indication of the abnormality of such behaviour in Time Warp.

However, we should not draw the conclusions too far. Figure 3.1 presents the results from a real system with 12 processes, while the above models make a number of significant simplifying assumptions. They are capable of modelling the interaction between only two process. It is assumed that the inter-process communication is instantaneous and that its overhead is negligible. More esoteric features of the Time Warp protocol such as anti-messages (and, consequently, rollback cascades) or optimism throttling are not modelled at all. All of these are bound to affect the level of synchronisation between processes, and hence could influence the distribution of rollback length.

Summarising, we shall compare our approach to what can be found in the literature discussed in Sec. 3.2. The techniques we used bear some similarity to those of Tay et al. [179], since in both cases a probabilistic approach is used. Regarding the properties of the simulation, self-sustaining systems such as the Ising spin model have been analysed by e.g. Nicol [131] or Felderman and Kleinrock [58]. However, all of these publications focus on the performance bounds, while we are more interested in the rollback behaviour. Rollback behaviour has been analysed by Lubachevsky et al. [120], but their approach assumes a network simulation problem, and focuses on identifying patterns in the rollback behaviour, not on its statistical analysis.

3.6 Simulator of a parallel kernel

Extending the analytical model presented above is non-trivial. This has prompted us to work on a different, possibly easier strategy: a discrete event simulation of a Time Warp simulation kernel itself.[†] To some extent, this can be seen as addressing the problem from the other end. While the analytical model focuses on simulating the run-time behaviour of the Ising spin model, the simulator we are about to build will simulate the Time Warp dynamics.

The kernel simulator has been implemented in Java, using JavaSim package (a Java version of a better known C++Sim package [115]) as a sequential discrete event simulator toolkit. JavaSim provides the process-oriented view of the simulation, and simulation entities are implemented as separate Java execution threads.

Figure 3.12 presents the main components of the implemented kernel simulator and the interactions between them. Every logical process is represented as a separate simulation thread (LP_1 through LP_n). The implementation of the logical processes is quite complete. Input and output queues are fully implemented, state saving is partially supported (only for the random number generator state). On rollbacks, anti-messages are

[†]To avoid confusion, we will be using the term *kernel simulator* when talking about the simulation of a parallel kernel, and *simulation kernel* when talking about the parallel simulation of another model.

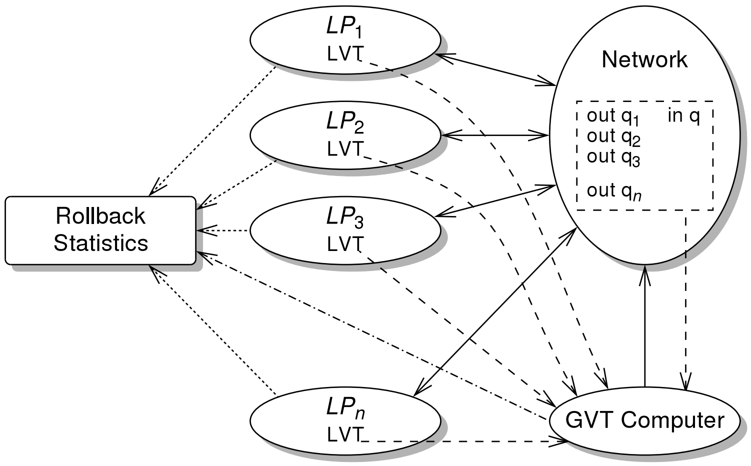


Figure 3.12: Main components of the kernel simulator.

sent. Optimism throttling via a virtual time window is supported. Garbage collection is performed when a new GVT value is received.

The processes communicate with each other via *Network*, which is implemented as a separate simulation thread. Such an architecture makes it easy to implement the message travel delay independently of the sending/receiving delays of the logical processes. When a process wants to schedule an event on another logical process, it appends the event to the Network's input queue inq, notifies the Network and suspends for the sending delay time. Network suspends for the travelling delay time and puts the message into an appropriate output queue $outq_x$. At its convenience, the destination process retrieves the message from the queue and suspends for the receiving delay time (our model, just like our Time Warp kernel, is *non-preemptive*, that is, the arrival of a new message does not interrupt the processing of the current event).

Another simulation thread is *GVT Computer*, which is responsible for periodic recalculation of the GVT estimate and its dissemination to all the logical processes. In the actual parallel simulation kernel, this is performed by the first logical process (the *root* process). Based on the data obtained from the other logical processes, the root process calculates the minimum of the local times on all the logical processes and of the messages currently in transit. The latter makes the whole calculation rather complicated, particularly since the data is not obtained on all the logical processes simultaneously and is somewhat obsolete by the time it is used (see also Sec. 5.5.2). However, in the simulator the current data from all the logical process threads as well as the network thread is easily accessible. Therefore, instead of implementing the GVT calculating code into the root process, we have created a separate thread that directly accesses variables in other threads. The newly calculated GVT is disseminated across the logical processes just like in the actual simulation kernel, by sending messages via Network.

The last important component is *Rollback Statistics*. In fact, it is not a separate thread, only a globally accessible object. When a process needs to roll back, it records this fact in

Rollback Statistics, together with the reason for the rollback, i.e. an externally scheduled positive or negative event that caused the rollback. This allows us to track the rollbacks through the whole simulated system, if deemed necessary.

We have thus built a kernel simulator that closely mimics the architecture and run-time behaviour of a real parallel simulation kernel, at the level of single events. With such a level of detail, a large number of parameters need to be specified. We have obtained most of them through the extraction from a running parallel simulation kernel, performing profiling analysis on a modified kernel if necessary, and by running some dedicated benchmarks. The following values have been extracted:

- simulation kernel: time to schedule a new event, time to advance to the next event, GVT algorithm frequency, time to perform the garbage collection, VTW size, number of logical processes,
- interconnecting network: time to send a message, travel time of a message, time to poll for a message, time to receive a message,
- target simulation: event scheduling pattern (initial events, function advancing the simulation time, function deciding whether to schedule events locally or on other logical processes), time to process an event.

3.6.1 Validation

Figure 3.13 presents a sample of the experiments performed to validate the kernel simulator. Because we were primarily interested in simulating the rollback behaviour, we have shown the number of rollbacks and rolled back events, and a rollback length histogram. In addition to that, the results for the simulation run time are also available.

Some of the results from the simulation do come quite close to the real runs, in particular in the number of rollbacks that have taken place (Fig. 3.13 (*top-left*)). In other cases, at least the shapes of the curves are quite similar. The largest differences can be observed in the rollback length histogram (Fig. 3.13 (*bottom-left*)). The differences visible for the smallest rollback lengths (1–2) should actually be ignored. The kernel simulator uses a simpler method of calculating the rollback length that can cause the results to be off by one event. This is insignificant for larger rollbacks, that we are focusing on. The source of the discrepancy that can be observed in that region cannot be explained given the facts presented so far (we will explain it in Sec. 3.6.3). The output from the kernel simulator generally indicates a better performance (shorter rollbacks, shorter run time) than what is observed in reality.

A number of factors contributing to this discrepancy can be identified, but we should state that it is in general very difficult to simulate a parallel system so closely that the simulated run-time behaviour will match that of the real system. The model under consideration has a simple representation of the computing and communication resources, with fixed costs assigned to various operations. In reality, with programs running under multi-tasking operating systems with virtual memory, the performance fluctuates. For example, the model does not consider the influence of the CPU cache. The communication

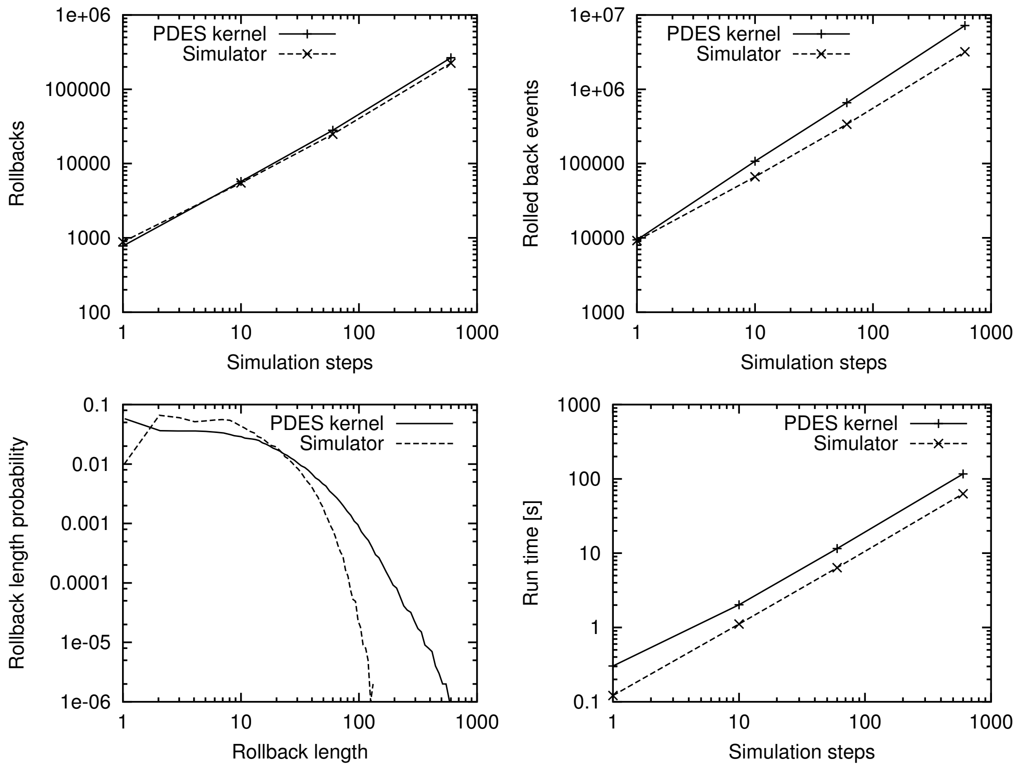


Figure 3.13: Validation of the kernel simulator against a real PDES simulation kernel, for 8 processes and the Ising spin model temperature of 2.5. Note the logarithmic scale on both axes.

network is an even larger source of fluctuations. We do not consider complex run-time behaviour such as congestions in the communication channels.

3.6.2 Simulating rollbacks

The validation performed above shows that our kernel simulator does not accurately predict the run-time behaviour of an actual parallel simulation kernel. To improve its operation, the simulator would need to be extended to handle even more intricacies of the real-world system.

However, building a performance-accurate simulator is not our main objective. Instead, we want to simulate the rollback behaviour. This can easily be achieved if, instead of trying to accurately model an actual simulation kernel, we use the simplified assumptions of the analytical model from Sec. 3.5.

Figure 3.14 presents the results of the verification against the analytical model. The match is nearly perfect. To achieve this result, we have disabled the features of the kernel simulator not supported by the analytical model, such as the optimism throttling, anti-messages and state saving. Most overheads have been removed, including the overheads

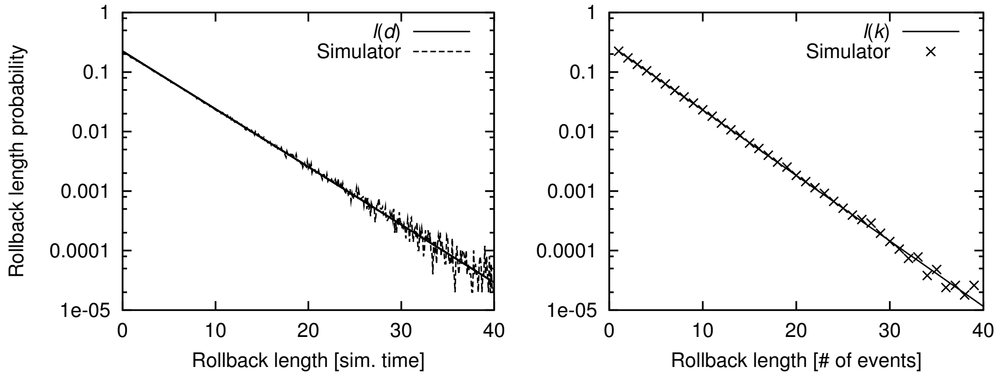


Figure 3.14: Verification of the kernel simulator against the analytical model, for two processes and the rollback probability $P_{\text{rb}} = 0.05$.

of inter-process communication and rollback handling. Only a fixed cost of processing locally scheduled events has been kept. The simulation has been limited to two logical processes.

The results thus prove that the kernel simulator accurately implements the analytical model under its simplifying assumptions. Given the difficulties of expanding the model further, we can instead expand the simulator to try to explain the complex behaviour observed in the real system.

Once the kernel simulator has been verified, we have re-enabled its extra features such as the optimism throttling and the generation of anti-messages, and we have increased the number of simulated processes to 8. Also, we have added a small, linear cost to the rollback handling, setting it arbitrarily to 10% of the cost of moving forward.

The results of the experiments can be found in Fig. 3.15, that shows the rollback length distributions. Several VTW sizes have been tried. We can see that the distributions

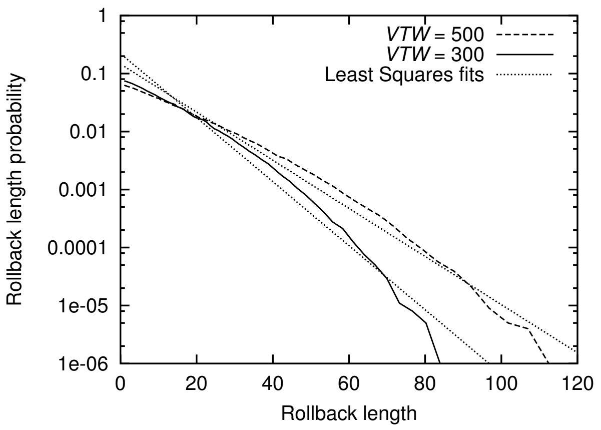


Figure 3.15: Rollback length distributions from the kernel simulator.

are decaying slightly faster than exponentially (fitting to straight lines is not completely successful anymore), probably because of the introduction of optimism throttling. No power-law behaviour can be observed, This is another indication that such behaviour in Time Warp is abnormal, and a much stronger one, given that the kernel simulator implements many more features of a parallel simulation kernel than does the analytical model.

3.6.3 Rollback cost

With all these indications, we have decided to look even more closely at the behaviour of the APSIS kernel running Ising spin simulation in sub-critical temperatures, where the power-law behaviour occurs.

As it eventually turned out, our assumption of a low, at most linear cost of rollback operations was incorrect. As implemented in APSIS, the cost of a rollback was quadratic in the rollback length. The reason was that locally scheduled events were annihilated one by one, individually. While efforts had been taken to optimise the annihilation process, the simple, double linked list implementation of the input event queue could potentially require a significant part of that list to be swept before an event was found. As a result, a large rollback, in the range of thousands, could take several seconds to perform! Instead of decreasing the overall stress in the system, such a rollback would actually increase it.

Figure 3.16 presents the rollback length distribution from the kernel simulator with a

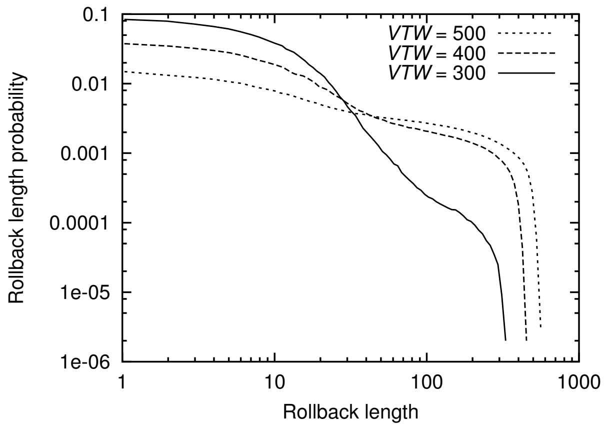


Figure 3.16: Rollback length distributions from the kernel simulator with a quadratic cost of rollback handling.

modified, quadratic cost of a rollback. Some striking similarities can be observed when comparing this figure to Fig. 3.1 (*left*) on p. 40, especially in case of strongly throttled runs of the kernel simulator, i.e. for *VTW* = 300.

Following up, we have attempted to introduce the square rollback cost in the non-simplified kernel simulator, as described in Sec. 3.6. Experiments performed along the lines of the validation tests from Sec. 3.6.1 indicate that such a simulator better matches the parallel simulation kernel. We do not present these results here for two reasons.

Firstly, we do not have enough data to provide a good estimate of the rollback cost, and we cannot obtain such data anymore, since the machine where the original data had been collected was decommisioned in the mean time. Secondly, the introduction of square rollback cost with a somewhat arbitrarily chosen cost function significantly destabilised the simulator, forcing us to decrease the VTW size and thus making it difficult to perform any meaningful comparison with the simulation kernel.

Needless to say, once we discovered the excessive cost of the rollback algorithm in APSIS, we rushed to fix the problem. Figure 3.17 presents a comparison of the run-time

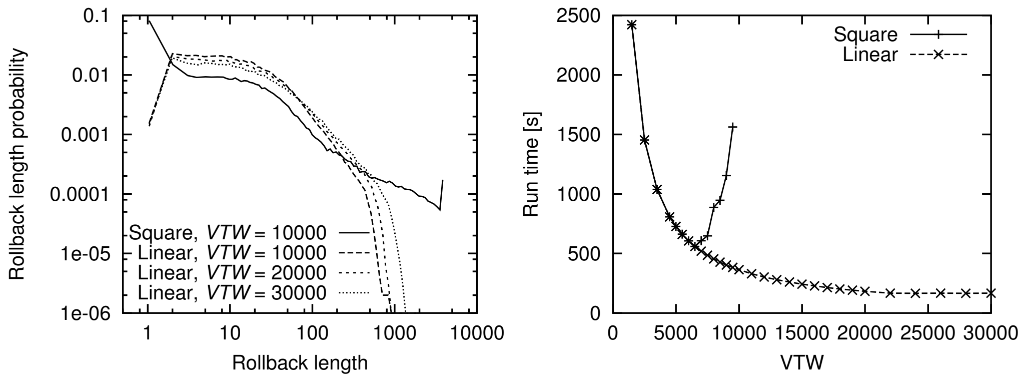


Figure 3.17: Comparison of the run-time behaviour of Time Warp kernels with square and linear rollback costs: rollback length histogram (*left*), elapsed run time (*right*).

behaviour of the "old" kernel (with a square rollback cost) and the "new" one (with a linear cost). As expected, with linear cost of a rollback no power-law behaviour can be observed (see Fig. 3.17 (*left*)). Further, the kernel has become a lot more stable, so, as shown in Fig. 3.17 (*right*), we can use much larger VTW sizes, significantly decreasing the simulation run time.

But why does a simulation kernel with a square rollback cost result in a rollback length distribution similar to the power law? Figure 3.18 presents a possible explanation. Suppose that there is a correlation between subsequent rollbacks, i.e. that it is possible with a certain probability to predict the length of the next rollback knowing the length of the last one. This is quite realistic, given that the cost of a rollback is a function of the rollback length, although we must stress the stochastic nature of this random walk. No events are processed while rolling back, so once a process resumes forward processing, it will lag behind the other processes by a predictable amount of simulation time, so the length of the next rollback in the system can be estimated.

If the rollback cost is linear, as in Fig. 3.18 (*left*), then the majority of rollbacks will be short. Namely, because the computational cost of a rollback is lower than that of moving forward, the next rollback is likely to be shorter than its predecessor. This is shown in Fig. 3.18 (*top-left*). In grey, we have shown the *rollback amplification region*: if the correlation curve is in this region, then the next rollback is likely to be longer than the current one (the region is simply the area above the $y = x$ line). For the linear rollback cost, the

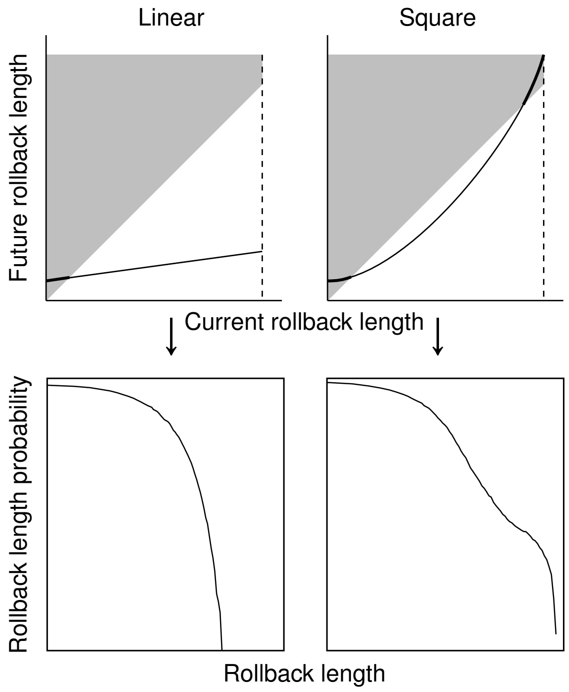


Figure 3.18: Influence of the correlation between the current and future rollback length (*top*) on rollback length histogram from the kernel simulator on a log-log scale (*bottom*), with linear and square rollback costs. The region of rollback amplification is marked with grey, the dashed lines denote the VTW border.

correlation curve is in that region for small rollback lengths. This is the case because we made a reasonable assumption that a rollback cost is a sum of a linearly increasing part and a non-zero fixed part, so for small rollback lengths, when the fixed part dominates, performing a rollback may actually increase the stress in the system. Although small rollbacks can result in an amplification, the system remains stable, because larger rollbacks result in a reduction. Hence, the majority of rollbacks take place in the amplification region or in its neighbourhood, as can be observed in Fig. 3.18 (*bottom-left*), showing the rollback length histogram from the kernel simulator on a log-log scale.

The situation is different if the rollback cost is quadratic, as shown in Fig. 3.18 (*right*). Like before, the correlation curve is in the amplification region for small rollback lengths, and this is not a problem. However, for sufficiently large rollbacks, the correlation curve eventually becomes steep enough to cross the amplification border again, never to come back. This is guaranteed to result in an instability: the processes will execute ever longer rollbacks without doing any useful work. The virtual time window (the vertical dashed lines in Fig. 3.18 (*top*)) can come to rescue, limiting the maximum rollback length. Still, if the VTW size is large enough to allow for some of the large rollbacks to occur, as in Fig. 3.18 (*top-right*), the result will be a characteristic power-law like rollback histogram as in Fig. 3.18 (*bottom-right*). It is in fact not a power-law curve, but more of a bi-modal one, with modes close to zero and to the VTW border. Increasing the VTW size even further will result in a loss of stability in the system.

3.7 Conclusion

In this chapter, we have described our various efforts in the area of modelling and simulation of a Time Warp parallel simulation kernel.

Our interest was triggered by a very particular behaviour of the Ising spin simulation. In the sub-critical model temperature range, the rollback behaviour appears to exhibit the properties of a self-organised critical system, namely, the rollback length histogram has a power-law like shape, instead of the expected exponentially decaying one. We have also observed that the power-law behaviour is dependent on the VTW size and the properties of the random number generator used. We decided to model the rollback behaviour of a Time Warp kernel to get an insight into what exact circumstances are necessary to get either shape.

We have shown the mechanism of how changing the properties of the random number generator can affect the simulation's rollback behaviour in the presence of a VTW throttling, confirming the previously obtained experimental results.

We have built an analytical model of the behaviour of a Time Warp kernel running the Ising spin simulation. Our aim was to derive the rollback length distribution function, so that we could compare it to the histograms obtained from the experiments. We built two models to achieve this goal: one where a rollback length is expressed in the simulation time, and another one where it is expressed in the number of events. In either case, we have obtained an exponentially decaying rollback length distribution function. However, the models are strongly simplified, being able to model just two processes, assuming instantaneous inter-process communication, and neglecting Time Warp protocol properties such as anti-messages.

Given the difficulties with extending the analytical models further, we subsequently decided to build a discrete event simulation of a PDES kernel. The kernel simulator features a high level of detail, striving to model a parallel simulation kernel as closely as possible. Nevertheless, its validation against a real-world system has shown that the simulator is not quantitatively accurate, probably due to an overly simplified representation of computational and communicational costs. Still, after the assumptions from our analytical models have been put into the simulator, the verification performed was quite successful. In the kernel simulator, we could easily add the features missing from the model, such as a support for more than two processes or the VTW optimism throttling. We have also introduced a small, linear rollback cost. Still, the rollback length histograms obtained were uniformly exponentially decaying, no power-law shape could be observed.

This has prompted us to look even more closely into the experiments resulting in the power-law behaviour. As it finally turned out, the overly simple rollback algorithm of the APSIS kernel resulted in a quadratic cost of a rollback. Once this was fixed, the power-law like shape disappeared, and additionally the kernel has become a lot more stable. Moreover, when we modified the kernel simulator to have a quadratic rollback cost, the power-law like shape could be observed again.

Based on these results, the previously made speculation that the Time Warp dynamics is a self-organised critical system can no longer be supported. Relaxations in a SOC system must be virtually instantaneous, whereas in a Time Warp kernel we can only observe a power-law like rollback length distribution if rollbacks (that act as relaxations) have a super-linear complexity.

The work described in this chapter has given us a good insight into the inner working of our parallel simulation kernel. With this knowledge, we are ready to perform some real measurements on the Grid, which will be the focus of the next chapter.

4

Event Cancellation Optimisations*

4.1 Introduction

Ever since the Time Warp mechanism was proposed by Jefferson in [94], the rollback mechanism has been attracting much attention. The influence of rollbacks on the performance of a parallel simulation has been extensively studied analytically (see the plethora of publications cited in Sec. 3.2) and also experimentally, see e.g. Fujimoto [73], Sokol and Stucky [171], Turner and Xu [184], or Avril and Tropper [9].

These studies have led to a better understanding of the rollback behaviour, and have also resulted in the development of new cancellation strategies.

In [156], Reynolds introduced a distinction between *aggressiveness* and *risk*. Aggressiveness allows events to be processed possibly out of order. Risk goes further, allowing the messages generated during aggressive event processing to be sent out immediately. Rollbacks are a consequence of aggressiveness, while risk necessitates the use of anti-messages, and can thus result in rollback cascades.

An aggressive approach without risk is certainly possible. One such implementation, the Riskfree TWOS (see Sec. 2.2 for a description of TWOS) has been described by Bellenot [23]. Positive events scheduled on other logical processes are only sent out when they are provably correct, that is, when GVT reaches their creation time. The performance of the scheme is found to be rather poor; good lookahead is needed to improve it. A different approach, called Local Time Warp, is described by Rajaei et al. [151]. It does not eliminate risk, but it significantly limits it by grouping LPs into logical clusters and only employing the Time Warp protocol within each cluster. Between the clusters, a

*This chapter is based on: K. A. Iskra, G. D. van Albada, and P. M. A. Sloot. Time Warp Cancellation Optimisations on High Latency Networks. In *Proceedings of the 7th IEEE International Symposium on Distributed Simulation and Real-Time Applications*, pp. 128–135, Delft, The Netherlands, Oct. 2003.

conservative synchronisation protocol is used, resulting in a hybrid approach.

Time Warp employs both aggressiveness (due to its optimistic nature) and risk (it uses anti-messages). In the standard cancellation scheme, from now on referred to as *aggressive cancellation*, for every erroneous externally scheduled event a separate anti-message must be sent out (see Sec. 4.2.1). This means that if a large rollback is taking place, many messages need to be sent in a very short time, which can be a burden on the communication layer. A number of improvements have been proposed to reduce the number of anti-messages.

By far the best known optimisation of this sort is *lazy cancellation*, proposed by Gafni in [77], that delays the sending of anti-messages (we will discuss it in detail in Sec. 4.2.2). An analytical comparison between the aggressive and lazy schemes was performed by Lin and Lazowska [111], while an early experimental data can be found in the work of Reiher et al. [154]. Even though no clear winner can be identified, lazy cancellation is found to have somewhat more potential. It often requires fewer messages for its operation, and it is capable of effectively shortening the critical path by processing the events from the path out of order.

Balsamo and Manconi [16] introduced what they call *lazy sending*, an optimisation making it possible to send just one anti-message per destination LP per rollback. From the description it looks very similar to the optimisation that we describe in Sec. 4.2.3. We call this optimisation *bulk anti-messages*, as in our opinion it much better describes the underlying mechanism. In another recent publication, Kalantery [98] discusses *group cancellation*, that also looks very similar to the optimisation discussed here. This optimisation requires FIFO communication channels for its operation.

A radical solution, getting rid of anti-messages altogether without eliminating the potential advantages of risk, has been suggested by Damani et al. [52]. When a process receives a straggler message and needs to roll back, it notifies all the other processes by sending a single, broadcast message. This is the only message needed to synchronise the whole system, no secondary messages from other logical processes are needed, so no rollback cascades take place. This is achieved using *transitive dependency tracking*, a scheme where simulation vectors similar to the vector clocks of Lamport [105] are used. Each logical process has its own simulation vector, where it keeps the up to date information about itself, and for the other processes the data corresponding to the latest states of these processes that the process depends on. The data stored for each process is an *incarnation number* (the number of times a process has rolled back), and the simulation time. The vectors make it possible to track the dependencies between events. Every event holds a copy of its creator's simulation vector, current at the time when the event was created. Thus, for external events, the destination process can update its vector by taking the maximum of its own vector and that of the currently processed message, and as a result it knows exactly on what state of each process it depends. The broadcast rollback announcement is simply the current incarnation number of the rolling back process and the straggler's timestamp. All processes need to drop the events that, in the sender's part of the simulation vector, have the same incarnation number and a timestamp higher than that of the rollback announcement message. If any dropped events have already been processed, a process needs to roll back, but only locally (no more broadcast announcements are needed). While elegant, such a scheme will unfortunately prove rather expensive if the number of LPs is large.

In this chapter, we are going to take our PDES kernel APSIS, implement some of the aforementioned cancellation optimisations, and perform a number of experiments on a single cluster and on a wide area distributed, multi-cluster Grid platform. By comparing the results obtained from the two platforms, we are hoping to be able to identify the primary sources of the performance degradation expected to be observed on the Grid.

4.2 Cancellation strategies

We will now expand on some of the cancellation strategies mentioned above, presenting them in a greater detail and discussing their implications.

4.2.1 Aggressive cancellation

Figure 4.1 presents the initial case of a scenario that we are going to consider for all

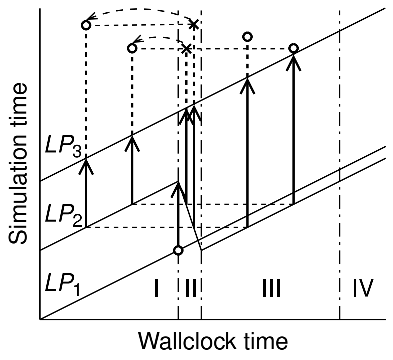


Figure 4.1: Initial scenario: aggressive cancellation, ordinary anti-messages.

the strategies. There are three logical processes, their progress in the simulation time is shown vertically. Ordinary (wall clock) time, shown horizontally, can be divided into four phases. In *phase I*, process LP_2 schedules two events on process LP_3. As in previous figures of this sort, the arrows indicate the receiver of a message, whereas the dotted lines with a small circle indicate the schedule time of an event, far into the future in this case. In order not to complicate the figure, the communication is assumed to be instantaneous, resulting in fully vertical arrows. After the two messages are scheduled, process LP_1 schedules an event on process LP_2. Because the schedule time of this event is in the past of LP_2, that process must roll back (*phase II*). While rolling back, LP_2 sends two anti-messages to LP_3, to annihilate the previously scheduled events (to simplify the figure, the order of anti-messages (marked with ×) is reversed). The curved dashed arrows at the top indicate the positive events annihilated by the anti-messages. In *phase III*, process LP_2 makes progress again, performing a re-run of the rolled back events. Again, it schedules two events on LP_3, although the first event is scheduled for a different time than before the rollback. In *phase IV*, all three processes make ordinary forward progress again, just like in *phase I*. We have chosen the times in Fig. 4.1 such that LP_3 never needs to roll back, but in reality the probability of multiple rollbacks occurring in such a scenario would be quite significant.

4.2.2 Lazy cancellation

We can observe in the scenario presented above, that the second positive event from LP_2 to LP_3 need not be annihilated, since eventually an identical event is re-scheduled in *phase III*. Lazy cancellation attempts to optimise the simulation by preventing spurious anti-messages from being sent. As shown in Fig. 4.2, with lazy cancellation no anti-

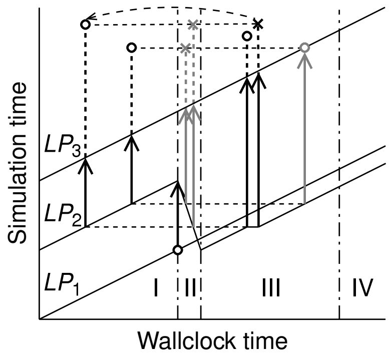


Figure 4.2: Rollback with lazy cancellation. Grey messages are not sent.

messages are sent during the rollback (*phase II*). They are prepared, but their eventual sending is deferred to the forward re-run (*phase III*). We use a shade of grey to indicate the messages that are not sent. During the re-run (*phase III*), for every external event scheduled, the queue of the deferred anti-messages is scanned for a matching one. If no such anti-message is found, which is the case for the first event scheduled in Fig. 4.2, the positive event is simply sent out. The unmatched lazy anti-message is sent soon afterwards, namely when the simulation time is about to increase, since from that point on no matching positive event could possibly be scheduled. On the other hand, if a matching lazy anti-message is found, as is the case for the second event in Fig. 4.2, there is no need to send either the anti-message or the (duplicate) positive event, which saves some communication and up to two rollbacks at the destination.

The success of lazy cancellation is ultimately dependent on how the arrival of a straggler affects the externally scheduled events. Lin and Lazowska [111] call this property the *sensitivity of output message*. If the majority of events are re-scheduled unchanged, the optimisation will be successful. Otherwise, it will only make things worse, as delaying the cancellation results in further propagation of erroneous computations. Lazy cancellation also introduces an extra overhead, because newly scheduled events must be compared to lazy anti-messages, and also because lazy anti-messages increase the memory usage of the simulation processes. Fujimoto [72] claims that lazy cancellation essentially exploits the lookahead property of a simulation model. As pointed out by Chen and Szymanski [43], it is in fact not lookahead, but *lookback* that is exploited. Lookback is an ability to process straggler messages without causing rollbacks. For example, events that do not change the simulation state (*query* events) can always be processed this way, whereas other sorts of events might only be safe within a *lookback window*. While lookback must be explicitly identified and supported by the simulation, lazy cancellation exploits it implicitly, dynamically, at run time. *Lazy re-evaluation* [190] is another optimisation based on the same principle. It compares the simulation state before and after processing the straggler, and if they are the same and there are no new events on the event list, the logical process can *jump forward*, skipping the re-run phase.

Conceptually, lazy cancellation is relatively straightforward. As implemented in APSIS (see Fig. 4.3), during a rollback the items from the output queue that would normally be

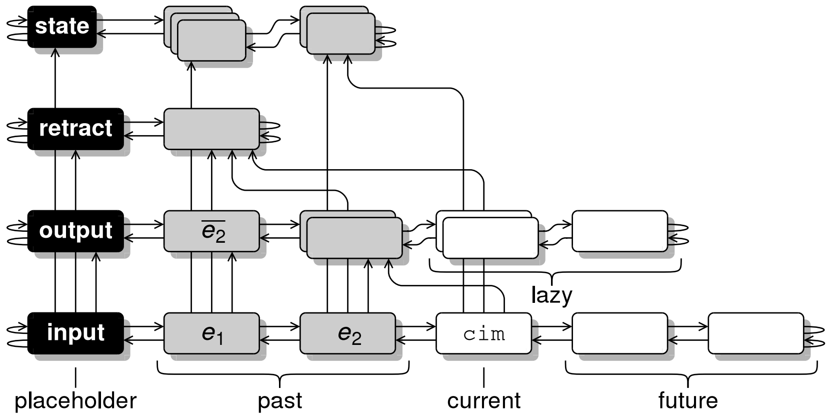


Figure 4.3: Influence of lazy cancellation on the data structures in APSIS (compare with Fig. 2.2 on p. 22).

sent out and released are kept there. As a result, the output queue begins resembling the input queue, having both items from the past and the future. The entries are sorted by their creation time, so the items of interest when looking for a match are conveniently always at the front of the future part of the output queue.

One issue that lazy cancellation complicates is the ordering of events. APSIS assumes the FIFO property of the communication channels (see Sec. 2.3.1), so events from one source should always arrive in the creation time order, helping to properly sort events with an identical timestamp (see Sec. 2.7.2). On the other hand, with lazy cancellation, events sent during the re-run phase after a rollback might have a lower creation time than the ones sent before the rollback that, due to the use of lazy anti-messages, have not been annihilated. To guarantee a deterministic event ordering, it might be necessary to force the annihilation of some of the "old" events before a new one can be scheduled.

Lazy cancellation can also help solve the zero lookahead loop problem (see Sec. 2.7.2), as shown in Fig. 4.4. Because the anti-message $\overline{e_1}$ does not get sent, event e_1 is not anni-

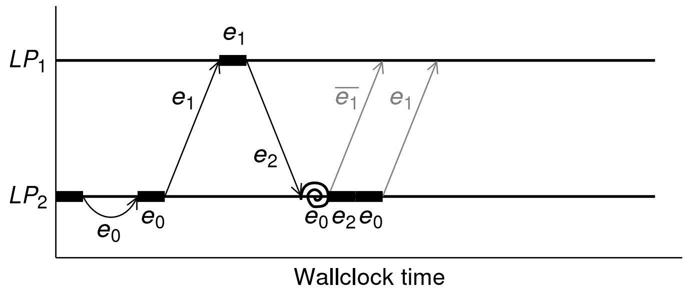


Figure 4.4: Influence of lazy cancellation on a simulation with zero lookahead loop (compare with Fig. 2.8 on p. 32).

hilated, and consequently neither is e_2, so the simulation never enters the endless loop. Applying lazy cancellation can thus result in a *semantic* improvement in certain corner cases, so, as reported by Lin and Lazowska [111], some have argued that lazy cancellation is more of a fix than an optimisation and that aggressive cancellation, due to its dangerous properties, should be abandoned.

4.2.3 Bulk anti-messages

As we have discussed earlier, with aggressive cancellation, when a process rolls back, it must annihilate all the external events it has generated in the simulation time period that is being rolled back (see Fig. 4.1 (*phase II*)). However, instead of annihilating them individually, it is possible to do so *in bulk*, as shown in Fig. 4.5.

The optimisation works as follows: when rolling back, only one, *bulk* anti-message per destination process is sent (see Fig. 4.5 (*phase II*), the thick black arrow). This specially crafted anti-message specifies the creation time of the oldest event to be annihilated at the destination. The destination process annihilates all the events earlier received from the source process with the creation time equal to or greater than the time specified in the bulk anti-message.

Two non-standard conditions must be met in order for this optimisation to work:

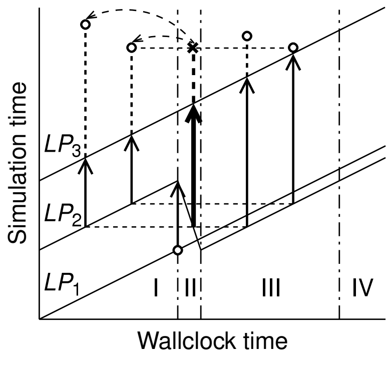


Figure 4.5: Rollback with bulk anti-messages.

- Every event sent must contain not only the schedule time, but also the creation time. The latter is not normally needed for the correct operation of the Time Warp protocol, although it can have some use even without the bulk anti-messages, for example during debugging.
- The communication layer must guarantee the FIFO property of the communication channels. Otherwise, the bulk anti-message could "overtake" some of the earlier sent positive events, which would prevent them from being annihilated. Most probably, the intentional lack of the FIFO requirement in the original Time Warp protocol is the reason why bulk anti-messages are not standard.

In order not to affect other areas of the Time Warp protocol, in particular the GVT calculation, the schedule time of the bulk anti-message is set to the minimum of the schedule times of all the events to be annihilated.

Unlike lazy cancellation, the use of bulk anti-messages is not an alternative to aggressive cancellation, it is rather a low-level improvement to it. Even with bulk anti-messages, the cancellation scheme stays aggressive. It would in fact be possible to use bulk anti-messages with lazy cancellation, as shown in Fig. 4.6. *Phase II* (rollback) is the same as in

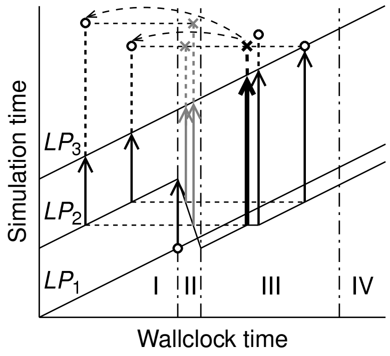


Figure 4.6: Rollback with lazy cancellation and bulk anti-messages used together.

lazy cancellation with ordinary anti-messages. However, during the re-run (*phase III*), if a lazy anti-message does not get matched and needs to be sent out, a bulk anti-message is sent instead, that annihilates all events sent from the given process, starting with the one corresponding to the not matched anti-message. In the case shown in Fig. 4.6, this means that both events from LP_2 to LP_3 will be annihilated, and consequently both need to be re-scheduled.

This example shows the greatest weakness of such a mixed approach: bulk anti-messages reduce the efficiency of lazy cancellation. When we implemented the scheme and compared the results to those of other cancellation strategies, it turned out to perform slightly, but consistently worse (by some 1–5%) than the ordinary lazy cancellation. We have thus abandoned it, and omitted the results from the figures presented later in this chapter.

4.3 Experiments

4.3.1 Single cluster

Figure 4.7 presents the results obtained using the cancellation strategies discussed in

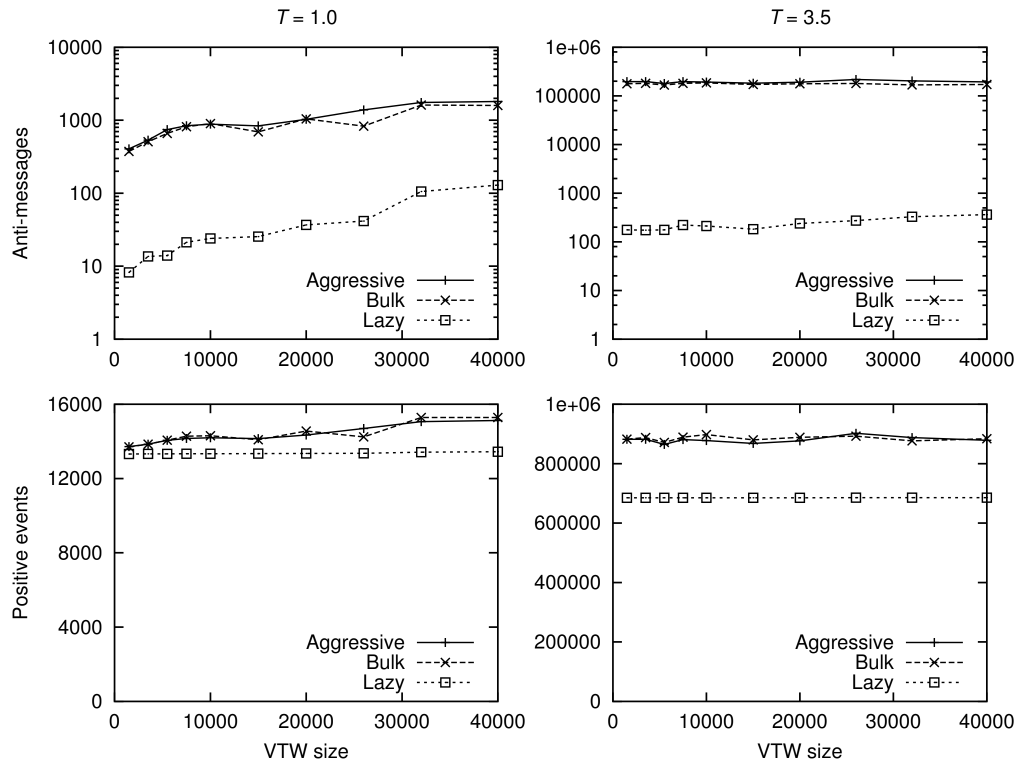


Figure 4.7: Influence of various cancellation strategies on the total amount of inter-process communication with the Ising spin simulation when running on a single cluster. Note the logarithmic scale on the *y* axis in the top plots.

Sec. 4.2, with 8 processes communicating using MPICH-GM, for a wide range of VTW sizes (for even larger sizes, the results slowly begin to deteriorate). We have used the Ising spin simulation (see Sec. 2.4.1) with two different model temperatures: 1.0 and 3.5 (there is a difference of two orders of magnitude in the communication volume between them).

The influence of these cancellation strategies on the average number of anti-messages received by one process is shown in Fig. 4.7 (*top*). Clearly, somewhat fewer messages need to be sent if bulk anti-messages are used (note that a logarithmic scale is used on the *y* axis, so the differences are in fact larger than it may seem), but it is lazy cancellation that is really impressive: an improvement by over one (model temperature 1.0) to three (model temperature 3.5) orders of magnitude can be observed! Our Ising spin simulation is thus very suitable for lazy cancellation. This is caused by the fact that in this model events received from other LPs only very slightly affect other events scheduled by the process, at least in the not too distant future.

The savings in the number of anti-messages with lazy cancellation are reflected in the average number of positive events received, shown in Fig. 4.7 (*bottom*) (note linear scale). With aggressive cancellation, the annihilated events need to be re-scheduled, increasing the total number of positive events. With lazy cancellation, only very few positive events are actually annihilated, and only they need to be re-scheduled. So the gain in the communication doubles. This is not the case with bulk anti-messages: they only save on the communication overhead when annihilating events, positive events are (re-)scheduled identically to aggressive cancellation.

Figure 4.8 presents rollback-related data for the temperature of 1.0. Figure 4.8 (*left*)

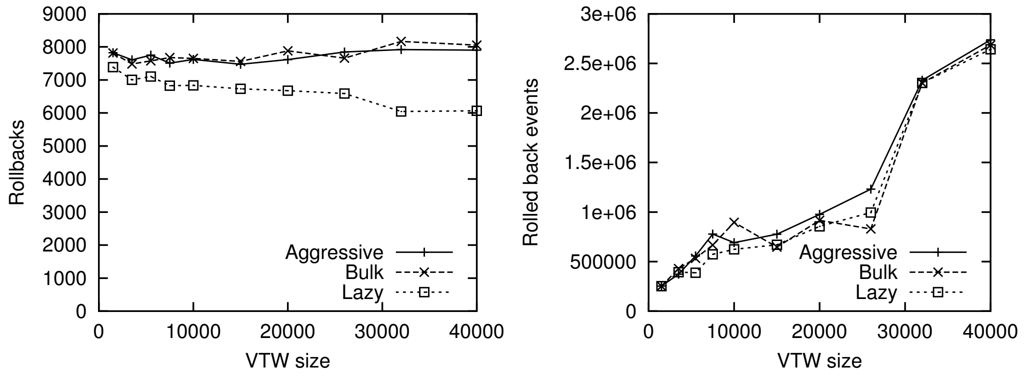


Figure 4.8: Influence of various cancellation strategies on the rollback behaviour with the Ising spin simulation when running on a single cluster, for $T = 1.0$: average rollback count (*left*), average number of rolled back events (*right*).

presents the average number of rollbacks that took place on a single logical process. This number is much larger than the number of anti-messages (see Fig. 4.7 (*top-left*)), since the majority of rollbacks are actually caused by positive events (stragglers). The number is approximately equal to the half of the number of positive events (see Fig. 4.7 (*bottom-left*)): because the external events in the Ising spin simulation are always scheduled for the current simulation time of the source process, the chance of arriving in the past at the destination process and causing a rollback there is close to 50%. While the rollback count is only slightly affected by the VTW size, the number of rolled back events (Fig. 4.8 (*right*)) is influenced fairly strongly. With a small virtual time window, the processes have less freedom, hence the maximum rollback size is limited, and so is consequently the total number of rolled back events. As the window size increases, the differences in the simulation time between the processes increase as well, so the average rollback size gets larger. With the largest window size the number of rolled back events reaches close to three million, but the efficiency is still close to 90%, since a process must on average commit almost 23 million events.

However, the in the end most important property, the run time, is disappointing, see

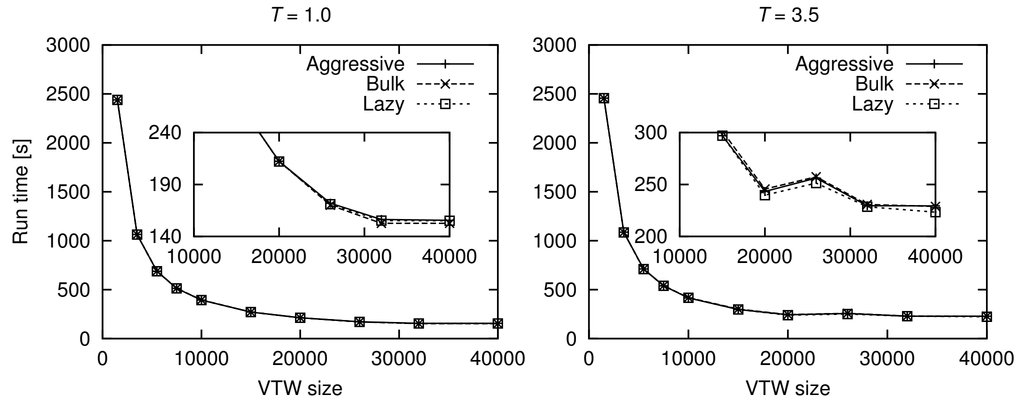


Figure 4.9: Influence of various cancellation strategies on the run time with the Ising spin simulation when running on a single cluster.

Fig. 4.9. In spite of the significant savings on communication, there is practically no improvement in the performance. Only with a 10× zoom some minimal, inconclusive differences can be observed: for the temperature of 1.0, runs using the bulk anti-messages perform marginally better than the rest, whereas for the temperature of 3.5 lazy cancellation leads. This is easy to explain. As we have noticed earlier, bulk anti-messages slightly reduce the communication overhead, so a small improvement over aggressive cancellation can be expected. On the other hand, lazy cancellation increases the processing overhead, and only becomes an advantage if the savings on the number of messages sent are large, as is the case with the temperature of 3.5. Still, the communication using MPICH-GM is so efficient, that these limited savings are simply insignificant for the total run time of the simulation.

4.3.2 Grid

The situation changes if a less efficient communication layer is used, such as MPICH-G2 (over MPICH-GM) distributed over a wide area Grid, as shown in Fig. 4.10. Again, we have used 8 processes, evenly distributed across two DAS-2 clusters (see Sec. 2.5), in Amsterdam and Leiden (the torus of the Ising spin model lattice (see Sec. 2.4.1) is divided along the shorter dimension, to minimise the inter-cluster communication). Even larger savings on the communication than before can be observed, especially in the decreased number of positive events sent: with lazy cancellation, there is a saving of over 20% with the temperature of 1.0, and by almost a factor of three with the temperature of 3.5. This time the savings in communication do translate into gains in run time, as shown in Fig. 4.11. For the model temperature of 1.0, a gain of 4% with lazy cancellation can be observed. For the model temperature of 3.5, the differences are a lot more significant: with bulk anti-messages the kernel manages to make a more than 4% improvement over aggressive cancellation, although for very large VTW values the standard cancellation does catch up. But the big winner here is lazy cancellation, achieving almost a factor

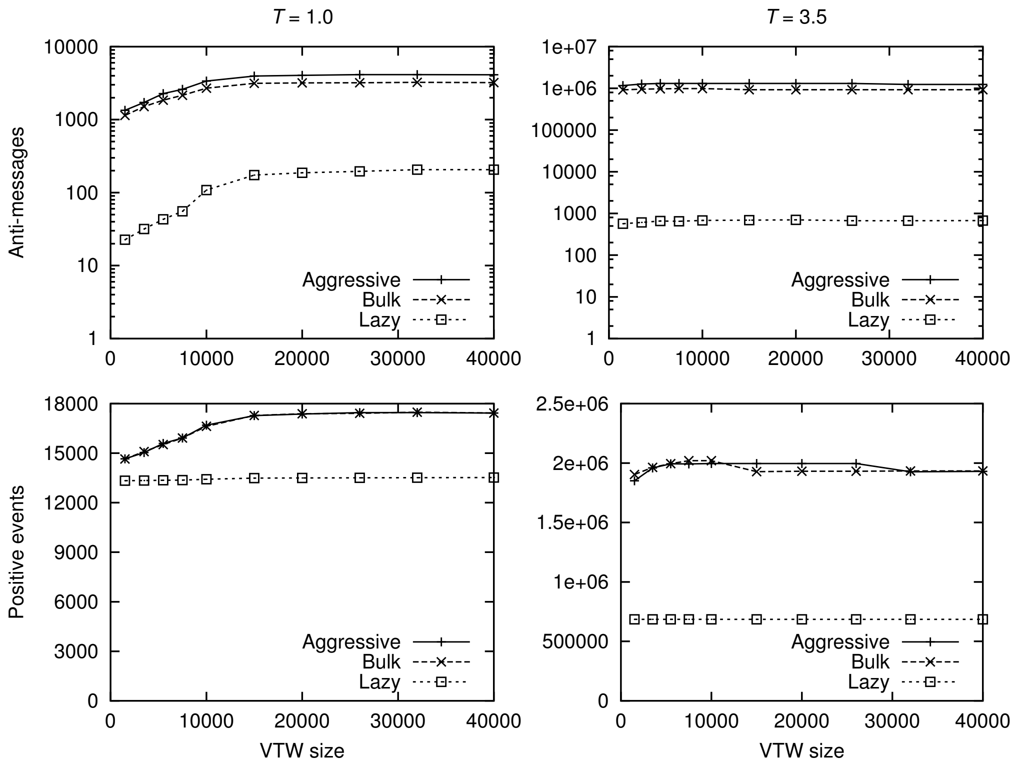


Figure 4.10: Influence of various cancellation strategies on the total amount of inter-process communication with the Ising spin simulation when running on the Grid. Note the logarithmic scale on the y axis in the top plots.

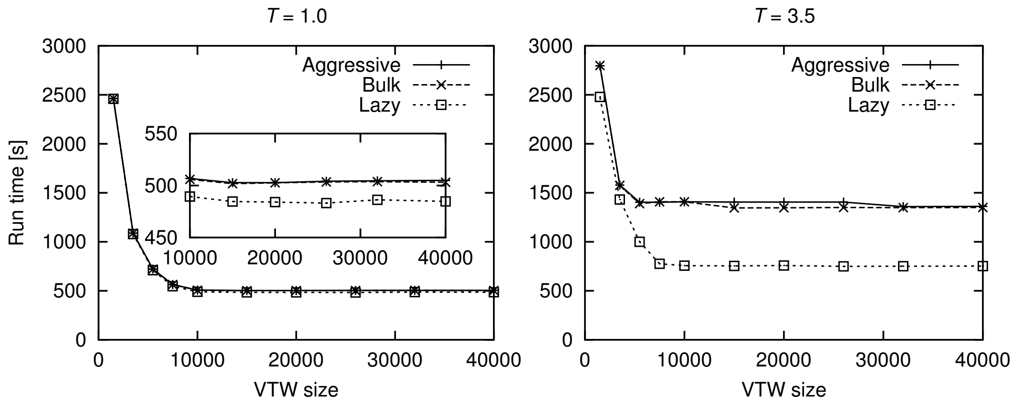


Figure 4.11: Influence of various cancellation strategies on the run time with the Ising spin simulation when running on the Grid.

of two decrease in the run time.

From the above results it appears that a slower communication layer requires more advanced cancellation strategies. We can investigate this further using the results presented in Fig. 4.12, where the run times of the Ising spin simulations using different com-

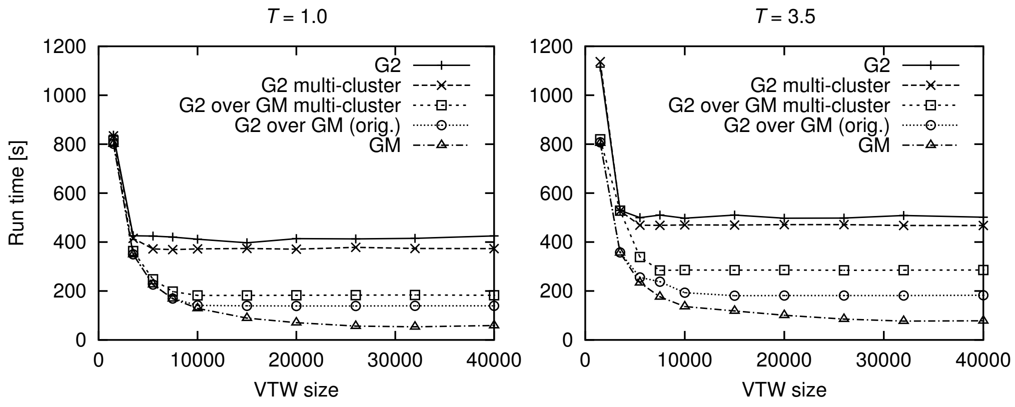


Figure 4.12: Influence of using various communication layers on the run time with the Ising spin simulation.

munication layers are compared. To make the experiment more challenging, we have decomposed the same problem on 24 processes this time, to intensify the inter-process communication. Lazy cancellation is used uniformly in all cases, as it was consistently giving the best or close to the best results. Just to give some idea about the intensity of the communication, suffice it to say that for the temperature of 3.5, each process received on average 380 thousand externally-scheduled positive events. Divided by the run time, this amounts to some 750 to 5000 messages/s for each of the 24 processors. We have used all the available flavours of MPICH (see Sec. 2.5) and performed the experiments on a single cluster and on multiple clusters. We received six sets of results, five of which are shown in Fig. 4.12. Not surprisingly, single-cluster MPICH-GM runs perform best. The runs obtained with the pure-TCP/IP MPICH-G2 perform worst. Given that the performance of MPICH-G2 for small messages is some 30–45 times worse than that of MPICH-GM, some 6–7 times longer run times with the former are not that dramatic. Surprisingly, *multi-cluster* runs of MPICH-G2 (with processes evenly divided between two DAS-2 clusters) actually perform slightly better than the single-cluster MPICH-G2 runs, in spite of the fact that the latency of the inter-cluster links is twice that of the intra-cluster ones. The cause of this phenomenon is unclear, but it was not investigated further because better options described below exist. Between the GM and G2 curves we find the results of the runs obtained with the hybrid *G2 over GM* version of MPICH. The *multi-cluster* runs of MPICH-G2 over GM perform visibly better than any of the "pure" MPICH-G2 runs, but there is still a performance gap of a factor of 3–4 compared to MPICH-GM. Single-cluster runs of MPICH-G2 over GM (marked in Fig. 4.12 as *(orig.)*), while clearly an improvement over the multi-cluster runs, fail to close the performance gap. This is caused by the unnecessary polling of TCP/IP sockets, as discussed in Sec. 2.5. With the modified MPICH version that does not perform these polls, the performance improves so much, that we have omitted the plot from Fig. 4.12, because it would be indistinguishable from MPICH-GM on this scale. Unfortunately, this modification only improves single-cluster runs.

One remaining issue is the somewhat disappointing efficiency of the bulk anti-messages. We have been expecting a significant improvement compared to ordinary aggressive cancellation, but the results presented so far are almost identical for both approaches. Fig. 4.13 presents a few results from a study conducted to explain the behaviour observed.

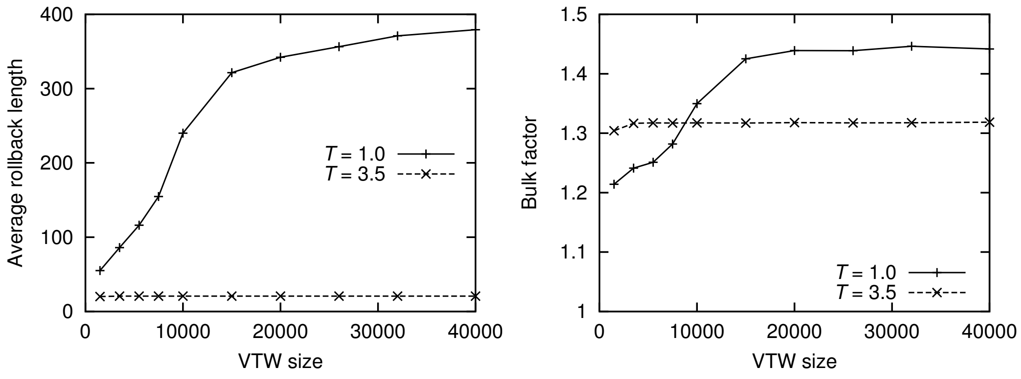


Figure 4.13: A study into the efficiency of bulk anti-messages with the Ising spin simulation on the Grid: average rollback length (*left*), bulk factor (*right*).

The results shown have been obtained for 24 processes, running in a multi-cluster Grid environment, using MPICH-G2 over GM. In Fig. 4.13 (*left*), average rollback length as a function of the VTW size is shown for the temperatures of 1.0 and 3.5. We can see that for the temperature of 3.5 the rollbacks are far shorter than for 1.0, up to a factor of almost 20. This is because the rollbacks for $T = 3.5$ are far more frequent, so the processes have little chance to significantly diverge from one another in the simulation time. For the temperature of 1.0, the simulation is far more computation-bound, and a large VTW size is needed to fully utilise the available processing power. Naturally then, the larger the VTW size, and hence the more optimism is allowed, the larger the average rollback length. Figure 4.13 (*right*) shows *bulk factor* as a function of the VTW size. Bulk factor is a measure of efficiency, defined as the average number of positive events annihilated by a single bulk anti-message. We can observe that the values achieved are rather low, below 1.5. It cannot be surprising then that the performance does not improve – bulk anti-messages hardly get a chance to show their potential. It is also interesting that the bulk factor for the temperature of 1.0 is not significantly better than for 3.5 (it is in fact worse for small VTW sizes), in spite of the fact that the average rollback size for $T = 1.0$ is far larger. The reason is that while the rollbacks for the lower temperature are larger, they consist mostly of rejected spin flip attempt events (see Sec. 2.4.1), that do not result in any events being scheduled on other logical processes. The percentage of externally scheduled events is thus very low with the temperature of 1.0, and it is only thanks to the much larger rollback sizes that the bulk factor eventually becomes higher than with the temperature of 3.5.

4.4 Conclusion

In this chapter, we have studied the influence of various cancellation optimisations on the performance of our APSIS simulation kernel running the Ising spin model simulation.

APSIS used to support aggressive cancellation only. We have extended it with lazy cancellation and a little-known cancellation optimisation, herein referred to as bulk anti-messages. While lazy cancellation significantly alters the behaviour of a Time Warp kernel, even resulting in semantic differences in certain corner cases, bulk anti-messages simply decrease the overhead of aggressive cancellation.

Experiments performed on a single cluster featuring a low-latency network, using a high-performance MPI communication library, show that employing bulk anti-messages decreases the communication overhead, but only slightly. For the simulated Ising spin model, the influence of lazy cancellation on the amount of communication taking place is far more visible. However, these savings do not translate into any significant improvement in the performance of the simulation kernel.

Identical runs repeated in a high-latency, multi-cluster Grid environment, using the hybrid MPICH-G2 over GM library, show even larger savings in communication. This time, the savings do positively influence the performance, especially in the configuration that requires intensive communication: with lazy cancellation, an improvement close to a factor of two over aggressive cancellation can be observed.

With bulk anti-messages, the results are less spectacular. Investigation shows that the simulation model is partly to blame: it simply schedules too few external events for bulk anti-messages to make any significant difference during a rollback. Still, we will keep this optimisation in the kernel and use it with aggressive cancellation, as it has the potential to decrease the bandwidth requirements without affecting the behaviour of the kernel at a higher level.

Repeating the experiments with lazy cancellation on other combinations of communication libraries and execution environments, and comparing the results, proved quite instructive. Among other things, we could observe that the performance gap between the multi-cluster and the single-cluster runs on top of MPICH-G2 over GM is similar to or even smaller than the gap between MPICH-G2 over GM and MPICH-GM, both running on a single cluster. The first gap is caused by a high latency inter-cluster network, with a latency two orders of magnitude greater than within the cluster. The second gap is simply a missed optimisation in the communication library: when running on a single cluster, there is no need to poll TCP/IP sockets for messages, since these could only arrive from other clusters. Once identified, the problem proved easy to solve. This brought the results obtained using the Grid-capable MPICH-G2 over GM in line with these of MPICH-GM, but only for the experiments performed on a single cluster.

For runs performed on multiple clusters, the gap remains. Given the two orders of magnitude difference in the latency, an increase of at most a factor of four in run time can be considered quite low. The experience from single-cluster runs proves that a significant part of this slowdown can be attributed to the local overhead of querying the TCP/IP sockets and not to the inherent network properties. We should thus try to optimise the multi-cluster runs to limit the non-inherent overheads as much as possible. This will be the focus of the next chapter.

5

Grid-Aware Topology*

5.1 Introduction

In Sec. 4.3.2, we have presented the results of the first experiments performed in a multi-cluster Grid environment. The approach we used to perform those experiments could be described as naive. A simulation kernel with no notion of the network topology had been compiled with a Grid-enabled version of MPICH and executed in a highly heterogeneous networking environment. Yet, the kernel was able to execute the experiments in a quite stable (albeit slow) fashion. The experiments have shown that a significant part of the slowdown can be attributed not to the inherent network properties, but to a local overhead of maintaining network connections.

In this chapter, we will build and evaluate an infrastructure designed to reduce these overheads as much as possible. To the best of our knowledge, no such attempts have previously been made in the area of the parallel discrete event simulation. However, a number of efforts have been described of porting the Time Warp to architectures such as networks of workstations. From these efforts, we will take the idea of using message aggregation to further optimise our communication. We will also look at the efficiency of the GVT calculation algorithm used, and modify it to better match our infrastructure.

We are going to evaluate the performance of our Time Warp kernel in far more detail than in Sec. 4.3. So far, we have been using the Ising spin simulation (see Sec. 2.4.1) for the benchmarking purposes. However, it is desirable to have a second model to ensure that the conclusions drawn are more generally applicable. We will use the PHOLD model (see Sec. 2.4.2) for this purpose. Unlike the Ising spin simulation, PHOLD is a message-

*This chapter is partly based on: K. A. Iskra, G. D. van Albada, and P. M. A. Sloot. Towards Grid-Aware Time Warp. In *Proceedings of the 18th IEEE/ACM/SCS Workshop on Parallel and Distributed Simulation*, pp. 63–70, Kufstein, Austria, May 2004.

initiated model. This should result in a significantly different run-time dynamics, which is precisely what we are looking for.

5.2 State of the art

Multi-cluster computing

A number of solutions similar to what we propose can be found in the broader domain of parallel computing. A good summary on Grid-enabled MPI implementations has been provided by Müller et al. in [129].

MagPIe [100] is an add-on library providing a replacement implementation of all collective MPI operations, optimised for wide area networks. The algorithms used are topology-aware: they guarantee that each wide area link is travelled through at most once to reduce the latency of the collective operations. Unfortunately, such a library is not useful for us, since the APSIS kernel does not use any collective MPI operations, because they are blocking. The library is actually inspired by earlier experiments of Plaat et al. [141], who studied the performance of several different parallel applications on systems with large differences in bandwidth and latency. They found that while the "out of the box" performance on such systems tends to be poor, with relatively simple changes it can be significantly improved.

We have already talked about **MPICH-G2** [99], a Grid-enabled version of MPICH, in Sec. 2.5. This is the MPI implementation that we are using, primarily because of its affinity with the Globus toolkit [65], which is used on DAS-2 for cross-cluster resource management. Globus IO with Globus Data Conversion for TCP/IP is the native communication protocol of MPICH-G2. However, it can also be configured to work on top of an existing, high-performance *vendor MPI* for intra-cluster communication. The collective operations of MPI are topology-aware. The cross-cluster point-to-point communication, on the other hand, is handled by establishing direct TCP/IP connections between the MPI processes that want to communicate with each other. A related MPICH-GQ [161] project attempts to address the issues of guaranteed quality of service for the interconnecting networks. Given the commonly complex, highly bursty nature of the communication patterns found in parallel applications, many existing mechanisms turn out to be unsuitable.

In **Stampi**, the Seamless Thinking Aid MPI [101], the communication between the clusters is handled by *router daemons*, acting as proxies transparent to the MPI job. The daemons can be configured selectively in or out, e.g. it is possible to have a router daemon on one cluster, but not on another one. Communication inside a cluster between the MPI processes and the daemon is performed using UDP/IP. This is unfortunately rather suboptimal, especially considering that there is support for vendor MPI (but only for communication between the MPI processes). Inter-cluster communication can use either the TCP or the UDP protocol.

PACX-MPI, the PArallel Computer eXtension [76] implementation of the MPI standard, while not as well integrated with Globus as MPICH-G2, appears to be the most mature of the solutions discussed here. PACX-MPI supports a vendor MPI, as well as optimised collective operations. In addition to that, it can use *communication daemons* (two per cluster) to handle the inter-cluster communication. Even if the computing nodes of a cluster have non-routable IP addresses or are behind a firewall, running the communication daemons on a node accessible from the outside makes the cross-cluster computing possible. The daemons communicate with computing nodes using the vendor MPI, and with the outside world using TCP/IP. This separation results in an improved performance, since the computing nodes do not need to maintain any TCP/IP links. PACX-MPI also supports data compression and conversion for the inter-cluster communication.

PDES on networks of workstations

Below, we will provide a summary of the efforts within the PDES community related to communication on networks of workstations, deemed relevant also on the Grid.

Carothers et al. [36] experimentally studied the effects of the communication overhead on the performance of Time Warp, with a conclusion that communication latency can significantly degrade the efficiency of Time Warp if events have small granularity (i.e. if the processing cost per event is low). In another experimental study, by Boukerche et al. [27], performance of a simulation distributed over loosely coupled domains has been analysed. The system described is capable of adjusting its behaviour to the changing network conditions, reducing the degree of optimism of individual processes as the network load increases. The kernel is only locally optimistic (Local Time Warp), between the domains a conservative protocol is used. Unfortunately, no real-world experiments have been conducted – the wide area network has been emulated by inducing artificial delays in the local communication channels.

The issue of optimising the communication has also been looked into by Chetlur et al. in [44]. Two strategies of dynamic message aggregation are described there, one where messages are aggregated for a constant age window (the aggregate is sent when the first aggregated message reaches a certain age), and a more advanced one where the size of the age window dynamically adapts to the changing message rate. In experiments performed on a network of workstations and on an SMP machine, a noticeable speedup has been observed for both computing platforms. The dynamic scheme is found to be superior for applications that change their behaviour during the simulation lifetime. Pham [139] compares several different aggregation schemes for conservative simulations, in particular the traditional sender-initiated one with two receiver-initiated ones (the sender keeps aggregating messages until the receiver requests them). The sender-initiated policy is found to be superior, because for the receiver-initiated schemes there is an extra overhead of initiation messages. In a related field of Distributed Interactive Simulation, Liang et al. [106] describe the experiences of using a message aggregation scheme that adaptively determines the optimal bundle size. Results obtained using a simulator show that the technique can be effective while preserving the real-time constraints.

Researchers have also been looking into load balancing, which can be very important on platforms with inhomogeneous resources, such as the Grid. Logical processes need to progress at approximately the same rate, otherwise the system can become unstable.

Glazer and Tropper [79] balance the load by controlling the amount of CPU time allocated to individual processes or, should the imbalance be too high, by migrating processes between nodes. Schlagenhaft et al. [163] study the load balancing of partitioned VLSI simulations, aiming to reduce the rollback time. In their scheme, fine-grain load balancing is available in the form of migrating individual logical processes between nodes. More recently, Carothers and Fujimoto [35] describe the *Background execution* algorithm capable not only of a dynamic redistribution of workload on an existing set of processors, but also of a dynamic allocation and release of computing resources. The algorithm takes into account both internal and external sources of imbalance, the latter being quite common on multi-user networks of workstations. Load balancing has also been studied in our research group, within the Dynamite project (see Sec. 6.3). It was not targetted specifically at PDES, but after some adjustments it could certainly be used for this purpose as well (see Sec. 7.3).

In order to allow the simulation to continue when some of the resources become unavailable, fault tolerance needs to be used. Making a distributed system fault tolerant is not an easy task, especially if good performance is required. Still, the fact that checkpoints and rollbacks are standard parts of the Time Warp protocol significantly simplifies the procedure. Agrawal and Agre [3] propose a replication-based approach, where several copies of each logical process run concurrently on separate nodes. The standard Time Warp protocol is used to maintain the consistency between the object replicas. If the majority of external events are read-only queries, the scheme can even help improve the performance, but if many events change the state of the destination, communication overhead becomes a problem because such events must be sent to all the replicas. A checkpoint-based approach is discussed by Damani and Garg [51]. Checkpoints are periodically saved on stable storage. A failure is modelled as a straggler event with the schedule time of the last recorded checkpoint. After restarting the failed process, the rollback mechanism can be used to synchronise the whole system again. The time of the last recorded checkpoint must be taken into account during the GVT calculation, which makes the latter estimate rather conservative if checkpoints are not saved frequently. Fault tolerance is also currently under investigation in our research group, within the Dynamite project (see Sec. 6.4).

GVT algorithms

GVT calculation is an interesting problem both from a theoretical and a practical point of view, so the literature on the subject is extensive. There are two main difficulties when calculating a GVT value, as pointed out e.g. by Lin and Lazowska [108] or Fujimoto [74]:

- the transient message problem: messages in transit need to be accounted for during the GVT calculation, but determining the set of messages that are in transit is non-trivial,
- the simultaneous reporting problem: local data from all logical processes should be obtained at exactly the same wallclock time, to prevent any events from “falling through the cracks”, i.e. being accounted for neither at the source nor at the destination.

The easiest way to address these problems is to require message acknowledgements and to use a synchronous algorithm. Any unacknowledged message is then assumed to be in transit, so constructing a complete set is trivial. In a synchronous algorithm, all the processes are stopped for the duration of the calculation. This prevents any race conditions from occurring, since no events at all are scheduled while the algorithm is in progress. Needless to say, such an algorithm would be very inefficient: the number of messages in the system would increase by a factor of two, and the processes would at times be blocked for prolonged periods.

An optimisation was proposed by Samadi [162]. The improved algorithm still requires message acknowledgements, but it is asynchronous. The simultaneous reporting problem is addressed by specially flagging message acknowledgements generated while the GVT calculation is in progress. From the point of view of the GVT algorithm, such acknowledgements are not received until the GVT calculation is finished, so the messages will be guaranteed to be accounted for at the source. Two optimisations to this algorithm have been proposed, both topology-related. Preiss [142] reduced the number of the GVT-related messages by arranging the processes in a ring and by introducing a token that travels in the ring, continually computing a new GVT value. Bellenot [22] improved the scalability by introducing a *message routing graph*, a static tree structure that reduces the algorithm's complexity from linear to logarithmic.

The need for message acknowledgements was eliminated in the algorithm proposed by Lin and Lazowska [108]. Essentially, the set of transient messages is determined actively. When the GVT calculation is initiated, processes inform one another about the smallest sequence numbers of the un-received messages. The senders then consult their local data structures to determine the minimum timestamp of the messages that are still in transit.

Another GVT algorithm has been developed by Mattern [124]. It is asynchronous and requires no message acknowledgements. It is based on the concept of distributed snapshots: each iteration of the algorithm requires two *cuts*; a snapshot is taken at each cut. Interestingly, the cuts do not need to be consistent, i.e. messages are allowed to travel from after the cut (the *future*) to before the cut (the *past*). The new GVT value is calculated along the second cut. After the first cut, the processes start recording minimum timestamps of newly scheduled messages. These timestamps are then used as estimates of the timestamps of the transient messages that cross the second cut. *Message colouring* is used to distinguish messages sent before the first cut (*white* messages) from those sent between the two cuts (*red* messages). The second cut cannot complete before all the white messages are accounted for. This can be achieved by counting white messages at the sender and the destination and comparing these counters. If they do not match, a process should simply wait for the messages to arrive. The algorithm, while elegant, is rather complex.

Bauer and Sporrer [19] introduce another, very simple asynchronous GVT algorithm that requires no message acknowledgements. This is the algorithm that we are using. Its simplicity stems from the fact that, unlike the algorithms discussed above, it requires FIFO communication channels. It works on the basis of keeping the counts of the messages sent and received via each channel. Periodically, every process reports these counts to a centralised location, along with the minimum timestamp of the messages sent since the previous report. If the counts from two ends of a channel are the same, then there are no messages in transit along the channel. Otherwise, the list of the message counts reported by the sender needs to be consulted. A minimum timestamp is calculated of all the entries with the count of the messages sent greater than the count of the messages received on the other side of the channel. This minimum is used as an estimate of the transient messages' timestamp. The quality of this estimate depends on the frequency of the reports sent to the centralised location. If this frequency is the same as that of the GVT algorithm, which is usually the case, then the estimate's quality will be rather low.

An interesting algorithm has been proposed by Concepcion and Kelly [45]. It is a hierarchical, multi-level token passing algorithm. The processes are arranged in a hypercube topology, and on each level GVT tokens are constantly circulating, one per a subcube face. Data on the minimum timestamp of the scheduled events are forwarded to the higher levels. A selected process on the highest level determines the next GVT value, that is subsequently circulated via tokens to its counterparts at the same level and forwarded to the lower levels. While the idea is elegant, the implementation, requiring message acknowledgements and using dedicated processes for GVT generation (one per logical process) has a rather high overhead. Another algorithm taking advantage of the hypercube topology has been described by Das and Sarkar in [54] (this publication also provides a good summary of many existing GVT algorithms). Unlike Concepcion and Kelly, they do not use a token passing algorithm, but rather use the hypercube to build a spanning binomial tree, and use this structure to pass the GVT-related messages. Their algorithm does not require message acknowledgements, but it is synchronous.

5.3 Routing processes

5.3.1 Process topology

Figure 5.1 (*a*) presents a single cluster of the multi-cluster simulation runs discussed in Sec. 4.3.2. Logical processes inside the cluster communicate with each other via Myrinet. A subset of them will also want to communicate with the logical processes on the other cluster(s). The messages are sent using TCP/IP via Fast Ethernet to a local switch/router (not shown), that has an uplink to a wide area network. The situation at the remote cluster(s) is identical. Because all the logical processes are allowed to communicate with each other, they must all poll both the Myrinet and the TCP/IP stack for any messages that might have arrived. This has a detrimental effect on the performance, as we have discussed in Sec. 4.3.2.

The solution we came up with to address this problem is shown in Fig. 5.1 (*b*). On every cluster, an additional *routing process* is reserved that is responsible for the wide area communication. Routing processes, while part of the MPI job, are not part of the simulation, i.e. no logical processes are assigned to them. Simulation processes are relieved of the wide area communication: whenever a message needs to be sent to a process at a remote cluster, it is sent (via Myrinet) to the local routing process instead. The

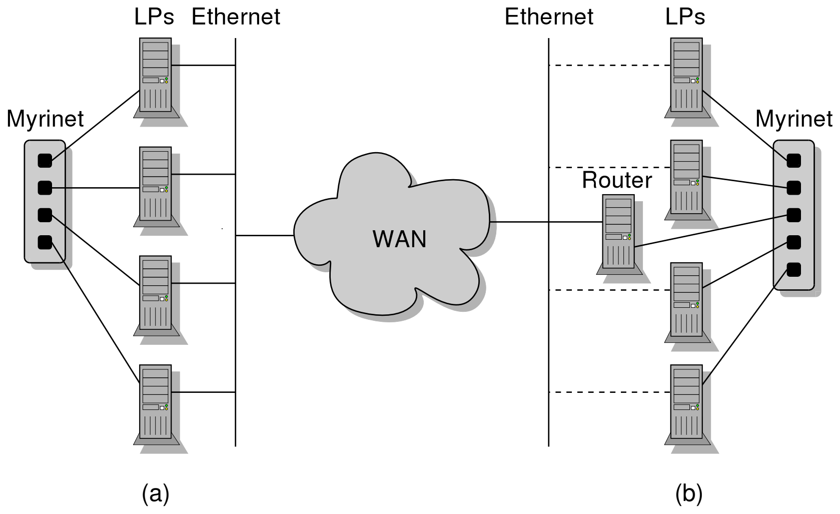


Figure 5.1: Process topology: original (*a*) and with a routing process (*b*).

routing process extracts the identifier of the intended destination process from the message header, consults its routing table and sends the message to the appropriate remote routing process over a wide area link. The remote routing process forwards the message (via Myrinet) to the intended destination LP.

To emphasise, no physical modifications to the cluster are made: every node is still capable of the direct wide area communication, this ability simply is not used by the resident processes anymore (hence the dashed lines in Fig. 5.1 (*b*)).

Some of the technical details of this approach are depicted in Fig. 5.2. The global MPI communicator `MPI_COMM_WORLD`, that encompasses all the processes across multiple clusters, is split into local sub-communicators, one for every cluster (`LAN_COMM_1` and `LAN_COMM_2` in Fig. 5.2). Simulation processes (small white rectangles in Fig. 5.2, four per cluster) only use their local communicators to exchange messages. In principle, an MPI communicator is just a convenient abstraction providing a separate communication

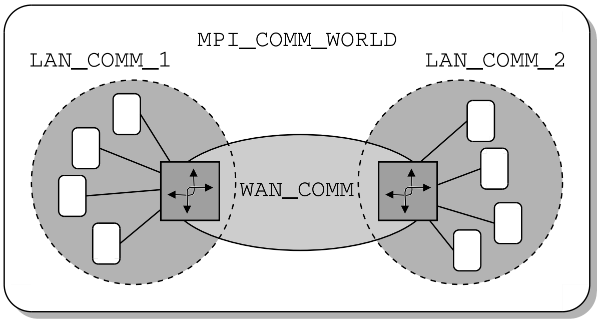


Figure 5.2: MPI communicators used with the Grid-aware topology.

context. However, if all the processes of a communicator belong to one cluster, there is no need for the expensive TCP/IP socket polls, since all the messages will arrive via Myrinet. If the modified version of MPICH-G2 over GM is used (see Sec. 2.5), TCP/IP polling will indeed be avoided. The creation of sub-communicators takes place automatically on initialisation. The simulation kernel uses the topology discovery extensions offered by MPICH-G2 [99] to find out how it is distributed. On every cluster, one process is selected to be the routing process (the two dark rectangles with a router symbol inside in Fig. 5.2). In addition to the local sub-communicators, a global sub-communicator is created that consists of the routing processes only (`WAN_COMM`). With the modified MPICH-G2 over GM, MPI routines that use this communicator only poll the TCP/IP sockets.

Compared to other solutions found in the literature, our implementation is most similar to PACX-MPI [76]. However, PACX-MPI is a complete, generic implementation of the MPI standard, and its communication daemons are elegantly transparent to the MPI application. Our solution is domain specific, and the routing processes are part of the MPI job. On the other hand, such an implementation makes it easy to implement additional optimisations that we are going to describe later.

5.3.2 Experiments

Figure 5.3 presents the performance results of the first experiments. The Ising spin simulation has been used, configured to use 24 simulation processes, for two model temperatures, and using either lazy (*top*) or aggressive (*bottom*) cancellation. Run times in seconds are shown as a function of the VTW size. Each plot contains three curves. *G2 over GM multi-cluster* and *GM* should already be known from Sec. 4.3.2 (in fact, the curves in the top plots are the same as in Fig. 4.12 on p. 72). New curves, called *Grid*, present the results of the multi-cluster runs with the modified MPICH-G2 over GM, with the routing processes. As we can see, the *Grid* runs perform visibly better than the ordinary *G2 over GM multi-cluster* most of the time. The improvement is particularly spectacular in case of lazy cancellation and the temperature of 1.0 (Fig. 5.3 (*top-left*)): the performance is very close to that of the single-cluster *GM* runs! This is the case when the least communication takes place. If we increase the communication volume, either by switching to aggressive cancellation or by increasing the model temperature, the performance is no longer so good. Still, a significant improvement compared to the multi-cluster runs without routing processes can be observed also in these cases. The communication volume is highest if aggressive cancellation and the temperature of 3.5 are used at the same time, as is the case in Fig. 5.3 (*bottom-right*). We can see that the multi-cluster runs perform much worse than in the other cases (note a different scale on the y axis). In fact, the *Grid* curve is nowhere to be found: the simulation runs were unstable under these circumstances and could not finish in any reasonable time. We will discuss this problem later.

We have performed similar experiments with PHOLD, as shown in Fig. 5.4. Instead of the run time, we calculated speedup, so the higher a value, the better. Also, we have disabled the virtual time window throttling. The motivation for the latter change is that for PHOLD the results with a disabled time window appear to be only slightly worse than

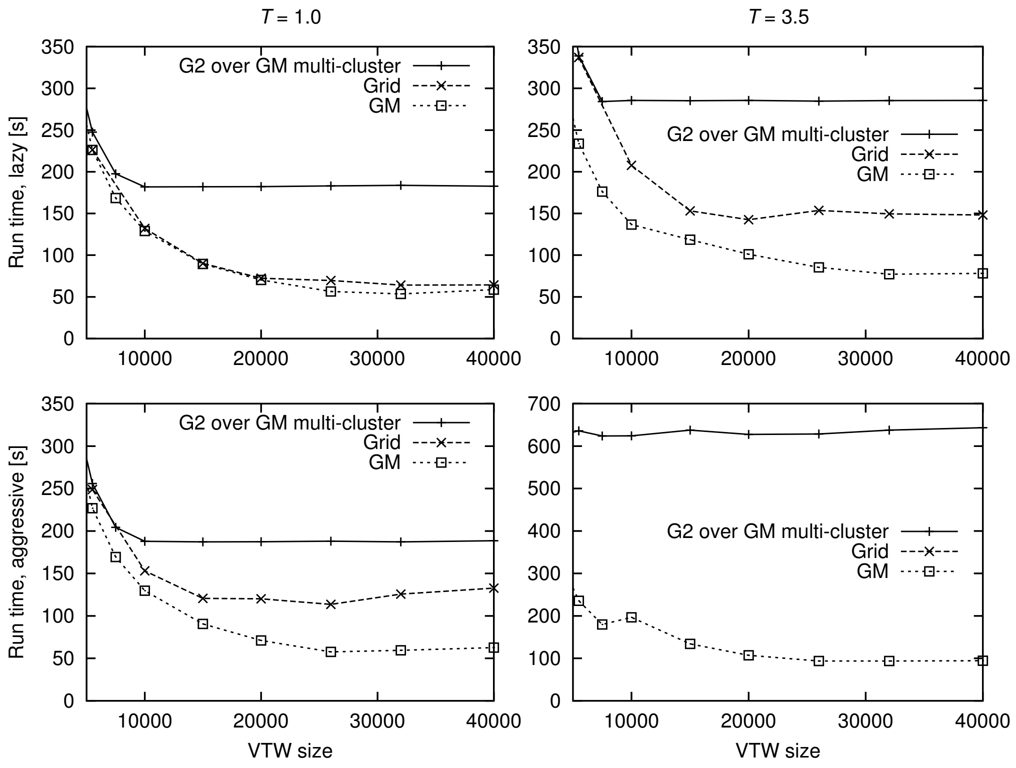


Figure 5.3: Run times of the Ising spin simulation when running on different communication layers, for different VTW sizes.

with an optimally tuned window size, while finding the optimal value for each experiment would significantly increase the number of runs to be performed. All the experiments with PHOLD have been performed with aggressive cancellation, using bulk anti-messages. The x axis shows the number of processes, making it possible to perform a scalability study. We have conducted the experiments for four values of the computational grain, ranging between 0 and 35 μs per event. Like in Fig. 5.3, the experiments have been performed on a single cluster (*GM*), on multiple clusters (*G2 over GM multi-cluster*), and on multiple clusters with the routing processes (*Grid*). The single-cluster experiments have been performed up to 32 processes, the multi-cluster ones up to 128 simulation processes. The latter were evenly split between two clusters up to and including 64 processes; the case of 128 processes was run on four clusters (which is one of the reasons why the performance is then lower than for 64 processes). Looking at the case of 0 μs grain (Fig. 5.4 (*top-left*)), we can observe that, not surprisingly, the single-cluster runs perform much better than the multi-cluster ones. The problem is that the *Grid* results are almost invisible: they are only there for the cases of two and four processes, that are hardly interesting. For larger numbers of processes, the kernel destabilises and the simulations progress poorly or stop completely. We have observed a similar behaviour for the Ising spin simulation, see Fig. 5.3 (*bottom-right*). Without the routing processes,

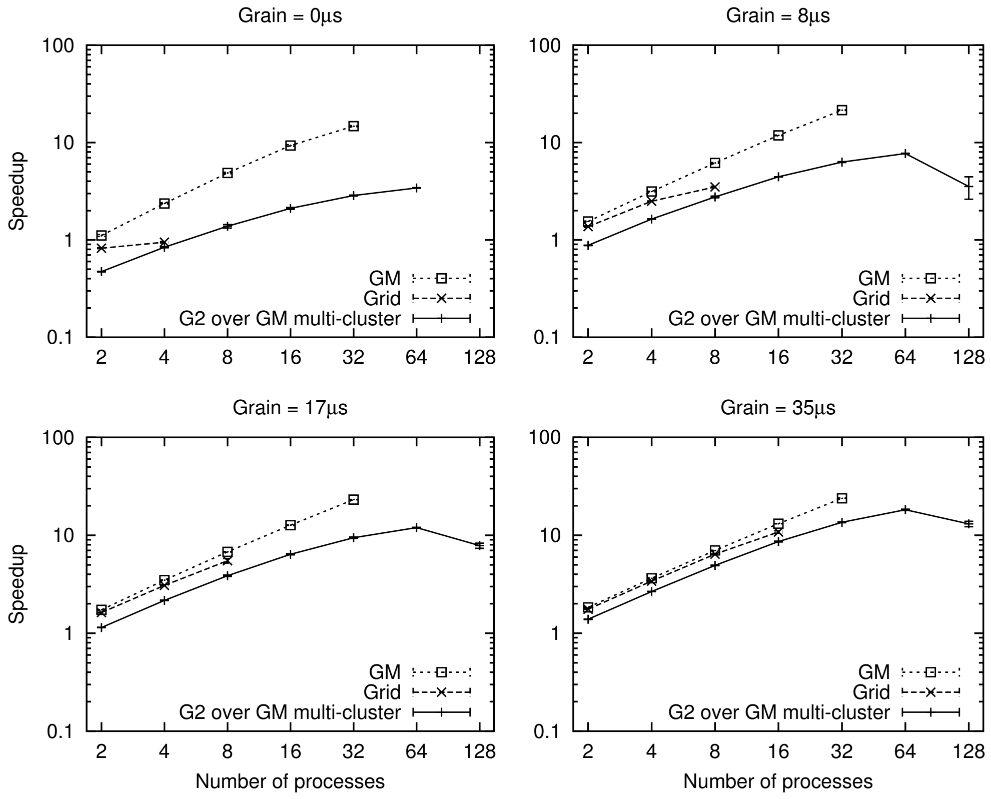


Figure 5.4: Speedup achieved on different communication layers with PHOLD, as a function of the number of processes, for different computational grains. Error bars indicate the standard deviation. Note the logarithmic scale on both axes.

the runs are far more stable, as they succeed up to and including 64 processes. Increasing the per-event computational grain improves the stability. Because a larger fraction of the execution time is then spent on computation, fewer events per wallclock time unit are scheduled, so the communication becomes less of a problem. With the grain values of 8 and 17 μs, the *Grid* runs succeed up to 8 processes. The performance is clearly better than that of the equivalent multi-cluster runs without the routing processes. Still, supporting 8 processes is a far cry from the 128 processors that are available. The case of 128 processes remains problematic also for runs without the routing processes: while the simulations succeed, the speedup is lower than for 64 processes. As already mentioned, for 128 processes the simulation is split between more clusters, negatively affecting the speedup. Another reason is that the communication rises rather quickly with an increasing number of processes, as discussed in Sec. 2.4.2. Therefore, as the number of processes increases, the speedup must start decreasing at some point. Increasing the grain to 35 μs further improves the stability: up to 16 processes can be used with a decent performance with the routing processes. However, the multi-cluster runs without the routing processes exceed this performance if 32 and more processes are used. In either case, a gap compared to the single-cluster runs remains.

5.3.3 Further improvements

All in all, the scheme with the routing processes is mostly successful for the Ising spin simulation, and is also successful for PHOLD, provided that it works. The problem is, that in many cases, especially with aggressive cancellation (which we always use with PHOLD), it just fails. We will focus on these cases for the time being. The routing processes apparently become a bottleneck: they cannot cope with the traffic in real time, so messages are forwarded with large delays. Handling of such delayed messages at the receiver's side requires huge rollbacks. Not only is this expensive, but it further destabilises the system, since anti-messages generated while rolling back cause secondary rollbacks at other processes resulting in rollback cascades. In the end, the processes almost permanently roll back without making any real progress. This is known as *rollback thrashing*. There are several reasons why simulations with routing processes are susceptible to this behaviour.

Firstly, there is the issue of how the routing processes should handle each of the two forwarding directions, e.g. should one direction have a priority over the other? In the experiments just described, no such preferences are given, and a single forwarding direction is handled as long as any messages in that direction are pending. The opposite direction is ignored for that time, which under high loads could lead to *starvation*. Indeed, the rollback thrashing observed is partly caused by this phenomenon. However, experiments we have performed with various forwarding strategies have shown the one currently used to be the most stable. Starvation can be prevented by introducing an artificial limit on the number of messages that can be forwarded in one direction at a time. However, this also decreases the peak throughput of the routing processes. At least with the Ising spin simulation, initially the communication tends to be very intensive (see Sec. 3.3.1), so a high peak throughput is important.

Secondly, the introduction of the routing processes increases the inter-cluster communication volume. This is caused by the fact that the simulation processes are no longer slowed down by the TCP/IP communication, so they can schedule more events in a (wallclock) time unit. Because of a discrepancy in the performance between the intra-cluster and inter-cluster links (see Sec. 2.5), the routing processes can become clogged with messages. A way to address this problem would be to prevent the simulation processes from generating too many messages by throttling them. Employing an ordinary Time Warp moving virtual time window turned out to be insufficient: even if the majority of the logical processes reached the top of their window, a few stragglers would keep on sending messages, causing the throttled processes to roll back and thus sustaining the excessive communication in the system. This could of course be addressed with a very conservative window size, but such a small window would limit the performance, often unnecessarily. The throttling mechanism should instead activate only when the communication layer is not keeping up with the load. One way to recognise this condition is by measuring the inter-arrival times of the messages belonging to the GVT protocol. Because the frequency of the GVT computation algorithm is constant, estimating the expected arrival time of the next GVT message is easy. If the message does not arrive on time, the logical process is throttled. Event processing is suspended, so no new messages are sent and the process can focus exclusively on receiving pending messages. Once the

GVT message is received, event processing is resumed. The experiments performed have proved this scheme to work, in the sense that the simulations regain stability and make progress. Unfortunately, the rate of this progress is unsatisfactory: sufficient communication throughput is a prerequisite to achieving a good performance.

Thirdly, and lastly, there is thus the issue of ensuring a sufficient throughput. Figure 5.5 presents the most straightforward solution: providing multiple routing processes per

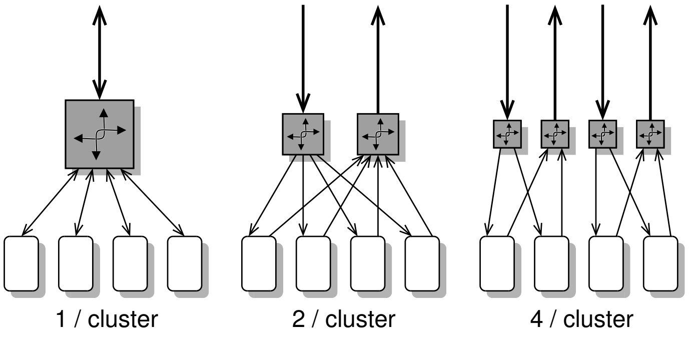


Figure 5.5: Several configurations of the routing processes per cluster.

cluster. If the support is turned on, the routing processes are split into two groups: those responsible only for receiving data from the local logical processes, and those receiving data only from the other clusters. In addition to increasing the throughput, this should also effectively solve the problem of when to switch the forwarding direction. With multiple routing processes, there is simply no need to do that.

Figure 5.6 presents a comparison of the speedup achieved with PHOLD, for three different configurations of the routing processes, as shown in Fig. 5.5. As we can see, for small numbers of processes, where the simulations are generally stable, introducing mul-

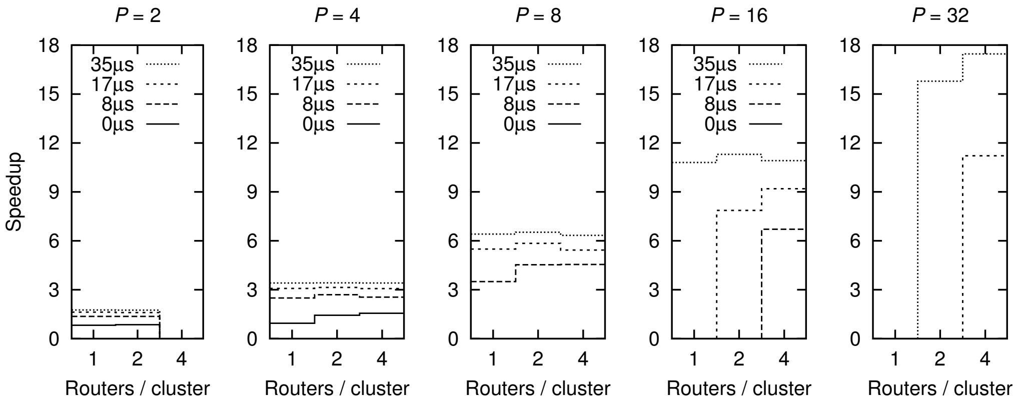


Figure 5.6: Speedup achieved with PHOLD as a function of the number of routing processes per cluster, for different computational grains and different numbers of the simulation processes. The key from $P = 16$ applies to $P = 32$ as well.

tiple routing processes per cluster has little or no influence on the performance. However, if the simulation runs are unstable with a single routing process per cluster, introducing multiple routing processes clearly helps. This can be observed for grain values of 8 and 17 μs for $P = 16$, and for grain of 17 and 35 μs for $P = 32$. Multiple routing processes per cluster thus fulfil our expectations.

Regarding the missing data for $P = 2$ with four routing processes per cluster: in this case, only two routing processes per cluster would actually be used. This is because, for any two simulation processes, exactly one route is always defined, with a uniquely identified routing process on the source cluster and on the destination cluster. Otherwise, the FIFO property could be violated.

Note that the results presented in Fig. 5.6 are in a strong contrast to experiments we have performed earlier. The results of the earlier experiments indicated that using two routing processes per cluster was a disadvantage. We had to repeat the experiments, because, upon closer examination, the results for four processes per cluster turned out to be invalid (due to a poorly chosen hashing function, they degenerated to two routing processes per cluster if the simulation processes were arranged in the torus topology). In both “old” and “new” experiments, care has been taken to ensure that the results were not unfairly influenced by some temporary external factors. However, the “new” experiments have been conducted after the operating system of the DAS-2 machine has been updated. It could be that the new version of Globus, 3.2, simply has a better optimised Globus IO module than the old 2.2.4 version. It could also be that the new, server-optimised operating system kernel of the Red Hat Enterprise Linux Advanced Server 3 has a better tuned TCP/IP stack. We must conclude that one of these factors, possibly both, is responsible for this change in behaviour.

5.4 Message aggregation

The existence of separate processes concerned only with the wide area communication makes it easy to implement an efficient message aggregation. Instead of performing the time consuming send operations over the wide area link separately for every message, the messages can be stored in a buffer, which is sent when it gets full or once a timeout expires. Figure 5.7 presents the results of some experiments performed over the inter-cluster link between the DAS-2 clusters of UvA (in Amsterdam) and in Leiden. As we can see, the travel time of a single 4 KB message is only moderately (28%) higher than that of a 156 byte message (a standard message size in the APSIS kernel), while the throughput increases by 327% over the same size range. Both quantities were measured using a dedicated benchmark, making use of MPICH-G2. The travel time was measured in a ping-pong experiment, and the throughput in a ping flood.

The effectiveness of our aggregation scheme is increased by the fact that the routing processes concentrate messages from multiple sources and for multiple destinations. This would not be the case if the message aggregation was implemented in the simulation processes themselves, as is the case with other implementations described in the literature. Message concentration improves the efficiency by increasing the average size

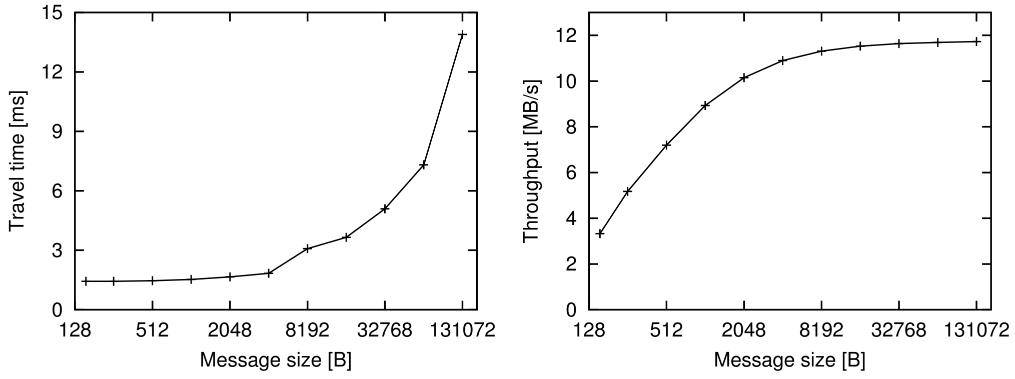


Figure 5.7: Performance characteristics of the inter-cluster link. Message travel time (*left*) and throughput (*right*), both as a function of the message size in bytes. Note the logarithmic scale on the x axis.

of the wide area messages sent.

Admittedly, the message aggregation adds latency by storing the messages in buffers instead of sending them immediately. However, in many situations this small loss is greatly outweighed by the increased throughput, which decreases the waiting time in the send queue. This should help the simulations that have high communication requirements. If the communication layer is overwhelmed by messages, aggregation will in fact reduce the effective latency.

The aggregation mechanism we have implemented is highly tunable. First, it can be turned off completely. The size of the aggregation buffer (which is separate for every remote routing process) can be adjusted. The default value is 4 KB, since for even larger values the travel times start rising quickly, while the throughput is already close to the maximum (see Fig. 5.7). Increasing the buffer size further was experimentally found to have negligible effect on the performance of the Ising spin simulation, so all the experiments presented below have been performed with the default buffer size. The aggregation timeout is also tunable. A value of zero is not quite the same as no aggregation at all: a routing process will still receive and aggregate the pending messages from the local simulation processes, but the buffer will be sent out as soon as a non-blocking receive attempt on the local communicator returns without a message.

5.4.1 Experiments

We have repeated the experiments with the Ising spin simulation shown earlier in Fig. 5.3, this time with the message aggregation in place. The results can be found in Fig. 5.8 (*Grid agg. timeout* x μs), compared as before to the single-cluster runs (*GM*) and the multi-cluster runs without the routing processes (*G2 over GM multi-cluster*). We have performed a number of experiments with different values of the aggregation timeout, the plots show the ones giving the best results (we will investigate the issue of the optimal timeout in Sec. 5.4.2). The results with the aggregation show a significant improvement. The run times for the temperature of 1.0 are very close to single-cluster results both for

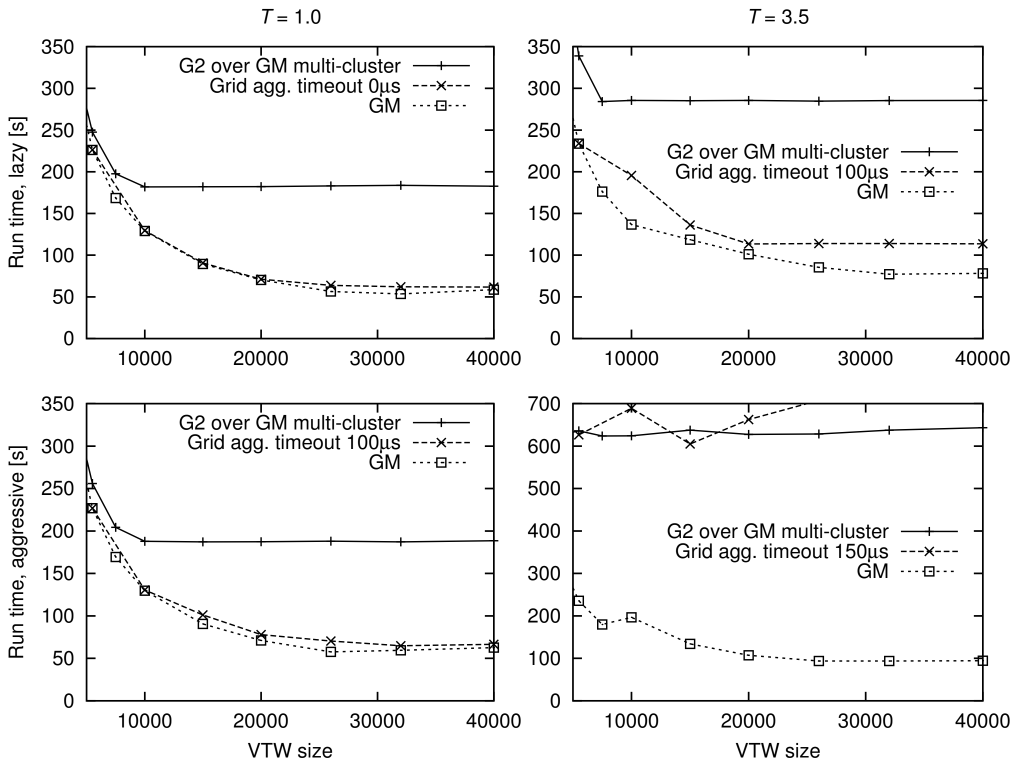


Figure 5.8: Run times of the Ising spin simulation when running on different communication layers, for different VTW sizes. *Grid* runs with message aggregation (compare with Fig. 5.3 on p. 85), with one routing process per cluster.

lazy and aggressive cancellation. Also the results for the temperature of 3.5 are improved. With lazy cancellation, the gap is reduced by half. With aggressive cancellation, most simulations at least successfully finish, although the kernel is still very unstable, so the results fluctuate and the performance remains unsatisfactory.

Similarly, we have repeated the PHOLD experiments shown earlier in Fig. 5.4. The results with message aggregation (timeout 100 μs) can be found in Fig. 5.9. Significant improvements in the stability can be observed. Where before we could only support up to four simulation processes for the computational grain of 0 μs, we can now run with up to 32 processes. The performance is also better than without the routing processes, although not by much. Still, looking at it objectively, the performance with $Grain = 0\,\mu$s is not good: for 32 processes we achieve a speedup of just 3.0. However, PHOLD is only a synthetic workload. For actual simulations, performing some computations while handling events is a perfectly natural behaviour. With a non-zero computational grain, the performance improves significantly, as can be observed in Fig. 5.9 for plots $Grain = 8\,\mu$s to $Grain = 35\,\mu$s.

Comparing the performance of *GM* and *Grid* runs, we can observe that for the largest computational grain of 35 μs, the speedup is quite similar, although in the *Grid* runs it

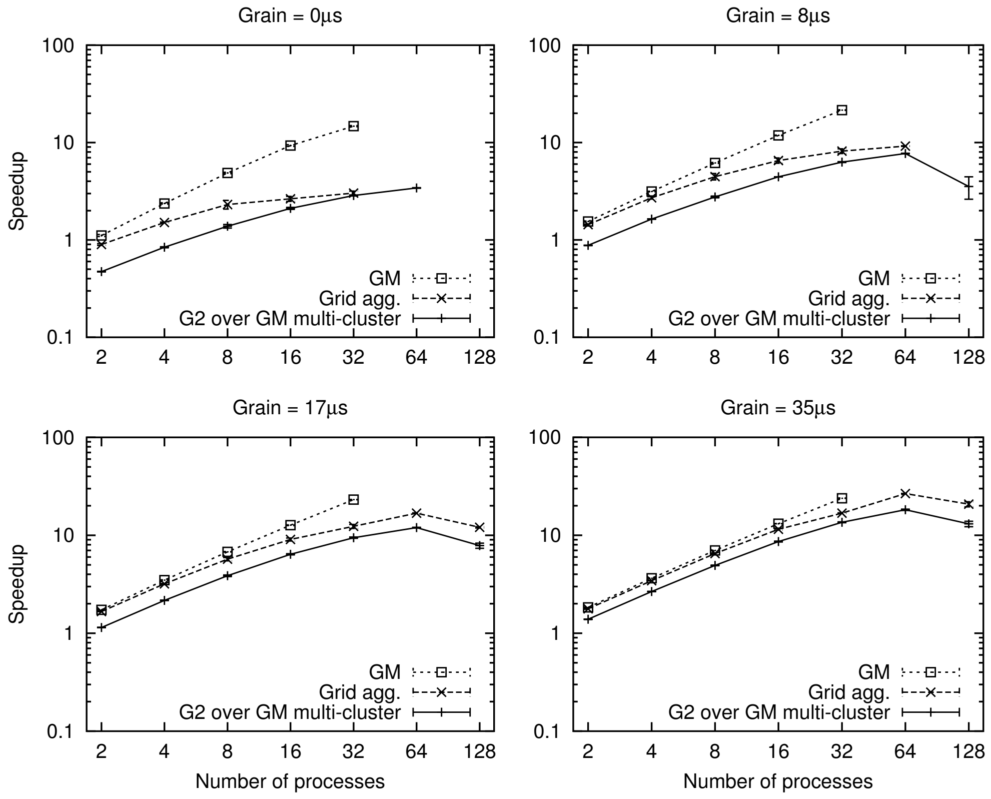


Figure 5.9: Speedup achieved on different communication layers with PHOLD, as a function of the number of processes, for different computational grains. *Grid* runs with message aggregation (compare with Fig. 5.4 on p. 86), with one routing process per cluster. Error bars indicate the standard deviation. Note the logarithmic scale on both axes.

rises slower with the increasing number of processes. However, the availability of larger resources in the distributed case allows us to cross this gap by increasing the number of processes: with 64 processes the multi-cluster *Grid* runs do perform better than their single-cluster counterparts. For smaller computational grains the comparison is not as favourable, but that was to be expected. The case of 128 processes remains a challenge: the speedup decreases. As discussed in Sec. 5.3.2, this is caused by the fact that the processes are then distributed across more clusters (four instead of two), and also because the communication intensity grows with an increasing number of processes. Performance does improve if we use more routing process (results not shown), but even with four routing processes per cluster, the performance with 128 processes is still below that of 64 processes.

5.4.2 Aggregation timeout

From the above experiments it should be clear that the message aggregation is generally an advantage: in every case we have encountered, turning it on resulted in a performance improvement. But how does changing the aggregation parameters, in particular the aggregation timeout, influence the run-time behaviour of the simulation kernel?

Figure 5.10 attempts to provide an answer to this question. We have taken measurements for a range of timeout values, with a special focus on the region of 0–200 μs. The influence of the aggregation timeout on both the simulation run time and the communication are presented. The experiments presented were performed with the Ising spin

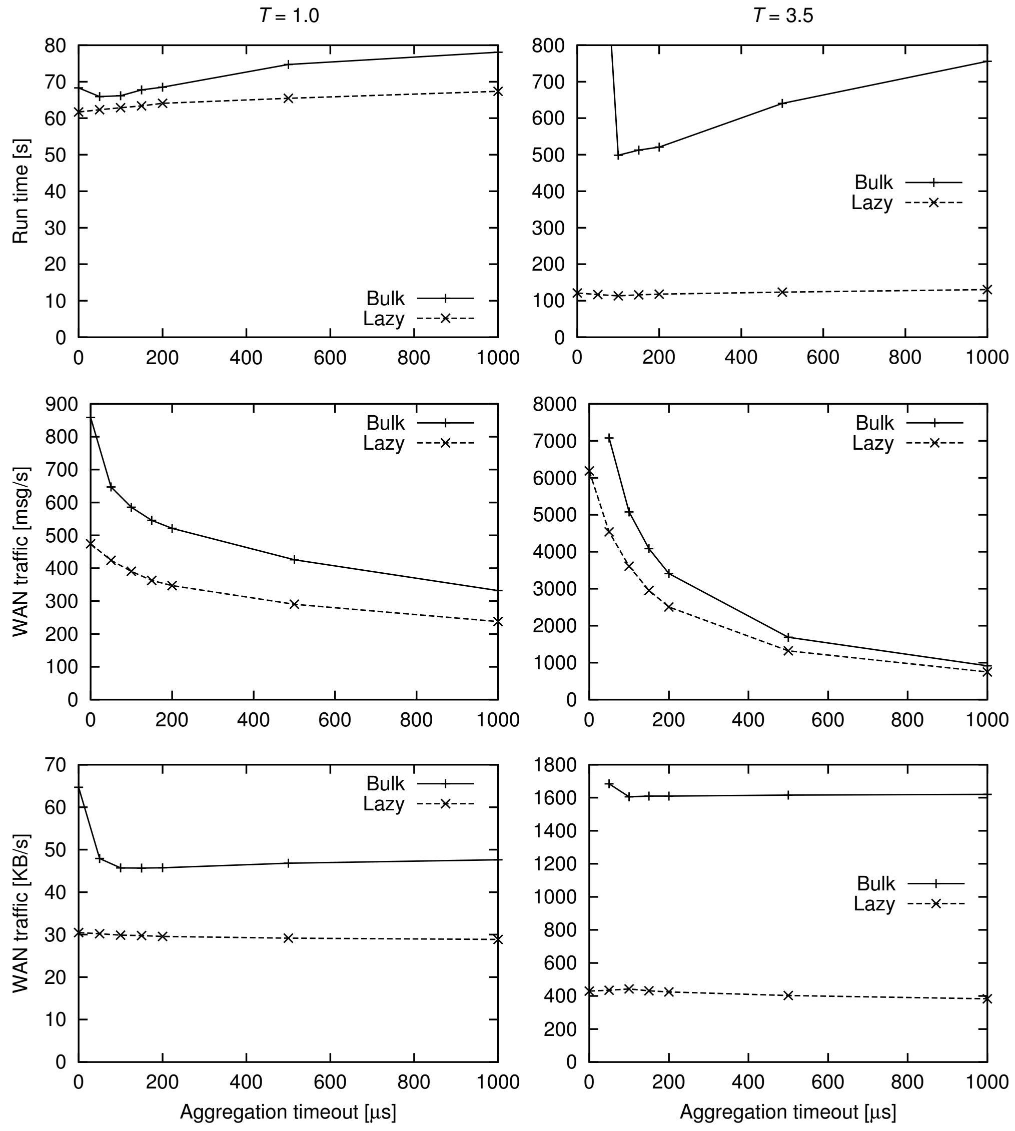


Figure 5.10: Influence of different aggregation timeouts on the behaviour of the Ising spin simulation: elapsed run time (*top*), average number of messages (*middle*) and data volume (*bottom*) sent by a single routing process via a wide area link in a time unit.

simulation on 24 simulation processes distributed evenly across two clusters, with one routing process per cluster. VTW size was 40000. Two model temperatures and two cancellation strategies have been tried: lazy cancellation and aggressive cancellation with bulk anti-messages. The latter tends to take less bandwidth and is slightly more stable than aggressive cancellation with individual anti-messages (as presented in Fig. 5.8), but apart from that the general trends are the same.

Figure 5.10 (*top*) presents the simulation run times. As we can see, with one exception, they are only slightly influenced by the aggregation timeout. The larger the timeout, the longer it takes an average message to reach its destination process, naturally thus the total simulation run time increases as well. If the communication is not very intensive, as is the case with lazy cancellation for the temperature of 1.0, the best run times are achieved for the timeout of zero (the semantics of this value of timeout has been discussed earlier). If the communication is more intensive, the run times are shortest if a small aggregation timeout is used.

As the timeout gets smaller, the number of messages that need to be sent across the wide area link increases sharply, as shown in Fig. 5.10 (*middle*). Because sending the wide area messages is expensive, if their number per time unit becomes too high, the performance suffers. Consequently, in configurations requiring a more intensive communication, a small aggregation timeout improves the performance by limiting the number of wide area messages. This is especially well visible for the temperature of 3.5 with bulk anti-messages: if the timeout is too low, the run times explode, as the routing processes cannot cope with the message load. In this particular case, the system is saturated with messages, which is probably why the run time is much more strongly affected by the increasing aggregation timeouts than in the other cases. In general, while the number of wide area messages quickly decreases with an increasing aggregation timeout, the throughput remains largely unaffected, as shown in Fig. 5.10 (*bottom*). With lazy cancellation, the throughput tends to decrease with an increasing aggregation timeout, while with aggressive/bulk cancellation the tendency seems to be reversed, but the differences are so small that it is difficult to draw any conclusions. We can see that the inter-cluster communication can be quite intensive: it reaches 1.6 MB/s with bulk anti-messages and the temperature of 3.5. An effective method of decreasing the bandwidth requirements, should that be needed, is to throttle the kernel by decreasing the VTW value (but that does increase the run time).

The maximum bandwidth requirements we have observed in the cases where the simulations were still able to successfully finish were around 2.5 MB/s. Such values can be achieved if aggressive cancellation with ordinary anti-messages is used. This is not much compared to the maximum throughput of close to 12 MB/s (see Fig. 5.7 on p. 90). However, that maximum was measured in a case of uni-directional communication, without any overheads, and with large buffer sizes. With the routing processes the communication is bi-directional (at least in the case of a single routing process per cluster), and there are overheads associated with analysing the message headers, copying the messages between buffers, querying the time source, etc. For the *Bulk* case with $T = 3.5$ from Fig. 5.10 the *message aggregation factor* (the ratio between the number of messages received from local processes and the number of wide area messages sent) is fairly high: between 5.2 (agg. timeout 50 μs) and 39.3 (agg. timeout 1000 μs). While the standard size of a message in APSIS is 156 bytes, when aggregating, the unused portion of the user data buffer is skipped, shrinking the size to (in case of the Ising spin simulation) at most 52 bytes. Average size of a wide area message is thus in the range of 280–2000 bytes, which is fairly small and thus further contributes to limiting the maximum throughput. In order to achieve a good performance, the average communication requirements of a simulation should be much smaller than the available throughput. Apparently, the communication is very bursty in nature, probably due to cascading rollbacks occassionally taking place, and a large margin is needed to handle such bursts efficiently. Using the message aggregation is an advantage precisely because it can provide a high peak throughput at such moments.

Back to the results, a general conclusion we can draw is that the aggregation timeout should be small. Sometimes, a timeout of 0 μs gives the best results, in other cases values around 100 μs are better (and this value has been chosen as the default), but values above 150 μs always perform sub-optimally.

One of the bonuses of implementing the message aggregation in the MPI application user space is the fact that we can influence the aggregation behaviour based on the message type. We have implemented two such mechanisms.

Firstly, if a message received by a routing process is a part of the GVT computation protocol, it forces an immediate flushing of the aggregation buffer, so that the GVT-related message can be delivered as quickly as possible. The experiments we have conducted to verify the usefulness of this optimisation were inconclusive: the differences in the performance were within the noise level. Nevertheless, we have kept the mechanism in, as it seems like the right thing to do, especially given the importance of the GVT protocol (see Sec. 5.5.1).

Secondly, we have implemented a similar aggregation buffer flushing for anti-messages. Since anti-messages are generated to cancel previously sent erroneous events, they should be given a preferential treatment. If they are delivered faster, fewer (or at least shorter) rollbacks should be needed. A performance evaluation of this mechanism can be found in Fig. 5.11. A comparison is presented between three different aggregation strategies: *No aggregation* (only routing processes), ordinary *Aggregation* and *Aggregation, negative flush*, where receiving an anti-message causes an immediate flushing of the aggregation buffer. All the experiments have been conducted with the Ising spin simulation, for 24 simulation processes distributed evenly between two clusters, with one routing process per cluster. The aggregation timeout of zero was used. The results are rather disappointing: forcing the premature sending has no effect for lazy cancellation, even with a high communication intensity (see Fig. 5.11 (*left*)) and for aggressive cancellation the performance gets worse, even if the temperature (and thus also the communication intensity) is low (see Fig. 5.11 (*right*)). In the former case too few anti-messages are sent to make any measurable difference, whereas in the latter a significant increase in the number of expensive wide area messages apparently outweighs the benefits of a faster delivery of anti-messages.

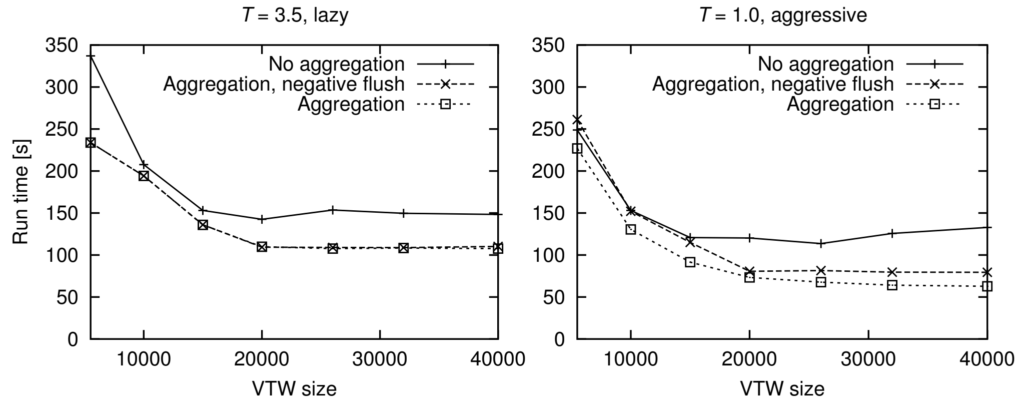


Figure 5.11: Performance of the Ising spin simulation with different configurations of the message aggregation, for different VTW sizes. Aggregation with a timeout of zero.

5.5 GVT algorithm

5.5.1 Motivation

The GVT algorithm used, based on the work of Bauer and Sporrer [19] (see Sec. 5.2), was originally implemented in one of the simulation processes. The experiments with the Ising spin simulation presented above were conducted in this configuration (see App. A). We have always found this assignment of an extra task to one simulation process to be inelegant, since it disturbs the balance of work in the system. With the introduction of dedicated routing processes, it became possible to move the GVT computation there.

An additional motivation for this move can be found in Fig. 5.12, that shows the speedup achieved with the Ising spin simulation as a function of the GVT computation frequency. The simulation has been configured for 24 simulation processes, with lazy

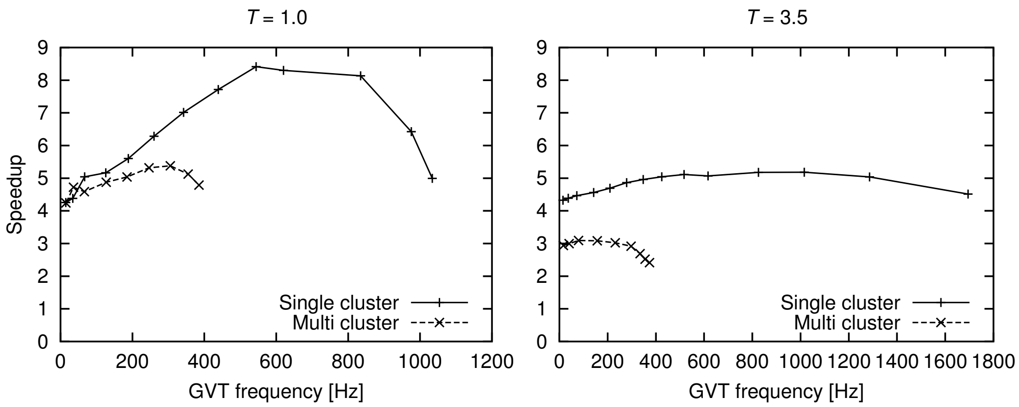


Figure 5.12: Influence of the GVT computation frequency on performance of the Ising spin simulation.

cancellation, and no VTW throttling. The experiments have been performed on a single cluster and on multiple clusters (two, to be precise, with one routing process per cluster and the message aggregation with timeout zero). Before these experiments were conducted, we always used the default GVT computation frequency of 20 Hz, assuming that its influence on the performance was negligible. As can be observed in Fig. 5.12, this is not always the case. An optimal performance for single-cluster runs is achieved for GVT frequencies of around 500–800 Hz. For the temperature of 1.0, an improvement in speedup of a factor of two can then be achieved. This might appear counter-intuitive at first: a more frequent GVT computation increases the overhead. However, it also decreases the number of past simulation states and events that need to be kept in the queues. A lower memory footprint of the simulation processes translates into a more efficient use of the CPU cache. This results in an overall performance improvement. An improvement can also be observed for multi-cluster runs, but it is much smaller then. One of the reasons is that the optimal GVT frequency simply cannot be achieved in the multi-cluster runs, as the wide area network latency limits the frequency to some 400–500 Hz. We should thus try to make a better use of the frequency range available, by improving the efficiency of the GVT algorithm.

5.5.2 Efficiency improvements

Figure 5.13 shows the changes made to the GVT protocol. In the original implementation

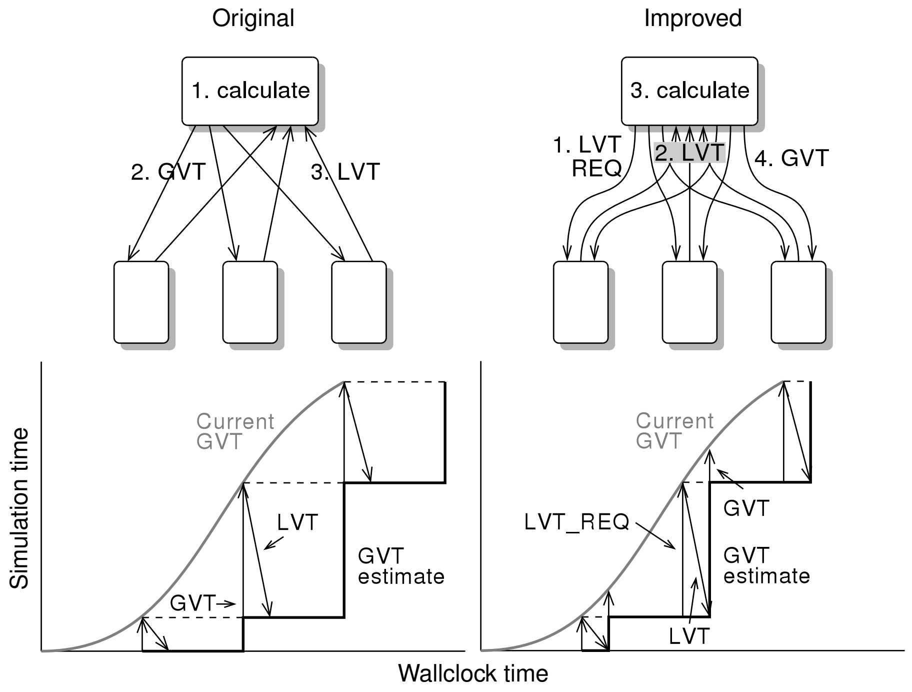


Figure 5.13: Efficiency improvements to the GVT algorithm.

(see Fig. 5.13 (*left*)), the algorithm minimises the amount of communication required. No requests for up to date information are sent; the process that calculates GVT does so based on the data already available to it (*1. calculate*). A newly calculated GVT estimate is disseminated to the other processes (*2. GVT*). These processes reply immediately by sending the current information on the communication channels (*3. LVT*). At low GVT frequencies, these data will be rather old by the time they are needed during the next iteration of the algorithm, making the already conservative GVT estimate (see Sec. 5.2) even more so.

Two optimisations have been put in place. Firstly (see Fig. 5.13 (*right*)), the process performing the GVT calculation sends a request for the latest data to the other simulation processes (*1. LVT_REQ*), and only performs the calculation when the data have been received. This increases the amount of the GVT-related communication in the system by 50%, but also allows the algorithm to calculate a GVT estimate far closer to the ideal value (see Fig. 5.13 (*bottom*)). Secondly, we have introduced message acknowledgements between the logical processes, so that the processes can estimate the timestamps of the transient messages much more accurately by considering only the as yet un-acknowledged messages. The communication overhead of the acknowledgements is virtually zero, as they have been implemented in the form of piggybacking the count of received messages to an ordinary message sent in the opposite direction. If no acknowledgements are received (e.g. because the communication between two processes is uni-directional), the algorithm falls back to its non-optimised mode of operation described above.

The influence of these optimisations is evaluated in Fig. 5.14, where the efficiency of three algorithms is compared. *Original* does not use any of the above discussed optimisations, *With requests* uses the *LVT_REQ* messages to request the latest data, and *With reqs and acks* uses the requests and also the message acknowledgements. The experiments shown have been performed with the Ising spin simulation, on 24 processes, on a single cluster, with lazy cancellation and a VTW size of 40000. The efficiency is shown

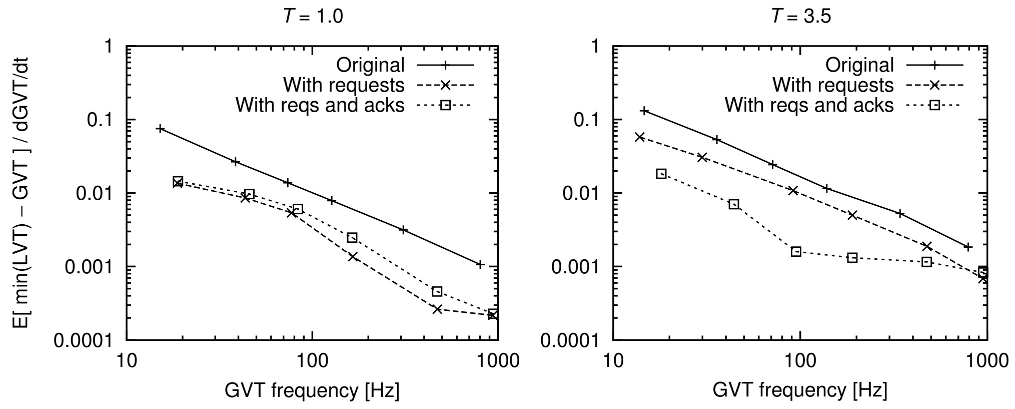


Figure 5.14: Average distance between the minimum LVT and the newly calculated GVT, with different GVT algorithms, as a function of the GVT computation frequency, for the Ising spin simulation. Note the logarithmic scale on both axes.

in a form of the average distance between the newly received GVT estimate and the minimum LVT of the simulation processes at that time. The necessary data have been collected using an instrumented version of the simulation kernel: each process generated a separate output file, containing a sample of its local data; the files have been processed post-mortem. To ensure that the overall performance of the simulation does not affect the results, the distances are divided by the progress rate. Otherwise, if for some possibly unrelated reason the simulation with one algorithm was progressing consistently faster than with another one, the average distance shown would also be larger, even if the efficiency of the two GVT algorithms was the same. Because of the transient messages that could be present in the system, the distance will usually be non-zero, but it should be relatively small. As expected, the distance with the original algorithm is largest, so the efficiency is lowest. Introducing data requests helps significantly (please note that a logarithmic scale is used). The efficiency of the algorithm making use of the message acknowledgements depends on the amount of communication. If the external events are scheduled rarely (see Fig. 5.14 (*left*)), message acknowledgements are infrequent, and the efficiency is similar to the one observed without the acknowledgements. If the communication is intensive (see Fig. 5.14 (*right*)), the estimate used with *Original* and *With requests* is more conservative, because the earliest external event is scheduled sooner after the local information is sent to the centralised location. *With reqs and acks* on the other hand benefits from the higher traffic, since the message acknowledgements are received sooner, so the estimate improves.

5.5.3 Distribution

Making use of these optimisations in the multi-cluster case is a little more complex. While the message acknowledgements are in themselves not a problem, increasing the number of the GVT messages per iteration certainly is. With a message travel time of 1400 μs, every extra message that needs to be exchanged can significantly reduce the maximum frequency of the GVT computation.

A more extensive use of the routing processes for the GVT computation helps to reduce this problem. Figure 5.15 presents two possible hierarchical solutions, a distributed and a centralised one. In both cases, the GVT calculation is moved from a simulation process to the routing processes. Periodically, on each cluster a routing process sends to its local simulation processes a request for up to date message counts (*1. LVT_REQ*). Once all the simulation processes reply (*2. LVT*), a new GVT estimate can be calculated.

In the *distributed* algorithm, shown in Fig. 5.15 (*left*), each routing process calculates a new GVT value on its own (*3. calculate*). Then it redistributes the new value to its local simulation processes (*4. GVT*), along with the message counts received from other processes, to facilitate the message acknowledgements if the communication frequency on some channels is unbalanced. The local data obtained from the simulation processes are also forwarded to the routing processes on other clusters (*5. LVT*), so that they can be used during the subsequent iterations of the algorithm there. Therefore, on each cluster the latest data from the local processes and older data from the remote processes are available. Consequently, each cluster will calculate a slightly different estimate of GVT.

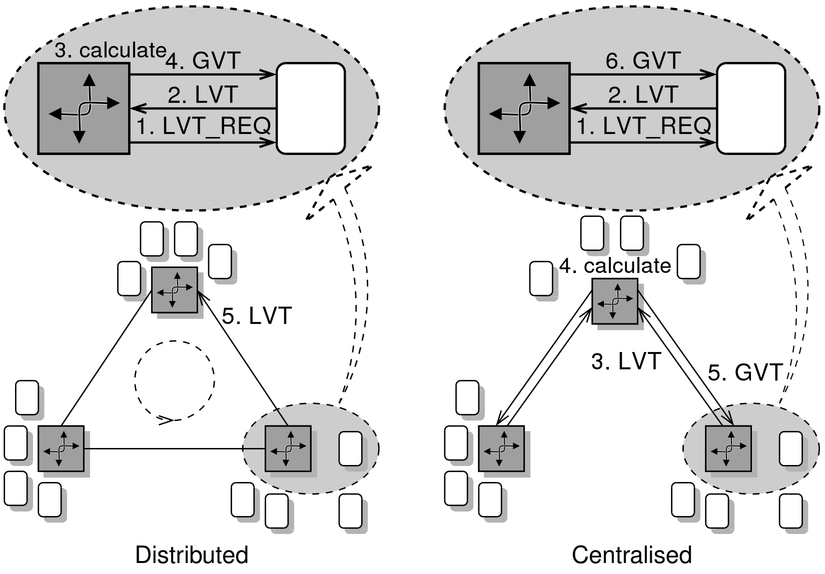


Figure 5.15: Two hierarchical GVT algorithms.

More importantly, the different age of the input GVT data obtained from the local and remote processes makes it very difficult to depend on the message acknowledgements, as shown in Fig. 5.16 (*a*). A local simulation process could send a message e_1 to a remote process and receive an acknowledgement ack_1, but during the subsequent iteration of the GVT algorithm the input LVT data received by the routing process RP_1 from the remote cluster containing process LP_2 would likely not yet take the received message into account. The local sender process LP_1 would not include the message either, since it has already received an acknowledgement for it. This is an example of the simultaneous reporting problem. A radical solution, currently in use, is not to send message acknowledgements to the processes on other clusters. Actually, the same problem could occur if the two simulation processes run on a single cluster (see Fig. 5.16 (*b*)), since the

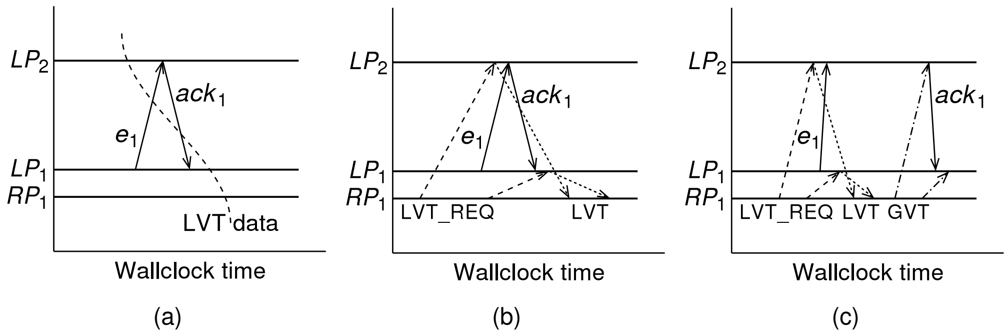


Figure 5.16: Potential problems with message acknowledgements: multi-cluster (*a*) and single-cluster (*b*), a solution to the latter (*c*).

LVT_REQ messages are not perfectly synchronised. However, for the local processes the problem is easily fixable (see Fig. 5.16 (*c*)) by not sending the acknowledgements during the time period between the *LVT_REQ* and *GVT* messages from the routing process, and this period is expected to be relatively short. This algorithm bears some resemblance to that of Concepcion and Kelly [45], because it is also hierarchical and the processes at the higher level of the hierarchy circulate data between themselves. However, the two-level hierarchy of our algorithm is intended to follow the topology of the multi-cluster execution environment, whereas the algorithm of Concepcion and Kelly creates a hypercube topology. Further, our algorithm does not have a centralised process responsible for the GVT calculation, and multiple GVT messages circulate between the routing processes (as many as the number of the routing processes themselves). The chief advantage of our algorithm is the small number of wide area messages required. Its disadvantages include its complexity and its limited efficiency at low frequencies (due to the use of old data from the other clusters).

The *centralised* approach, shown in Fig. 5.15 (*right*), makes it possible to avoid these problems. The data obtained from the local simulation processes are not used locally, but are forwarded to a central routing process (*3. LVT*). This process calculates a new GVT value (*4. calculate*) and sends it back (*5. GVT*) to the other routing processes, that in turn can disseminate it to the local simulation processes (*6. GVT*). Because with this algorithm the age of the message counts from all the clusters is the same, message acknowledgements can be used also for the inter-cluster communication (although, as before, the acknowledgements need to be suspended while the GVT calculation is in progress).

The algorithm would be even better synchronised if the central routing process initiated the computation by sending a request for data to the other routing processes. Instead, each routing process initiates the algorithm on its own. The central routing process attempts to synchronise these local initiation times by measuring the arrival times of the *3. LVT* messages, calculating the recommended time shifts and sending them back to the other routing processes with the *5. GVT* messages. We have done it this way to avoid the extra communication step that would otherwise be required. The *centralised* algorithm already requires one more step than the *distributed* one, that effectively cuts the maximum GVT computation frequency by half.

Figure 5.17 presents a comparison of the speedup achieved with the *Original*, not optimised algorithm implemented in one of the simulation processes, and the two improved algorithms proposed above: *Distributed* and *Centralised*. The experiments have been performed with the Ising spin simulation, for 24 simulation processes, with one routing process per cluster, lazy cancellation and no VTW throttling. As discussed earlier, the original algorithm calculates the worst estimate of GVT, so we could expect the speedup with this algorithm to be the lowest, at least at low GVT frequencies. Surprisingly, that is not the case: for the temperature of 1.0, and at low and medium GVT frequencies, it performs better than the other two algorithms, although the differences are not large. The centralised algorithm calculates the best estimate of GVT and thus performs quite well, especially for the temperature of 3.5, but only at low GVT frequencies. It quickly looses as the frequency increases, because of its more complex message exchange protocol. Eventually, at high GVT frequencies, the distributed algorithm is the winner. It

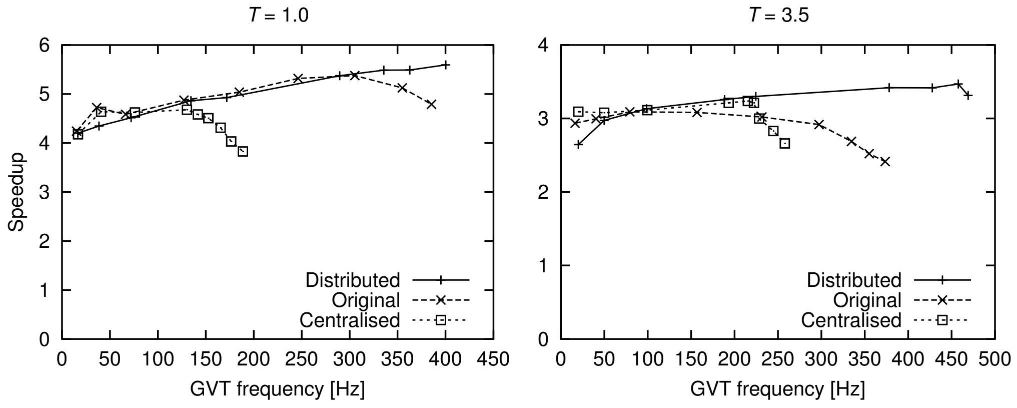


Figure 5.17: Performance of the multi-cluster Ising spin simulation under varying GVT frequencies for different GVT algorithms.

works much better under such conditions than the original algorithm, that, being implemented in a simulation process, apparently cannot deal efficiently with the overhead. Nevertheless, the improvement over the original algorithm is not that spectacular: it is less than 5% for the temperature of 1.0 and some 12% for the temperature of 3.5.

We should add that we have performed similar experiments with the PHOLD model, but it turned out to be far less sensitive to the GVT computation frequency. Significant changes in the performance could only be observed for the runs with low numbers of processes, since in these cases the memory footprint of individual processes is largest. However, our attention focuses on the experiments with larger numbers of processes (32 and more), and in these cases the differences on a single cluster were within 10%.

5.6 Conclusion

In this chapter, we have further studied the behaviour of the APSIS kernel in multi-cluster Grid environments.

Based on the conclusions from earlier experiments, we have built an infrastructure dedicated to wide area communication. Auxiliary routing processes have been introduced as the sole maintainers of the wide area links between the clusters. Logical processes, no longer hindered by the TCP/IP communication, can perform far more efficiently as a result.

The majority of the experiments performed with the Ising spin simulation confirm the effectiveness of this approach. Especially with a low model temperature and with lazy cancellation, performance close to that of the single-cluster runs can be observed. On the other hand, if a high temperature coupled with aggressive cancellation is used, the simulation becomes unstable. Experiments performed with PHOLD provide similar results: if a simulation run is stable, it performs better with the routing processes in place than without them. Unfortunately, even with a large computational grain, multi-cluster runs with the routing processes, configured to run using 32 processes or more, are unstable. Introducing a special throttling mechanism stabilises the simulations, but their performance remains poor. Increasing the number of the routing processes per cluster to two or four provides a far more stable infrastructure, so that many more simulations using 16 and 32 processes succeed with a good performance.

Subsequently, we have extended the routing processes with message aggregation, to further optimise the wide area communication. Because the cost of sending a single message across a wide area link is high, and for small messages it only increases slightly with an increasing message size, aggregating multiple small messages can improve the performance by decreasing the number of the wide area messages. Repeating the experiments with the Ising spin simulation and PHOLD fully supports this claim: the performance is further improved, and the simulations are far more stable. Experiments indicate that message aggregation with a small timeout is generally an advantage. While the influence of the aggregation timeout on the throughput requirements is limited, the number of messages exchanged across the wide area links is significantly reduced.

For the Ising spin simulation, especially at low model temperatures and for the single-cluster runs, large increases of the GVT frequency can significantly improve the performance. The effect is less visible for multi-cluster runs, because the GVT frequency is limited by the wide area network latency. Instead of increasing the frequency, we have thus decided to improve the efficiency of the GVT algorithm used. We modified the protocol so that the algorithm works on the latest data possible, and we added low-cost message acknowledgements to improve the algorithm's accuracy. Further, the GVT computation has been moved to the routing processes and made hierarchical to improve the load balance and scalability. Two different distribution approaches have been tried. Eventually, the more distributed algorithm, featuring the simplest inter-cluster communication protocol, turned out to be the most efficient, but the improvement in speedup is rather small. GVT frequency does not appear to have a significant influence on the PHOLD simulation.

Routing processes thus serve multiple purposes, and they could serve even more. For example, Noronha and Abu-Ghazaleh [133] describe the *early cancellation* optimisation that makes it possible to cancel messages within the communication layer, not only in the simulation kernel. While they apply the optimisation to programmable network interfaces, it would certainly be possible to put this functionality in the routing processes as well.

We have taken efforts to ensure that all these optimisations can efficiently co-exist. Multiple routing processes per cluster can safely be used together with the message aggregation. If the communication is intensive, the performance obtained with such a combination tends to be better than with either of the optimisations turned off. Message aggregation code also ensures that GVT messages are forwarded without delay, maximising the GVT computation frequency. On the other hand, multiple routing processes per cluster potentially present a problem for the new, distributed GVT algorithm. This is because the routing process that receives LVT data from the other clusters is then not the one responsible for the GVT computation. However, with a simple message forwarding between the local routing processes the problem was easy to solve.

While all these new features can improve the performance, they also add more dimensions to the configuration space, making it more difficult to find the combination resulting in the most optimal run-time behaviour. A solution to this problem would be to add a mechanism dynamically adjusting the configuration parameters at run time, based on the feedback from the kernel. Such adaptive schemes have been investigated before, e.g. Radhakrishnan et al. [148] describe a mechanism capable of adjusting the configuration of state saving, cancellation strategy, and message aggregation. Tuning many parameters is not easy, so in [150] they apply the concepts from control theory to build a more intelligent, hierarchical control system.

6

Polder Metacomputing Environment*

6.1 Introduction

In this last chapter of the thesis before the conclusion, we will provide a short introduction to the Polder Metacomputing Environment [88, 186], another Grid-related project that we have been involved in. In the future, some elements of Polder could be incorporated into the APSIS kernel. Polder is an experimental environment comprising software and hardware for high-performance computing and interactive simulation in the field of computational science.

Computational science strives to assist researchers in studying complex problems involving phenomena that can be described by computable models. These may involve physical or chemical processes, but are certainly not limited to these domains: biology, economy and traffic management are some of the other important application areas. Typically, such a study involves an evaluation of the computational model for a range of model parameters. The goal of the study influences how the parameters must be chosen, and how the results of each model calculation are processed. Generally, the parameter space is very large, making a dense sampling of this space prohibitively expensive. When the goal is to study the behaviour of a system over a wide range of parameters, it is important to quickly locate the regions where interesting phenomena occur, e.g. phase transitions. When the goal of the study is to optimise the behaviour of the system according to certain criteria, an efficient optimisation method is needed.

Problem solving environments aim to provide tools to perform the actual model evaluation, combined with data analysis and presentation tools to analyse the results from

*This chapter is partly based on: K. A. Iskra, R. G. Belleman, G. D. van Albada, J. Santoso, P. M. A. Sloot, H. E. Bal, H. J. W. Spoelder, and M. Bubak. The Polder Computing Environment, A System for Interactive Distributed Simulation. *Concurrency and Computation: Practice and Experience*, vol. 14, no. 13–15, pp. 1313–1335, 2002.

a collection of simulations for a range of parameters, and ideally, tools to efficiently select new parameter values on the basis of earlier results. Specifically, the selection or adjustment of model parameters is a difficult problem, and in many situations is best performed in an interaction between the human expert and the problem solving environment. This leads to the creation of interactive simulation environments. The details of such an environment are strongly influenced by the specific application; the underlying mechanisms should be as general as possible to allow re-use of code.

The planning of surgical procedures is an interesting and important area for the application of computational science techniques, see e.g. Sloot [166]. By providing appropriate support to the surgeon, the cost of the planning procedure can be reduced and, more importantly, the success rate of the procedure can be improved. For example in vascular surgery, when planning a bypass procedure, surgeons aim to construct the bypass in such a way that the blood flow is restored as well as possible, while also ensuring that new blockages (stenosis) are unlikely to form. The deposition rate of clots is found to depend on the shear-stress in the flow. Thus, a surgeon can be greatly helped if the blood flow resulting from an intervention can be predicted through interactive simulation of the flow before and after the construction of the bypass. Such a system has been under development in our research group for some time, see e.g. Belleman [21]. It uses the geometry of the patient's arteries derived from magnetic resonance techniques as input, an immersive interactive environment (a CAVE [48]) for presentation and interaction, and a lattice-Boltzmann code as a flow simulator. This set-up requires a reliable and high-performance computing and networking environment that can give near real-time response.

This takes us to the computer science aspects of the Polder Metacomputing Environment. Polder was designed to provide an accessible, versatile experimental environment to an extended research community within a part of the Netherlands. Where current Grid technology aims to create a global computing environment (see Sec. 1.4.2), the objectives of Polder are more confined. It adheres to a Municipal Area Network and intranet paradigm, rather than a WAN and internet paradigm. Polder predates the Grid, and thus could not then make use of the extensive facilities in the area of security and authorisation that are now available through e.g. the Globus toolkit [65]. Polder demonstrates the possibilities of coupling and combining heterogeneous and independently managed facilities at the participating institutes for a multitude of research projects. The reasons for keeping the scope limited lie in the better manageability and security that could be realised in this way, as well as a more predictable performance and a higher reliability. Many of the tools, concepts and applications developed within the Polder initiative have subsequently been ported to the Grid, e.g. in the European CrossGrid project, see Sloot et al. [169] or Tirado-Ramos et al. [180–182].

DAS, the Distributed ASCI Supercomputer, played a central role in this metacomputing environment. DAS was built based on the same principles as its current successor – the DAS-2 machine, described in Sec. 2.5. It was a wide area distributed multi-cluster. DAS was connected to various high-performance systems, including the CAVE and the supercomputers at the national computing centre, SARA. *Space-sharing* was used to schedule the nodes of the DAS, and this is still the case on DAS-2. In order to improve the utilisation in general while retaining the responsiveness of the system for high-priority tasks, we have been developing a dynamic, time sharing scheduling system for parallel programs, Dynamite [89–93, 185]. Dynamite allows the migration of tasks in a parallel program. This dynamic resource management makes it possible to allocate CPU cycles in a dynamic, priority-based manner, honouring the requirements of high priority programs, without unnecessarily starving lower priority jobs.

The realisation of an interactive simulation environment imposes a variety of requirements on the system, besides the performance and reliability provided by a system like DAS. The need for a combination of high performance computing and an immersive presentation and interaction environment imposes requirements on the interoperability of these systems. The need for interaction with a time-dependent simulation imposes requirements on the management and co-ordination of the virtual (simulation) time within the various components of the system. Portability to other systems and adaptability to other applications implies a need for a precise separation of functionalities into modular components. These requirements have led us to adopt the High Level Architecture (see Sec. 2.2) as a middleware layer and to implement various integrative and support functions in intelligent agents, see e.g. Zhao [199].

In the rest of this chapter we will focus on Dynamite, the dynamic task migration component of Polder, because this is the part that we were directly involved in. Further, Dynamite could in the future be integrated with APSIS to provide load balancing and fault tolerance (see Sec. 7.3).

6.2 State of the art

Checkpointing

Checkpointing is not a standard feature on most UNIX systems. Hence, over the years many projects have been developed to fill this gap. However, nowadays, with the quick development pace of operating systems such as GNU/Linux, finding a checkpointer that works “out of the box” with a current version of the operating system kernel and the system libraries proves rather difficult.

Kernel-level checkpointers normally have the most features, but also “age” fastest, often being suitable only for one specific revision of the operating system. MOSIX [17] and its offspring openMosix are well known examples of this approach. Completely unmodified binaries can be transparently migrated at any time. Only the user-level part of the process is actually migrated, the kernel-level component, called a *deputy*, stays on the original node and communicates with the user-level part via a network socket. Non-local system calls, such as I/O operations, are forwarded via the socket and executed on the original node. A migrated process can make use of most UNIX features, with the exception of shared memory and low-level direct I/O. Another relatively recent kernel-level checkpointer is EPCKPT, a part of Nomad [140]. Its feature set is similar to that of MOSIX. As we write this, neither supports the current stable Linux kernel 2.6 series.

User-level checkpointers tend to be easier to implement. Probably the best known checkpointer of this sort is the one found in Condor [144]. User-level checkpointers typically lack support for more esoteric features of the UNIX programming interface. Support for shared libraries is a particularly challenging problem. It is difficult for the checkpointer to find out what shared libraries are used, and it is even more difficult to ensure that, when restoring, the same libraries will be loaded into the same memory slots. Condor is not an exception here, e.g. under Linux it requires the binaries to be statically linked to old versions of system libraries. Better support for recent Linux versions is offered by another user-level checkpointer called CKPT that, in addition to supporting shared libraries, makes it possible to checkpoint an already running, unmodified process, see Zandy et al. [197]. Zandy and Miller [196] also developed a system providing transparent network connection mobility, that can be used for reliable socket communication in the presence of task migration.

Migrating parallel tasks

An ability to checkpoint a process is not enough to migrate a task of a parallel application. Dedicated solutions are needed to preserve the communication consistency.

CoCheck [175] is one such solution for PVM applications. Interestingly, it is more of an add-on to PVM than a modification, as the inner workings of PVM remain unchanged. All communication functions are wrapped. The wrappers are responsible for blocking the migration while a PVM function is being executed, and for ensuring that the user code does not notice that a task has been migrated. When a migration is to take place, the connections between all the tasks of the application are flushed and broken. This ensures that no messages will be lost in transit, but is of course suboptimal if only one task is to be migrated. An improved protocol of this sort can be found in ChaRM [53]. Only connections with the task(s) to be migrated are flushed, the rest can communicate with one another without problems. Even the messages sent to the migrating tasks while the migration is in progress do not block, because they are temporarily stored in a delaying buffer in the source task.

While implementing the task migration without modifying PVM is attractive from the maintenance point of view, it is prone to sub-optimal behaviour in various corner cases. MPVM [38] takes a different approach, by directly modifying the internals of the PVM library. The implementation has a number of interesting characteristics. For example, the migration protocol is fully asynchronous: only the migrating task is immediately notified of the fact, the rest of the system is informed in a lazy fashion. Messages can thus at times be sent to an old location, and need to be forwarded to the current one.

Similar solutions also exist for MPI. There is even an MPI version of CoCheck [174], based on the same principles as the PVM version discussed above. Another, fairly complete solution is offered by Hector [157]. It provides not only support for task migration, but also for fault tolerance, and it includes a monitoring and scheduling systems for optimal task allocation. It can also fill in the role of a cluster management software, given that it can manage multiple MPI programs running simultaneously, belonging to different users. Other fault-tolerance implementations of MPI include MPICH-V [26], that relies on uncoordinated checkpoint/rollback and distributed message logging, and FT-

MPI [57], that can respawn crashed tasks, but depends on the user to recover the lost computations. Open MPI [75] appears to be a major implementation currently under development.

6.3 Task migration

Dynamite, the time sharing scheduling component of Polder, was developed within the European Esprit project 23499, in collaboration with ESI, the Paderborn Center for Parallel Computing and Genias Benelux.

Essentially, Dynamite consists of three components: monitoring, scheduling and migration, as shown in Fig. 6.1. An application has to be decomposed into subtasks already.

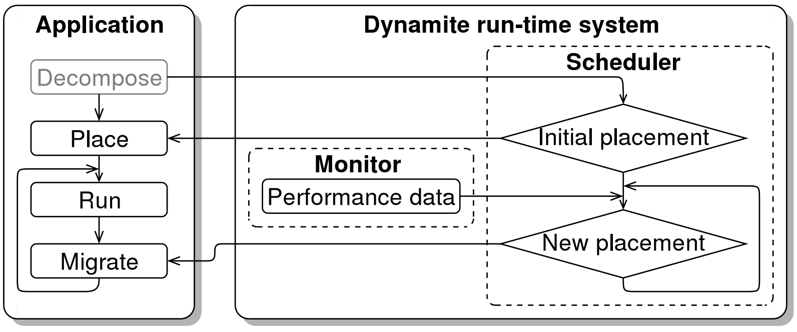


Figure 6.1: The architecture of Dynamite.

The scheduler determines an initial placement, one that might not be optimal yet. While the application is running, the distributed monitoring system constantly checks the resource usage of each subtask, as well as the resource availability on each node, and reports back to the centralised scheduler. If the scheduler finds the load imbalance to exceed a pre-defined threshold, it instructs the migration component to move one or more subtasks between the nodes to create a new, more optimal load distribution.

The main motivation for continuously ensuring an optimal task allocation is that both resource availability and resource requirements can dynamically change in time. The former is commonly the case on networks of workstations and on clusters with time sharing scheduling. The latter depends on the application: some computational problems have a property of radically changing their resource requirements as the computation progresses. Further, in many parallel applications the overall performance is determined by the slowest running subtask, so maintaining a good task allocation can significantly reduce the execution time. Finally, the ability to migrate the processes makes it possible to release the computational resources used by long-running programs without breaking the computations. This could be useful e.g. if the resources in question need to be brought down for maintenance.

The task migration component was built in our research group, based on the previous experiences with DynamicPVM [56, 137, 187].

Dynamite supports applications written for PVM 3.3.*x*, running under Solaris/UltraSPARC 2.5.1 and 2.6 and Linux/ia32 2.*x* kernels (libc 5 and GNU libc 2.0 binaries are supported, see Sec. 6.4 for information on support for latest library releases). From the user's perspective, it is quite transparent. All that is needed is to relink the application with the Dynamite's version of the PVM libraries and with the Dynamite dynamic loader. Moreover, the checkpointing mechanism can be used for non-PVM applications as well.

Parallel PVM applications consist of a number of processes (also called *tasks*) running on interconnected nodes forming a PVM *virtual machine.* A PVM daemon runs on every node and communicates with other daemons using the UDP/IP protocol. PVM tasks communicate with each other and with PVM daemons using a message-passing protocol. This protocol is reliable: no message can be lost, corrupted or duplicated, messages between two individual tasks arrive in the order sent.

In Dynamite, modified PVM daemons assist in task migration. The migration essentially consists of two parts: preserving the memory image of a running process and preserving the communication consistency.

Preserving the memory image of a process boils down to writing the process's address space to a file (checkpointing) and retrieving its contents on a different node (restoring it to memory). This includes the memory segments of the program itself (code, data, heap and stack) and those of the shared libraries used, to make the checkpoint file self-contained. In addition, the contents of the processor registers have to be taken care of. Extra care must be taken to ensure that the most important parts of the kernel-space state of a running process, such as the signal handlers or the open files, are properly preserved.

In Dynamite, the checkpointing and restoring support has been implemented in the *dynamic loader*, a low-level user-space component of a running UNIX process normally responsible for the loading of shared libraries. Such an implementation of the checkpointer makes it easy to preserve the shared libraries, to execute an initialisation code before the application starts running, and to wrap certain dynamically resolved function calls to catch the necessary kernel-space state. There is no need to modify or re-compile the user code, it only needs to be re-linked. Further, no administrative privileges are needed to install or use the checkpointer.

Preserving the communication, while not as low-level, is much more difficult to do right, because of a danger of race conditions occurring while the migration is being performed. Every PVM task has a socket connection (typically, TCP/IP) with the local PVM daemon. This connection is used for the *indirect routing*, i.e. when messages between tasks are routed through the PVM daemons. PVM tasks can also establish point-to-point *direct* TCP/IP communication channels with each other to improve the performance. Extra care must be taken when migrating PVM tasks to ensure that they do not permanently lose the connection with the rest of the parallel application, and that the PVM message protocol as outlined above is not violated.

In Dynamite, before a process is checkpointed, all direct connections are flushed and closed, and the task is disconnected from the local PVM daemon. Once a process is restored, it is reconnected to the local PVM daemon on the new node. Direct connections between tasks are re-established as soon as any new messages are to be sent. Modified PVM daemons maintain a routing table with the current location of migrated tasks (normally, the location would be encoded in the PVM task identifier). For the sake of efficiency, it is possible to migrate a task while it is in the middle of a communication operation. However, this leads to a number of difficult to solve race conditions. We had to add support for message forwarding between the routing processes and also had to extend message headers with sequencing information to ensure that messages "chasing" a migrating process will eventually catch up with it and that they will arrive in the right order. Partly established direct connections turned out to present a particular difficulty if a migration is to take place, because such a connection may already be fully initialised from one direction, but not from the other. We had to identify a number of critical sections where a migration must be deferred and we also had to fix some corner cases of the direct connection establishment protocol to make it recover gracefully if one of the tasks is migrated.

After this introduction, below we will present the most interesting extensions to Dynamite designed to extend its application domain.

6.3.1 Migrating sockets

Dynamite was limited to sequential applications or to parallel applications making use of PVM. In the course of his M.Sc. project, D. Żbik from the University of Mining and Metallurgy in Kraków, Poland, in a close cooperation with us, designed and developed *msocket* [29–32, 198], a library allowing any program that uses the generic TCP or UDP socket interface to be migrated without loosing its open socket connections. The library uses the Dynamite checkpointer to preserve the memory image of a migrating process, the rest of the system has been implemented from scratch.

Figure 6.2 presents an overview of the environment. In the figure, *Process C* has just been migrated from *Node 3* to *Node 4*. The connection redirection protocol is fairly complex, for the sake of brevity we will only identify its key components here. User data are transmitted over direct process-to-process links. *Daemons* play a managerial role during a migration, forcing the peers of the migrating process to redirect established connections to the new location. A *mirror* process is left at the original location to forward the on-the-fly user data to the new location. This process is terminated as soon as all such data have been forwarded. Migrating sockets are built on top of the ordinary system TCP and UDP stack, and use this stack for both the user-level communication and the internal library communication. To provide location independence, the library does not expose the physical addresses to the user code, making use of virtual addresses instead. *Address servers* are responsible for mapping the virtual addresses to the current physical locations of the processes.

6.3.2 Cross-cluster migration

Dynamite has originally assumed that the source and destination node shared a filesystem. This was necessary for the transfer of the checkpoint file, as shown in Fig. 6.3. The file was saved on the shared filesystem on the source node (*2. write*), and it was assumed that it was readily available under the same name on the destination node for restoring

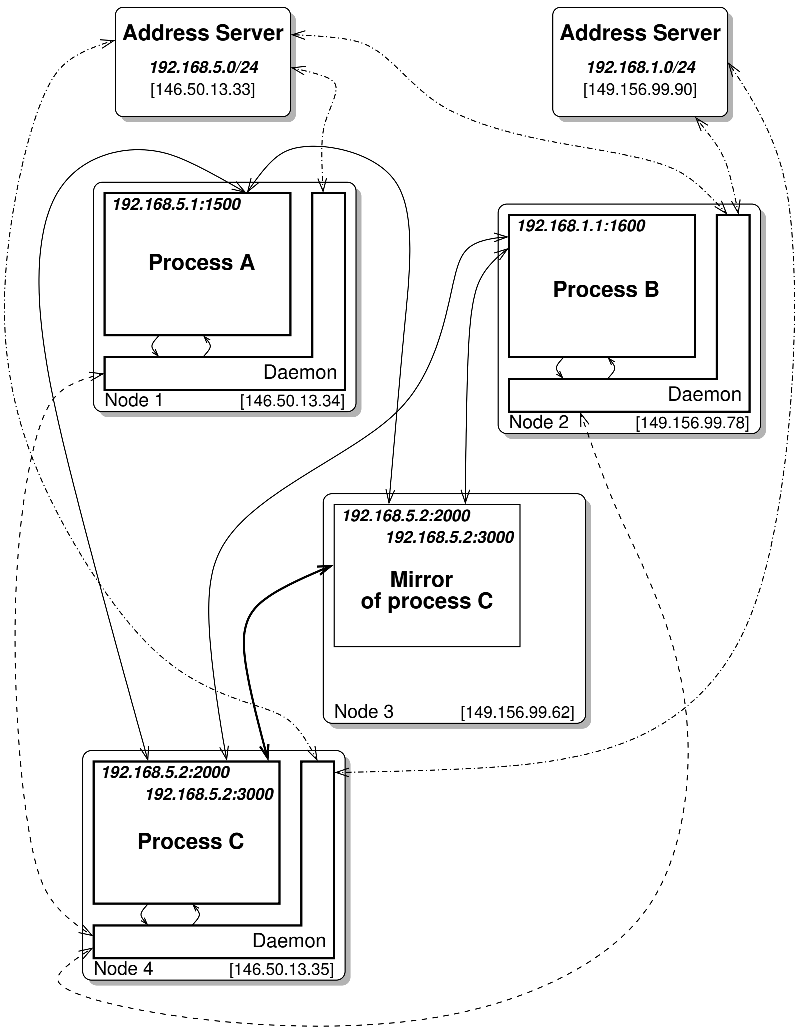


Figure 6.2: System overview of the migrating sockets. *Picture courtesy of Dariusz Żbik, UMM, Poland.*

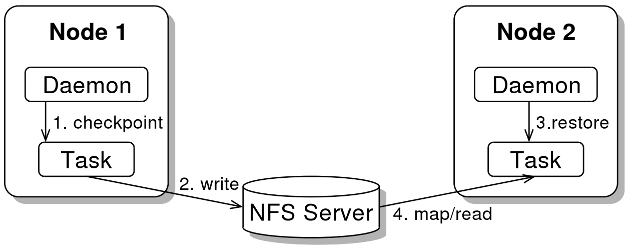


Figure 6.3: PVM task migration over a shared filesystem.

(*4. map/read*). Since individual nodes on a cluster or a network of workstations typically use a network filesystem for users' home directories, this was not a problem. However, this limited Dynamite's usability to single-cluster environments.

Under our direct supervision, in the course of her M.Sc. project, J. W. Wang [188] lifted this limitation. The modified protocol is shown in Fig. 6.4. A TCP link is established

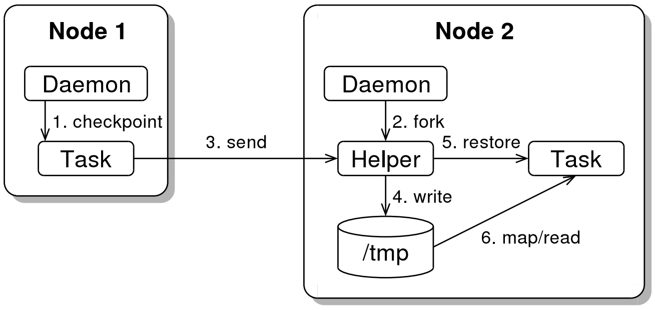


Figure 6.4: PVM task migration over a TCP link. *Picture based on the work of Jinghua Wang, UvA (now at C-LAB, Germany).*

between the checkpointing task on the source node and a helper task spawned by the PVM daemon on the destination node. The checkpoint data are transmitted over this link (*3. send*). For the sake of simplicity, the data are saved to a temporary file on a local disk (*4. write*), creating a checkpoint file that can then be restored as usual.

6.3.3 Cross-cluster I/O access

Another important step towards turning Dynamite into a full-featured multi-cluster environment was made under our direct supervision in the course of his M.Sc. project by A. Wibisono [191].

As mentioned above, it is important for a checkpointing mechanism to preserve the state of the open files used by the checkpointed process. Just like with the checkpoint file, Dynamite assumed that the open files would be readily available on the destination node under the same names.

This problem is far more difficult to solve than the assumption about the checkpoint file, because the access pattern to the checkpoint file is well known: it is a temporary file and it is always used in a read-only mode. In comparison, the access pattern to files opened from within the user code is unpredictable. For example, such files may be written to, making synchronisation in multi-cluster environments extremely difficult. Even if they are used solely for reading, they may be so large as to make a transfer of the complete file problematic.

In Dynamite, it has been decided to use the readily available Globus Access to Secondary Storage (GASS) module to provide the cross-cluster I/O support. GASS stores a copy of a remote file in the local cache and, if the file is modified, synchronises it with the remote location when the file is closed (an exception is made for append-only files, which are not cached, but synchronised immediately). This provides a reasonable compromise between the complexity and efficiency, and covers the most common access patterns. GASS provides replacement I/O initialisation and termination functions that, using the function wrapping facilities of Dynamite, can transparently replace the corresponding standard C and UNIX functions.

Figure 6.5 shows the protocols used for two sample scenarios. Initially (before a mi-

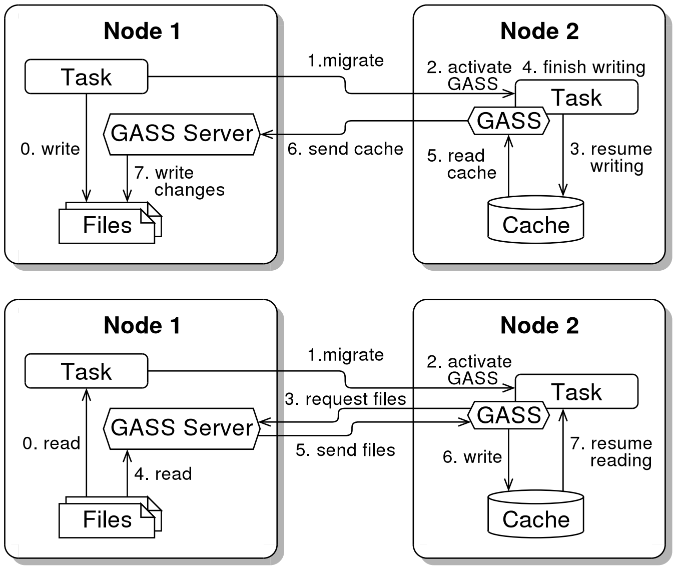


Figure 6.5: Protocols used for files open for writing (*top*) and reading (*bottom*). *Picture based on the work of Adianto Wibisono, UvA.*

gration takes place), the files are accessed directly, since they are available locally. After a migration, GASS needs to be activated. If a file has been opened exclusively for writing (Fig. 6.5 (*top*)), the local cache only needs to contain the data on the writing operations performed after the migration. Once the file is closed (e.g. because the application is finished or because the task using the file is migrated again), the local cache is sent to the GASS server so that the modifications made can be incorporated into the original file. If a file has been opened exclusively for reading (Fig. 6.5 (*bottom*)), it needs to be transferred to the local cache before the task can be resumed after the migration. There is no need to synchronise anything in this case, so once the file is no longer in use, the contents of the cache can simply be dropped.

6.4 Current efforts

Work on Polder-like environments is continuing in our research group on several fronts, to name only the already mentioned CrossGrid project. Here, we shall discuss the developments on the front of task migration, given our more direct involvement in this area.

Recently, T. Gubała has re-implemented the checkpointer. The old version was based on the Linux ELF dynamic loader release 1.9.9. This version was originally meant for the Linux/ia32 libc 5 libraries, and with some effort could also be made to work under Linux/ia32 with GNU libc 2.0, as well as with native libraries under Solaris/UltraSPARC. However, it did not work with any GNU libc releases starting with 2.1, seriously limiting its application on any contemporary machine.

The new checkpointer is based on the dynamic loader that comes with GNU libc 2.*x*. This ensures that it is compatible with the current GNU libc releases. However, the libraries turn out to be so tightly bound to the dynamic loader that most of the time the checkpointer must be based on the sources exactly matching the library version, down to the local modifications made by the distributor. Compared to the old checkpointer, the new one is considerably cleaner. Because it only needs to support Linux systems, it makes use of the `/proc` filesystem to obtain the information on memory mappings, open files, etc. It no longer needs to wrap many library calls, and the modifications to the dynamic loader sources are far more confined, significantly simplifying the maintenance.

In parallel, B. Ó Nualláin and D. Kaarsemaker have been working on extending the migration support to MPI applications. Eventually, it is planned to provide a service similar to that of DynamicPVM, so that single tasks can dynamically be migrated at run time. However, in the first phase, the work is focusing on the ability to consistently checkpoint parallel applications as a whole. This makes it possible to migrate a whole job to a different cluster in a fixed number of steps, or to provide fault tolerance by restarting a job from the checkpoint should a cluster node suddenly fail. Support for such operations is probably more immediately needed by the users of parallel applications. It should also be easier to implement reliably: since all of the parallel tasks have to be stopped anyway, many of the race conditions we have encountered in Dynamite with PVM will simply have no chance of occurring.

7

Conclusion

Dr. McCoy: '[If you] pick up enough speed, you're in time warp. If you don't, you're fried!'
Star Trek IV: The Voyage Home

7.1 Summary

In the work described in this thesis, we have taken a challenging high performance computing problem – an optimistic parallel discrete event simulation kernel, and ported it to the Grid – an emerging wide area distributed computing infrastructure. We wanted to find out how a latency-sensitive problem such as PDES would be affected by execution on wide area distributed resources. This would allow us to evaluate the fitness of the existing Grid components for high performance computing problems, and, if necessary, to design, implement and verify techniques to improve this fitness.

Chapter 1 provided an introduction to our work. There we discussed the importance of analytical modelling and computer simulation, and the relation between the two. We provided an introduction to the area of parallel discrete event simulation, introducing the synchronisation problem and two key approaches towards solving it: conservative and optimistic (Time Warp). Regarding the optimistic approach, which was the focus of this thesis, we introduced important concepts such as rollback and the distinction between local and global virtual time. We also briefly discussed the most popular contemporary high performance computing platforms: clusters and the emerging Grid.

Our first experiences with APSIS, a locally developed parallel simulation kernel that we used for our work, are described in Ch. 2. We briefly discussed the architecture of the kernel, the simulation models used (Ising spin and PHOLD) and the target machine used for the majority of the experiments. We also presented the first modifications we had made: generalising the process topology and making the event processing deterministic. The latter was made possible by implementing random number generators that roll back their state with the rest of the simulation, and by defining a strict processing order of simultaneous events. While evaluating the influence of these modifications on the run-time behaviour of the kernel, we observed a strange phenomenon: with the Ising spin simulation, introducing a rollback-aware random number generator very significantly reduced the maximum rollback length observed. We studied this issue further in Ch. 3. The dynamic run-time behaviour of the Ising spin simulation had been studied in our research group before, and the rollback length distribution had been found to adhere to the power law in the sub-critical model temperature range. Based on this, a conjecture had been made, that the Time Warp dynamics could be characterised as a self-organised critical system. Our later experiments cast some doubt on this claim. In particular, with a rollback-aware random number generator, the virtual time window size needed to be increased to get the expected rollback length distribution. Otherwise, the distribution was exponentially decaying. With the help of a few diagrams, we showed that the same window size can have different effects on the simulation, depending on the properties of the random number generator used. Further, we built two analytical models of the rollback behaviour for a Time Warp kernel running a simulation model such as the Ising spin. One model measured rollback length in the simulation time, the other in the number of events. In either case, the derived rollback length distribution was exponential, questioning the general validity of the power-law shape observed with APSIS. Because the models were strongly simplified and difficult to extend further, we built a discrete event simulation of a parallel simulation kernel, and used it to perform a further, more realistic study. The rollback length distributions obtained were again exponential. Finally, we found the rollback algorithm in the actual parallel simulation kernel to be the culprit, as it turned out to have a quadratic computational complexity. With an algorithm with linear complexity, the apparent power-law behaviour disappeared. Similarly, we obtained a power-law like bimodal shape with the discrete event simulation of the kernel when a rollback cost was increased from linear to quadratic. These experiments proved that the previously made speculations about the self-organised critical behaviour in a Time Warp kernel were inaccurate. This is because in a self-organised critical system, relaxations must be virtually instantaneous, whereas in Time Warp we get a power-law like behaviour only if rollbacks (that act as relaxations) have a high computational complexity.

The experiences with the parallel simulation kernel after extending it with multiple cancellation strategies are described in Ch. 4. Next to the already implemented aggressive cancellation, we added lazy cancellation and optimised aggressive cancellation using bulk anti-messages. Experiments performed on a single cluster were inconclusive: while the number of messages sent was significantly reduced, especially with lazy cancellation, the run time was virtually unaffected. Results obtained on the Grid, with two clusters communicating over a wide area network, were more interesting: up to a factor of two decrease in the run time could be observed with lazy cancellation. Still, there remained a difference of a factor of 3–4 in the performance between the single-cluster and the multi-cluster runs. Further experiments showed that a significant part of this slowdown could be attributed not to the inherent properties of the wide area network, but to a local overhead of querying TCP/IP sockets. This issue was investigated closely in Ch. 5. To avoid the expensive TCP/IP operations in the simulation processes, we shifted the wide area communication to newly introduced, dedicated routing processes. With a modified version of the communication library, multi-cluster experiments with the Ising spin simulation showed a significant performance improvement, especially with lazy cancellation and low communication. Aggressive cancellation with the Ising spin, as well as with the second simulation model used, PHOLD, proved more difficult. Because of excessive communication volume, the routing processes would become bottlenecks, destabilising the simulation kernel. Increasing the per-event computational grain, or introducing multiple routing processes per cluster, improved the stability. The introduction of message aggregation in the routing processes provided even larger performance and stability improvements. The performance of the majority of the Ising spin experiments was brought close to that of the single-cluster runs. A scalability study performed with PHOLD showed that with a sufficient number of processes and a high computational grain, the performance of multi-cluster experiments exceeded that of the single-cluster ones. We found the message aggregation with a small timeout to have a uniformly positive influence on the performance, as it significantly decreased the number of inter-cluster messages that needed to be sent. For simulations with a large state, such as the Ising spin, further improvement could be achieved by increasing the frequency of the global virtual time calculation. Because achieving an optimal frequency was difficult in multi-cluster runs, we focused on improving the algorithm instead, making it more efficient and moving it from a simulation process to the routing processes. This improved the scalability of the algorithm at high frequencies, but the overall performance improvement was not that impressive. Apparently, with the introduction of the changes described above we have collected most of the low-hanging fruit, so further improvements are more difficult.

Chapter 6 introduced the Polder Metacomputing Environment, an experimental environment for high performance computing and interactive simulation. We focused on the dynamic task migration component of it, called Dynamite, that continuously ensures the optimal task allocation of parallel applications that make use of the PVM library. We discussed the most interesting extensions to Dynamite: migrating sockets allow us to migrate processes that use the generic network socket interface, cross-cluster migration eliminates the need for a shared filesystem for the checkpoint file, and cross-cluster I/O access makes use of Grid services to implement remote I/O support.

7.2 Discussion

Integration of distributed resources is an important trend in contemporary computing. Given the traditional attention of the high performance computing community to efficiency, embracing the use of distributed resources is not an easy matter, considering the potential performance degradation. And yet, the HPC community does not want to stay behind. This is how the idea of the computational Grid was born. Major projects such as the Globus toolkit are under development to facilitate the interoperability of high performance computing resources and to enable HPC applications to run efficiently in distributed environments.

Our contribution to this effort was a move of a latency-sensitive parallel application – an optimistic Time Warp parallel discrete event simulation kernel – to a Grid environment. To the best of our knowledge, we were the first to do that. This thesis describes our experiences.

Have we met our objective? Definitely.

We have found that while the communication latency does negatively influence the performance of a PDES kernel on the Grid, the slowdown is nowhere near as bad as could have been feared. While an increase in run time by a factor of four compared to single-cluster experiments cannot of course be ignored, it is not a bad result given an increase in the latency by two *orders* of magnitude. What is more, the stability of the simulations running on the Grid was actually very good.

We have observed that some well known optimisation techniques that have only a small influence on the performance when running on a single cluster, such as lazy cancellation, can have a profound effect in a Grid environment. Thus, investing some effort into modifying the communication pattern can bring large gains in return.

Another action that generally helps is a separation of inter-cluster and intra-cluster communication, because it limits the number of processes that need to be concerned with expensive maintenance of wide area connections. Dedicated processes needed to implement this separation in user-space also create an additional level of infrastructure that can be used for other, domain-specific optimisations. We have used them for message aggregation and distributed global virtual time algorithms.

We can state that with these improvements in place, we have reached our goal, as the performance measurements made on the Grid have come close enough to those made on a single cluster to make Grid a viable computing platform for our parallel simulation kernel.

Naturally, there were also some downsides.

Our experiences have shown that the Grid components used were not best suited for high-performance applications "out of the box". In order to implement the separation of communication domains, we not only had to introduce additional user-level processes, but also had to make several low-level modifications to the communication library used.

Also, our optimisations, while mostly successful, have somewhat decreased the stability of the simulation kernel in the cases of intensive communication. What is more, the extra options added have increased the overal complexity of the system, making it (even) less accessible for a casual user.

But, in the end, we are very positive about our experiences. Taking a challenging HPC application such as a Time Warp kernel and distributing it using basically off-the-shelf components proved quite viable with a relatively small number of adjustments. Had we taken a less challenging application, or had we implemented a dedicated, highly tuned inter-cluster solution on our own, we would have probably achieved even better results. However, our conclusions would then lack generality. Having done things the way we did and still having achieved a successful outcome, we can with a greater confidence claim that high performance Grid computing is not a myth. It can be done, here and now.

Our approach makes a number of implicit assumptions. Foremost is the assumption about a multi-level hierarchy of communication latencies. In our research, we had access to a machine with three levels of communication latency: a base level within each SMP node, an order of magnitude higher latency between the nodes, and another two orders of magnitude higher value between two clusters (see Sec. 2.5). We strived to map the processes in the 3-D torus topology in such a way as to ensure that the processes running on the same node would be logical neighbours. However, this was a minor optimisation and further than that we made no disinction between the base (node) level and the second (cluster) level of the latency. On the other hand, the assumption about a large difference between the second (cluster) and the third (wide area) level of the latency was crucial. It was the primary motivation for introducing dedicated routing processes, so that simulation processes would not need to handle expensive wide area TCP/IP communication. Without a low-latency local area network, both intra-cluster and inter-cluster communication would be carried out using TCP/IP. Additional forwarding operations between the routing processes and the local simulation processes would not only become prohibitively expensive, but also mostly useless, since the simulation processes could then send the messages to other clusters directly. From the point of view of a simulation process, there is little difference in the cost of sending a message to a local or a remote cluster.

We should point out that the applicability of our findings is not limited to the Grid. As processors get faster, access time delays of any sort become more and more of a problem. This is the case not only in highly distributed environments, but also in popular computing architectures such as clusters. Even the highly complex and expensive multiprocessors are not immune to the problem – the existence of non-uniform memory access (NUMA) machines is clear evidence of that.

If current trends in the computing world continue (see e.g. Flynn and Dubey [63]), it is not unimaginable that a latency hierarchy of similar proportions to what we have encountered could become common within a single cluster in the not too distant future. It is enough if the processors in the currently common two-way SMP nodes are replaced by multi-core ones (i.e. ones having multiple independent processing units in a single case, typically sharing a common last-level cache), possibly featuring simultaneous multithreading on top of that. This will make each node appear as a four- or even an eight-way system. While each node will still be an SMP machine (the memory will be shared), the cache hierarchy will be non-uniform, making the communication between the processing units within one processor more efficient. A three-level hierarchy of communication latency within a cluster will thus be a fact. Allocating a single logical processing unit within each node for the inter-node communication, as we did on the cluster level, could thus become an advantage at that point, see e.g. Regnier et al. [152].

7.3 Future work

Naturally, this work could be carried on further. Several possible directions to take include:

- Load balancing: in our experiments, computational resources were distributed, but homogeneous. In a more generic Grid scenario, resources could be highly heterogeneous. To be more applicable in such cases, APSIS would need to be augmented with a load balancing subsystem. The viability of using Dynamite for this purpose should be investigated. The scheduler would certainly need modifications, since it currently has no notion of applications that make use of speculative computing. Without any feedback from the simulation kernel, the scheduler would be unable to determine which processes perform useful work and which constantly roll back.

- Fault tolerance: further Grid integration would need to include a scheme allowing the simulation kernel to continue even if some of the resources suddenly become unavailable. Again, suitability of Dynamite for this purpose should be investigated.
- Throttling algorithm: in spite of all efforts, if the communication is very intensive, the stability of the simulation kernel with the routing processes remains lower than without them. The stabilisation mechanism currently in place behaves like a safety valve: it only activates if the communication layer is already overwhelmed, and works by completely blocking a simulation process until the communication layer catches up. A more efficient throttling mechanism would instead gradually slow the process down, attempting to prevent it from ever overwhelming the communication layer. We have experimented with a mechanism that, if rollback thrashing is detected, does not send newly scheduled external events, but keeps them at the source process for a short period. This way, if an event needs to be cancelled, the operation will take place at the source, and thus will not involve the communication layer at all. However, the experiments were inconclusive; further research would be required.
- Inter-cluster communication: MPICH-G2 is currently used for this purpose. While adequate, this solution did show its shortcomings a few times. We had to tune the receiving routines to bring the performance closer to what we had expected. Attempts to increase the TCP window size, using a documented API, resulted in crashes. With very intensive communication, we observed a number of times that the processes would mysteriously hang inside of MPICH functions as if some messages were lost. It would be interesting to see what sort of performance and stability improvements could be achieved with a dedicated solution implemented from scratch, possibly using a different communication protocol such as UDP/IP.
- Optimistic vs. conservative: in our simulation kernel, we uniformly use optimistic Time Warp protocol across all links, simply because this approach was the most straightforward. However, it is not at all obvious that this is the best strategy: maybe a conservative protocol should be used between the clusters?
- Larger latency: the wide area network used in our experiments has a latency of around 1.4 ms. Not much, but that is what one can realistically expect over fast links in a small country like the Netherlands. It could be interesting to see to what extent the performance would be affected over a larger distance, on a link with latency measured in tens of milliseconds.

A

History of APSIS

Over the course of our research, we have made a number of improvements to the parallel discrete event simulation kernel APSIS. These improvements have been incrementally described throughout the thesis. However, in some cases it might not be immediately obvious what particular version of APSIS was used to conduct some of the experiments discussed. For this reason, we have put all the major developments together in Tab. A.1 (see next page). From top to bottom, the table incrementally describes the changes made, so later versions include the features of earlier ones. To clarify, both versions from 2001 were used on the old DAS machine (see Sec. 6.1), and all the subsequent versions of APSIS were run on DAS-2. For each version, we have put information on the section(s) of the thesis where the new features are discussed, and we have provided the references to figures showing the results of the experiments conducted with this version.

Table A.1: A summary of the development work on the APSIS simulation kernel carried out within the scope of this thesis.

Date	Key features	Applicable sections of the thesis	Figures based on experiments conducted using this version
2001–01	• original version from B. J. Overeinder • running on DAS machine	2.3	3.1
2001–06	• generalised topology • state-saving RNG • deterministic event ordering	2.6, 2.7	2.9, 2.10, 3.3, 3.13
2002–10	• support for MPICH-GM and G2 • running on DAS-2 machine	2.5	3.2, 3.17
2003–03	• linear rollback cost	3.6.3	3.17
2003–05	• lazy cancellation • bulk anti-messages	4.2, 4.3	4.7, 4.9, 4.10, 4.11, 4.12, 4.13
2003–12	• routing processes • message aggregation	5.3, 5.4	5.3, 5.8, 5.10, 5.11
2004–03	• multiple routing processes per cluster • tunable GVT frequency • GVT inter-arrival based throttling	5.3.3, 5.5.1	5.12, 5.14
2004–04	• GVT efficiency improvements	5.5.2	5.14
2004–10	• hierarchical GVT algorithms	5.5.3	5.4, 5.6, 5.9, 5.17

Nederlandse samenvatting

Het werk beschreven in dit proefschrift betreft een veeleisend *high performance computing* vraagstuk – een programma (*kernel*) voor optimistische parallelle simulatie van discrete gebeurtenissen (*PDES*), dat geschikt is gemaakt voor het Grid – een *wide area* gedistribueerde computer infrastructuur van snel groeiend belang. Het doel was om uit te zoeken hoe een vraagstuk dat gevoelig is voor communicatie-latentie, zoals PDES, wordt beïnvloed door het gebruik van gedistribueerde hulpbronnen. Dit maakt het voor ons mogelijk om de geschiktheid van bestaande Grid componenten voor high performance computing te evalueren en indien nodig technieken te ontwerpen, te implementeren en te testen om de geschiktheid ervan te verbeteren.

Hoofdstuk 1 geeft een introductie op ons werk. Daar hebben we het belang van analytisch modelleren besproken, van computer simulaties en de relatie tussen beide. We hebben een introductie op het gebied van parallelle simulatie van discrete gebeurtenissen gegeven en we hebben het synchronisatieprobleem en de twee belangrijkste methoden om dat op te lossen besproken: de conservatieve en de optimistische aanpak (Time Warp). Wat betreft de optimistische aanpak, die de kern van het in dit proefschrift beschreven onderzoek vormt, hebben we belangrijke begrippen geïntroduceerd, zoals een *rollback* en het onderscheid tussen een lokale en globale virtuele tijd. We hebben ook kort de meest algemene high performance computing platforms besproken: clusters en het steeds belangrijker Grid.

Onze eerste ervaringen met APSIS, een parallelle simulatie kernel die lokaal is ontwikkeld en die wij voor ons werk hebben gebruikt, zijn beschreven in hoofdstuk 2. We hebben de architectuur van de kernel kort besproken, alsmede de gebruikte simulatie modellen (Ising spin en PHOLD) en de machine die gebruikt is voor de meeste experimenten. Ook hebben we hier de eerste aanpassingen gepresenteerd die door ons gemaakt zijn: de generalisatie van de procestopologie en de introductie van het deterministisch verwerken van gebeurtenissen. Het laatste werd mogelijk door de implementatie van generatoren voor pseudo-random getallen die bij rollbacks hun toestand herstellen samen met de rest van de simulatie en door het definiëren van een strikte verwerkingsorde van samenvallende gebeurtenissen. Tijdens een onderzoek van de invloed van deze aanpassingen op het *run-time* gedrag van de kernel trad een opmerkelijk fenomeen op: bij de Ising spin simulatie beperkt de introductie van een rollback-bewuste random generator de maximale optredende rollback lengte significant. We hebben deze kwestie in hoofdstuk 3 nader bestudeerd. Het dynamische run-time gedrag van de Ising spin simulatie was in onze onderzoeksgroep al eerder bestudeerd, waarbij was gevonden dat de distributie van de rollback lengte een *power-law* verdeling leek te volgen in het subkritische temperatuurbereik van het model. Hierom werd aangenomen dat Time Warp zich als een zichzelf organiserend kritisch systeem (*SOC*) gedroeg. Onze latere experimenten hebben vraagtekens geplaatst bij deze aanname. In het bijzonder moest in het geval van een rollback-bewuste random generator het virtueel tijd venster worden vergroot om de verwachte verdeling van de rollback lengte opnieuw te verkrijgen, anders was de verdeling exponentieel dalend. Met behulp van een aantal diagrammen hebben we laten zien dat dezelfde venstergrootte verschillende effecten op een simulatie kan hebben, afhankelijk van de eigenschappen van de gebruikte random generator. Verder hebben we twee analytische modellen gebouwd voor het rollback-gedrag van een Time Warp kernel die een simulatie draait zoals bijvoorbeeld het Ising spin model. Eén model mat een rollback lengte in termen van de simulatie tijd, de andere in termen van het aantal gebeurtenissen. In beide gevallen was de afgeleide distributie van de rollback lengte exponentieel, daarmee de algemene geldigheid van de vorm van de power-law zichtbaar met APSIS in twijfel trekkend. Omdat de modellen sterk vereenvoudigd waren en verder moeilijk uitbreidbaar, hebben we een discrete-event simulatie van een parallelle simulatie kernel gebouwd om een verdere, meer realistische studie te verrichten. Ook deze keer waren de distributies van de rollback lengte exponentieel. Uiteindelijk werd gevonden dat het rollback algoritme in de echte parallelle simulatie kernel hiervoor verantwoordelijk was, omdat deze een kwadratische computationele complexiteit bleek te hebben. Bij een algoritme met lineaire complexiteit is het schijnbare power-law gedrag verdwenen. Ook hebben we een power-law-achtig bimodale vorm verkregen van de simulatie van de kernel toen de kosten van een rollback van lineair naar kwadratisch werden verhoogd. Deze experimenten toonden aan dat de eerder gemaakte speculaties over het zichzelf organiserend kritisch gedrag in een Time Warp kernel voorbarig waren. Dit omdat in een zichzelf organiserend kritisch systeem de relaxaties vrijwel instantaan moeten zijn, terwijl we in Time Warp een power-law-achtig gedrag krijgen alleen als de rollbacks (die als relaxaties optreden) een hoge computationele complexiteit hebben.

Hoofdstuk 4 beschrijft de ervaringen met een versie van de APSIS kernel na uitbreiding met meerdere *cancellation* strategieën. Naast de reeds eerder geïmplementeerde *aggressive cancellation* hebben we *lazy cancellation* en een geoptimaliseerde aggressive cancellation met *bulk anti-messages* toegevoegd. Experimenten op een enkelvoudig cluster gaven geen verbetering in de prestatie te zien: terwijl het aantal verstuurde berichten aanzienlijk was verkleind, vooral met lazy cancellation, bleef de doorlooptijd vrijwel onveranderd. Interessanter waren de resultaten die verkregen werden op het Grid, met twee clusters die communiceren over een wide area netwerk: een vermindering in de doorlooptijd met een factor twee was zichtbaar voor lazy cancellation. Toch bleef er een verschil van een factor 3–4 in prestatie tussen de uitvoeringen op een enkelvoudig cluster en op meerdere clusters. Uit verdere experimenten bleek dat een aanzienlijk deel van deze vertraging niet toegeschreven kon worden aan inherente eigenschappen van het wide area netwerk, maar aan een lokale overhead veroorzaakt door het gebruik van TCP/IP sockets. Deze kwestie werd nader onderzocht in hoofdstuk 5. Om dure TCP/IP operaties in de simulatie processen te voorkomen, hebben we nieuwe routeringsprocessen geïntroduceerd die uitsluitend de wide area communicatie verzorgen. Met een aangepaste versie van de communicatie bibliotheek lieten experimenten van de Ising spin simulatie op meerdere clusters een aanzienlijke verbetering in prestatie zien, vooral met lazy cancellation en weinig communicatie. Aggressive cancellation bleek moeilijker, zowel met de Ising spin als met PHOLD, het tweede gebruikte simulatie model. Door de excessieve hoeveelheid communicatie zijn de routeringsprocessen knelpunten geworden, die de simulatie kernel instabiel maakten. Door verhoging van de computationele granulariteit of de introductie van meerdere routeringsprocessen per cluster wordt de stabiliteit verbeterd. Introductie van berichtaggregatie in de routeringsprocessen heeft nog grotere prestatie- en stabiliteitsverbeteringen opgeleverd. De prestatie van de meerderheid van de Ising spin experimenten is in de buurt van uitvoeringen op enkelvoudig cluster gekomen. Een schaalbaarheidsstudie uitgevoerd met PHOLD liet zien dat met een voldoende aantal processen en een hoge computationele granulariteit de prestatie van experimenten op meervoudige clusters die van enkelvoudige kon overtreffen. De berichtaggregatie met een kleine *timeout* bleek steeds een positieve invloed op de prestatie te hebben, omdat het aantal inter-cluster berichten dat verstuurd moest worden aanzienlijk werd verkleind. Voor simulaties met een omvangrijke toestand, zoals de Ising spin, konden verdere verbeteringen worden bereikt door de verhoging van de frequentie van de berekening van de globale virtuele tijd. Omdat het bereiken van een optimale frequentie van uitvoeringen op meerdere clusters moeilijk was, hebben we ons in plaats daarvan op het verbeteren van het algoritme toegespitst, door het efficiënter te maken en door het te verplaatsen van simulatie- naar routeringsprocessen. Dit heeft de stabiliteit van het algoritme bij hoge frequenties aanzienlijk verbeterd, maar de globale verbetering van de prestatie was minder indrukwekkend. Blijkbaar hebben we met de hierboven beschreven aanpassingen de meest voor de hand liggende dingen al opgepikt, en dus zijn de verdere aanpassingen moeilijker.

Hoofdstuk 6 staat enigszins op zichzelf en introduceert de *Polder Metacomputing Environment*, een experimentele omgeving voor high performance rekenen en interactieve simulaties. We hebben hier met name de component beschreven die verantwoordelijk is voor dynamische taak migratie, genaamd Dynamite. Deze component zorgt doorlopend voor een optimale taakallocatie van parallelle applicaties die gebruik maken van de PVM bibliotheek. We hebben de meest interessante uitbreidingen op Dynamite besproken: *migrating sockets* maken het mogelijk om processen te migreren die de generieke socket interface gebruiken, cross-cluster migratie neemt een eis van een gedeeld filesysteem voor de *checkpoint* file weg, en cross-cluster I/O toegang maakt gebruik van Grid services die *remote* I/O ondersteuning bieden.

Streszczenie po polsku

Przedmiotem badań opisanych w tej pracy jest złożony problem z dziedziny obliczeń dużej wydajności, a mianowicie opracowanie jądra do równoległej symulacji metodą kolejnych zdarzeń (*PDES*) i przeniesienie go na Grid – wyłaniającą się infrastrukturę do obliczeń rozproszonych. Celem pracy było zbadanie, jak problem wrażliwy na opóźnienia w transmisji, taki jak PDES, zachowa się w przypadku uruchomienia go na zasobach rozproszonych geograficznie. Pozwoliłoby to oszacować przydatność istniejących komponentów gridowych do zastosowań z dziedziny obliczeń dużej wydajności oraz, o ile okazałoby się to potrzebne, można byłoby zaprojektować, zaimplementować i przetestować techniki służące poprawie tej przydatności.

W rozdziale 1 przedstawiono wprowadzenie do pracy. Przedyskutowano tam wagę modelowania analitycznego i symulacji komputerowej oraz występujące pomiędzy nimi powiązania. Przedstawiono wstęp do równoległej symulacji metodą kolejnych zdarzeń, omawiając problem synchronizacji oraz dwa podstawowe podejścia do jego rozwiązania: konserwatywne i optymistyczne (Time Warp). W odniesieniu do podejścia optymistycznego, na którym skupiła się ta praca, przedstawiono ważne pojęcia, takie jak *rollback* oraz rozróżnienie pomiędzy wirtualnym czasem lokalnym a globalnym. Zostały też omówione najpopularniejsze współczesne platformy do obliczeń dużej wydajności: klastry i wyłaniające się systemy gridowe.

W rozdziale 2 opisano pierwsze doświadczenia z jądrem do symulacji równoległej, zwanym APSIS, opracowanym w naszej grupie badawczej i użytym w przedstawionych tutaj badaniach. Przedyskutowano w skrócie architekturę jądra, modele wykorzystane do symulacji (model Isinga i PHOLD) oraz opisano sprzęt wykorzystany do większości eksperymentów. Przedstawiono także pierwsze z wprowadzonych modyfikacji jądra: uogólnienie topologii procesów oraz deterministyczne przetwarzanie zdarzeń. To ostatnie stało się możliwe poprzez zaimplementowanie generatorów liczb losowych, które przywracają swój stan razem z resztą symulacji, oraz poprzez zdefiniowanie ścisłego porządku przetwarzania zdarzeń jednoczesnych. Podczas mierzenia wpływu tych modyfikacji na zachowanie jądra zauważono nieoczekiwane zjawisko: w przypadku symulacji modelu Isinga, wprowadzenie generatora liczb losowych przywracającego swój stan podczas rollbacków bardzo znacznie zredukowało maksymalną długość obserwowanych rollbacków. Zagadnienie to przestudiowano dogłębniej w rozdziale 3. Dynamiczne zachowanie symulacji modelu Isinga było już wcześniej studiowane w naszej grupie badawczej, kiedy to stwierdzono, że rozkład prawdopodobieństwa długości rollbacków w podkrytycznym przedziale temperatur modelu jest zgodny z prawem potęgowym. Bazując na tym domniemywano, że dynamikę jądra Time Warpa można scharakteryzować jako samoorganizujący się system krytyczny. Przedstawione w obecnej pracy eksperymenty naruszyły tę teorię. W szczególności, przy generatorze liczb losowych przywracającym swój stan podczas rollbacków, aby otrzymać spodziewany rozkład długości rollbacków, należało powiększyć rozmiar okna czasu wirtualnego. W przeciwnym wypadku rozkład zanikał wykładniczo. Na kilku diagramach pokazano, że ten sam rozmiar okna może różnie wpływać na symulację, w zależności od własności użytego generatora liczb losowych. Zbudowano także dwa modele analityczne zachowania rollbacków w jądrze Time Warpa wykonującym symulację taką jak model Isinga. W pierwszym modelu długość rollbacków była mierzona w jednostkach symulowanego czasu, w drugim w liczbie zdarzeń. W obu przypadkach wyliczony rozkład długości rollbacków był wykładniczy, poddając w wątpliwość ogólną poprawność prawa potęgowego obserwowanego w APSIS-ie. Ponieważ modele były poważnie uproszczone, a ich dalsza rozbudowa była trudna, zbudowano symulację (metodą kolejnych zdarzeń) jądra do symulacji równoległej i użyto jej do przeprowadzenia dalszych, bardziej realistycznych badań. Także i tym razem otrzymany rozkład długości rollbacków był wykładniczy. W końcu odkryto, że przyczyną był algorytm rollbacków w prawdziwym jądrze do symulacji równoległej, ponieważ okazało się, ze miał on kwadratową złożoność obliczeniową. Przy algorytmie ze złożonością liniową domniemane zachowanie potęgowe zniknęło. Podobnie, przy użyciu symulacji jądra otrzymano kształt bimodalny przypominający potęgowy, kiedy złożoność obliczeniowa rollbacków została podwyższona z liniowej na kwadratową. Te eksperymenty wykazały, że wszeœniej poczynione spekulacje co do samoorganizującego się krytycznego zachowania jądra Time Warpa były nietrafne. Wynika to z tego, że w samoorganizującym się krytycznym systemie relaksacje muszą być praktycznie natychmiastowe, podczas gdy w Time Warpie zaobserwowano zachowanie potęgowe tylko gdy rollbacki (które pełnią rolę relaksacji) mają wysoką złożoność obliczeniową.

Doświadczenia z jądrem do symulacji równoległej po jego rozszerzeniu o różne strategie anulowania (ang. *cancellation strategies*) zostały opisane w rozdziale 4. Obok zaimplementowanego już wcześniej anulowania agresywnego (ang. *aggressive cancellation*), dodano anulowanie odroczone (ang. *lazy cancellation*) oraz zoptymalizowane anulowanie agresywne z wiadomościami anulującymi masowo (ang. *bulk anti-messages*). Eksperymenty przeprowadzone na pojedynczym klastrze nie przyniosły rozstrzygnięcia: choć liczba wysłanych wiadomości znacząco się zmniejszyła, szczególnie w przypadku anulowania odroczonego, czas wykonania pozostał praktycznie taki sam. Rezultaty uzyskane na systemie gridowym, z użyciem dwóch klastrów komunikujących się poprzez sieć rozległą, były bardziej interesujące: w przypadku anulowania odroczonego czas wykonania uległ zmniejszeniu do dwóch razy. Niemniej jednak pomiędzy eksperymentami na pojedynczym klastrze i na wielu klastrach pozostała trzy–czterokrotna różnica w wydajności. Dalsze eksperymenty wykazały, że znacząca część tego spowolnienia może być przypisana nie nieodłącznym cechom sieci rozległych, ale lokalnemu narzutowi związanemu ze sprawdzaniem stanu gniazd TCP/IP. Sprawa ta została wnikliwie przebadana w rozdziale 5. Aby uniknąć kosztownych operacji TCP/IP w procesach wykonujących symulację, obsługę komunikacji rozległej przeniesiono do nowo utworzonych, dedykowanych procesów rozsyłających (ang. *routing processes*). Eksperymenty przeprowadzone przy użyciu symulacji modelu Isinga na wielu klastrach z wykorzystaniem zmodyfikowanej wersji biblioteki komunikacyjnej wykazały znaczącą poprawę wydajności, szczególnie w przypadku anulowania odroczonego i przy niskiej komunikacji. Anulowanie agresywne w połączeniu z modelem Isinga, a także z drugim używanym do symulacji modelem, PHOLD, było trudniejsze do realizacji. Z powodu nadmiernej komunikacji, procesy rozsyłające stały sie wąskim gardłem, destabilizując jądro. Stabilność poprawiła się po zwiększeniu ziarnistości obliczeniowej lub wprowadzeniu wielu procesów rozsyłających na każdym klastrze. Wprowadzenie mechanizmu agregacji wiadomości w procesach rozsyłających doprowadziło do jeszcze większej poprawy wydajności i stabilności. Wydajność większości eksperymentów przy użyciu modelu Isinga została doprowadzona do poziomu bliskiego eksperymentom na pojedynczym klastrze. Studium skalowalności przeprowadzone przy użyciu modelu PHOLD wykazało, że przy odpowiednio dużej liczbie procesów i wysokiej ziarnistości obliczeniowej wydajność eksperymentów wieloklastrowych może przewyższyć wydajność na pojedynczym klastrze. Stwierdzono, że agregacja wiadomości z niewielkim limitem czasu ma generalnie pozytywny wpływ na wydajność, ponieważ znacząco zmniejsza liczbę wiadomości, które muszą zostać przesłane pomiędzy klastrami. W przypadku symulacji charakteryzujących się stanem o znacznym rozmiarze, takich jak symulacja modelu Isinga, dalszą poprawę udało się osiągnąć poprzez zwiększenie częstotliwości obliczania globalnego czasu wirtualnego. Ponieważ uzyskanie optymalnej częstotliwości w eksperymentach wieloklastrowych okazało się trudne, zamiast tego skupiono się na usprawnieniu algorytmu, czyniąc go bardziej efektywnym i przenosząc go z procesu wykonującego symulację do procesów rozsyłających. Polepszyło to znacząco skalowalność algorytmu przy wysokich częstotliwościach, ale ogólna poprawa wydajności nie była aż tak imponująca. Wygląda na to, że wprowadzając opisane powyżej ulepszenia wyczerpano większość znajdujących się w zasięgu ręki możliwości, tak więc dalsze usprawnienia są trudniejsze.

W rozdziale 6 omówiono *Polder Metacomputing Environment*, eksperymentalne środowisko do obliczeń dużej wydajności oraz symulacji interaktywnych. Skupiono się na komponencie do dynamicznej migracji zadań, o nazwie Dynamite, który zapewnia optymalny przydział zadań aplikacji równoległych używających biblioteki PVM. Przedyskutowano najbardziej interesujące rozszerzenia Dynamite'a: *migrating sockets* umożliwiają migrację procesów używających standardowego interfejsu gniazd sieciowych, migracja *cross-cluster* neutralizuje potrzebę posiadania współdzielonego systemu plików dla pliku *checkpoint*, a *cross-cluster I/O* używa serwisów gridowych do implementacji zdalnego I/O.

Acknowledgements

While a Ph.D. research as such is an inherently solitary activity, it would've been much harder and less fun if it wasn't for all the people around offering their help and providing support.

First and foremost, I express my sincere thanks to my thesis supervisor, Peter Sloot. We first met in person in my home town, Cracow in Poland, where Peter came to attend a scientific conference nearby, while I was stuck waiting for my Dutch visa with a work permit to get through. I think he might still remember :-). While he is a very busy man, he has always been there when I really needed his assistance. I appreciate his trust in me, his enthusiasm and guidance. Coming from a country where funding *does* tend to be a problem, I can't fail to appreciate his ability to shield me from such harsh realities of life. And of course I won't forget his at first worrying ability to ask relevant questions after *any* presentation :-).

Then, I would like to thank my thesis co-supervisor, Dick van Albada. On the day I arrived in the Netherlands, he was the one waiting for me outside the Duivendrecht station (a "middle of nowhere" according to a friendly person on the platform questioning my intention of heading to the nearest exit :-). Dick is the sort of person who seems to know *everything*, or at the very least *something* about everything. It's been a very rewarding experience having him as my daily supervisor. With his sharp, questioning mind and attention to detail, his contribution to my work has been invaluable. Also, I think he might have actually read this thesis more times than I did :-).

Special thanks also go to Zeger Hendrikse. We've shared an office at UvA for over four years while he was still working there, and gosh, that was fun! :-) At some point, he even proclaimed himself my *mental coach*, relaying the wisdoms he had collected during his own Ph.D. research, providing some healthy scepticism, but also supporting me when needed. Being the closest person I knew that had a car in Amsterdam, he has helped me to move at least twice, and I think he even once drove me to Schiphol when NS (the

Dutch Railways) were on strike. His help was also invaluable for me in the understanding of Dutch customs and culture. His patience as a conversation partner helped me to put into practice what I had learnt during the Dutch language courses. He also corrected countless language errors in the Dutch version of the thesis summary.

I would like to thank all colleagues from the Section Computational Science of UvA. My current office-mates Alfredo and Breanndán are great (especially since they are often not in, and then I have the whole office room for myself :-), and with Michael Sipior across the corridor, we all have plenty of fun both at work and afterwards. Special thanks to Michael for proofreading parts of this thesis. Many thanks to the regulars forming the "12:30 lunch group": Alessia, Lilit, Zhiming, Roman, Michael Scarpa, Yves, and Jordi. I would like to thank Rob, Elena, Simon, and Denis, and Jiangjun, Maxim, and Evghenii. Special thanks to the remaining staff members: Alfons Hoekstra, Jaap Kaandorp, and Walter Hoffmann. Thank you, those of you working here part-time: David, Drona, Mark, Dennis, Derek, Bas, and Berry, and colleagues working abroad: Lera, Kasia, and Tomek. I would like to thank the past members of the group, especially my old office-mates Artoli, Judhi, Ronald, and Edwin, and Piero, Roeland, David Dubbeldam, Benno, Martin, and Arjen.

I would like to thank the colleagues from other groups of the Informatics Institute of UvA that I had various contacts with: Frank, Andy, Bas, Marcel, Wojtek, Dmitry, Vladimir, Adam, Adianto, Ersin, Victor, Hakan, Hans, Simon, Jelle, and Matthijs.

Special thanks to Frank van der Linden and Benno Overeinder, the "forefathers" of the two projects I've been involved in, respectively Dynamite and APSIS. Helping two students, Jinghua Wang and Adianto Wibisono, complete their Dynamite-related Master projects was a very pleasant experience and I thank them for that.

How would we all manage without the secretaries? These days they are so distributed that I don't even know the names of some of them. Thank you all, and especially Erik, Coco, Jacqueline, Marianne, Virginie, and Saskia, and the administrative manager Richard. Special thanks to the people from the personnel department, especially Anny, Elvire, Marianne, and Willem, for helping me fight the never-ending battle with the immigration services.

ICT department plays a vital role in an institute like ours. Given the inherently background nature of their work, I don't know many of our administrators personally. Still, thank you all, and especially Jan (always willing to drop by and exchange the latest gossip), Ruud, Frans, and Stephan, and their bosses Gert, Jeroen, and Bert (always thank the bosses :-). They sure like to run their infrastructure, shall we say, "according to the best industry standards." Luckily, I never made the fatal mistake of requesting the workstation on my desk to be taken out of their control. As an innocent bystander I could thus fully enjoy watching others fight an uphill battle reminding me of my favourite *Bastard Operator From Hell* series :-).

Special thanks to the maintainers of the computing facilities I used for my research, in particular the DAS-2 machine. I can't praise enough the dedication and patience of Kees Verstoep, and the promptness of his reactions to reported problems. I would like to thanks David Groep, the Dutch Grid expert, and other DAS-2 administrators, in particular Djuhaeri Harapan, the administrator of the UvA cluster at NIKHEF.

I acknowledge the efforts of the people who agreed to serve as members of my Ph.D.

committee. Thank you, Maarten Boasson, Marian Bubak, Rob Meijer, Benno Overeinder, and Stephen Turner for the time you have spent reading the manuscript, for your suggestions, and for your presence during the defence itself (you *will* be there, right? :-).

Special thanks to the Polish friends I've met in Amsterdam. Without you this would have been some boring years! Thanks for the discussions, trips, sport activities, games, meals, drinks, and movies we enjoyed together (did I forget about something? :-). Thank you, Agnieszka Szynol, Marcin, and Agnieszka Kubacka, and those who no longer live here: Kasia, Maciek, Ania, and Tomek.

I would like to thank other fellow-foreigners I've met here, in particular the seemingly always cheerful and enthusiastic Heidi, and my present and past neighbours.

Special thanks to my fellow fencers from the fencing club Esprit, especially Stephen, Sietske, Oliver, Jacco, Micha, Josje, and Filip, and of course our maître Paul van Polanen Petel. I took the introduction course in fencing mainly because it was too late then to subscribe for ice skating. In over three years that have passed, I didn't regret it for a moment (but in the meantime I did learn to ice skate as well :-).

Many thanks to my friends from Poland: Michał and Karolina, Darek and Dorota (and their son Bartosz). When I was with a visit to Poland, they always tried to find time to meet me. I would like to thank my friends from the studying days, especially Piotrek, Radek, and Paweł, as well as Daniel, a Brazilian with a Polish heart.

Last but not least, I express my deepest thanks to my family: my parents, my older brother Marcin, sister-in-law Bożena, and niece Kasia, and my younger brother Wojtek. During the past years when I have been living in Amsterdam, my parents have always been willing to listen to what I had to say during our weekly phone conversations, and I was always welcome to visit them. My younger brother would even make my old room (now *his* kingdom) available to me for the duration of my stay – I have no idea why (I probably wouldn't do it if I was him :-).

These acknowledgements refer to 116 people, if my count is correct. Being only a human, I must have forgotten somebody. My sincerest apologies for that.

Once again, thank you all!

Kamil Iskra
Amsterdam, April 2005

Publications

[1] K. A. Iskra, Z. W. Hendrikse, G. D. van Albada, B. J. Overeinder, and P. M. A. Sloot. Experiments with Migration of PVM Tasks. In *Proceedings of the ISThmus 2000, Research and Development for the Information Society*, pp. 295–303, Poznań, Poland, Apr. 2000.

[2] K. A. Iskra, Z. W. Hendrikse, G. D. van Albada, B. J. Overeinder, and P. M. A. Sloot. Dynamic Migration of PVM Tasks. In *Proceedings of the 6th Annual Conference of the Advanced School for Computing and Imaging*, pp. 206–212, Lommel, Belgium, June 2000.

[3] K. A. Iskra, F. van der Linden, Z. W. Hendrikse, B. J. Overeinder, G. D. van Albada, and P. M. A. Sloot. The Implementation of Dynamite – an Environment for Migrating PVM Tasks. *ACM Operating Systems Review*, vol. 34, no. 3, pp. 40–55, July 2000.

[4] K. A. Iskra, Z. W. Hendrikse, G. D. van Albada, B. J. Overeinder, and P. M. A. Sloot. Performance Measurements on Dynamite/DPVM. In *Proceedings of the 7th European PVM/MPI User's Group Meeting*, vol. 1908 of *Springer Lecture Notes in Computer Science*, pp. 27–38, Lake Balaton, Hungary, Sept. 2000.

[5] K. A. Iskra, Z. W. Hendrikse, G. D. van Albada, B. J. Overeinder, P. M. A. Sloot, and J. Gehring. Experiments with Migration of Message Passing Tasks. In *Proceedings of the 1st IEEE/ACM International Workshop on Grid Computing*, vol. 1971 of *Springer Lecture Notes in Computer Science*, pp. 203–213, Bangalore, India, Dec. 2000.

[6] M. Bubak, W. Funika, D. Żbik, G. D. van Albada, K. A. Iskra, P. M. A. Sloot, R. Wismüller, and K. Sowa-Piekło. Performance Measurement, Debugging and Load Balancing for Metacomputing. In *Proceedings of the ISThmus 2000, Research and Development for the Information Society*, pp. 409–418, Poznań, Poland, Apr. 2000.

[7] M. Bubak, D. Żbik, G. D. van Albada, K. A. Iskra, and P. M. A. Sloot. Portable Library of Migratable Sockets. In *Proceedings of the SGI Users' Conference*, pp. 175–184, Kraków, Poland, Oct. 2000.

[8] M. Bubak, D. Żbik, G. D. van Albada, K. A. Iskra, and P. M. A. Sloot. Migratable Sockets for Dynamic Load Balancing. In *Proceedings of the 9th International Conference on High-Performance Computing and Networking*, vol. 2110 of *Springer Lecture Notes in Computer Science*, pp. 23–31, Amsterdam, The Netherlands, June 2001.

[9] M. Bubak, D. Żbik, G. D. van Albada, K. A. Iskra, and P. M. A. Sloot. Portable Library of Migratable Sockets. *Scientific Programming*, vol. 9, no. 4, pp. 211–222, 2001.

[10] K. A. Iskra, G. D. van Albada, and P. M. A. Sloot. Rollbacks in Time Warp – Analysis and Modelling. In *Proceedings of the 8th Annual Conference of the Advanced School for Computing and Imaging*, pp. 88–94, Lochem, The Netherlands, June 2002.

[11] K. A. Iskra, R. G. Belleman, G. D. van Albada, J. Santoso, P. M. A. Sloot, H. E. Bal, H. J. W. Spoelder, and M. Bubak. The Polder Computing Environment, A System for Interactive Distributed Simulation. *Concurrency and Computation: Practice and Experience*, vol. 14, no. 13–15, pp. 1313–1335, 2002.

[12] K. A. Iskra, G. D. van Albada, and P. M. A. Sloot. Time Warp Cancellation Optimisations on High Latency Networks. In *Proceedings of the 7th IEEE International Symposium on Distributed Simulation and Real-Time Applications*, pp. 128–135, Delft, The Netherlands, Oct. 2003.

[13] K. A. Iskra, G. D. van Albada, and P. M. A. Sloot. Towards Grid-Aware Time Warp. In *Proceedings of the 18th IEEE/ACM/SCS Workshop on Parallel and Distributed Simulation*, pp. 63–70, Kufstein, Austria, May 2004.

[14] K. A. Iskra, G. D. van Albada, and P. M. A. Sloot. Towards Grid-Aware Time Warp. *Simulation: Transactions of The Society for Modeling and Simulation International*, 2005. Accepted.

Bibliography

[1] D. Abramson, R. Buyya, and J. Giddy. A Computational Economy for Grid Computing and its Implementation in the Nimrod-G Resource Broker. *Future Generation Computer Systems*, vol. 18, no. 8, pp. 1061–1074, Oct. 2002.

[2] D. Abramson, R. Sosic, J. Giddy, and B. Hall. Nimrod: A Tool for Performing Parameterised Simulations Using Distributed Workstations. In *Proceedings of the 4th IEEE International Symposium on High Performance Distributed Computing*, pp. 112–121, Washington, DC, USA, Aug. 1995.

[3] D. Agrawal and J. R. Agre. Replicated Objects in Time Warp Simulations. In *Proceedings of the 24th Conference on Winter Simulation*, pp. 657–664, Arlington, VA, USA, Dec. 1992.

[4] I. F. Akyildiz, L. Chen, S. R. Das, R. M. Fujimoto, and R. F. Serfozo. Performance Analysis of Time Warp with Limited Memory. In *Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, pp. 213–224, Newport, RI, USA, June 1992.

[5] M. al Mourabit. Time Warp Performance: An Empirical Study of Rollback in the APSIS Time Warp Kernel. Master's thesis, University of Amsterdam, The Netherlands, Aug. 2000.

[6] G. M. Amdahl. Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. In *Proceedings of the AFIPS Conference*, vol. 30, pp. 483–485, Atlantic City, NJ, USA, Apr. 1967.

[7] T. E. Anderson, D. E. Culler, D. A. Patterson, and the NOW team. A Case for NOW (Networks of Workstations). *IEEE Micro*, vol. 15, no. 1, pp. 54–64, Feb. 1995.

[8] H. Avril and C. Tropper. The Dynamic Load Balancing of Clustered Time Warp for Logic Simulation. In *Proceedings of the 10th Workshop on Parallel and Distributed Simulation*, pp. 20–27, Philadelphia, PA, USA, May 1996.

[9] H. Avril and C. Tropper. On Rolling Back and Checkpointing in Time Warp. *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 11, pp. 1105–1121, Nov. 2001.

[10] R. Bagrodia, R. Meyer, M. Takai, Y.-a. Chen, X. Zeng, J. Martin, and H. Y. Song. Parsec: A Parallel Simulation Environment for Complex Systems. *Computer*, vol. 31, no. 10, pp. 77–85, Oct. 1998.

[11] M. L. Bailey, J. V. Briner, Jr, and R. D. Chamberlain. Parallel Logic Simulation of VLSI Systems. *ACM Computing Surveys*, vol. 26, no. 3, pp. 255–294, Sept. 1994.

[12] P. Bak. *How Nature Works: The Science of Self-Organized Criticality.* Copernicus Books, June 1996.

[13] P. Bak, C. Tang, and K. Wiesenfeld. Self-Organized Criticality. *Physical Review A*, vol. 38, no. 1, pp. 364–374, July 1988.

[14] H. E. Bal, R. A. F. Bhoedjang, R. F. H. Hofman, C. J. H. Jacobs, T. Kielmann, J. Maassen, R. van Nieuwpoort, J. Romein, L. Renambot, T. Rühl, R. Veldema, K. Verstoep, A. Baggio, G. Ballintijn, I. Kuz, G. Pierre, M. van Steen, A. S. Tanenbaum, G. Doornbos, D. Germans, H. J. W. Spoelder, E.-J. Baerends, S. J. A. van Gisbergen, H. Afsermanesh, G. D. van Albada, A. S. Z. Belloum, D. Dubbeldam, Z. W. Hendrikse, L. O. Hertzberger, A. G. Hoekstra, K. A. Iskra, B. D. Kandhai, D. C. Koelma, F. van der Linden, B. J. Overeinder, P. M. A. Sloot, P. F. Spinnato, D. H. J. Epema, A. J. C. van Gemund, P. P. Jonker, A. Radulescu, C. van Reeuwijk, H. J. Sips, P. M. W. Knijnenburg, M. S. Lew, F. Sluiter, L. Wolters, H. Blom, C. de Laat, and A. van der Steen. The Distributed ASCI Supercomputer Project. *ACM Operating Systems Review*, vol. 34, no. 4, pp. 76–96, Oct. 2000.

[15] V. Balakrishnan, R. Radhakrishnan, D. M. Rao, N. B. Abu-Ghazaleh, and P. A. Wilsey. A Performance and Scalability Analysis Framework for Parallel Discrete Event Simulators. *Simulation Practice and Theory*, vol. 8, no. 8, pp. 529–553, July 2001.

[16] M. S. Balsamo and C. Manconi. Rollback Overhead Reduction Methods for Time Warp Distributed Simulation. *Simulation Practice and Theory*, vol. 6, no. 8, pp. 689–702, Dec. 1998.

[17] A. Barak, O. La'adan, and A. Shiloh. Scalable Cluster Computing with MOSIX for LINUX. In *Proceedings of the 5th Linux Expo*, pp. 95–100, Raleigh, NC, USA, May 1999.

[18] J. Basney and M. Livny. Improving Goodput by Co-scheduling CPU and Network Capacity. *International Journal of High Performance Computing Applications*, vol. 13, no. 3, pp. 220–230, 1999.

[19] H. Bauer and C. Sporrer. Distributed Logic Simulation and an Approach to Asynchronous GVT-Calculation. In *Proceedings of the 6th Workshop on Parallel and Dis-*

tributed Simulation, vol. 24 of *SCS Simulation Series*, pp. 205–208, Newport Beach, CA, USA, Jan. 1992.

[20] H. Bauer and C. Sporrer. Reducing Rollback Overhead in Time-Warp Based Distributed Simulation with Optimized Incremental State Saving. In *Proceedings of the 26th Simulation Symposium*, pp. 12–20, Mar. 1993.

[21] R. G. Belleman. *Interactive Exploration in Virtual Environments*. PhD thesis, University of Amsterdam, The Netherlands, Apr. 2003.

[22] S. Bellenot. Global Virtual Time Algorithms. In *Proceedings of the SCS Multiconference on Distributed Simulation*, vol. 22 of *SCS Simulation Series*, pp. 122–127, San Diego, CA, USA, Jan. 1990.

[23] S. Bellenot. Performance of a Riskfree Time Warp Operating System. In *Proceedings of the 7th Workshop on Parallel and Distributed Simulation*, pp. 155–158, San Diego, CA, USA, May 1993.

[24] R. Beraldi and L. Nigro. A Time Warp Mechanism Based on Temporal Uncertainty. *Transactions of The Society for Modeling and Simulation International*, vol. 18, no. 2, pp. 60–72, June 2001.

[25] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, vol. 15, no. 1, pp. 29–36, Feb. 1995.

[26] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Hérault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Néri, and A. Selikhov. MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, Baltimore, MD, USA, Nov. 2002.

[27] A. Boukerche, A. Fabbri, and A. R. Mikler. Distributed Simulation over Loosely Coupled Domains. In *Proceedings of the 4th International Workshop on Distributed Simulation and Real-Time Applications*, pp. 18–25, San Francisco, CA, USA, Aug. 2000.

[28] R. E. Bryant. Simulation of Packet Communication Architecture Computer Systems. Technical Report LCS-TR-188, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, USA, Nov. 1977.

[29] M. Bubak, W. Funika, D. Żbik, G. D. van Albada, K. A. Iskra, P. M. A. Sloot, R. Wismüller, and K. Sowa-Piekło. Performance Measurement, Debugging and Load Balancing for Metacomputing. In *Proceedings of the ISThmus 2000, Research and Development for the Information Society*, pp. 409–418, Poznań, Poland, Apr. 2000.

[30] M. Bubak, D. Żbik, G. D. van Albada, K. A. Iskra, and P. M. A. Sloot. Portable Library of Migratable Sockets. In *Proceedings of the SGI Users' Conference*, pp. 175–184, Kraków, Poland, Oct. 2000.

[31] M. Bubak, D. Żbik, G. D. van Albada, K. A. Iskra, and P. M. A. Sloot. Migratable Sockets for Dynamic Load Balancing. In *Proceedings of the 9th International Conference on High-Performance Computing and Networking*, vol. 2110 of *Springer Lecture Notes in Computer Science*, pp. 23–31, Amsterdam, The Netherlands, June 2001.

[32] M. Bubak, D. Żbik, G. D. van Albada, K. A. Iskra, and P. M. A. Sloot. Portable Library of Migratable Sockets. *Scientific Programming*, vol. 9, no. 4, pp. 211–222, 2001.

[33] C. D. Carothers, D. Bauer, and S. Pearce. ROSS: A High-Performance, Low Memory, Modular Time Warp System. *Journal of Parallel and Distributed Computing*, vol. 62, no. 11, pp. 1648–1669, Nov. 2002.

[34] C. D. Carothers and R. M. Fujimoto. Efficient Optimistic Parallel Simulations Using Reverse Computation. *ACM Transactions on Modeling and Computer Simulation*, vol. 9, no. 3, pp. 224–253, July 1999.

[35] C. D. Carothers and R. M. Fujimoto. Efficient Execution of Time Warp Programs on Heterogeneous, NOW Platforms. *IEEE Transactions on Parallel and Distributed Systems*, vol. 11, no. 3, pp. 299–317, Mar. 2000.

[36] C. D. Carothers, R. M. Fujimoto, and P. England. Effect of Communication Overheads on Time Warp Performance: An Experimental Study. In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation*, pp. 118–125, Edinburgh, Scotland, UK, July 1994.

[37] C. D. Carothers, R. M. Fujimoto, and Y.-B. Lin. A Case Study in Simulating PCS Networks Using Time Warp. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, pp. 87–94, Lake Placid, NY, USA, June 1995.

[38] J. Casas, D. L. Clark, R. B. Konuru, S. W. Otto, R. M. Prouty, and J. Walpole. MPVM: A Migration Transparent Version of PVM. *Computing Systems*, vol. 8, no. 2, pp. 171–216, 1995.

[39] R. D. Chamberlain and D. P. Discher. SimBench: A Logic Simulation Benchmark Set. In *Proceedings of the Mentor User's Group Conference*, Oct. 2000.

[40] S. Chandrasekaran and M. D. Hill. Optimistic Simulation of Parallel Architectures Using Program Executables. In *Proceedings of the 10th Workshop on Parallel and Distributed Simulation*, pp. 143–150, Philadelphia, PA, USA, May 1996.

[41] K. M. Chandy and J. Misra. Distributed Simulation: A Case Study and Design and Verification of Distributed Programs. *IEEE Transactions on Software Engineering*, vol. 5, no. 5, pp. 440–452, Sept. 1979.

[42] K. M. Chandy and J. Misra. Asynchronous Distributed Simulation via a Sequence of Parallel Computations. *Communications of the ACM*, vol. 24, no. 4, pp. 198–206, Apr. 1981.

[43] G. Chen and B. K. Szymanski. Lookback: A New Way of Exploiting Parallelism in Discrete Event Simulation. In *Proceedings of the 16th Workshop on Parallel and Distributed Simulation*, pp. 153–162, Washington, DC, USA, May 2002.

[44] M. Chetlur, N. B. Abu-Ghazaleh, R. Radhakrishnan, and P. A. Wilsey. Optimizing Communication in Time-Warp Simulators. In *Proceedings of the 12th Workshop on Parallel and Distributed Simulation*, pp. 64–71, Banff, AB, Canada, May 1998.

[45] A. I. Concepcion and S. G. Kelly. Computing Global Virtual Time Using the Multi-Level Token Passing Algorithm. In *Proceedings of the SCS Multiconference on Ad-*

vances in Parallel and Distributed Simulation, vol. 23 of *SCS Simulation Series*, pp. 63–68, Anaheim, CA, USA, Jan. 1991.

[46] D. Conklin, J. Cleary, and B. Unger. The Sharks World: A Study in Distributed Simulation Design. In *Proceedings of the SCS Multiconference on Distributed Simulation*, vol. 22 of *SCS Simulation Series*, pp. 157–160, San Diego, CA, USA, Jan. 1990.

[47] J. H. Cowie, D. M. Nicol, and A. T. Ogielski. Modeling the Global Internet. *IEEE Computing in Science and Engineering*, vol. 1, no. 1, pp. 42–50, Jan. 1999.

[48] C. Cruz-Neira, D. J. Sandin, and T. A. DeFanti. Surround-Screen Projection-Based Virtual Reality: The Design and Implementation of the CAVE. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 135–142, Aug. 1993.

[49] K. Czajkowski, D. Ferguson, I. Foster, J. Frey, S. Graham, T. Maguire, D. Snelling, and S. Tuecke. From Open Grid Services Infrastructure to WS-Resource Framework: Refactoring & Evolution, Mar. 2004.

[50] J. S. Dahmann, R. M. Fujimoto, and R. M. Weatherly. The Department of Defense High Level Architecture. In *Proceedings of the 29th Conference on Winter Simulation*, pp. 142–149, Atlanta, GA, USA, Dec. 1997.

[51] O. P. Damani and V. K. Garg. Fault-Tolerant Distributed Simulation. In *Proceedings of the 12th Workshop on Parallel and Distributed Simulation*, pp. 38–45, Banff, AB, Canada, May 1998.

[52] O. P. Damani, Y.-M. Wang, and V. K. Garg. Optimistic Distributed Simulation Based on Transitive Dependency Tracking. In *Proceedings of the 11th Workshop on Parallel and Distributed Simulation*, pp. 90–97, Lockenhaus, Austria, June 1997.

[53] P. Dan, D. Wang, Y. Zhang, and M. Shen. Quasi-Asynchronous Migration: A Novel Migration Protocol for PVM Tasks. *ACM Operating Systems Review*, vol. 33, no. 2, pp. 5–14, Apr. 1999.

[54] S. K. Das and F. Sarkar. A Hypercube Algorithm for GVT Computation and its Application in Optimistic Parallel Simulation. In *Proceedings of the 28th Simulation Symposium*, pp. 51–60, Santa Barbara, CA, USA, Apr. 1995.

[55] S. R. Das, R. M. Fujimoto, K. Panesar, D. Allison, and M. Hybinette. GTW: A Time Warp System for Shared Memory Multiprocessors. In *Proceedings of the 26th Conference on Winter Simulation*, pp. 1332–1339, Orlando, FL, USA, Dec. 1994.

[56] L. Dikken, F. van der Linden, J. J. J. Vesseur, and P. M. A. Sloot. DynamicPVM: Dynamic Load Balancing on Parallel Systems. In *Proceedings of the 2nd International Conference and Exhibition on High-Performance Computing and Networking*, vol. 797 of *Springer Lecture Notes in Computer Science*, pp. 273–277, Munich, Germany, Apr. 1994.

[57] G. E. Fagg and J. J. Dongarra. FT-MPI: Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World. In *Proceedings of the 7th European PVM/MPI Users' Group Meeting*, vol. 1908 of *Springer Lecture Notes in Computer Science*, pp. 346–353, Lake Balaton, Hungary, Sept. 2000.

[58] R. E. Felderman and L. Kleinrock. Bounds and Approximations for Self-Initiating Distributed Simulation Without Lookahead. *ACM Transactions on Modeling and Computer Simulation*, vol. 1, no. 4, pp. 386–406, Oct. 1991.

[59] R. E. Felderman and L. Kleinrock. Two Processor Time Warp Analysis: Some Results on a Unifying Approach. In *Proceedings of the SCS Multiconference on Advances in Parallel and Distributed Simulation*, vol. 23 of *SCS Simulation Series*, pp. 3–10, Anaheim, CA, USA, Jan. 1991.

[60] A. Ferscha. Parallel and Distributed Simulation of Discrete Event Systems. In A. Y. Zomaya, editor, *Parallel and Distributed Computing Handbook*, pp. 1003–1041. McGraw-Hill, Jan. 1996.

[61] A. Ferscha, J. Johnson, and S. J. Turner. Distributed Simulation Performance Data Mining. *Future Generation Computer Systems*, vol. 18, no. 1, pp. 157–174, Sept. 2001.

[62] M. J. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, vol. C-21, no. 9, pp. 948–960, Sept. 1972.

[63] M. J. Flynn and P. Dubey. Guest Editors' Introduction: Hot Chips 15 – Scaling the Silicon Mountain. *IEEE Micro*, vol. 24, no. 2, pp. 7–9, Mar. 2004.

[64] I. Foster. What is the Grid? A Three Point Checklist. *GRID today*, vol. 1, no. 6, July 2002.

[65] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputer Applications and High Performance Computing*, vol. 11, no. 2, pp. 115–128, 1997.

[66] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, Nov. 1998.

[67] I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration, June 2002.

[68] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of High Performance Computing Applications*, vol. 15, no. 3, pp. 200–222, 2001.

[69] S. Franks, F. Gomes, B. Unger, and J. Cleary. Saving for Interactive Optimistic Simulation. In *Proceedings of the 11th Workshop on Parallel and Distributed Simulation*, pp. 72–79, Lockenhaus, Austria, June 1997.

[70] R. M. Fujimoto. Lookahead in Parallel Discrete Event Simualtion. In *Proceedings of the International Conference on Parallel Processing*, vol. III, pp. 34–41, University Park, PA, USA, Aug. 1988.

[71] R. M. Fujimoto. Time Warp on a Shared Memory Multiprocessor. *Transactions of the Society for Computer Simulation*, vol. 6, no. 3, pp. 211–239, July 1989.

[72] R. M. Fujimoto. Parallel Discrete Event Simulation. *Communications of the ACM*, vol. 33, no. 10, pp. 30–53, Oct. 1990.

[73] R. M. Fujimoto. Performance of Time Warp Under Synthetic Workloads. In *Proceedings of the SCS Multiconference on Distributed Simulation*, vol. 22 of *SCS Simulation Series*, pp. 23–28, San Diego, CA, USA, Jan. 1990.

[74] R. M. Fujimoto. *Parallel and Distributed Simulation Systems.* Wiley, Dec. 1999.

[75] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Proceedings of the 11th European PVM/MPI Users' Group Meeting*, pp. 97–104, Budapest, Hungary, Sept. 2004.

[76] E. Gabriel, M. Resch, and R. Rühle. Implementing MPI with Optimized Algorithms for Metacomputing. In *Proceedings of the 3rd MPI Developers' and Users' Conference*, pp. 31–42, Starkville, MS, USA, Mar. 1999.

[77] A. Gafni. Rollback Mechanisms for Optimistic Distributed Simulation Systems. In *Proceedings of the SCS Multiconference on Distributed Simulation*, vol. 19 of *SCS Simulation Series*, pp. 61–67, San Diego, CA, USA, July 1988.

[78] A. Geist, A. Beguelin, J. J. Dongarra, W. Jiang, R. Manchek, and V. S. Sunderam. *PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press, Cambridge, MA, USA, Nov. 1994.

[79] D. W. Glazer and C. Tropper. On Process Migration and Load Balancing in Time Warp. *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 3, pp. 318–327, Mar. 1993.

[80] K. Gödel. Über Formal Unentscheidbare Sätze der Principia Mathematica und Verwandter Systeme, I. *Monatshefte für Mathematik und Physik*, vol. 38, pp. 173–198, 1931.

[81] V. Govindan and M. A. Franklin. Speculative Computation: Overcoming Communication Delays. In *Proceedings of the 23rd International Conference on Parallel Processing*, vol. III, pp. 12–16, North Carolina State University, NC, USA, Aug. 1994.

[82] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface (2nd Edition).* MIT Press, Cambridge, MA, USA, Nov. 1999.

[83] A. Gupta, I. F. Akyildiz, and R. M. Fujimoto. Performance Analysis of Time Warp with Multiple Homogeneous Processors. *IEEE Transactions on Software Engineering*, vol. 17, no. 10, pp. 1013–1027, Oct. 1991.

[84] P. Hontalas, B. Beckman, M. DiLoreto, L. Blume, P. L. Reiher, K. Sturdevant, V. Warren, J. J. Wedel, F. Wieland, and D. R. Jefferson. Performance of the Colliding Pucks Simulation on the Time Warp Operating System (Part 1: Asynchronous Behavior & Sectoring). In *Proceedings of the SCS Multiconference on Distributed Simulation*, vol. 21 of *SCS Simulation Series*, pp. 3–7, Mar. 1989.

[85] D. Howe. The Free On-line Dictionary of Computing, 1993. *http://www.foldoc.org/*.

[86] M. Hybinette and R. M. Fujimoto. Latency Hiding with Optimistic Computations. *Journal of Parallel and Distributed Computing*, vol. 62, no. 3, pp. 427–445, Mar. 2002.

[87] E. Ising. Beitrag zur Theorie des Ferromagnetismus. *Zeitschrift für Physik*, vol. 31, pp. 253–258, 1925.

[88] K. A. Iskra, R. G. Belleman, G. D. van Albada, J. Santoso, P. M. A. Sloot, H. E. Bal, H. J. W. Spoelder, and M. Bubak. The Polder Computing Environment, A System for Interactive Distributed Simulation. *Concurrency and Computation: Practice and Experience*, vol. 14, no. 13–15, pp. 1313–1335, 2002.

[89] K. A. Iskra, Z. W. Hendrikse, G. D. van Albada, B. J. Overeinder, and P. M. A. Sloot. Dynamic Migration of PVM Tasks. In *Proceedings of the 6th Annual Conference of the Advanced School for Computing and Imaging*, pp. 206–212, Lommel, Belgium, June 2000.

[90] K. A. Iskra, Z. W. Hendrikse, G. D. van Albada, B. J. Overeinder, and P. M. A. Sloot. Experiments with Migration of PVM Tasks. In *Proceedings of the ISThmus 2000, Research and Development for the Information Society*, pp. 295–303, Poznań, Poland, Apr. 2000.

[91] K. A. Iskra, Z. W. Hendrikse, G. D. van Albada, B. J. Overeinder, and P. M. A. Sloot. Performance Measurements on Dynamite/DPVM. In *Proceedings of the 7th European PVM/MPI Users' Group Meeting*, vol. 1908 of *Springer Lecture Notes in Computer Science*, pp. 27–38, Lake Balaton, Hungary, Sept. 2000.

[92] K. A. Iskra, Z. W. Hendrikse, G. D. van Albada, B. J. Overeinder, P. M. A. Sloot, and J. Gehring. Experiments with Migration of Message Passing Tasks. In *Proceedings of the 1st IEEE/ACM International Workshop on Grid Computing*, vol. 1971 of *Springer Lecture Notes in Computer Science*, pp. 203–213, Bangalore, India, Dec. 2000.

[93] K. A. Iskra, F. van der Linden, Z. W. Hendrikse, B. J. Overeinder, G. D. van Albada, and P. M. A. Sloot. The Implementation of Dynamite – an Environment for Migrating PVM Tasks. *ACM Operating Systems Review*, vol. 34, no. 3, pp. 40–55, July 2000.

[94] D. R. Jefferson. Virtual Time. *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 3, pp. 404–425, July 1985.

[95] D. R. Jefferson. Virtual Time II: Storage Management in Distributed Simulation. In *Proceedings of the 9th ACM Symposium on Principles of Distributed Computing*, pp. 75–89, Quebec City, QC, Canada, Aug. 1990.

[96] D. R. Jefferson, B. Beckman, F. Wieland, L. Blume, M. DiLoreto, P. Hontalas, P. Laroche, K. Sturdevant, J. Tupman, V. Warren, J. J. Wedel, H. Younger, and S. Bellenot. Distributed Simulation and the Time Wrap Operating System. In *Proceedings of the 11th ACM Symposium on Operating System Principles*, pp. 77–93, Austin, TX, USA, Nov. 1987.

[97] V. Jha and R. Bagrodia. Simultaneous Events and Lookahead in Simulation Protocols. *ACM Transactions on Modeling and Computer Simulation*, vol. 10, no. 3, pp. 241–267, July 2000.

[98] N. Kalantery. Time Warp – Connection Oriented. In *Proceedings of the 18th Workshop on Parallel and Distributed Simulation*, pp. 71–77, Kufstein, Austria, May 2004.

[99] N. T. Karonis, B. Toonen, and I. Foster. MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface. *Journal of Parallel and Distributed Computing*, vol. 63, no. 5, pp. 551–563, May 2003.

[100] T. Kielmann, R. F. H. Hofman, H. E. Bal, A. Plaat, and R. A. F. Bhoedjang. MagPIe: MPI's Collective Communication Operations for Clustered Wide Area Systems. In *Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 131–140, Atlanta, GA, USA, May 1999.

[101] T. Kimura and H. Takemiya. Local Area Metacomputing for Multidisciplinary Problems: A Case Study for Fluid/Structure Coupled Simulation. In *Proceedings of the 12th International Conference on Supercomputing*, pp. 149–156, Melbourne, Australia, July 1998.

[102] A. Kolakowska, M. A. Novotny, and G. Korniss. Algorithmic Scalability in Globally Constrained Conservative Parallel Discrete Event Simulations of Asynchronous Systems. *Physical Review E*, vol. 67, no. 046703, Apr. 2003.

[103] A. Kolakowska, M. A. Novotny, and P. A. Rikvold. Update Statistics in Conservative Parallel Discrete Event Simulations of Asynchronous Systems. *Physical Review E*, vol. 68, no. 046705, Oct. 2003.

[104] G. Korniss, M. A. Novotny, H. Guclu, Z. Toroczkai, and P. A. Rikvold. Suppressing Roughness of Virtual Times in Parallel Discrete-Event Simulations. *Science*, vol. 299, no. 5607, pp. 677–679, Jan. 2003.

[105] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, July 1978.

[106] L. A. H. Liang, W. Cai, B.-S. Lee, and S. J. Turner. Performance Analysis of Packet Bundling Techniques in DIS. In *Proceedings of the 3rd International Workshop on Distributed Interactive Simulation and Real-Time Applications*, pp. 75–82, College Park, MD, USA, Mar. 1999.

[107] Y.-B. Lin and P. A. Fishwick. Asynchronous Parallel Discrete Event Simulation. *IEEE Transactions on Systems, Man and Cybernetics, Part A*, vol. 26, no. 4, pp. 397–412, July 1996.

[108] Y.-B. Lin and E. D. Lazowska. Determining the Global Virtual Time in a Distributed Simulation. In *Proceedings of the International Conference on Parallel Processing*, vol. III, pp. 201–209, Urbana-Champaign, IL, USA, Aug. 1990.

[109] Y.-B. Lin and E. D. Lazowska. Exploiting Lookahead in Parallel Simulation. *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 4, pp. 457–469, Oct. 1990.

[110] Y.-B. Lin and E. D. Lazowska. Optimality Considerations of Time Warp Parallel Simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation*, vol. 22 of *SCS Simulation Series*, pp. 29–34, San Diego, CA, USA, Jan. 1990.

[111] Y.-B. Lin and E. D. Lazowska. A Study of Time Warp Rollback Mechanisms. *ACM Transactions on Modeling and Computer Simulation*, vol. 1, no. 1, pp. 51–72, Jan. 1991.

[112] Y.-B. Lin and B. R. Preiss. Optimal Memory Management for Time Warp Parallel Simulation. *ACM Transactions on Modeling and Computer Simulation*, vol. 1, no. 4, pp. 283–307, Oct. 1991.

[113] Y.-B. Lin, B. R. Preiss, W. M. Loucks, and E. D. Lazowska. Selecting the Checkpoint Interval in Time Warp Simulation. In *Proceedings of the 7th Workshop on Parallel and Distributed Simulation*, pp. 3–10, San Diego, CA, USA, May 1993.

[114] R. J. Lipton and D. M. Mizell. Time Warp vs. Chandy-Misra: A Worst-Case Comparison. In *Proceedings of the SCS Multiconference on Distributed Simulation*, vol. 22 of *SCS Simulation Series*, pp. 137–143, San Diego, CA, USA, Jan. 1990.

[115] M. C. Little and D. L. McCue. C++ SIM. *C Users Journal*, vol. 12, no. 3, pp. 119–136, Mar. 1994.

[116] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor – A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, pp. 104–111, San Jose, CA, USA, June 1988.

[117] L. Z. Liu and C. Tropper. Local Deadlock Detection in Distributed Simulations. In *Proceedings of the SCS Multiconference on Distributed Simulation*, vol. 22 of *SCS Simulation Series*, pp. 64–69, San Diego, CA, USA, Jan. 1990.

[118] B. D. Lubachevsky. Efficient Parallel Simulations of Dynamic Ising Spin Systems. *Journal of Computational Physics*, vol. 75, no. 1, pp. 103–122, Mar. 1988.

[119] B. D. Lubachevsky and A. Weiss. Synchronous Relaxation for Parallel Ising Spin Simulations. In *Proceedings of the 15th Workshop on Parallel and Distributed Simulation*, pp. 185–192, Lake Arrowhead, CA, USA, May 2001.

[120] B. D. Lubachevsky, A. Weiss, and A. Shwartz. An Analysis of Rollback-Based Simulation. *ACM Transactions on Modeling and Computer Simulation*, vol. 1, no. 2, pp. 154–193, Apr. 1991.

[121] M. Marín. Time Warp on BSP Computers. In *Proceedings of the 12th European Simulation Multiconference*, pp. 225–229, Manchester, UK, July 1998.

[122] D. E. Martin, T. J. McBrayer, and P. A. Wilsey. WARPED: A Time Warp Simulation Kernel for Analysis and Application Development. In *Proceedings of the 29th Hawaii International Conference on System Sciences*, vol. I, pp. 383–386, Maui, HI, USA, Jan. 1996.

[123] E. Mascarenhas, F. Knop, and V. Rego. ParaSol: A Multithreaded System for Parallel Simulation Based on Mobile Threads. In *Proceedings of the 27th Conference on Winter Simulation*, pp. 690–697, Arlington, VA, USA, Dec. 1995.

[124] F. Mattern. Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation. *Journal of Parallel and Distributed Computing*, vol. 18, no. 4, pp. 423–434, Aug. 1993.

[125] B. C. Merrifield, S. B. Richardson, and J. B. G. Roberts. Quantitative Studies Of Discrete Event Simulation Modelling of Road Traffic. In *Proceedings of the SCS Multiconference on Distributed Simulation*, vol. 22 of *SCS Simulation Series*, pp. 188–193, San Diego, CA, USA, Jan. 1990.

[126] Message Passing Interface Forum. MPI: A Message Passing Interface Standard. *International Journal of Supercomputer Applications and High Performance Computing*, vol. 8, no. 3/4, pp. 159–416, 1994.

[127] N. C. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. Equation of State Calculation by Fast Computing Machines. *Journal of Chemical Physics*, vol. 21, no. 6, pp. 1087–1092, June 1953.

[128] J. Misra. Distributed Discrete Event Simulation. *ACM Computing Surveys*, vol. 18, no. 1, pp. 39–65, Mar. 1986.

[129] M. Müller, M. Hess, and E. Gabriel. Grid Enabled MPI Solutions for Clusters. In *Proceedings of the 3rd International Symposium on Cluster Computing and the Grid*, pp. 18–25, Tokyo, Japan, May 2003.

[130] Myricom, Inc. MPICH-GM Webpage. *http://www.myri.com/scs/*.

[131] D. M. Nicol. Performance Bounds on Parallel Self-Initiating Discrete-Event Simulations. *ACM Transactions on Modeling and Computer Simulation*, vol. 1, no. 1, pp. 24–50, Jan. 1991.

[132] D. M. Nicol and R. M. Fujimoto. Parallel Simulation Today. *Annals of Operations Research*, vol. 53, pp. 249–286, Dec. 1994.

[133] R. Noronha and N. B. Abu-Ghazaleh. Early Cancellation: An Active NIC Optimisation for Time-Warp. In *Proceedings of the 16th Workshop on Parallel and Distributed Simulation*, pp. 43–50, Washington, DC, USA, May 2002.

[134] B. J. Overeinder. *Distributed Event-driven Simulation – Scheduling Strategies and Resource Management*. PhD thesis, University of Amsterdam, The Netherlands, Nov. 2000.

[135] B. J. Overeinder, A. Schoneveld, and P. M. A. Sloot. Spatio-Temporal Correlations and Rollback Distributions in Optimistic Simulations. In *Proceedings of the 15th IEEE/ACM/SCS Workshop on Parallel and Distributed Simulation*, pp. 145–152, Lake Arrowhead, CA, USA, May 2001.

[136] B. J. Overeinder and P. M. A. Sloot. Application of Time Warp to Parallel Simulations with Asynchronous Cellular Automata. In *Proceedings of the European Simulation Symposium*, pp. 397–402, Delft, The Netherlands, Oct. 1993.

[137] B. J. Overeinder, P. M. A. Sloot, R. N. Heederik, and L. O. Hertzberger. A Dynamic Load Balancing System for Parallel Cluster Computing. *Future Generation Computer Systems*, vol. 12, no. 1, pp. 101–115, May 1996.

[138] A. C. Palaniswamy and P. A. Wilsey. An Analytical Comparison of Periodic Checkpointing and Incremental State Saving. In *Proceedings of the 7th Workshop on Parallel and Distributed Simulation*, pp. 127–134, San Diego, CA, USA, May 1993.

[139] C.-D. Pham. Comparison of Message Aggregation Strategies for Parallel Simulations on a High Performance Cluster. In *Proceedings of the 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pp. 358–365, San Francisco, CA, USA, Aug. 2000.

[140] E. Pinheiro and R. Bianchini. Nomad: A Scalable Operating System for Clusters of Uni and Multiprocessors. In *Proceedings of the 1st IEEE International Workshop on Cluster Computing*, pp. 247–254, Melbourne, Australia, Dec. 1999.

[141] A. Plaat, H. E. Bal, R. F. H. Hofman, and T. Kielmann. Sensitivity of Parallel Applications to Large Differences in Bandwidth and Latency in Two-Layer Interconnects. *Future Generation Computer Systems*, vol. 17, no. 6, pp. 769–782, Apr. 2001.

[142] B. R. Preiss. The Yaddes Distributed Discrete Event Simulation Specification Language and Execution Environments. In *Proceedings of the SCS Multiconference on Distributed Simulation*, vol. 21 of *SCS Simulation Series*, pp. 139–144, Mar. 1989.

[143] B. R. Preiss and W. M. Loucks. Memory Management Techniques for Time Warp on a Distributed Memory Machine. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, pp. 30–39, Lake Placid, NY, USA, June 1995.

[144] J. Pruyne and M. Livny. Managing Checkpoints for Parallel Programs. In *Proceedings of the 2nd Workshop on Job Scheduling Strategies for Parallel Processing*, vol. 1162 of *Springer Lecture Notes in Computer Science*, pp. 140–154, Honolulu, HI, USA, Apr. 1996.

[145] F. Quaglia. Combining Periodic and Probabilistic Checkpointing in Optimistic Simulation. In *Proceedings of the 13th Workshop on Parallel and Distributed Simulation*, pp. 109–116, Atlanta, GA, USA, May 1999.

[146] F. Quaglia, V. Cortellessa, and B. Ciciani. Trade-Off between Sequential and Time Warp-Based Parallel Simulation. *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 8, pp. 781–794, Aug. 1999.

[147] F. Quaglia and A. Santoro. Nonblocking Checkpointing for Optimistic Parallel Simulation: Description and an Implementation. *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, no. 6, pp. 593–610, June 2003.

[148] R. Radhakrishnan, N. B. Abu-Ghazaleh, M. Chetlur, and P. A. Wilsey. On-line Configuration of a Time Warp Parallel Discrete Event Simulator. In *Proceedings of the International Conference on Parallel Processing*, pp. 28–35, Minneapolis, MN, USA, Aug. 1998.

[149] R. Radhakrishnan, D. E. Martin, M. Chetlur, D. M. Rao, and P. A. Wilsey. An Object-Oriented Time Warp Simulation Kernel. In *Proceedings of the International Symposium on Computing in Object-Oriented Parallel Environments*, vol. 1505 of *Springer Lecture Notes in Computer Science*, pp. 13–23, Santa Fe, NM, USA, Dec. 1998.

[150] R. Radhakrishnan and P. A. Wilsey. Software Control Systems for Parallel Simulation. In *Proceedings of the 16th Workshop on Parallel and Distributed Simulation*, pp. 135–142, Washington, DC, USA, May 2002.

[151] H. Rajaei, R. Ayani, and L.-E. Thorelli. The Local Time Warp Approach to Parallel Simulation. In *Proceedings of the 7th Workshop on Parallel and Distributed Simulation*, pp. 119–126, San Diego, CA, USA, May 1993.

[152] G. Regnier, S. Makineni, R. Illikkal, R. Iyer, D. Minturn, R. Huggahalli, D. Newell, L. Cline, and A. Foong. TCP Onloading for Data Center Servers. *Computer*, vol. 37, no. 11, pp. 48–58, Nov. 2004.

[153] P. L. Reiher. Parallel Simulation Using the Time Warp Operating System. In *Proceedings of the 22nd Conference on Winter Simulation*, pp. 38–45, New Orleans, LA, USA, Dec. 1990.

[154] P. L. Reiher, R. M. Fujimoto, S. Bellenot, and D. R. Jefferson. Cancellation Strategies in Optimistic Execution Systems. In *Proceedings of the SCS Multiconference on Distributed Simulation*, vol. 22 of *SCS Simulation Series*, pp. 112–121, San Diego, CA, USA, Jan. 1990.

[155] P. L. Reiher, F. Wieland, and P. Hontalas. Providing Determinism in the Time Warp Operating System – Costs, Benefits, and Implications. In *Proceedings of the 2nd Workshop on Experimental Distributed Systems*, pp. 113–118, Huntsville, AL, USA, Oct. 1990.

[156] P. F. Reynolds, Jr. A Spectrum of Options for Parallel Simulation. In *Proceedings of the 20th Conference on Winter Simulation*, pp. 325–332, San Diego, CA, USA, Dec. 1988.

[157] J. Robinson, S. H. Russ, B. K. Flachs, and B. Heckel. A Task Migration Implementation of the Message-Passing Interface. In *Proceedings of the 5th International Symposium on High Performance Distributed Computing*, pp. 61–68, Syracuse, NY, USA, Aug. 1996.

[158] R. Rönngren, R. Ayani, R. M. Fujimoto, and S. R. Das. Efficient Implementation of Event Sets in Time Warp. In *Proceedings of the 7th Workshop on Parallel and Distributed Simulation*, pp. 101–108, San Diego, CA, USA, May 1993.

[159] R. Rönngren, L. Barriga, and R. Ayani. An Incremental Benchmark Suite for Performance Tuning of Parallel Discrete Event Simulation. In *Proceedings of the 29th Hawaii International Conference on System Sciences*, vol. I, pp. 373–382, Maui, HI, USA, Jan. 1996.

[160] R. Rönngren and M. Liljenstam. On Event Ordering in Parallel Discrete Event Simulation. In *Proceedings of the 13th Workshop on Parallel and Distributed Simulation*, pp. 38–45, Atlanta, GA, USA, May 1999.

[161] A. Roy, I. Foster, W. Gropp, N. T. Karonis, V. Sander, and B. Toonen. MPICH-GQ: Quality-of-Service for Message Passing Programs. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, Dallas, TX, USA, Nov. 2000.

[162] B. Samadi. *Distributed Simulation Algorithms and Performance Analysis.* PhD thesis, University of California, Los Angeles, CA, USA, Jan. 1985.

[163] R. Schlagenhaft, M. Ruhwandl, C. Sporrer, and H. Bauer. Dynamic Load Balancing of a Multi-Cluster Simulator on a Network of Workstations. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, pp. 175–180, Lake Placid, NY, USA, June 1995.

[164] A. Schoneveld. *Parallel Complex Systems Simulation.* PhD thesis, University of Amsterdam, The Netherlands, Dec. 1999.

[165] L. N. Shchur and M. A. Novotny. Evolution of Time Horizons in Parallel and Grid simulations. *Physical Review E*, vol. 70, no. 026703, Aug. 2004.

[166] P. M. A. Sloot. Simulation and Visualisation in Medical Diagnosis: Perspectives and Computational Requirements. In A. Marsh, L. Grandinetti, and T. Kauranne, editors, *Advanced Infrastructures for Future Healthcare*, pp. 275–282. IOS Press, 2000.

[167] P. M. A. Sloot and B. J. Overeinder. Time Warped Automata: Parallel Discrete Event Simulation of Asynchronous CA's. In *Proceedings of the 3rd International Conference on Parallel Processing and Applied Mathematics*, pp. 43–62, Kazimierz Dolny, Poland, Sept. 1999.

[168] P. M. A. Sloot, B. J. Overeinder, and A. Schoneveld. Self-organized Criticality in Simulated Correlated Systems. *Computer Physics Communications*, vol. 142, no. 1–3, pp. 76–81, Dec. 2001.

[169] P. M. A. Sloot, A. Tirado-Ramos, A. G. Hoekstra, and M. Bubak. An Interactive Grid Environment for Non-Invasive Vascular Reconstruction. In *Proceedings of the 2nd International Workshop on Biomedical Computations on the Grid*, pp. 309–319, Chicago, IL, USA, Apr. 2004.

[170] L. Smarr and C. E. Catlett. Metacomputing. *Communications of the ACM*, vol. 35, no. 6, pp. 44–52, June 1992.

[171] L. M. Sokol and B. K. Stucky. MTW: Experimental Results for a Constrained Optimistic Scheduling Paradigm. In *Proceedings of the SCS Multiconference on Distributed Simulation*, vol. 22 of *SCS Simulation Series*, pp. 169–173, San Diego, CA, USA, Jan. 1990.

[172] P. F. Spinnato. *Hybrid Systems for N-body Simulations.* PhD thesis, University of Amsterdam, The Netherlands, Sept. 2003.

[173] J. S. Steinman. SPEEDES: Synchronous Parallel Environment for Emulation and Discrete Event Simulation. In *Proceedings of the SCS Multiconference on Advances in Parallel and Distributed Simulation*, vol. 23 of *SCS Simulation Series*, pp. 95–103, Anaheim, CA, USA, Jan. 1991.

[174] G. Stellner. CoCheck: Checkpointing and Process Migration for MPI. In *Proceedings of the 10th International Parallel Processing Symposium*, pp. 526–531, Honolulu, HI, USA, Apr. 1996.

[175] G. Stellner and J. Pruyne. Resource Management and Checkpointing for PVM. In *Proceedings of the 2nd European PVM Users' Group Meeting*, pp. 131–136, Lyon, France, Sept. 1995.

[176] T. Sterling, D. J. Becker, D. Savarese, J. E. Dorband, U. A. Ranawake, and C. V. Packer. BEOWULF: A Parallel Workstation for Scientific Computation. In *Proceedings of the 24th International Conference on Parallel Processing*, vol. I, pp. 11–14, Urbana-Champain, IL, USA, Aug. 1995.

[177] V. S. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, vol. 2, no. 4, pp. 315–339, Dec. 1990.

[178] B. K. Szymanski, A. Saifee, A. Sastry, Y. Liu, and K. Madnani. Genesis: A System for Large-scale Parallel Network Simulation. In *Proceedings of the 16th Workshop on Parallel and Distributed Simulation*, pp. 89–96, Washington, DC, USA, May 2002.

[179] S. C. Tay, Y. M. Teo, and R. Ayani. Performance Analysis of Time Warp Simulation with Cascading Rollbacks. In *Proceedings of the 12th Workshop on Parallel and Distributed Simulation*, pp. 30–37, Banff, AB, Canada, May 1998.

[180] A. Tirado-Ramos, J. M. Ragas, D. P. Shamonin, H. Rosmanith, and D. Kranzlmüller. Integration of Blood Flow Visualization on the Grid: the FlowFish/GVK Approach. In *Proceedings of the 2nd European Across Grids Conference*, vol. 3165 of *Springer Lecture Notes in Computer Science*, pp. 77–79, Nicosia, Cyprus, Jan. 2004.

[181] A. Tirado-Ramos, P. M. A. Sloot, A. G. Hoekstra, and M. Bubak. An Integrative Approach to High-Performance Biomedical Problem Solving Environments on the Grid. *Parallel Computing*, vol. 30, no. 9–10, pp. 1037–1055, Sept. 2004.

[182] A. Tirado-Ramos, K. Z. Zając, Z. Zhao, P. M. A. Sloot, G. D. van Albada, and M. Bubak. Experimental Grid Access for Dynamic Discovery and Data Transfer in Distributed Interactive Simulation Systems. In *Proceedings of the International Conference on Computational Science*, vol. 2657 of *Springer Lecture Notes in Computer Science*, pp. 284–292, Melbourne, Australia and St. Petersburg, Russia, June 2003.

[183] D. L. Turcotte. Self-Organized Criticality. *Reports on Progress in Physics*, vol. 62, no. 10, pp. 1377–1429, Oct. 1999.

[184] S. J. Turner and M. Q. Xu. Performance Evaluation of the Bounded Time Warp Algorithm. In *Proceedings of the 6th Workshop on Parallel and Distributed Simulation*, vol. 24 of *SCS Simulation Series*, pp. 117–126, Newport Beach, CA, USA, Jan. 1992.

[185] G. D. van Albada, J. Clinckemaillie, A. H. L. Emmen, J. Gehring, O. Heinz, F. van der Linden, B. J. Overeinder, A. Reinefeld, and P. M. A. Sloot. Dynamite – Blasting Obstacles to Parallel Cluster Computing. In *Proceedings of the 7th International Conference on High-Performance Computing and Networking*, number 1593 in Springer Lecture Notes in Computer Science, pp. 300–310, Amsterdam, The Netherlands, Apr. 1999.

[186] A. W. van Halderen, B. J. Overeinder, P. M. A. Sloot, R. van Dantzig, D. H. J. Epema, and M. Livny. Hierarchical Resource Management in the Polder Metacomputing Initiative. *Parallel Computing*, vol. 24, no. 12–13, pp. 1807–1825, Nov. 1998.

[187] J. J. J. Vesseur, R. N. Heederik, B. J. Overeinder, and P. M. A. Sloot. Experiments in Dynamic Load Balancing for Parallel Cluster Computing. In *Proceedings of the Workshop on Parallel Programming and Computation and the 4th Nordic Transputer Conference*, pp. 189–194, Linkoping, Sweden, June 1995.

[188] J. W. Wang. Cross-Cluster Migration of Parallel Tasks. Master's thesis, University of Amsterdam, The Netherlands, Sept. 2001.

[189] X. Wang, S. J. Turner, M. Y.-H. Low, and B.-P. Gan. Optimistic Synchronization in HLA Based Distributed Simulation. In *Proceedings of the 18th Workshop on Parallel and Distributed Simulation*, pp. 123–130, Kufstein, Austria, May 2004.

[190] D. West. Optimizing Time Warp: Lazy Rollback and Lazy Re-evaluation. Master's thesis, University of Calgary, Alberta, Canada, Jan. 1988.

[191] A. Wibisono. Cross Cluster Migration Using Dynamite – Remote File Access Support. Master's thesis, University of Amsterdam, The Netherlands, Sept. 2002.

[192] F. Wieland. Practical Parallel Simulation Applied to Aviation Modeling. In *Proceedings of the 15th Workshop on Parallel and Distributed Simulation*, pp. 109–116, Lake Arrowhead, CA, USA, May 2001.

[193] S. Wolfram. *A New Kind of Science*. Wolfram Media, May 2002.

[194] R. Wolski, N. T. Spring, and J. Hayes. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. *Future Generation Computer Systems*, vol. 15, no. 5–6, pp. 757–768, Oct. 1999.

[195] G. Yeung, M. Takai, R. Bagrodia, A. Mehrnia, and B. Daneshrad. Detailed OFDM Modeling in Network Simulation of Mobile Ad Hoc Networks. In *Proceedings of the 18th Workshop on Parallel and Distributed Simulation*, pp. 26–34, Kufstein, Austria, May 2004.

[196] V. C. Zandy and B. P. Miller. Reliable Network Connections. In *Proceedings of the 8th Annual International Conference on Mobile Computing and Networking*, pp. 95–106, Atlanta, GA, USA, Sept. 2002.

[197] V. C. Zandy, B. P. Miller, and M. Livny. Process Hijacking. In *Proceedings of the 8th International Symposium on High Performance Distributed Computing*, pp. 177–184, Redondo Beach, CA, USA, Aug. 1999.

[198] D. Żbik. Portable Library of Migratable Sockets. Master's thesis, University of Mining and Metallurgy, Kraków, Poland, Aug. 2000.

[199] Z. Zhao. *An Agent Based Architecture for Constructing Interactive Simulation Systems*. PhD thesis, University of Amsterdam, The Netherlands, Dec. 2004.

Index