

# Time Warp performance

An empirical study of rollback in the APSIS  
Time Warp kernel

Mostapha al Mourabit  
31st August 2000



MASTER'S THESIS



UNIVERSITEIT VAN AMSTERDAM  
*Faculty of Science*

THE NETHERLANDS



# **Time Warp performance**

An emperical study of rollback in the APSIS  
Time Warp kernel

Mostapha al Mourabit

31st August 2000

MASTER'S THESIS



UNIVERSITEIT VAN AMSTERDAM

*Faculty of Science*

THE NETHERLANDS

This document was prepared with L<sup>A</sup>T<sub>E</sub>X2 $\epsilon$ . A British spelling checking program was used to check the spelling of this thesis.

Copyright © 2000 by Mostapha al Mourabit, University of Amsterdam, the Netherlands.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without prior written permission by the copyright owner.

Illustration on the cover: *Persistence of Memory* by Salvator Dali.

# Acknowledgements

First of all I would like to thank my parents, brothers, sisters and the rest of the family without whom I probably would not have gotten this far.

I would also like to thank my supervisor Dr. G. D. van Albada for the discussions, his many suggestions and his guidance.

I also thank Prof. Dr. P. M. A. Sloot for accepting this project and Drs. B. J. Overeinder for his explanations about the APSIS Time Warp kernel.

At last, I would like to thank Dr. J. Kaandorp for providing me the solution to the basic Lotka-Volterra mathematical model.



# Contents

<b>Acknowledgements</b>	<b>i</b>
<b>List of Figures</b>	<b>v</b>
<b>List of Tables</b>	<b>vii</b>
<b>Abstract</b>	<b>2</b>
<b>1 Introduction</b>	<b>3</b>
1.1 Systems and models . . . . .	4
1.2 Thesis goals . . . . .	7
1.3 Thesis results . . . . .	7
1.4 Thesis structure . . . . .	8
<b>2 Discrete Event Simulation</b>	<b>9</b>
2.1 Sequential DES . . . . .	9
2.2 Parallel DES . . . . .	11
2.2.1 Causality . . . . .	11
2.2.2 Conservative protocols . . . . .	13
2.2.3 Optimistic protocols . . . . .	15
<b>3 The Time Warp mechanism</b>	<b>19</b>
3.1 Basic TW algorithm . . . . .	19
3.1.1 Space . . . . .	21
3.1.2 Time . . . . .	21
3.2 Performance enhancing extensions . . . . .	22
3.2.1 State saving . . . . .	22
3.2.2 Reclaiming memory . . . . .	23

3.2.3	Rollback . . . . .	23
3.2.4	Lazy Re-evaluation . . . . .	24
3.2.5	Optimism control . . . . .	24
3.2.6	Message aggregation . . . . .	25
3.3	A TW implementation: APSIS . . . . .	25
3.3.1	APSYS queues . . . . .	26
3.3.2	APSYS communication . . . . .	27
3.3.3	GVT computation algorithm . . . . .	27
3.3.4	The APSIS primitives . . . . .	30
<b>4</b>	<b>Workload for APSIS experiments</b>	<b>33</b>
4.1	The PHOLD algorithm . . . . .	34
4.2	PHOLD implementation . . . . .	35
<b>5</b>	<b>Communication</b>	<b>39</b>
5.1	APSYS communication primitives . . . . .	40
5.2	PVM library . . . . .	42
5.3	MPI library . . . . .	43
5.4	FM library . . . . .	44
<b>6</b>	<b>Experiments</b>	<b>49</b>
6.1	Experiment environment and tools . . . . .	49
6.2	Communication library performance . . . . .	50
6.3	Testing APSIS . . . . .	51
<b>7</b>	<b>Discussion and conclusions</b>	<b>55</b>
7.1	Discussion . . . . .	55
7.2	Future work . . . . .	59
<b>A</b>	<b>APHOLD</b>	<b>61</b>
<b>B</b>	<b>Experiment results</b>	<b>67</b>
B.1	Communication libraries experiments . . . . .	67
B.2	PHOLD experiments . . . . .	73
	<b>Bibliography</b>	<b>79</b>

# List of Figures

1.1	<i>Continuous vs. discrete system state plots</i>	4
1.2	<i>Basic Lotka-Volterra model.</i>	6
2.1	<i>Basic DES algorithm.</i>	9
2.2	<i>DES example: post-office</i>	10
2.3	<i>Sketch of a PDES run</i>	12
2.4	<i>Rollback and resumed execution at A</i>	15
2.5	<i>Breathing Time Bucket</i>	18
3.1	<i>Communication topology</i>	28
3.2	<i>A lower bound for LVT</i>	29
3.3	<i>A typical APSIS LP</i>	30
5.1	<i>Simplified communication scheme</i>	40
5.2	<i>Hello world example using FM</i>	46
A.1	<i>main APHOLD program</i>	61
A.2	<i>main event loop</i>	62
A.3	<i>computational grain function</i>	62
A.4	<i>time increment function</i>	63
A.5	<i>movement function</i>	63
A.6	<i>scheduling the initial event; first part</i>	64
A.7	<i>scheduling the initial event; second part</i>	65
A.8	<i>initial time distribution</i>	65
A.9	<i>initial space distribution</i>	66
B.1	<i>Median time against message size for PVM on the Sun machines.</i>	67
B.2	<i>Median time against message size for MPI on the Sun machines.</i>	67

B.3	<i>Median time against message size for PVM on the DAS.</i>	68
B.4	<i>Median time against message size for MPI on the DAS.</i>	68
B.5	<i>PVM communication delay distribution on Sun machines</i>	69
B.6	<i>MPI communication delay distribution on Sun machines</i>	70
B.7	<i>PVM communication delay distribution on DAS nodes</i>	71
B.8	<i>MPI communication delay distribution on DAS nodes</i>	72
B.9	<i>Number of LPs on the DAS</i>	73
B.10	<i>Initial message population on the DAS</i>	74
B.11	<i>Movement function on the DAS</i>	75
B.12	<i>Computational grain on the DAS</i>	76
B.13	<i>Normalised turnaround time when varying the number of LPs experiments</i>	77
B.14	<i>Normalised turnaround time when varying the number of messages</i>	77
B.15	<i>Normalised turnaround time when the computational grain is varied</i>	77

# List of Tables

4.1	<i>Movement function</i> . . . . .	37
5.1	<i>PVM functions in APSIS</i> . . . . .	43
5.2	<i>MPI functions in APSIS</i> . . . . .	44
5.3	<i>FM functions in APSIS</i> . . . . .	47
6.1	<i>Default APHOLD parameters</i> . . . . .	52
6.2	<i>Mutated APHOLD parameters</i> . . . . .	53
7.1	<i>Communication timings</i> . . . . .	55



# Abstract

Because of physical limitations it may be impossible to efficiently execute (very) large problems on one computing machine alone. However such problems may be divided into a set of smaller pieces which each may need fewer computing resources, such as processor time. Executing these subproblems on a number of processing nodes at the same time will often lead to a reduced execution time for the whole parallel program compared to the execution time of the original sequential problem. In most cases, these pieces will have to interact because local information at one piece is needed at another piece. In addition, because of problem specific properties and/or differences in processing speeds of the computing nodes, each piece may execute at a different program stage at any given time. They should be synchronised so that a subproblem which is at a different stage in the program execution compared to another subproblem will be able to use the correct information from that piece at the correct time.

One of the important computing applications is Discrete Event Simulation (DES). A DES is basically an approximation of the behaviour of a physical or theoretical system in which periods of inactivity are skipped and only actual activities (the events) are processed. These events take place at random (simulation) times, the time of occurrence. At the end of the simulation, all events should have been processed in the order of their time of occurrence. One of the possible synchronisation methods for parallel DES is the Time Warp (TW) protocol. TW detects out of order execution of events when a straggler has been received; a straggler is an event with an occurrence time which is smaller than that of the last processed event just before the straggler had been received. After detection the simulation is rolled back, that is, the simulation state is restored to a state generated before the occurrence time of the straggler and simulation is then resumed. The greatest effects on TW performance come from these rollbacks. The rollbacks may lead to an explosion of messages between the various entities. It is therefore important to choose a suitable communication library for any TW implementation.

In this thesis we have selected a suitable communication library for the APSIS TW implementation and studied rollbacks in APSIS with a synthetic workload called PHOLD.

We found that regarding the APSIS message size (156 bytes) PVM is a slightly better choice for APSIS communication than MPI. We also concluded that the APSIS message size might be increased up to 750 bytes, so that users have more freedom regarding their event sizes, while they lose very little in communication performance.

The PHOLD behaviour can be controlled by a number of parameters. While varying the number of PHOLD processes, we discovered that the topology in which the processes were put was rather important. Most rollbacks in this scheme emerged when processes were put in a ring topology. But the gain in parallelism when the number of processes is increased, while the number of total events are

kept the same, was more important than the increased probability for rollbacks, so that the turnaround time decreased substantially with increasing number of processes.

Increasing the number of initial events, led to the conclusion that the the extra resulting work and communication is more important than the decrease in the fraction of rolled back events, resulting in an increase in turnaround time.

Experiments with the way that processes communicate showed that processes which communicate as one big group exchanging many events work more efficiently than processes which work in distinct subgroups; they encountered fewer events which had to be rolled back during a run.

In the last test the computational grain per event was increased. We found hardly any change in rollback behaviour; only the variance of the measured quantities decreased with increasing grain.

# Chapter 1

## Introduction

Computer based simulation is almost as old as the computer itself. It has been used to study a wide range of real world and artificial systems, such as queueing networks, computer systems, population dynamics, VLSI circuits, fluid dynamics, airport traffic, etc. A simulation is a model (imitation) of a system, describing some of its features as well as possible. It tries to mimic that part of the system behaviour a particular observer is interested in. One of the widely used simulation paradigms is Discrete Event Simulation (DES). Central to this thesis are DESs which divide a problem to be simulated into a state and a number of entities and events. The state contains information about the simulation at any instance in simulation time. The entities are the objects of interest (e.g. a plane, creature, CPU, etc.). Events represent instantaneous changes in the state caused by the entities or by other events: e.g. the fact that a plane starts to descend, the fact that a creature goes to sleep, the fact that a CPU has finished processing an instruction, etc.

Small simulations can be executed on one (sequential) computer with little to no problems. However large simulations, for example the simulation of a large car factory, may not complete within reasonable time if executed on one sequential machine or may need a large amount of memory, which is not available on one machine alone. Such a large simulation can be executed within reasonable time and with enough resources available by processing it on a number of computing nodes at the same time, that is, in parallel. The large problem is divided into a number of smaller ones, each requiring only part of the computing requirements of the original problem. Each “sub-simulations” can then be executed on one of the computing nodes. The car factory, for example, could be divided into several disjoint objects representing the various production units. One or more of such objects could then be simulated on separate computing nodes.

Usually, the sub-simulations will interact, e.g. a car might be transported from one object to another in order to complete its assembly. This interaction may be implemented by using some kind of message passing system, that is, by sending messages over the commu-

nication hardware from a source sub-simulation (executing on one node) to its destination sub-simulation (usually executing on another node). In addition, each sub-simulation may proceed at different speeds through the global simulation: a synchronising method is needed to keep the information at one node consistent with the information at another node. Take the factory example. A car should not be transported from one object to another while the destination object is still executing at a simulation time which is smaller than the time at the source object — or else the future would be erroneously influencing the past. The destination object might still be occupied, for example, while a new car to be assembled is presented to this production unit, because it is assumed that the busy unit is at a time at which it should have completed its last job.

This thesis deals with *Time Warp*, a popular method used to synchronise DES programs which have been designed to be executed in parallel. We will first look at a brief introduction into systems and models, followed by a description of the goals and the contents of this thesis. The reader is expected to be familiar with basic Computer Science terminology.

## 1.1 Systems and models

In order to study a system a simplification of that system is needed. A system can be described as a set of objects of interest, the entities, whose actions and interactions change some sort of state ([12], [34]). The state describes the characteristics of the system at any particular time and is recorded through a collection of variables. We call that which causes the state to change an event. The entities, events and state represent only that part of the system which is important to the observer: because of this simplification some part of the system may be lost. It is the responsibility of the modeller not to throw away important system information.

When systems are modelled, they can be classified as discrete or continuous systems,

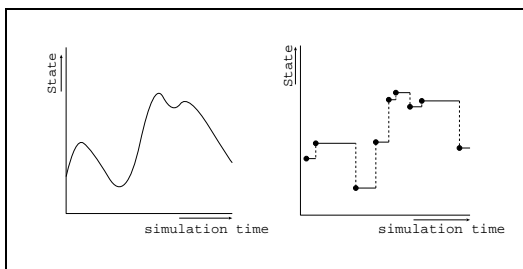


Figure 1.1: *Continuous (left) vs. discrete (right) system state plots.*

depending on the way the system state changes in time according to the modeller (figure 1.1). The state of continuous systems changes uninterrupted in time, i.e. plotting the state vector against time results in a continuous graph. A discrete system state, on the other hand, changes at certain, separated points in time: a plot of state against time results in a discrete graph.

An example of a continuous system is a moving car: its location in space (e.g.  $x$ ,  $y$  co-ordinates), an element of the state vector if relevant for the study objectives, changes continuously in time. A post office, with clients entering and leaving the office, can be described as a discrete system if we are only interested in the number of clients: the customers enter and leave the office at discrete points in time. It can

also be regarded as a continuous system, though, if it would be important how the customers move through the building. The classification depends on the way the modeller perceives the system to be modelled.

When studying a particular system, an experimenter can use the actual physical system or a model for his/her experiments. Whenever experiments with the real system are impossible or impractical, due to costs or times involved, an abstraction must be used instead. This abstraction, the model, should give a consistent and correct view of those aspects of the actual system which are important.

The abstractions can be categorised into physical and mathematical models. The mathematical models can be further subdivided into analytical solutions and simulations. It is possible to use a mixture of any of the models and the real systems, like in virtual reality or military combat training. In the latter case actions in real time of real tanks, bataljons etc. are used next to those of simulated troops, tanks etc. and combined into one real time simulation model. However, the focus in this thesis will be on some of the simulation models.

Physical models are models like a scaled down version of a plane or car, e.g. for measurements in a wind tunnel. Analytical and simulation models use formal reasoning techniques, that is mathematical deduction, to get to a representation of the system. This representation can be used to implement an exact analytical solution or a simulation model. If an analytical solution for the model is available and can be computed with reasonable costs (i.e. concerning times and resources involved), it is often better to use this method to study the system.

Take the example of growth models in population dynamics. We could use a mathematical representation using partial differential formulas like in one of the Lotka-Volterra [35] predator-prey models (figure 1.2 on the next page) to describe the relationship between predator and prey populations. The model can be studied by applying different model parameters to the analytical solution.

However, there are many cases where the complexity of the analytical solution is much higher, making it difficult or even impossible to efficiently implement and compute the model; in those cases a simulation can be used instead. Other reasons for using a simulation include model accuracy and certain types of model extensions. Sometimes using simulation results in a more accurate approximation of the real system. At other times some aspects, like spatial effects, need to be incorporated in the model which is much easier done in a simulation model or which is even impossible in analytical models.

Population dynamics, for example, could be simulated by using stochastic variables, i.e. variables that change in a "random" way according to a given distribution, to simulate the behaviour and interactions for the individual animals of both types. Instead of a birth rate for the whole population each individual animal would give birth with a certain probability. Other model parameters could be incorporated in the same way. The behaviour of the simulation model should closely approximate the mathematical model.

The basic Lotka-Volterra (LV) model describes two species [35], a predator and a prey species. While the predator feeds on the prey, the LV model assumes that the preys feed on a resource which is available in abundance and is not explicitly described by the model. When there is no more prey left, the predator species becomes extinct. If a predator catches a prey it gets fed and gains a number of calories. It is assumed that the prey population loses more calories than the predator population gains. This is described by an efficiency factor, which represents the calory flow per interaction.

The model has the following parameters (the calory flow is transformed into predator offspring): the predator population at time  $t$ ,  $x(t)$ , the prey population at time  $t$ ,  $y(t)$ , the excess death rate of the predator population if there are no prey,  $a$  ( $a > 0.0$ ), the excess birth rate of the prey population if there are no predators,  $b$  ( $b > 0.0$ ), the efficiency with which predators convert prey encounters into offspring,  $c$  ( $0.0 < c \leq 1.0$ ) and the rate of decrease of prey due to encounters with predators,  $d$ .

These are used in the following two formulas to describe the population dynamics of the two species:

$$\begin{aligned}x'(t) &= -ax(t) + cdx(t)y(t) \\y'(t) &= by(t) - dx(t)y(t)\end{aligned}$$

An exact analytical solution can be computed and is given by the following equation [2]:

$$b \ln \frac{x}{x_0} + a \ln \frac{y}{y_0} = cd(y - y_0) + d(x - x_0)$$

The time dependence of the original equations has disappeared in the analytical solution. This solution describes the relation between population size of the different species at any given time. The start population sizes of respectively the predator and the prey species are represented by  $x_0$  and  $y_0$ . In order to study species dependency, an  $x - y$  graph can be drawn using the solution given above for different model parameters.

Figure 1.2: *Basic Lotka-Volterra model.*

or more random variables in the model.

Most sequential algorithms can be ported to a parallel version and implemented on an MPP<sup>†</sup> or a distributed machine yielding a substantial amount of speedup [13]. In order to execute programs in parallel most of the time some sort of synchronisation method is needed. This certainly holds for a parallel DES implementation. A number of synchronising protocols have been developed. Time Warp is one of the so called optimistic protocols. For a more extensive

Due to the tremendous increase in computer power, simulation of large systems has gained more and more popularity among researchers. One of the simulation paradigms is Discrete Event Simulation (DES). While in other (simulation) models the simulation progresses through simulation time in time steps of  $\Delta t^*$  (time driven simulation), in event driven simulation it advances from event to event, skipping any periods of inactivity.

DES is a discrete, dynamic and stochastic model, in which time can take any value from  $\mathbb{R}$ . It is a discrete model in the sense that the state is also discrete in time analogous to the state in a *discrete system*. Changes in state, the so called events, can happen at any separate point in (simulation) time. It is dynamic because its state changes in time — *time variant* behaviour — as opposed to static models, such as a building floor plan. Generally, those models do not change over time — *time invariant* model behaviour. The model is called stochastic as the output of a DES is not fully determined when its input is known, due to stochastic behaviour of the DES entities and the existence of one

\*  $\Delta t$  can have a constant value or vary dynamically throughout the simulation depending on model parameters; in Simulated Annealing [34], for example,  $\Delta t$  depends on the varying temperature of the model. <sup>†</sup> Massively Parallel Processor

overview of computer simulation in general we refer to [34], [12] and [4].

## 1.2 Thesis goals

The goal of this thesis is to investigate the rollback behaviour of the APSIS Time Warp implementation. An attempt has been made to identify which settings influence execution speed and efficiency.

First, the PVM and MPI communication libraries which can be used for Time Warp have been investigated. Then one of those libraries has been selected to be used for the second series of experiments. Finally APSIS has been tested using a synthetic workload called PHOLD [8]. The tests have been performed by varying one PHOLD parameter at a time. An extensive description of the experiments will be given in chapter 6.

## 1.3 Thesis results

We found that regarding the APSIS message size (156 bytes) PVM is the better choice for APSIS communication. We also concluded that the APSIS message size might be increased up to 750 bytes, so that user have more freedom regarding their event sizes, while they loose very little in communication performance.

Regarding the PHOLD experiments we found that the choices made for the default PHOLD parameter values would minimise the probability for rollbacks on the DAS. However some rollback did emerge in the various PHOLD experiments. While varying the number of PHOLD processes, we discovered that the topology in which the processes were put was rather important. Most rollbacks in this scheme emerged when processes were put in a ring topology. But the gain in parallelism when the number of processes is increased, while the number of total events are kept the same, was more important than the increased probability for rollbacks, so that the turnaround time decreased substantially with increasing number of processes.

Increasing the number of initial events, led to the conclusion that the the extra resulting work and communication is more important than the decrease in the fraction of rolled back events, resulting in an increase in turnaround time.

Varying the communication patterns showed that processes which communicate as one big group exchanging many events work more efficiently than processes which work in distinct subgroups; they encountered less events which had to be rolled back during a run. However the differences were small, mainly due to the chosen default PHOLD parameters.

In the last test the computational grain per event was increased. We found hardly no change in rollback behaviour; only the variance of the measured quantities decreased with increasing grain.

## 1.4 Thesis structure

This thesis is organised as follows. In chapter 2 a survey of discrete event simulation will be given. Chapter 3 deals with Time Warp, the auxiliary mechanism for performing parallel discrete event simulation. Chapter 4 describes PHOLD, a workload model used to test Time Warp. Chapter 5 contains an overview of various communication libraries — PVM, MPI and FM — which can be used for parallel programming. Chapter 6 describes the various experiments that have been conducted. The results of these experiments are given in appendix B. Finally, in chapter 7 the experiment results will be discussed, conclusions will be drawn and recommendations for future work will be stated.

## Chapter 2

# Discrete Event Simulation

### 2.1 Sequential DES

In this thesis we consider Discrete Event Simulation (DES) as a model which divides a system into a set of distinct objects, the so called *entities*, scheduling and processing events [4]. For example, entities could model post-office clerks when a post-office is simulated or the CPU or a memory module in the case of a computer system. The characteristics of the system at any given time are maintained in a system state which is divided into disjoint parts and distributed over the entities. The entities interact by scheduling and processing time stamped events which represent instantaneous changes in the system state. In the post-office simulation the arrival of a customer could be an event and in the computer system it could be the arrival of a job to be processed. Each event can cause zero or more new events to occur after it has been processed by an entity. A job in the computer system example, for instance, could need to perform a memory operation. A new event is then scheduled which must be processed by the “memory” entity in order to simulate this operation. An outline of the DES algorithm is sketched in figure ( 2.1).

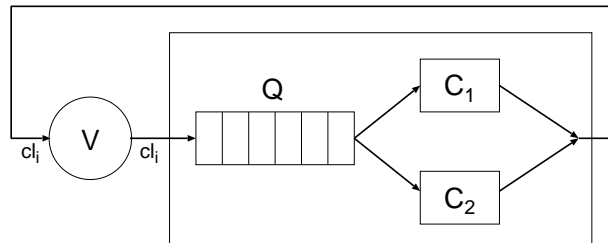
A clock keeps track of the simulation time to which the system has advanced. Every time an event should occur it is stamped with its scheduled (simulation) time of occurrence  $T_O$  and put in a priority queue: the *event list*. The events are removed sequentially from this list in order to be processed. Which event is taken first depends on the timestamps: the smallest timestamp will be chosen first, the one with the remaining smallest stamp next, etc. Every time an event has been chosen the clock is set to its scheduled time of occurrence.

Sequential DES algorithm
clock := 0;
event list(EL) := {initialization events};
<b>while</b> termination criterium is not met <b>do</b>
remove event $e$ with smallest $T_O$ from EL;
process $e$ ;
update event list (insert/delete events);
clock := $T_O(e)$ ;
<b>endwhile</b>

Figure 2.1: *Basic DES algorithm.*

A simple DES example describing a post-office is given in figure 2.2. The office model contains a (fixed) number of clients  $cl_i$ , a fixed set of possible client activities (e.g. posting a letter or sending a package, etc.), two clerks  $C_1$  and  $C_2$ , one queue  $Q$  for clients entering the office and a (small) village  $V$  where the clients stay while they are not in the office. The clients, clerks, queue and village make up the entities of the model, and customers entering, waiting in the queue, being serviced and leaving the office make up the events which take place during the simulation. Clients in the village may enter the office with some type of job. They have to wait at  $Q$  whenever both clerks are busy at the same time. The clerks can immediately start servicing the next client, after they have completed their task. When both clerks are free at the same time, they have equal probability to be chosen by a client for service. Each job arriving at a clerk has a random service time depending on the type of job and on the clerk. After a client has been serviced he/she leaves the office and goes into the village. The client stays there for some random time after which he/she will re-enter the office.

Events may occur at the same time: for example, at simulation time 3 the queue  $Q$  is empty and a customer entering the office (i.e. event 1,  $cl_1$ ) can go to either of the clerks without having to wait at  $Q$ . Subsequently the next event (event 2, client at  $C_1$ ) occurs immediately and is thus also scheduled to happen at time 3.



Event number	1	2	3	4	5	6
Event description	$cl_1$ at $Q$	$cl_1$ at $C_1$	$cl_2$ at $Q$	$cl_2$ at $C_2$	$cl_3$ at $Q$	$cl_1$ at $V$
Occurrence time	3	3	8	8	9	11
Event number	7	8	9	10	11	12
Event description	$cl_3$ at $C_1$	$cl_4$ at $Q$	$cl_5$ at $Q$	$cl_2$ at $V$	$cl_6$ at $Q$	$cl_3$ at $V$
Occurrence time	11	14	16	18	18	19

Figure 2.2: Simple DES: graph and sample event list of a simple post-office model.

## 2.2 Parallel DES

Parallel DES (PDES) can speed up\* the simulation by processing events in parallel. This is typically done on a multi-processor, using an MPMD† paradigm. However, several PDES methods make use of a message passing communication scheme and can, therefore, be used on distributed systems too. Only if this is needed, the distinction between parallel and distributed computations will be explicitly made.

According to [21] events can be processed in parallel if they are independent of one another. Two events  $e$  and  $e'$  are independent if processing them in any sequential order ( $[e, e']$  or  $[e', e]$ ) results in the same changes in state and event list.

With PDES the entities or groups of entities which process the events, can be distributed over multiple tasks (e.g. UNIX processes). One global simulation clock is used to keep track of the overall progress of the simulation, while different entities can be executing at different points in simulation time (figure 2.3(a) on the following page). Messages can be used to schedule events at other entities, if the results of a processed event affect other entities and/or cause new events to occur at the other entities. A message  $m(TAG, SRC, DEST, T_C, T_O)$  may be identified by five parameters: the type ( $TAG$ ) of the message, which is often the type of the resulting event when  $m$  is processed, the entity which caused it ( $SRC$ ), the one which will have to process it and to which it will be sent ( $DEST$ ), the simulation time it was caused ( $T_C$ ), and the point in simulation time at which the resulting event should occur ( $T_O$ ).

### 2.2.1 Causality

At some point an entity A which has progressed to simulation time  $T$  could receive a message from another entity B, containing an event  $e_B$  scheduled to occur at simulation time  $T_O(e_B) < T$ . So this event, which A has not processed yet, is scheduled to occur in the “simulation past” of A (figure 2.3(b) on the next page). If  $e_B$  is processed without any adjustments, that could lead to serious causality violations.

One causality problem can emerge when “younger” events — events with scheduled simulation time greater than  $e_B$ ’s time — depend on the state resulting from processing  $e_B$  itself. If these younger events have been processed before  $e_B$ , they will not have had the opportunity to use the changes  $e_B$  has made to the state. Causality, the fact that future events depend on events from the past, is thus wrongly ignored and that could lead to an erroneous simulation result for the program.

Another causality violation could emerge when younger events change the system state in such a way that  $e_B$  will be using a wrong state: some of the variables needed for processing  $e_B$  could have been altered. In this case the future influences the past.

---

\* Various speedups for PDES over DES are demonstrated in [20] and [21]. † Multiple Programs Multiple Data

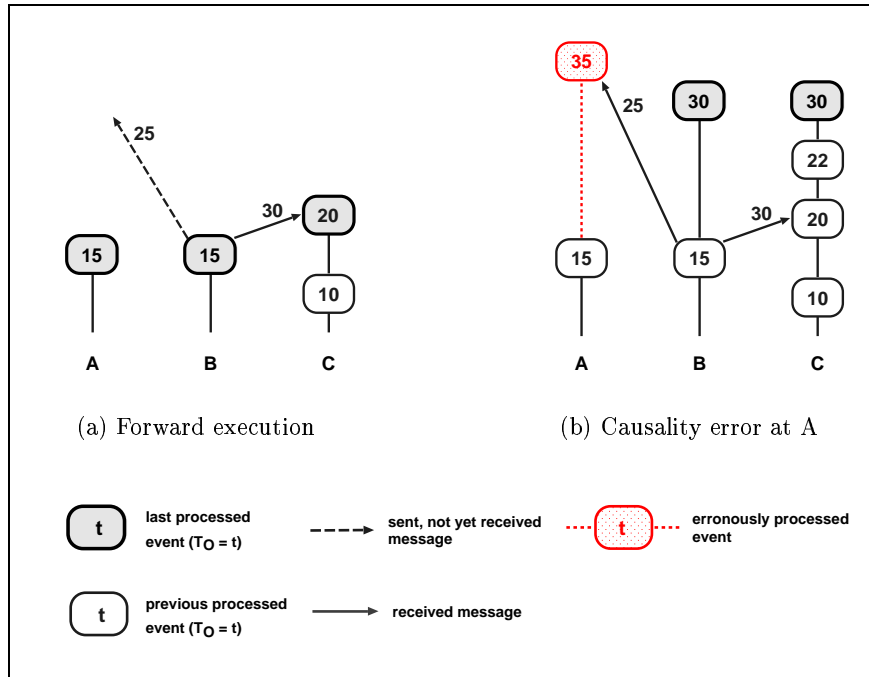


Figure 2.3: Sketch of a PDES run. At the start of the execution, fig. 2.3(a), a message is sent by B to A containing an event with  $T_O=25$ . When A receives this message, fig. 2.3(b), it has already (erroneously) advanced to simulation time 35.

So what is causality? In order to define causality the following notations will be used:

- $ent(e)$ : the entity event  $e$  lives in
- $T_O(e)$ : the simulation time at which event  $e$  was scheduled to occur
- $e \rightsquigarrow m$ : the occurrence of event  $e$  leading to message  $m$  being sent
- $m \rightsquigarrow e$ : the processing of message  $m$  leading to the occurrence of event  $e$

Assuming that each generated message can be uniquely identified\*, a definition of causality can be given as (see also [10]):

**Definition.**  $e_i$  causes  $e_j$ ,  $e_i \rightarrow e_j$ , iff for  $e_i \neq e_k \neq e_j$ :

- $ent(e_i) = ent(e_j) \wedge T_O(e_i) < T_O(e_j)$ , or
- $\exists e_k : e_i \rightarrow e_k \wedge e_k \rightarrow e_j$ , or
- $\exists m : e_i \rightsquigarrow m \wedge m \rightsquigarrow e_j$

---

\* for example, using a unique sequence number in combination with the id of the sending process

Note the second item in the definition, which allows for a sequence of causal related events: i.e.  $e_1 \rightarrow e_n$ , if there exists a chain of relations  $e_1 \rightarrow e_2 \rightarrow \dots \rightarrow e_{n-1} \rightarrow e_n$ .

This is a relative simple definition: there are cases possible where there is no real (intuitive) causality between events, even though they would be classified as such by the definition. Let e.g.  $e_i, e_{i+1}, e_{i+2}$  be three events living in one entity and let  $T_O(e_i) < T_O(e_{i+1}) < T_O(e_{i+2})$ . Suppose  $e_{i+2}$  is scheduled to occur as a result of processing  $e_i$ . Then according to the definition,  $e_{i+1}$  is also taken to be a cause of  $e_{i+2}$ . Processing  $e_{i+1}$  however does not have to result in  $e_{i+2}$  being scheduled, nor does any of  $e_{i+1}$  and  $e_{i+2}$  need to be dependent on state variables changed by the other event. Still, this is not a serious shortcoming of the definition as it includes the intuitive causality relationship and as most PDES methods treat causality as defined above.

Events which are causally related, will have to be processed in the correct order, i.e. in a sequential way. In general, others can be safely processed concurrently. Yet, care should be taken when events should occur at the same simulation time. Although they should be independent and therefore no restrictions should be imposed on the order in which they are supposed to be executed, this is not always the case: sometimes differing order of event execution can result in differing simulation outcomes for the same program input. For possible actions that can be taken to get a correct simulation in these cases see [41].

To avoid causality violation problems a number of schemes have been developed to ensure local causality, thus ensuring global causality[21]. Local causality is preserved if each task executes its events effectively in non-decreasing order. This constraint is sufficient, but not necessary. For example, two *independent* events at the same task could be executed out of timestamp order while the simulation still remains valid.

Traditionally, the schemes can be put along a scale ranging from purely conservative, via hybrid to purely optimistic approaches. Conservative schemes allow for the execution of only those events which do not lead to causality errors. Optimistic scheme do allow causality errors, but perform recovery steps to undo the errors when causality is violated. Hybrid approaches combine the conservative and optimistic methods in various ways.

### 2.2.2 Conservative protocols

One end of the scale is formed by the conservative approaches. Most conservative protocols, like the CMB (Chandy/Misra/Bryant) protocol [21] divide the simulation task into a set of communicating *Logical Processes* (LPs) each simulating one or more of the system entities [4]. Each LP progresses more or less independently through simulation time while it processes safe events only. Safe events are events in an LP's event list up to a simulation time  $T_{safe}$  for which the LP is guaranteed not to receive any message with timestamp  $< T_{safe}$  and which will consequently not result in causality violations when it is processed. If no such events are

available then an LP will block until other events become safe to process or new safe events arrive in its event list.

In order to determine if events are safe, the protocols use a fixed number of FIFO\* communication channels per LP: throughout the whole simulation, each LP can communicate with a (fixed) subset of LPs only. The timestamp of the last message received over a communication channel is associated with that channel.  $T_{safe}$  is the minimum of the timestamps associated with each channel; if each LP sends messages with increasing  $T_O$  no message will arrive with timestamp smaller than  $T_{safe}$ . In order to generate message streams with increasing  $T_O$  each new event  $e_n$  should be scheduled at  $T_O(e_n) > T_O(e_c)$  if it was directly caused by event  $e_c$ .

When an LP knows that processing the next event with timestamp  $T_O$  will certainly increment its local simulation time with  $la$ , the *lookahead*, then it can put a so called *null message*, with a timestamp of  $T_O + la$  on each of its outgoing channels. This increases the number of events which other LPs can process safely. Null messages carry the current simulation time only and are not related to the underlying simulation model.

In addition to increasing concurrency, these messages are also used for deadlock avoidance. A deadlock situation occurs, when a set of LPs are blocked, because they are waiting for safe events to arrive from one of the other set members. In other words, when  $LP_i$  is waiting for  $LP_{i+1}$ , which is waiting for  $LP_{i+2}$ , which is waiting for  $LP_{i+3}$ , etc., while  $LP_{i+n}$  is waiting for  $LP_i$  (a cycle containing  $n + 1$  LPs ( $LP_i, LP_{i+1}, \dots, LP_{i+n}$ )).

In order to avoid deadlock null messages are sent whenever an event is processed which does not generate a message for some other LP. If each closed cycle of communicating channels contains at least one message which increments its timestamp, the protocol guarantees deadlock free execution. This is always true if there is a minimum lookahead for the whole system and each message carries a non zero time increment. This deadlock avoidance protocol is straightforward to implement, but can result in an explosion of null messages during the simulation. Consequently a number of optimisations such as sending null messages on demand only, etc., have been proposed.

Another way to handle deadlock is by using a deadlock detection/recovery protocol. Such a protocol allows for deadlock to occur, but will detect any one occurring and will take appropriate steps to resolve the deadlock. It is for example always safe to process the event with the smallest timestamp in a deadlock situation. Most protocols use a centralised deadlock manager to resolve deadlocks.

Conservative methods have a number of advantages. First of all they are straightforward to implement: they have simple control and data structures. Another advantage is their relative small memory consumption compared to optimistic methods. Although both can suffer from memory overflow, this can be a much bigger problem for the optimistic methods. Yet another advantage is that performance is not degraded, when events are not uniformly distributed over

---

\* First In First Out: message are delivered in the same order they were sent

space and time. For optimistic methods, however, performance can degrade substantially in those cases.

Nevertheless, they also have some (big) disadvantages. Lookahead is often very difficult to determine, even though it is essential for (efficient) execution. Another disadvantage is the static communication topology, making it difficult to dynamically reschedule LPs over the various physical processors. Last, but not least, conservative protocols cannot fully exploit the parallelism available in the underlying simulation model; in some cases the protocols behave overly pessimistic even if causalities between events are rare. This is explained in more detail in the following section about optimistic methods.

### 2.2.3 Optimistic protocols

The other end of the causality enforcing protocol-scale is formed by the purely optimistic approaches. Entities here, keep on processing all available events until a causality violation is detected. The system is then rolled back to some earlier saved state where there was no causality violation (figure 2.4). It starts executing from that point in its past, but now using the correct execution order for the old events and the new event which caused the error (the so called *straggler* event).

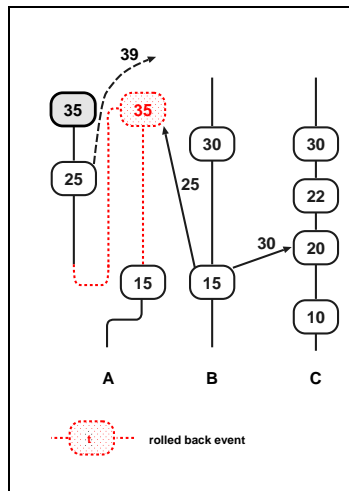


Figure 2.4: Rollback by A to simulation time 15. At local simulation time 35, A had to perform the rollback because of the arrival of a straggler message (with  $T_O = 25$ ). After the rollback A (re-)executes its events in the correct order.

Optimistic approaches have three major advantages over conservative ones. First, they allow for more parallelism and secondly they rely less on application specific information. Another advantage is the deadlock free execution of the optimistic methods: deadlock is intrinsic to all conservative algorithms due to blocking LPs when there are no safe events available.

When two events could affect each other conservative approaches will not take any risk and

will always process them sequentially, even if they actually are independent, i.e. if there is no causality relationship. Optimistic ones, on the other hand, will process them concurrently and in doing so will exploit all the parallelism available.

Assume two events  $e_a, e_b$  are available for processing, each in a different location in LP-space, i.e. each in a different LP, and also assume  $T_O(e_a) < T_O(e_b)$ . There is a possible dependency now, because  $e_a$  could e.g. schedule an event  $e_c$ , with  $T_O(e_c) < T_O(e_b)$ , while the computation of  $e_b$  could depend on variables altered by the computation of  $e_c$ . A conservative system has to wait for all results of  $e_a$  before it can process  $e_b$ , thus in effect, processing these events sequentially. An optimistic system however, can process both concurrently: recovery steps are only needed, when  $e_c$  really does emerge.

The second advantage of optimistic systems lies in their transparency. All conservative approaches need to know some application specific information (lookahead) essential for good performance. For one of the widely used conservative methods, CMB, lookahead is even necessary to make it operable. Although optimistic schemes can also use this additional information about the application for better performance, they do not depend on it. This simplifies the programmers task as he/she needs to know less about the internals of the simulation library used, such as synchronisation.

Optimistic models, however, can suffer from performance penalties, due to the need for recovery. These penalties depend on the number and (computational) size of the performed recovery steps. Another drawback is the amount of memory needed to make recovery possible. State saving and other bookkeeping steps can consume large amounts of memory. It is also very hard to use the protocols for implementing interactive simulators, but still much easier than when using conservative methods.

A number of optimistic PDES methods have been developed, like Time Warp, Moving Time Window, Breathing Time Buckets and Breathing Time Warp.

**Time Warp** (TW) is based on the concepts of so called logical processes (LPs) and virtual time (VT). Using TW the system entities are divided among a number of LPs. While each LP keeps track of its progress through virtual or simulation time a global clock (GVT) keeps track of the overall progress of simulation time. GVT is computed by using some global reduction algorithm. The LPs are allowed to progress independently through simulation time without any restriction. When causality is violated the appropriate LPs are rolled back. TW uses so called anti-messages in order to track down and annihilate erroneously sent messages. See also chapter 3 for a comprehensive discussion of TW.

**Moving Time Window** (MTW) is another protocol based on system entities distributed over LPs [24]. It uses a "window" moving over simulation time to advance the simulation. Only events  $e$  with  $T_O(e) < t + \Delta$  are processed in the window  $[t, t + \Delta)$ . Other events are postponed until one of the next time windows  $[t + (i - 1)\Delta, t + i\Delta)$ , (for  $i = 2, 3, \dots$ ). The protocol computes the next lower window edge by performing a global reduction operation

much like GVT computation in TW. MTW does not await the completion of all events within  $[t, t + \Delta)$  but attempts to move the window as soon as the number of events to be executed falls below a certain threshold. In this way it tries to avoid idle waiting time at the end of each window. By using a window, MTW limits LP progression, diminishing the changes for rollback. MTW favours simulation models with a low variation of event occurrence distances relative to the window size  $\Delta$ : i.e. it assumes an approximately uniform distribution of events over simulation time. If this is not the case MTW performs badly. Another drawback is determining the length of the optimal  $\Delta$  for a given simulation.

In **Breathing Time Buckets** (BTB) the event list and system state are distributed over a number of tasks (LPs), which can be processed in parallel [36]. Each LP processes events optimistically, while it uses a conservative policy when new messages should be sent. Messages are sent only after all events in the current so called *time bucket* have been processed. Each bucket contains the maximum number of causally independent events (which can be executed concurrently). The bucket size is determined dynamically with the aid of the so called *event horizon*. The local event horizon for an LP is the minimum timestamp of any new scheduled events as a result of processing current events. The global (or true) event horizon GEH is the minimum of all local horizons and defines the lower time edge of the next time bucket — this is also the upper time edge of the current bucket. After GEH has been determined by some global reduction scheme — usually an update is requested after each LP has passed its local horizon — the LPs can start sending out the postponed messages (figure 2.5 on the next page).

Using BTB no anti-messages are needed: every rollback is local to the LP and does not affect any other LPs. Upon rollback the only steps which need to be taken, are restoring an older state and discarding scheduled messages. This is a big advantage over Time Warp which can suffer from an explosion of anti-messages, degrading overall performance. However in order to remain efficient, BTB needs to process enough events on the average per time bucket. Depending on the underlying simulation model, this is not always possible. This is no problem for the original TW paradigm, which does not use buckets and can process events with unlimited optimism.

**Breathing Time Warp** (BTW) aims to combine the best of TW and BTB. Steinman [37] notes that the main problem suffered by TW is the rapidly growing possibility of cascading antimessage explosions when runaway LPs get more and more out ahead of the rest. Events with timestamp far away from the current GVT tend to have a larger chance of being rolled back than events close to GVT, due to straggler events sent by slower LPs. So BTW slows down the send out policy of messages at the faster LPs by using BTB. This will prevent the network from getting flooded with a large number of messages in case of a rollback. In this way both CPU power and network bandwidth are saved, as fewer messages will be processed. A second mechanism used is throttling of optimism by temporarily stopping event processing.

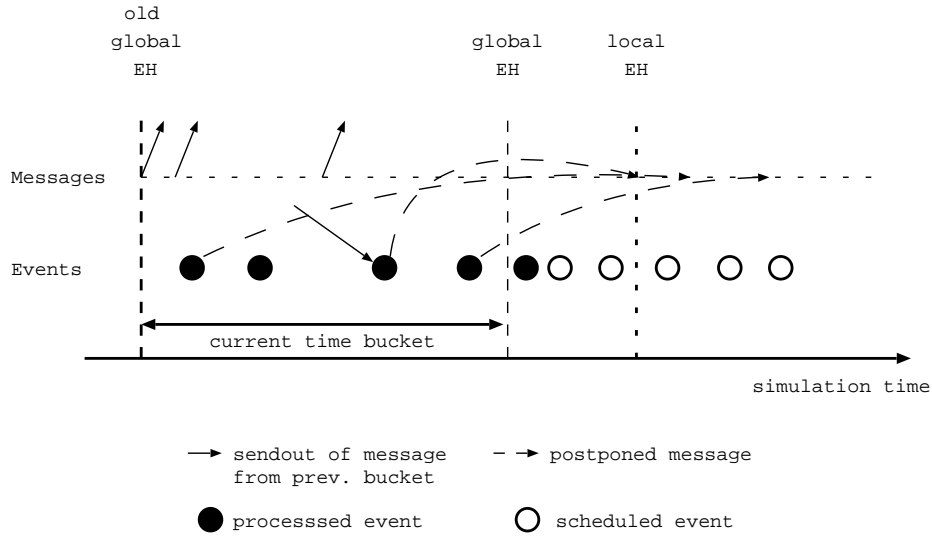


Figure 2.5: *Breathing Time Bucket: events are processed optimistically, while new generated messages are postponed until the global EH (event horizon) has been determined, using the local EH of each simulation node. New messages for the current node need not be postponed and can be optimistically processed.*

The BTW algorithm [37] can be described as follows: before the next GVT is determined, the first  $N_{risk}$  (a user selected run-time parameter) events will be processed locally on each LP using the TW mechanism. After that, events are processed using BTB. When  $N_{gvt}$  events have been processed, or when the event horizon has been crossed, each LP requests a GVT update; GVT is updated after all LPs have made that request. Whenever an LP processes  $N_{optimism}$  events it stops event processing until the next GVT value is determined.

See [4] for a comparison of some of these protocols; Time Warp will be discussed in the next chapter.

## Chapter 3

# The Time Warp mechanism

This chapter describes the Time Warp (TW) mechanism. First the theory behind the basic scheme will be described. Then some possible improvements to this scheme will be given. Finally, the implementation used for this thesis, APSIS, will be examined.

### 3.1 Basic TW algorithm

When using *Time Warp* the system to be simulated is divided into a number of tasks called Logical Processes (LPs). These LPs can be processed on a parallel machine, allowing one or more LPs to be executed per processing node (i.e. CPU or workstation). The LPs process and schedule events according to the basic DES algorithm given in figure 2.1 on page 9. Events are scheduled at other LPs by sending messages for each such event.

Special synchronisation (using rollback) is needed to preserve causality (see also section 2.2.1). In TW this synchronisation mechanism is based on the notion of Virtual Time [14]. Each LP keeps track of its *local virtual time* (LVT). This is the simulation time to which the LP has progressed and is equal to the timestamp of the last processed event. Because of the rollbacks, the LVTs jump back and forth through simulation time, while each LVT does eventually progress ahead into the simulation future.

An LP has to perform a rollback whenever it receives a message — a so called *straggler* message — scheduling an event with a timestamp which is smaller than the LVT to which the receiving task has progressed. It restarts using the most recent state which was saved at a (simulation) time before the straggler's timestamp and has to “unsend” any output message\* since the last time it (re)produced the saved state. To accomplish this, for each output message, an anti-message — also called negative message, as opposed to normal, “positive” messages — must be sent to the appropriate tasks. When another task receives such a negative message it annihilates the dual, the corresponding positive event. If it already

---

\* a message sent to another LP

had processed the positive event, it also needs to perform a rollback to the most recent state with a save time smaller than the event's timestamp. All rollback propagations eventually terminate, while all erroneous events are unsent recursively [14].

Another important quantity is the *Global Virtual Time* (GVT). It is the simulation time to which the simulation as a whole has progressed, i.e. the time up to which the total simulation can be considered to be valid. It is also the smallest time to which any LP could need to rollback, because all timestamps are generated in non-decreasing order. Besides indicating simulation progress, GVT is also important for committing irrevocable actions and for memory management. All irrevocable actions such as (file) I/O and error reports have to be postponed until the GVT has passed the simulation time at which those actions had to be performed.

Basically, GVT is equal to the smallest timestamp in the simulation. In addition to the LVT at each LP, the  $T_O^*$  of each unprocessed or not yet completely processed event should be taken into account when the GVT is calculated: each such event could lead to rollback. Determining the minimum VT at a particular (wallclock) time is therefore not so trivial: the event with the smallest  $T_O$  could be hidden while it is in transit in some communication channel between the various nodes.<sup>†</sup> So, in addition to the  $T_O$ s in the various event lists,  $T_O$ s of transient messages should also be considered.

Care should also be taken when collected LVTs and information about transient messages are combined. Usually, messages initiating a particular GVT-computation will be received at each LP at different instances in (real) time. Each snapshot to determine this information will (thus) have been taken at different points in real time and the reply messages of each LP will also be received at different times. This is due to two phenomena: real time clock differences and communication latencies. Because usually the real time clocks of each computing node will differ substantially it is impossible to determine LVTs at a particular instance in time, even if LPs receive the messages initiating GVT computation or send local GVT information at the same instance in real time (according to the local clocks at each LP). And because of the uncertainty of latency length in communication systems in combination with the use of an asynchronous communication scheme, snapshot initiating and reply messages will be received at different real time moments. So, snapshot messages can be incorrectly combined if no precautions are taken.

Several GVT calculation algorithms have been proposed. They can be classified into two kinds of schemes according to [11]: acknowledgement schemes and message numbering schemes. The first kind of schemes use acknowledgement messages to keep track of transient messages during the GVT reduction, which substantially increases the number of messages in the network. The second kind of schemes do not require an LP to acknowledge received messages. Instead bookkeeping is done by numbering each produced message.

---

\* The scheduled time of occurrence; however some extensions to TW use a different definition for GVT, like using  $T_C$ , the time at which a message was caused instead of  $T_O$  (see also section 3.2) <sup>†</sup> when we are using asynchronous communication

### 3.1.1 Space

Due to the (frequent) state saves needed for rollback, TW can consume large amounts of memory. Memory consumption in TW depends mainly on four data-structures used at each LP: an *input*, an *output history*, a *state* and a *gvt queue*. Messages which schedule events for the current LP are put in the *input queue*. It contains messages from its past (from GVT up to the current LVT)\*, its present (the current LVT) and its future (timestamps beyond the current LVT). The local *event list* is implicitly given by this queue. The *output history queue* contains negative messages for annihilating prematurely sent positive messages. The *state queue* contains state information from its past and present. The *gvt queue* contains the reply messages sent by each LP to determine new GVTs.

The standard way to minimise memory consumption is by periodically performing so called *garbage or fossil collection*. Every now and then old states and messages will be deleted from memory when they become obsolete. Saved states and messages become obsolete when GVT has progressed beyond their simulation time (stamp), as future rollbacks will not use any of these states or messages. However, garbage collection alone may not be sufficient: some extensions to TW are needed to reclaim extra memory when the simulation runs out of memory.

The *output history*, *state* and *gvt* queues will have both FIFO and LIFO<sup>†</sup> behaviour: new elements are added at the head of the queue, while remove operation take place at both the head (rollback) and at the tail (garbage collection). Both operations will have a  $\mathbf{O}(1)$  behaviour per element.<sup>‡</sup> The input queue is a priority queue in which new elements may be added anywhere in the queue according to their timestamp. While future messages could be stored in a (heap) tree based queue, present and past messages are best stored in an ordered linear queue, because a rollback or garbage collection may roll back/delete a number of adjacent elements within a particular timestamp range. Dequeue operations from the linear queue will have  $\mathbf{O}(1)$  complexity and enqueue operations  $\mathbf{O}(n)$ , while in the heap tree both adding and removing elements will be  $\mathbf{O}(\log n)$ . Of course using a combination of both data-structures will give extra overhead.

### 3.1.2 Time

TW performance is in general difficult to predict, but a number of observations can be made. First of all, cascading rollbacks can lead to an explosion of messages, which could consume considerable amounts of bandwidth, slowing down the communication network. In addition a lot of work has to be undone, leading to inefficient use of resources<sup>§</sup> due to rolled back events which have been processed prematurely and which might be re-executed later on during

---

\* needed for possible rollbacks    † FIFO: First In First Out; LIFO: Last In First Out    ‡ using a linked list, with pointers to the head and tail of the queue    § like memory, CPU cycles, etc.

the simulation. Of course, small but (very) frequent rollbacks will also lead to performance degradation for the same reasons.

Secondly the probability for cascading rollbacks gets bigger as LPs progress in a more differing pace through simulation time. If LPs get too far behind others then the probability that a faster LP with a larger LVT will receive a straggler message could increase substantially (depending on the communication patterns of the LPs). A faster LP is an LP with a higher time rate ( $\frac{\Delta T}{\Delta t}$ ), the change of simulation time per real time unit at that LP. This rate depends on the hardware used (speeds of the processors, memory, communication system, etc.), policies of the operating system (like process scheduling), the number of processes per node (and their behaviour), the implementation choices for the TW kernel and the simulated model, and properties intrinsic to the underlying simulation model (computational granularity, communication, etc.) — which could change dynamically during the simulation. If we assume that all LPs and processing nodes are homogeneous, that is that each LP operates in the same way using a similar implementation, that all the computing nodes — hardware and supporting software — and communication channels are equal and that each node executes only one LP exclusively and no other software,\* then the simulation time advance of each LP will be mainly influenced by the properties of the underlying model. The underlying model determines the computational granularity, that is the amount of work per event, communication frequency and partners, and the way events are distributed over (LP) space and over (simulation) time, that is how many events each LP contains and at what (simulation) times they are scheduled to occur.

Other important influences on performance are the time lost by: enqueueing and dequeueing elements in the various queues, sending messages to other LPs, saving and recovering states and (periodically) calculating the next GVT. These actions do not only cost execution time, but will also increase the risk of LPs getting farther behind on others and will thus increase the probability for (cascading) rollbacks.

## 3.2 Performance enhancing extensions

The TW algorithm sketched in the previous section can be augmented by a number of refinements, which aim to minimise some of the penalties suffered by the basic scheme [4]. Often a mixture of these refinements is used when TW is implemented.

### 3.2.1 State saving

State saving can be done in a number of ways. Originally each time a message has been processed a full state save is performed. With *incremental state saving* only state variables

---

\* For example, these conditions may be easily met when we use a multi-processor.

$s_i$  affected by a state change are logged, thus minimising memory consumption. The state recovery becomes more complex, however, as the desired states have to be reconstructed by following back the appropriate state saves.

Instead of saving the state when each message is processed, with *periodic state saving* state logging happens periodically whenever a specific number of messages have been processed or some (real) time period has passed since the last save. This is a major gain concerning state saving overhead (time and memory space involved). Rollback overhead, on the other hand, will increase, as the appropriate state must be recomputed from the most recent saved state older than the straggler message.

### 3.2.2 Reclaiming memory

*Message sendback* was the first approach used for freeing extra memory in addition to memory de-allocation by garbage collection [4]. Whenever an arriving message causes the TW system to run out of memory, the input queue is used to free memory by sending some of the saved messages back to its sender; these need not include the one just received. Younger messages are returned first, since it is more likely for messages with larger timestamps to be rolledback and since annihilating them will cause less side-effects like cascading rollbacks, than older messages.

In contrast to the previous method *Gafni's protocol* reclaims memory by freeing memory from any of the three queues: input, output history or state queue [4]. The protocol is invoked whenever memory overflow occurs due to an arriving input message or the logging of a state or output message. The youngest element (message or state) is selected for annihilation irrespective of its type. If an *output message* is selected, it is deleted, a corresponding antimessage is sent and the state saved just before sending the original message is restored. If an *input message* was selected it is sent back to its sender where it is annihilated with its dual and may lead to rollback at that LP (like in the Message sendback method). If a *saved state* was selected it is deleted and may be recomputed later on, upon rollback.

A memory-performance trade-off is made in both mechanisms: performance may degrade due to additional messages and rollbacks, caused while memory is reclaimed. On the other hand supplying more memory will lead to less invocations of the schemes and may thus substantially improve performance.

### 3.2.3 Rollback

Originally *aggressive cancellation* (AC) was the strategy used to implement TW rollback ([14], [23]). More recently another mechanism has been proposed called *lazy cancellation* (LC). The two methods differ with regard to the moment, when message cancellation is actually performed. In AC, antimessages for each message  $m$  which has to be cancelled, are sent immedi-

ately after a causality error has been detected. In LC, on the other hand, the propagation of an antimessage for  $m$  is delayed until  $LVT = T_O(m)$ . If this resimulation produced a message  $m'$  which is equal to  $m$ , no antimessage needs to be sent for  $m$ . In this way unnecessary cancelling of correct messages is avoided. Drawbacks are, however, the extra memory and bookkeeping overhead (potential antimessages must be maintained in a rollback queue).

AC can out perform LC and vice versa; it cannot be predicted which strategy works best. For some applications using AC results in better performance, for other LC outperforms AC.

### 3.2.4 Lazy Re-evaluation

In *lazy re-evaluation* (LR), when at  $LVT = lvt$  a straggler message  $m$  has been received with  $T_O(m)$ , the computed states during the simulation period  $[T_O(m), lvt]$  will not be discarded after rollback until the simulation has progressed again beyond  $lvt$ . Should the re-computation after rollback reach a state that exactly matches one of the logged states and should all the messages waiting to be processed correspond with the ones needed at the matching state then immediately a jump is performed to  $lvt$ . Unnecessary recomputing correct states is avoided in this way. Again, as for LC, additional memory and bookkeeping overheads will be suffered when using this optimisation. Another drawback is the TW implementation complexity, which will increase substantially.

### 3.2.5 Optimism control

It is generally assumed that the amount of optimism allowed has a large impact on performance. Consequently, a large number of optimism controlling protocols have been proposed. [3] reports the following *non-adaptive* protocols: allowing the execution of only those LPs that lie within a specified (static) window in virtual time, limiting the number of events each LP may execute beyond GVT, generating artificial rollbacks and using lookahead to limit optimism. Many of them use one or more "tuning" parameters (like window size in a time-window based schemes) without clearly specifying how users should set these parameters. In addition these methods do not adapt to dynamically varying workloads. In most cases, it is not clear which values to use for the parameters in a specific application, nor how the parameters should be changed during the program's execution.

*Adaptive* protocols show more flexibility. Early efforts to such protocols include a scheme which manipulates the process scheduling algorithm to reduce the frequency of executing processes that have experienced many rollbacks in the past and a protocol which schedules processes with LVT within a time window which is automatically adjusted by monitoring the execution. More recently introduced protocols include a scheme which checks the inter-arrival times (both in simulation and in real time) of received messages to predict whether an event arriving later on in one of the input channels will cause a rollback. Another protocol uses a

probabilistic, adaptive scheme which predicts the arrival pattern of incoming messages to an LP using certain time-series based forecasting, and blocks the LP according to the forecast. A third protocol relies on the availability of information on some aspects of the global simulation state to estimate an error potential for each individual LP; each LP may be blocked based on its error potential.

### 3.2.6 Message aggregation

In addition to messages generated to schedule events at other LPs for the underlying model, other messages are needed by the TW kernel for synchronisation (anti-messages) and GVT-computation. [26] note that schemes developed to minimise communication overhead focus on minimising the number of generated messages, like using improved partitioning strategies to reduce the number of messages generated by the underlying model, or methods to reduce TW-kernel messages (e.g. lazy cancellation, efficient GVT computation, optimism control). In the *dynamic message aggregation* scheme proposed by [26] send-requests by an LP are not carried out directly, but are aggregated for each destination LP. Periodically, aggregated messages are sent to their destination using one physical message per destination, thus reducing communication overhead. The physical message is sent, each time the first of the aggregated messages gets a certain age. This age can be determined by using a fixed "age-window". Alternatively, this window can be varied according to the message arrival rate of the LP.

The number of rollbacks may increase, because messages are delayed: LPs could receive delayed messages with time stamps far into their past causing the LP to rollback. This might not have been the case if the messages were received in time. However the communication overhead may be reduced to such an extent that in the long run information travels faster between the LPs thus reducing rollback lengths or causing less rollbacks, because LPs are more tightly coupled.

In studies carried out by [26] significant performance improvements — up to 30% in the best case — have been claimed for certain simulated models, while a large number of messages were aggregated without increasing rollback behaviour.

## 3.3 A TW implementation: APSIS

In this thesis the *Amsterdam Parallel Simulation System* (APSYS), a Time Warp implementation\* developed at the University of Amsterdam has been examined. APSIS has been designed with Asynchronous Cellular Automata (ACA, [30]) models in mind. ACA models are based on a discretisation of space into a number of cells. Each cell has its own state which changes asynchronously with respect to other cells by a number of rules local to the cell. Applications implemented using APSIS should consist of one or more logical processes (LPs), each

---

\* written in the C programming language

managing a number of cells. APSIS is based on the basic TW kernel using three extensions: *Incremental State Saving*, a *static (virtual) time window* scheme and *direct event cancellation*. *Incremental state saving* is used, because the state of each LP in ACA applications is often large while most of the time only few state variables change. A *static Time Window* scheme is used in order to limit optimism. Whenever the difference between LVT and GVT is greater than the time window allows, event processing is skipped: newly received positive messages are put in the input queue and others (GVT and negative messages) are processed in the usual way.

The third extension to the basic scheme is *direct event cancellation*. The simulation model may need to cancel, or preempt, previously scheduled events. For example, suppose we are modelling two kind of fish swimming in a pool: prey fish and predator fish feeding on that prey. Suppose that each predator fish will starve if it cannot catch a fish within some specific time span. An event might be scheduled to cause one of the predators to die. The predator should die after the death-event has been processed, if it did not catch any prey before the indicated time span has elapsed. However, if it does catch a prey, the death-event should be cancelled and a new death-event should be scheduled to occur at a later time.

A call to *tw\_cancel()* removes the cancelled message from the input or output history queue and saves a copy of the message in the cancel queue. Furthermore, if the cancelled message was an output message then a corresponding negative message is sent to annihilate its dual at the destination LP. A pointer (CQP) in the current input queue element\* is set to point to the new cancel queue element. The standard rollback mechanism is used to undo the effects of a cancelled event: negative messages are sent for each output message caused by the cancelled event. Message processing will then be performed in the usual way. However, the standard rollback mechanism alone is not sufficient: when a cancel message *cm* itself should be rolled back then the message cancelled by *cm* must be resent. When a rollback occurs, containing messages for which CQP is set, then the CQP pointer is used to re-send saved positive messages to undo the cancellations.

### 3.3.1 APSIS queues

APSYS uses 5 queues: an *input*, *output history*, *state*, *cancel* and *gvt* queue. All the queues are implemented as doubly linked circular lists using pointer structures. Each element in the *input queue* contains an input message and 3 extra pointers into the other queues: into the output history queue (OQP), into the state queue (SQP) and into the cancel queue (CQP). Whenever an input queue element, *iqe*, is processed and one or more output messages are generated as a result, the OQP of *iqe* is set to the last output message in the output history queue. If state information is saved, a copy of the information is inserted at the head of the state queue and the SQP of *iqe* is set to point to this state queue element. A pointer *cim*

---

\* which is the message directly causing the cancellation

is used to keep track of the next pending input message in the input queue. Each element in the *output history queue* contains a (negative) output message. The *state queue* contains the LVT at which the state save was requested, the memory location of the data to be saved, the size of the data (in bytes) and a copy of the data itself. The *cancel queue* contains the positive messages which have been cancelled by the user (a pointer to the appropriate cancel queue element is set in the input queue element causing the cancellation). The *gvt* queue is only needed at one (root) LP because of the GVT computation scheme used by APSIS (see section 3.3.3).

### 3.3.2 APSIS communication

The communication primitives used by APSIS can be implemented using any communication library desired, like PVM or MPI. We refer to chapter 5 for a short description of low level APSIS communication primitives and the libraries used to test APSIS. Each message has a fixed size: it contains a per LP unique sequence number, the identifiers of the source and destination LPs, 128 bytes of free space which will contain the user event data, the number of bytes used by the user event,  $T_O$ ,  $T_C$  and a field to identify the type of message (e.g. positive, negative, GVT, etc.). When the user issues a call to *tw\_send()* in order to send a message (event) to another LP, APSIS immediately sends this message to the other LP and puts a negative copy of the message in the output history queue. If *tw\_send()* is called for a message for the current LP, then it is immediately put in the input queue without using the communication subsystem.

Currently, LPs must be put in a torus communication topology for correct execution; this is done by calling *tw\_2dtorus\_create()*. This function requires the caller to specify the  $x$  and  $y$  dimension of an LP grid which will be converted into a torus topology by the kernel. These values must be chosen such that if  $n$  is the number of LPs, then  $x * y = n$ . Each LP has four fixed communication channels pointing to its four neighbours: a North, South, East and West channel (figure 3.1 on the next page). If  $n = 1$  then the channels of the only existing LP,  $LP_0$ , will point to the LP itself. If  $n$  is prime and  $y = 1$  then the West and East channels of each LP  $LP_i$  will point to its left and right neighbour respectively, while the North and South channels will point to  $LP_i$  itself. A similar scheme is used for  $n$  prime and  $x = 1$  with the North and South channel pointing to the upper and lower neighbour and the West and East channels to the LP itself.

### 3.3.3 GVT computation algorithm

APSYS uses a GVT computation scheme based on the algorithm constructed by Bauer and Sporrer [1], which is one of the message sequencing schemes. This algorithm assumes FIFO\*

---

\* messages are delivered in the same order they were sent

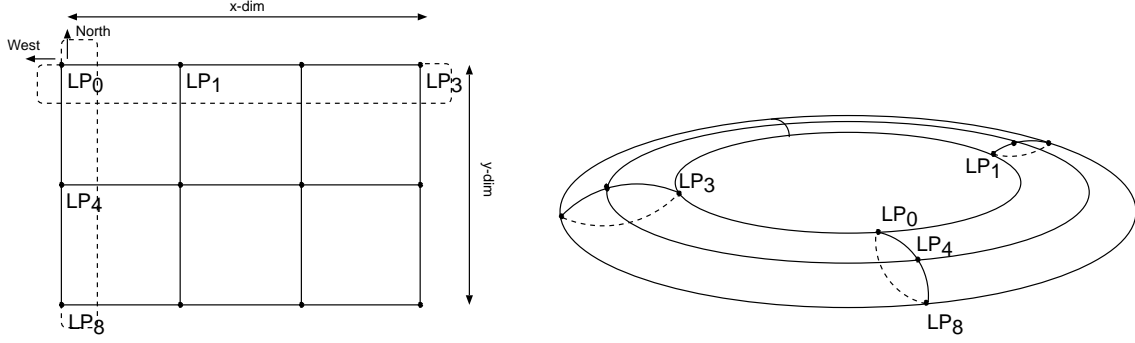


Figure 3.1: *APSYS torus communication topology for 12 LPs put in a  $4 \times 3$  grid (only  $LP_0$  and its neighbours are labelled here).*

communication channels which are error-free. Each LP frequently sends the information needed to determine the next GVT to one central GVT calculating LP,  $LP_0$ . This LP can also perform "normal" simulation functions, which are part of the simulated system, as the GVT computation itself needs relative few CPU cycles. It is not necessary to obtain GVT computation information at the same time. An LP can send its part periodically, whenever it finds it suitable. In this way there is minimal interference between the actual GVT computation and the simulation at other LPs.  $LP_0$  watches the LVTs and the timestamp of events in transit, i.e. events still in the communication channels, but not yet received.

Bauer-Spörrer claim their estimated GVT to be close to the real, theoretical, GVT, despite the fact that only a lower bound is calculated and the fact that the information needed can originate from different moments in real time.

Their estimate for GVT,  $\widehat{GVT}$ , is given by:

$$\min(LVT^a(t_{II}^a), LVT^b(t_{II}^b), T_{min}^{b \gg a}(t_I^b, t_{II}^b), T_{min}^{a \gg b}(t_I^a, t_{II}^a)) \leq \widehat{GVT}(t_{II}^a) \leq GVT$$

for every pair of processes  $LP_a, LP_b$  that share a communication channel. The symbols in this formula have the following meaning:

- $t_i^a$ : the real time at  $LP_a$  when the simulation time snapshot  $i$  was taken
- $LVT^a(t_i^x)$ : the LVT of  $LP_a$  at time  $t_i^x$
- $T_{min}^{b \gg a}(t_{i-1}^x, t_i^y)$ : the minimum simulation time  $T_O$  among the messages sent over the communication channel from  $LP_b$  to  $LP_a$  during the time period  $[t_{i-1}^x, t_i^y]^*$

In addition to the last two quantities each LP has to send the following two pieces of information in order to get a consistent snapshot:

---

\* that is the minimum  $T_O$  of the messages sent since the last time a GVT information messages was sent

- $n^{a>b}(t_i^x)$ : the number of messages sent from  $LP_a$  to  $LP_b$  in the time interval  $[0, t_i^x]$
- $r^{b<a}(t_i^x)$ : the number of messages received by  $LP_b$  from  $LP_a$  in the time interval  $[0, t_i^x]$   
(this is also the sequence number of the last received message up to  $t_i^x$ )

Figure 3.2 illustrates how this information can be used to determine the next GVT. First  $t_I^a$  and  $t_{II}^b$  are determined for channel  $a > b$ , such that  $n^{a>b}(t_I^a) \leq r^{b<a}(t_{II}^b)$ , i.e. all messages sent by  $a$  at time  $t_I^a$  have been received by  $b$  at time  $t_{II}^b$ . Next  $t_I^b$  and  $t_{II}^a$  are determined for channel  $b > a$ , such that  $n^{b>a}(t_I^b) \leq r^{a<b}(t_{II}^a)$ .

Finally, after computing  $T_{min}^{a>b}(t_I^a, t_{II}^a)$  for channel  $a > b$  and  $T_{min}^{b>a}(t_I^b, t_{II}^b)$  for channel  $b > a$ ,\* the partial  $\widehat{GVT}$  can be computed for  $LP_a$  and  $LP_b$ . The new  $\widehat{GVT}$  along all channels can be determined by computing the minimum of all partial  $\widehat{GVT}$ s.

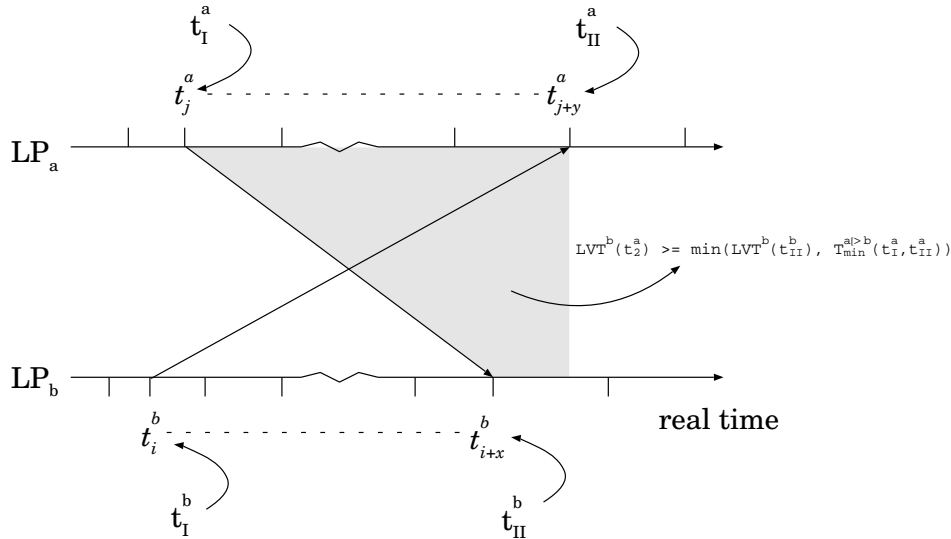


Figure 3.2: Lower bound for  $LVT^b$  in  $[t_{II}^b, t_{II}^a]$  if  $n^{a>b}(t_I^a) \leq r^{b<a}(t_{II}^b)$  and  $n^{b>a}(t_I^b) \leq r^{a<b}(t_{II}^a)$ .

In APSIS  $LP_0$  periodically computes the new  $\widehat{GVT}$  and sends this to the other LPs.<sup>†</sup> Upon receiving a message containing the new  $\widehat{GVT}$  each LP updates its copy of the  $\widehat{GVT}$  and send the appropriate information back to  $LP_0$  for a future  $\widehat{GVT}$  computation.

$LP_0$  logs for each LP separately, the information messages in the order they were received, so that they are sorted in the order the snapshots were taken at each LP.<sup>‡</sup>

\* for example, if  $t_I^b = t_i^b$  and  $t_{II}^b = t_{i+x}^b$  then  $T_{min}^{b>a}(t_I^b, t_{II}^b) = \min(T_{min}^{b>a}(t_{i-1}^b, t_i^b), T_{min}^{b>a}(t_{i+x-1}^b, t_{i+x}^b))$  <sup>†</sup> The length of this period is hard coded into the APSIS kernel. <sup>‡</sup> recall that we assume error-free, FIFO communication channels

### 3.3.4 The APSIS primitives

The TW engine is implemented as a library which should be linked with the user application at compile time. The virtual time representation, that is whether to use *double* or *integer* virtual time values, can be specified by defining respectively *VTIME\_FLOAT* or *VTIME\_INT*. The application can then use *vtime\** to appropriately refer to the virtual time type (for example to define a variable which will hold some virtual time value). The local and the global virtual time can be determined by calling respectively *tw\_lvt()* and *tw\_gvt()*. The number of LPs participating in the simulation must be given as one of the command line arguments; the command line should also be passed to *tw\_init()*. This function initialises the TW engine and returns with the number of LPs which have been started. The APSIS identification number of the current LP can be determined by calling *tw\_pid()*. Figure 3.3 illustrates how APSIS is typically used. Each LP starts processing events in a loop, after it has executed code to initialise the TW engine and communication topology.

```

initialise TW engine;           /* with tw_init() */
initialise comm. topology;     /* tw_2dtorus_create() */
schedule initialisation events; /* with tw_send() */
while tw_recv(e) > -1 do     /* while events pending */
  if termination criterium is not met then
    save state variables if needed; /* tw_state_save() */
    process e;
    schedule new events if needed; /* tw_send() */
  else
    break; /* exit while loop */
  endif
endwhile
terminate; /* tw_finalize() */

```

Figure 3.3: A typical APSIS LP.

At initialisation time the optimism limiting time window length, *vt\_window*, is read from an APSIS resource file in the directory where the APSIS application has been started. The communication topology is initialised with a call to *tw\_2dtorus\_create()*. Using a loop iterating over the pending events is compulsory in order for rollback and other TW functions to be performed correctly. New events can be scheduled with *tw\_send()* and state variables can be saved with *tw\_state\_save()*. Events can be preempted (cancelled) by calling *tw\_cancel()*. The next pending event is accessed by calling the primitive *tw\_recv()*. This primitive will also execute some of the other TW functions, like receiving and buffering any incoming message which schedules a new event, etc.

A loop is executed at each *tw\_recv()* call in which the TW kernel tries to receive as many messages as possible and performs any TW administration function necessary when a message has been received: performing a rollback when a straggler has been received, updating the GVT (queue) in case of a new GVT message, inserting a message scheduling a user event into the input queue, etc. If no message has been received and  $LVT - GVT \geq vt\_timewindow$  then the previous steps will be repeated until LVT is inside the optimism limiting time window — because of a GVT update or a rollback. If no message has been received and  $LVT - GVT < vt\_timewindow$  then the kernel sleeps for 100 microseconds, sets LVT to the

---

\* which will be set by APSIS library

timestamp of the next input message to be processed, copies the user data of this message into the memory location which must be given as an argument to *tw\_recv()* and returns the size of the copied data. However, if the input queue is empty then LVT is set to  $\infty$  and the next loop iterations are executed until there are new pending events or GVT is  $\infty$ . When GVT equals  $\infty$  then *tw\_recv()* will return a negative value. The simulation should be terminated by the user by calling *tw\_finalize()*, which will cleanup the APSIS state.



## Chapter 4

# Workload for APSIS experiments

The best way to test a library is to do it on the same computer using the same configuration and performing the same actions a user intends to do with his program. In practice, however, the library will be used by different users on a variety of systems and under a variety of programming conditions. Any test should cover as much of this as possible, should be well-defined and results obtained by the test should be reproducible. One can use "normal" or "synthetic" benchmarks for this purpose, although the benchmarks should be designed and used carefully in order to avoid misleading conclusions.

A *normal benchmark* contains a "typical" mixture of (parts of) real programs, for which "typical" program parameters have been set. Typical settings can be determined by examining the every day applications of the tested library. However, it is often not exactly clear which test programs to use in the mixture and how the parameters of each program should be set in order to get a good approximation of the choices made in practice. A benchmark that contains mostly integer programs, for example, cannot be expected to perform well for floating point applications. Besides, it can be hard to distinguish between performance effects caused by the (particular implementation of the) test programs and effects due to the library internals itself. This can be especially hard for simulations, which lack deterministic behaviour, because of the use of various stochastic variables.

A special kind of benchmark, the *synthetic workload*, does not try to simulate any real world system, but attempts to capture the essential characteristics of a range of typical simulations from the practice. However, the synthetic workload may not capture all the possible behaviours of real programs. The workloads are designed to be relatively simple in order to facilitate understanding the behaviour of the tested library. They allow for the assessment of performance without getting lost in the details and peculiarities of a specific simulation. This thesis uses the synthetic workload PHOLD [8] to test the Time Warp library. First we will examine the original PHOLD algorithm. Then a description will be given of the implementation used for the experiments performed in chapter 6.

## 4.1 The PHOLD algorithm

[8] state that PHOLD is a symmetric and homogenous workload which behaves like simulations of symmetric, closed queueing networks. The algorithm contains a number of Time Warp logical processes (LPs) processing and exchanging event scheduling messages. Every time an event scheduling message has been processed exactly one other message is generated and sent to some other LP. In this way the number of events in the system at any (physical) moment in time stays constant throughout the whole simulation. Each process  $LP_i$  occupies a unique, integral cartesian coordinate position in a two-dimensional communication plane and may only send messages to a fixed set of other processes. This set is called the neighbourhood of  $LP_i$ . It is assumed that the neighbourhood consists of the LPs closest to  $LP_i$  in the two dimensional coordinate plane. The following PHOLD parameters have been described in [8]:

1. **number of LPs**  
the number of Logical Processes participating in the simulation;
2. **message population**  
the number of event scheduling messages with which the simulation is started;
3. **initial event distribution**  
location in (simulation) time and LP-space of each initial event.
4. **timestamp increment function**  
function returning the timestamp increment for each newly generated message;
5. **movement function**  
function returning the destination LP for each newly generated message;
6. **computation grain per event**  
the amount of work associated with each incoming event (excluding the time required to schedule the next event)

The model assumes that each LP behaves in the same way, that is, it assumes that choices made for the model parameters are the same for all LPs. Each of the parameters 3, 4, 5 and 6 are generated according to some probabilistic distribution. In addition to these parameters two other simulation characteristics have been defined in [8]: spatial locality and the lookahead of an LP.

The performance of parallel programs is substantially influenced by the spatial locality exhibited by the communication actions between the various program entities. This spatial locality can be defined as the degree to which entities will communicate with entities closer by. A larger degree of locality means that more communication will take place with entities closer by, while a smaller degree of locality communication implies more communication with

entities farther away. In PHOLD this quantity is implicitly incorporated in parameter 5: the region from which a destination LP will be chosen, the neighbourhood, controls the degree to which processes exhibit spatial locality.

The second, extra parameter defined by [8] is the lookahead of an LP (see also section 2.2.2 of chapter 2). PHOLD emulates lookahead as follows, assuming that a process  $LP_i$  processes some event  $e$  which will cause it to schedule a new event  $e'$  to occur at  $T = T_O(e')$ . Lookahead is emulated by demanding that  $LP_i$  should first schedule a local event\*  $e''$  which will immediately schedule  $e'$  to occur at  $T$ , when  $e''$  itself has been processed. In addition, if  $la$  is the lookahead, then  $T_O(e'')$  should be equal to  $T - la$ . If an LP has lookahead equal to the time increment of the next event to schedule,  $e'$ , then the message scheduling that event can be sent immediately — which is what in a real simulation would be done if it is known what kind of new event, with what time increment will be scheduled after the current event will be processed. However, a smaller lookahead will lead to postponing the next event until the LP is within  $la$  of  $T_O(e')$  — a real simulation cannot schedule new events until after enough information is known about these new events. So if a new event should occur at time  $T$  it can only be scheduled by a former event with a  $T_O$  equal to  $T - la$ .

## 4.2 PHOLD implementation

The PHOLD implementation has been tailored to the APSIS library. We will henceforth call this implementation APHOLD (AP<sub>SIS</sub> PHOLD), while we use PHOLD to refer to the original algorithm itself. PHOLD has been implemented as described above, except for some parameters such as spatial locality and lookahead. APHOLD's peculiarities and deviations from the standard algorithm will be explained below. Pseudo code for the main APHOLD functions are given in Appendix A.

The various distributions are generated using the *erand48()* pseudo-random number generator.<sup>†</sup> This function takes a random generator state array as an argument. This array is used to compute the next random number and is adjusted at each computation. The Unix man page [38] states that: “*By using different arguments, functions erand48(), ... allow separate modules of a large program to generate several independent streams of pseudo-random numbers, that is, the sequence of numbers in each stream will not depend upon how many times the routines have been called to generate numbers for the other streams.*”

APHOLD reserves random generator state vectors for each PHOLD parameter when the command line is parsed at the start of the program. The state, state size and the parameters of each requested distribution are stored in an array (*distribution[]*).

For example, take the function *nextInitEventOccurrenceTime()*, returning the occurrence time

---

\* That is an event scheduled to occur at  $LP_i$  itself. <sup>†</sup> using the linear congruential algorithm and 48-bit integer arithmetic like in the *drand48()* function [38])

of the next initial event (figure A.8 on page 65). Assume that the occurrence time of the initial events should have an  $exp(\lambda)$  distribution. At the start of the program an entry in *distribution[]* is created and the index to this entry is saved in *initialTimeDid*. This entry will contain an *erand48()* state, the state size and the parameter  $\lambda$ . The pointer *initialTimeDistr()* will refer to a function returning numbers which are exponentially distributed with mean  $\lambda$ . This function takes the distribution identifier *initialTimeDid* as an argument and uses it to calculate the next number using the appropriate state and parameter information stored in *distribution[]*. In order to implement correct rollback behaviour, *tw\_state\_save()* is used to save the *erand48()* state before a new number is computed.

All PHOLD parameters are given to APHOLD as command line options. In order to implement PHOLD parameter 3, a special LP,  $LP_0$ , computes the distributions of the initial events over virtual time and over LP-space. After doing so,  $LP_0$  sends messages to each process  $LP_i$  containing the number of initial events which should be scheduled at  $LP_i$  and the  $T_Os$  of those events. The initial events are then scheduled by each LP itself to occur locally. Figure A.6 on page 64 and figure A.7 on page 65 illustrate how this is implemented using the function *scheduleInitialEvents()*. The functions *async\_send()* and *async\_recv()* are low level APSIS primitives, which can be used to distribute initialising information, like the initial event distributions. The first function, *async\_send()*, implements nonblocking asynchronous send and the second function blocking receive (see also section 5.1 of chapter 5). The command line must contain separate arguments describing the dispersion of the initial events over (virtual) time and over the LPs.

Parameter 4, the timestamp increment function is described in figure A.4 on page 63. The current LVT is passed to *nextEventOccurrenceTime()* which returns a timestamp equal to the LVT plus some time increment.

Parameter 5, the movement function is implemented by *nextEventLocation()* and is described in figure A.5 on page 63. The APSIS library allows each LP to communicate with 5 LPs: itself and its South, West, North and East neighbours (see also chapter 3). Consequently spatial locality as defined by PHOLD is meaningless for APHOLD. Instead APHOLD uses 5 directions along which a new message can be sent: *Forward*, *Back*, *Right*, *Left* and *Self*. One of these directions will be selected according to some probability distribution function supplied by the user and depending on the source of the next input message to be processed. If the source LP was the West LP, for example, and if, after drawing a stochastic variable, it is determined that the next communication direction to follow is Forward, then the new message will be sent to the East neighbour (see table 4.1 on the next page for the other movement function values). The user should define probabilities for all or part of the directions on the command line. If part of the probabilities are specified then the unspecified probabilities will be computed in such a way that the remaining directions will be chosen with equal probability and that the sum of all probabilities (both defined and computed ones) will be equal to 1.

Special care should be taken when the source of a message is the same as its destination. If the direction selected is other than *Self*, then it is not obvious where the next message should be sent to. APHOLD remembers the last received message which originates from one of the neighbouring processes and uses the source of this message to determine the destination of the next message.

Table 4.1: *Movement function returning the destination of a new message depending on the source of the current input message and the selected direction. The direction is selected using a user specified probability function. The destination will be selected depending on the last message received with a source equal to one of the neighbours, when the current message originates from Self and the chosen direction is different from Self.*

Direction \ Source	South	East	North	West	Self
Forward	North	West	South	East	$\begin{array}{c c} S & E \\ \hline N & W \end{array}$
Back	South	East	North	West	$\begin{array}{c c} S & E \\ \hline N & W \end{array}$
Right	East	North	West	South	$\begin{array}{c c} S & E \\ \hline N & W \end{array}$
Left	West	South	East	North	$\begin{array}{c c} S & E \\ \hline N & W \end{array}$
Self	Self	Self	Self	Self	Self

PHOLD parameter 6, the computational grain per event, has been implemented in function *processEvent()* which will execute a delay loop every time it is called (figure A.3 on page 62). An attempt has been made to emulate work like it is done in real simulations, by executing a loop in which  $x = 10 \sin(x)$  is repeatedly computed. The output of the expression at each iteration is fed as the input to the sine call of the next iteration.\* We take 10 times the sine function because the series  $x_i = \sin(x_{i-1})$  will go to zero very fast. The parameter for the very first call (at the start of the program) is initialised to the (mathematical) number  $e$ . This scheme is used as an effort to minimise compiler optimisations such as trivial sine values being looked up from a table instead of being computed. The number of iterations can be chosen according to a particular distribution function specified on the command line.

The lookahead characteristic is not fully implemented as we will not use it to test lookahead impact on the APSIS library (see also chapter 6). Instead we will assume full lookahead: as

---

\* So  $x_i = 10 \sin(x_{i-1}), i = 1, 2, \dots$

soon as the next event to be processed,  $e$ , is known, a new one is scheduled before  $e$  is actually processed (see also the main event processing loop in figure A.2 on page 62). This concludes the APHOLD description and this chapter.

## Chapter 5

# Communication

A parallel job can be viewed as a set of subtasks, the logical processes, which repeatedly perform some local computation followed by inter-process communication of local information. Messages are used to send local information to other processes.

Generally, each communication request will be issued to a communication library. This library provides high level primitives which hide the details of the rest of the communication system. The library itself uses services provided by the operating system kernel. At the lowest level, the message is sent over the communication hardware to its destination. Figure 5.1 on the following page illustrates the basic communication scheme.

Message passing processes can communicate in two ways: synchronous or asynchronous. In addition, the communication primitives can be either blocking or nonblocking. The terms synchronous and asynchronous are used to specify when a send operation will be finished. A call to a synchronous send finishes when the corresponding receiver has performed a call to a matching receive. This is equivalent to a telephone conversations. A conversation between two parties can only start after the callee has picked up the telephone. Equivalently, a call to a synchronous send function will only finish after the receiver has called a matching receive and the send buffer can be safely re-used. The synchronous send will always block until the communication action has finished.

An asynchronous send on the other hand does not depend on a matching receive: it does not need to know what the receiver is doing. This resembles the e-mail system. A sender is done with sending a letter after he/she has posted it. The sender can then post a new letter whether or not the receiver knows about the former one. Equivalently, an asynchronous send will be finished when the send buffer can be safely re-used, whether the message has been sent over the communication hardware or not.

A call to a nonblocking operation will return immediately after the communication has been initiated. A polling mechanism must be used to check whether the operation has finished. On the other hand, a call to a blocking communication operation will only return after the

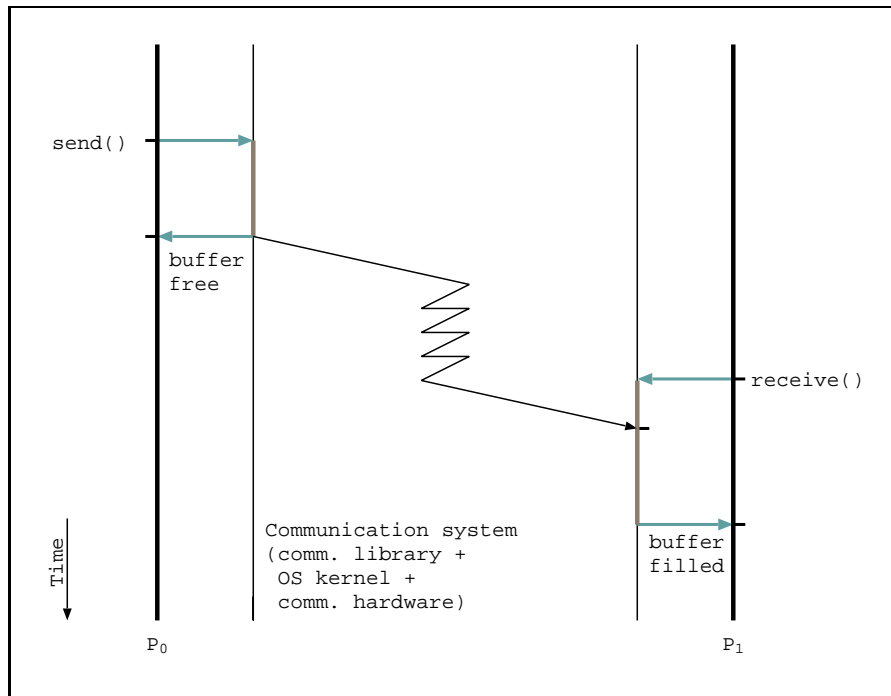


Figure 5.1: A simplified view on communication between 2 processes, from  $P_0$  to  $P_1$ .  $P_0$  issues a send-request to the communication system in order to send a message to  $P_1$ . After the send buffer has been copied to the communication system it can be safely re-used. The communication system at  $P_1$  copies the received message into user space after which the receive buffer can be consumed (if a matching receive-request had been issued).

operation has finished. No extra polling is needed.

The rest of this chapter will describe the communication primitives implemented in APSIS and the communication libraries with which APSIS can be tested: PVM, MPI and FM.

## 5.1 APSIS communication primitives

APGIS provides 5 communication primitives which can be implemented using a number of communication libraries. These primitives are: `async_comm_init()`, `async_comm_finalize()`, `async_send()`, `async_recv()` and `async_nrecv()`. The first two routines, `async_comm_init()` and `async_comm_finalize()`, are respectively used to initialise and to cleanup state on behalf of the communication library. The first function must be called before any of the other communication functions are called. This is automatically done when an APSIS user calls `tw_init()` (see also section 3.3.4 of chapter 3). The other function, `async_comm_finalize()` is called in `tw_finalize()`.

The primitive `async_send()` is used to send data to other processes. It is a blocking, asynchronous operation and requires that the destination, a pointer to the buffer containing the

message, the size of the message and a message tag is specified for each message. The primitive will block until the send buffer can be safely re-used.

The function *async\_recv()* is used to poll for a message and to receive the message after it has arrived. Polling is a nonblocking and receiving is a blocking operation. The last function, *async\_nrecv()* is also used for message polling and reception, but the receive operation is a nonblocking operation. Both functions require the following parameters to be specified: the source of the message, a pointer to a buffer for the received data and the expected message size and tag. If the source is set to -1 then the first received message will be copied into the buffer, irrespective of its source. The message tag of a pending message is ignored, if the tag is set to -1. If a message has arrived from the requested source with the requested message tag, it will be copied into the supplied buffer. The boolean FALSE will be immediately returned if no message is pending.

A boolean value TRUE will be returned after the message has been fully copied into the buffer. If a message is pending then *async\_recv()* will block until all the received data has been copied. The other receive function, *async\_nrecv()*, will always return immediately without blocking. However, it will return with the value FALSE, while a pending message has not yet been (fully) copied.

The functions *async\_send()* and *async\_nrecv()* are used in respectively *tw\_send()* and *tw\_recv()* (see also sections 3.3.2 and 3.3.4 of chapter 3). A process can schedule new events with *tw\_send()*. Messages scheduling non-local events are sent by calling *async\_send()*. The function *async\_nrecv()* is used to poll and receive messages at the beginning of the *tw\_recv()* loop. The other receive function, *async\_recv()* can be used to communicate initialisation messages (before *tw\_recv()* has been called).

Nonblocking, asynchronous send is possible, if the *async\_send()* implementation is extended. A new send operation should not be started while a previous send has not yet finished.\* A loop polling for the completion of the previous send is needed before the new send may be started. The send function could be extended in such a way, that it will return a boolean FALSE while the previous send operation has not finished. After finishing an ongoing send the function should return with boolean TRUE. A process could perform some user defined work while a new send cannot be started.

It is not possible to use synchronous communication with APSIS as this type of communication may led to deadlock. Consider a simulation system consisting of two processes,  $LP_0$  and  $LP_1$ . If  $LP_0$  sends an (event scheduling) message to  $LP_1$  while at the same time  $LP_1$  sends a message to  $LP_0$  both processes will block and wait for a matching receive to be executed at the other process.

---

\* the new send has to wait for the send buffer to be free

## 5.2 PVM library

The Parallel Virtual Machine (PVM) version 3.3.11 is a message passing system that enables a heterogeneous network of computers to be used as a single, distributed memory parallel computer [27]. This network is referred to as the virtual machine. A PVM program may run any number of tasks or processes using an SPMD or MPMD paradigm.\*

A communication daemon (called *pvmd*) is started on each host of the virtual machine. Each message sent by a process will be forwarded to the *pvmd* on the same host as the one the process is living on. This *pvmd* sends the message to the *pvmd* on the host of the destination process. The *pvmd* at the other host will then forward the message to its destination.

However processes can also be set to communicate directly with each other by setting the option `PvmRouteDirect` at the start of an APSIS program. A communication channel will be established between two processes the first time they try to communicate. Once a link is established, it persists until the program finishes.

PVM uses a blocking buffered asynchronous send primitive, *pvm\_send()*. User data must be “packed” (copied) into the send buffer before it can be sent. The send primitive returns as soon as the message is safely on its way to the receiving process. Nonblocking asynchronous and (blocking) synchronous sends are not defined.

Blocking synchronous send can be emulated by using *pvm\_send()* in conjunction with some of the other PVM routines. This send mode can be emulated by using *pvm\_joiningroup()* and *pvm\_barrier()*. The first function can be used by a process to join a group of processes — a process can be a member of more than one group. A call to the second function with a given group id will block until a user specified number of group members have reached the barrier.†

A receive operation can be blocking or nonblocking. A blocking receive will simply block until the data has been put into the receive buffer. A nonblocking receive returns immediately, indicating whether or not the operation has finished. This receive operation must be called repeatedly until its return value indicates that the message data has been copied into the receive buffer.

The PVM library guarantees that message order is preserved and that message delivery is reliable. If process 1 sends two consecutive messages A and B to process 2, message A will arrive at process 2 before message B. Moreover, if both messages arrive before process 2 does a receive then a receive set for any message will always return message A.

PVM has other advantages besides the good support for heterogeneous environments. Two of the most notable are dynamic process scheduling and the ability to write fault tolerant applications. PVM allows tasks to be dynamically scheduled, that is tasks may migrate from one host to another and they may leave and enter the program whenever desired. Applications

---

\* SPMD: Single Program, Multiple Data. MPMD: Multiple Programs, Multiple Data. † However, blocking synchronous mode is useless for APSIS; see section 5.1

can proceed even when hosts or tasks fail or loads change dynamically due to outside influence. The PVM implementation used on the DAS is built on top of the Panda 4.0 [17] low level communication library. Table 5.1 shows which PVM functions have been used to implement the various APSIS communication primitives.

Table 5.1: APSIS calls to PVM primitives

APSYS function	implemented using	PVM functionality
async_comm_init()	pvm_spawn()	spawn the participating processes
	pvm_setopt()	set pvm options used here to set PvmRouteDirekt
	pvm_mcast()	send a message to a group of tasks used here to send the task id's to each task
	pvm_rcv() pvm_unpackbyte()	blocking receive unpack message bytes into user memory called after a message has been received
async_comm_finalize()	pvm_exit()	cleanup
async_send()	pvm_initsend()	initialise pvm send buffer
	pvm_pkbyte()	copy message bytes into send buffer
	pvm_send()	blocking, asynchronous send the pvm send buffer
async_rcv()	pvm_rcv()	blocking receive
	pvm_unpackbyte()	see async_comm_init()
async_nrecv()	pvm_nrecv()	nonblocking receive
	pvm_unpackbyte()	see async_comm_init()

### 5.3 MPI library

In an article in which PVM and MPI are compared, [16]\*, G. A. Geist *et al.* conclude that MPI is more usefull on a multicomputer, while PVM is a better choice for heterogeneous systems. They note that MPI was designed specifically for Massively Parallel Processors (MPPs) and that it would be tuned by each MPP vendor to produce high communication performance. However, this has lead to some sacrifices. Applications written in MPI on one vendor's MPP are easily ported to another vendor's MPP, but an MPI application executing on one MPP is not able to communicate with an MPI program on another vendor's MPP. Another sacrifice is the lack of fault tolerance. All that is guaranteed for a parallel program is the ability to exit, if a processing node or task should fail due to an error. A possible advantage of MPI, besides the higher expected performance, might be that MPI has more point-to-point and collective communication options than PVM.

---

\* PVM and MPI: a Comparison of Features

PVM was specifically designed for message passing programs running on heterogeneous systems. This has led to some extra overhead, e.g. because data may need to be converted from one host type to another or extra checks should be performed for fault tolerance, etc. So MPI is expected to always be faster than PVM on MPPs because MPI is specifically implemented towards those MPPs. However, [16] also note that some comparison studies show a comparable performance for both libraries.

The Message Passing Interface standard (MPI) version 1.1 has been developed for an SPMD based programming style [5]. The library uses a number of different communication modes. The APSIS functions are implemented using the so called standard mode when the MPI library is actually used. In this mode communication is normally blocking and asynchronous. However MPI may decide to use a more synchronous scheme instead. MPI decides whether to use a buffer for a send operation or not. If a buffer is used then a send may complete before the invocation of a matching receive. If a buffer is not used (for performance reasons) or if there is no space available then a send can complete only if a matching receive has already been posted and all data has been moved to the receiver. Messages are delivered reliably and in-order: the messages are delivered at their destination error free and in the same order they were sent.

APSYS uses the MPICH 1.1.1 implementation (using the `ch_p4` device) on the Suns and MPICH 1.1.1 on top of the Panda 4.0 communication library (using the `ch_panda4` device) on the DAS. The MPI functions used to implement the various APSIS communication primitives are given in table 5.2.

Table 5.2: *APSYS calls to MPI primitives*

<b>APSYS function</b>	<b>implemented using</b>	<b>MPI functionality</b>
<code>async_comm_init()</code>	<code>MPI_Init()</code>	initialise the communication library
	<code>MPI_Comm_size()</code>	get the number of participating processes
<code>async_comm_finalize()</code>	<code>MPI_Finalize()</code>	cleanup
<code>async_send()</code>	<code>MPI_Send()</code>	blocking send in standard mode
<code>async_recv()</code>	<code>MPI_Recv()</code>	blocking receive
<code>async_nrecv()</code>	<code>MPI_Irecv()</code>	start receive, nonblocking
	<code>MPI_Test()</code>	test for communication completion, nonblocking

## 5.4 FM library

As the Fast Messages (FM) library is less generally known than PVM and MPI we will discuss this library in more detail. FM version 2.0 is a low-level messaging layer that uses an SPMD programming model [32]. It was designed to deliver as much of the hardware's raw performance to applications and higher-level messaging layers as possible. FM is based

on "message streams", behaving much like C socket streams: while each message is sent as an individual bounded stream, the data within each stream is treated as a stream of bytes. A message is not perceived as a single atomic unit, but as an entity from which each byte can be processed as soon as it is extracted from the network. This is the big advantage of FM: whereas with PVM and MPI data processing can only start when the last byte of a message has been copied to user space, within FM this can happen even before the sender has completed the send operation.

A sender must specify a handler function to be used by the receiver to process the incoming data. Data is extracted from the network whenever possible. Each time some data is extracted the appropriate handler function is called automatically. The handlers are executed concurrently with the rest of the program using threads. More than one instantiation of a handler may be executed at a time: if a handler is busy extracting a message and a new message should be extracted with the same handler, a new instance of that handler is started.

A handler must be declared using pragma directives: *FM\_declare\_handler*, *FM\_begin\_decls* and *FM\_end\_decls*. The first directive is used to identify the start of a handler declaration. The second and third directives are used to declare variables local to the handler. A handler must be registered by putting a pointer to the function in the array *FM\_handler\_table[]*. The handler can then be identified by its array index when a message stream is started. An auxiliary program (*streamify*) should be called to parse the pragma directives before the program can be compiled. For an example of the pragma directives see lines 3 up to 7 in figure 5.2 on the next page.

Sending a message involves 3 primitives. First, a message stream should be opened using *FM\_begin\_message()*. This function needs three parameters: the message destination, the message size and the identifier of the handler to be invoked at the receiver. Next, zero or more calls to *FM\_send\_piece()* are used to send parts of the message. This is a blocking asynchronous function and will return as soon as the data has been buffered. If the library runs out of buffer space, the primitive will block until memory has been freed or the data has been received by the receiver. Finally *FM\_end\_message()* should be called to mark the end of the message and close the stream.

FM uses a polling mechanism to check for incoming messages. Polling and receiving is done through the use of two primitives: *FM\_extract()* and *FM\_receive()*. *FM\_extract()* should be called by a process whenever it needs or is ready to receive data. When *FM\_extract()* is called a user specified amount of data is extracted from the network and passed to the appropriate handler. The user needs to call this function frequently: if data is extracted to infrequently from the network then send operations to the current process will eventually block when a sender runs out of buffer space.

*FM\_receive()* can only be called within a handler and is used to copy extracted data into user space so that it can be processed by the program. A user specified global variable may be

set in the handler to indicate when the entire message has been copied (or processed) by the handler. Figure 5.2 shows a small FM example program. Process 0 sends a string ("hello world") to another process which will print the string after receiving it.

```

1  #define MSGSIZE 13
2  int Done=0;                               /* message not received yet */
3
4  #pragma FM_declare_handler
5  int helloHandler(FM_stream *stream, unsigned int sender){
6  #pragma FM_begin_decls                     /* begin local var declarations */
7  char GetBuffer[MSGSIZE];                 /* end of local var declarations */
8  #pragma FM_end_decls
9
10 /* receive data */
11 FM_receive(GetBuffer, stream, MSGSIZE);
12 Done=1;
13 printf("%s\n", GetBuffer);
14 return FM_CONTINUE;
15 }
16
17 int main(){
18 char *MsgBuff="Hello World.";
19 int helloHandlerId=0;
20 FM_stream *Strm;
21
22 /* init FM and register handler */
23 FM_initialize();
24 FM_handler_table[helloHandlerId]=helloHandler;
25
26 if(0==FM_nodeid){ /* I'm process 0 */
27 Strm=FM_begin_message(1, MSGSIZE, helloHandlerId);
28 FM_send_piece(Strm, MsgBuff, MSGSIZE);
29 FM_end_message(Strm);
30 }else{ /* I'm process 1 */
31 do
32 FM_extract(~0);                             /* extract ~0=MAXLONG bytes */
33 while (!Done);
34 }
35 }

```

Figure 5.2: *Hello world example*

FM guarantees reliable and in-order communication. Table 5.3 on the next page shows the FM primitives used to implement the various APSIS communication primitives.

Table 5.3: *APSYS calls to FM primitives*

<b>APSYS function</b>	<b>implemented using</b>	<b>FM functionality / notes</b>
<code>async_comm_init()</code>	<code>FM_initialize()</code>	initialize the communication library
<code>async_comm_finalize()</code>	<code>FM_finalize()</code>	cleanup
<code>async_send()</code>	<code>FM_begin_stream()</code>	start FM message stream
	<code>FM_send_piece()</code>	blocking, asynchronous send piece of data may be called a number of times to send all desired data
	<code>FM_end_stream()</code>	end stream
<code>async_rcv()</code>	<code>FM_extract()</code>	nonblocking poll for a message
	<code>FM_receive()</code>	blocking receive, copies data to user space may only be called within a handler handler is executed concurrently with main program
<code>async_nrcv()</code>	<code>FM_extract()</code>	see <code>async_rcv()</code>
	<code>FM_receive()</code>	see <code>async_rcv()</code> a flag is set when all data has been received



## Chapter 6

# Experiments

As one can see in chapter 3 a lot of research has been done considering Time Warp performance. It is generally accepted that rollbacks have a large impact on Time Warp performance. This thesis is an empirical study of the TW rollback behaviour. We will determine which library out of a number of communication libraries is best suited for APSIS and what the effects of various PHOLD parameters are on rollback behaviour. First we will describe the tools with which the experiments have been performed. Then we will describe the experiments with the communication libraries discussed in the previous chapter. Finally we will examine the rollback behaviour of APHOLD.

### 6.1 Experiment environment and tools

There is a growing tendency in parallel computing to use a network of workstations as one big parallel virtual machine. These machines may be shared with other applications which may interrupt the execution of the parallel program for an arbitrary time. In addition, communication costs are generally larger on the network of workstations than on a multi-processor and the network often contains different kinds of machines (processors running at different clock speeds, etc). So, the same parallel program may exhibit different behaviour using the same program input on each type of parallel machine. A parallel program performance study should thus be conducted on both a cluster of workstations and on an MPP.

The experiments had to be performed on both Sun Workstation clusters and on a multi-processor (the so called DAS [39]). However, APHOLD on top of APSIS and PVM 3.3.11 could not be successfully run on the workstations: the runs could not complete, because of deadlocked LPs or LPs which ended prematurely. This was probably caused by a GVT state updating bugs in the APSIS GVT computing code, but it might also have been caused by PVM bugs. The runs crashed at unpredictable times, making it hard to track the bugs down; most of the time they appeared for APHOLD settings which lead to long execution times

(about half an hour or more). The experiments with APHOLD have only been performed on the DAS, as the bugs could not be found within a short time.

The test programs were written in *C* and compiled with the SunOs/Sparc *cc 5.0* compiler on the workstations and the *gcc 2.8.1* compiler on the DAS. The programs have been compiled without setting any compiler optimisation options. Time measurements on the Sun workstations were performed using the *gethrtime()* function — returning time in nanoseconds. On the DAS nodes *gettimeofday()* — returning time in microseconds — was used, as *gethrtime()* was not available on the Linux distribution used at the DAS nodes.

The experiments were performed on Sun Ultra-SPARC-III workstations running at clock speeds of 270 MHz and containing 64 MB of main memory. Each machine uses SunOs 5.6 as its operating system and contains a 100 Mbit/sec Ethernet card with which users can communicate with any of the other workstations.

Each DAS node contains a Pentium Pro processor running at 200 MHz. Each node has 128 MB of EDO-RAM in DIMM modules and a 1.2 Gbit/sec (full duplex) Myrinet card for user level communication (regular OS related communication traffic, like NFS, rsh, etc. uses a separate 100 Mbit/sec Fast Ethernet card). The DAS nodes use Redhat 6.2 with Linux kernel 2.0.36 as their operating system.

## 6.2 Communication library performance

In chapter 3 section 3.1.2 we have concluded that rollbacks have a major influence on TW performance. Rollbacks may cost a lot of time and may lead to an explosion of messages sent to undo erroneous computations. These messages should be delivered as fast as possible in order to prevent LPs from progressing far into erroneous computations. The more events an LP has executed, the longer the rollbacks will be and the larger the probability will be for future cascading rollbacks, as LPs execute more and more out of pace. Carothers *et al.* conclude in their study about the effects of communication on Time Warp performance [6], that increasing communication delays significantly reduces performance. They found the greatest performance degradation for applications with small computational grain.

We have determined the communication delay, throughput and distributions of communication times for each communication library. Only the PVM and MPI libraries have been tested, as the FM library was not available on the DAS at the time when the final experiments were performed. The communication libraries have been tested with a so called "ping-pong" program. The program contains two processes,  $LP_0$  and  $LP_1$ .  $LP_0$  executes a loop in which it first sends a message to  $LP_1$  and then waits for the message to be returned by  $LP_1$ .  $LP_1$  executes a loop in which it repeatedly receives a message from  $LP_0$  and then sends it back to  $LP_0$ . At  $LP_0$  the time it takes to perform each iteration is divided by 2 in order to get one-way communication costs. This experiment has been performed for a number of message

size, including the size of an APSIS message, 156 bytes. Results for the experiments are given in section B.1 on page 67; the frequencies of measured communication times for a number of the tested message sizes are given in figures B.5, B.6, B.7 and B.8. Due to the large number of tests not all results have been shown.

### 6.3 Testing APSIS

First we have determined how long each iteration in the computational grain loop of the APHOLD program takes to execute so that one can translate the number of iterations used into the time that the program was executing inside the computational grain function. The computational grain length has been determined by measuring the time it takes to execute a loop in which the *processEvent()* function of APHOLD (figure A.3 on page 62) is called 200 times. While *processEvent()* in APHOLD randomly selects the number of delay loop iterations, we set this quantity to  $10^6$  in this experiment and compute the mean time per iteration by dividing the execution time of the loop containing the APHOLD function by  $200 \cdot 10^6$ . On the Sun Sparc workstations we found an execution time of  $1.019 \mu\text{sec}$  per iteration and on the DAS nodes  $0.605 \mu\text{sec}$  per iteration.

Next, the rollback behaviour has been examined in relation to a number of APHOLD parameters: the number of LPs ( $N_{lp}$ ), the message population ( $N_{msg}^{init}$ ), the computational grain ( $C_{grain}$ ) and the movement function ( $F_{move}$ ). We used APSIS on top of PVM because of the lower delay of PVM for smaller message sizes.

Measurements have been performed to determine the turnaround time,  $T$ , and the total rollback time ( $T_{rb}$ ), that is the time during which the state of a process is restored to the last saved valid state in the past.  $T_{rb}$  does not include the time lost by event thrashing, i.e. the time lost by prematurely processing events which will be rolled back eventually. In addition to the time measurements, we will look at the total number of processed events per LP,  $E_p$ , the total number of events rolled back per LP,  $E_{rb}$ , the total number of rollbacks per LP,  $N_{rb}$ , and the fraction of committed events\* per LP,  $F_c = \frac{E_p - E_{rb}}{E_p}$ . We will also look at the relative efficiency,  $E(p) = \frac{T}{pT_p}$ , for the  $N_{lp}$  experiments (with  $p = N_{lp}$ ), the mean time used to process each message thread,  $\frac{T}{N_{msg}^{init}}$ , for the  $N_{msg}^{init}$  experiments and the grain normalised turnaround time  $\frac{T}{C_{grain}}$  for the  $C_{grain}$  experiments. A message thread is the computation started by an initial message and consists of all the messages which have been caused by that initial message.

APHOLD has been run 4 times and the minimum, median and maximum over all LPs and all runs have been determined for  $E_p$ ,  $E_{rb}$ ,  $N_{rb}$  and  $F_c$  for each parameter setting. For  $T$  the maximum turnaround time has been used, as the minimum and median were very close to

---

\* Committed events are events for which the GVT has passed beyond their time stamp without being annihilated.

the maximum (except for the movement function, see figure B.11 on page 75). The random generators have been reset at each new run, so that each run uses a new stream of random numbers.

In order to analyse the behaviour of the TW system only one APHOLD parameter is changed at a time; table 6.1 summarises the default APHOLD parameters used at each experiment. The parameters which should not change are set to the default values given in this table. All APHOLD runs are set to end as soon as  $GVT \geq 20000$  on the DAS and  $GVT \geq 6000$  on the Suns; this is the simulation stop times for which the most time consuming experiments will end after approximately 1 hour. The APSIS optimism limiting time window length is set to 3000.

Table 6.1: *Default APHOLD parameters*

parameter	value
1. number of LPs	6
2. message population	180
3. initial event distribution	simulation time: 0 distribution over LPs: uniform 0 5
4. time stamp increment	1
5. movement function	equal probability to all directions
6. computational grain	1000

Table 6.2 on the facing page contains the values with which the APHOLD parameters have been varied. When the number of LPs are varied, the initial event distribution is changed such that for each chosen number of LPs,  $n$ , the events are uniformly distributed over  $LP_0, \dots, LP_{n-1}$ . Experiments with parameter 5 use 36 LPs in a  $6 \times 6$  torus topology, so that the intended communication patterns do actually emerge (for example, using only a  $2 \times 2$  topology greatly restricts the number of possible patterns irrespective of the probabilities of the movement function). Ideally we want to set  $N_{lp}$  to a much larger value (e.g.  $15^2$ ) but there were not that many DAS nodes or Sun workstations available.

Increasing the value of *parameter 1*, the number of LPs (while keeping the message population constant) will lead to more event processing parallelism being exploited. So an increase of the number of LPs will have a decreasing effect on the turnaround time. However more LPs will increase the probability for LPs running out of pace. This in turn will increase the probability for rollbacks; more rollbacks will increase the turnaround time.

If only *parameter 2* is increased then each LP will have more events to process during a program run. So LPs can increase their LVT more often and send a new LVT more often to the other LPs which will decrease the probability for out of pace execution of LPs. Less and less rollbacks are suffered so that the program execution time will decrease more and more.

Table 6.2: *Mutated APHOLD parameters. In 4 sets of experiments, one parameter is changed at a time, while the other parameters are set to the default values given in the previous table. In the experiments with parameter 5 the number of LPs is set to 36 instead of the default value. The movement function values are labelled with the first few letters of the alphabet for convenience.*

parameter	values
1. number of LPs	1, 2, 3, 4, 5, 6, 7, 10, 11, 13, 15, 16, 17, 18, 19, 20, 25, 35, 37, 45
2. message population	1, 2, 3, 4, 5, 6, 10, 20, 30, 40, 50, 60, 180, 600, 4000
5. movement function	Self $10^{-4}$ Forward $10^{-6}$ Back $10^{-6}$ (A)
	Self $10^{-4}$ Forward $10^{-7}$ Back $10^{-7}$ (B)
	Self $10^{-6}$ Left $10^{-4}$ Right $10^{-4}$ (C)
	Self 0.9999 (D)
	Self $10^{-6}$ (E)
	Self 0.2 (F)
6. computational grain	1, 25, 500, $10^3$ , $3 \times 10^3$ , $10^4$ , $2.5 \times 10^4$ , $5 \times 10^4$ , $7.5 \times 10^4$ , $10^5$

However, at the same time more work and communication has to be performed as more events must be processed leading to an increase in the execution time.

The movement function, *parameter 5*, is used to examine a number of communication patterns. We can expect the following experiment results for the schemes used in table 6.2.

- A Messages travel mostly left or right in this scheme. We can expect a number of interwoven threads of communicating LPs extending over the 2-dimensional LP space. We do not have maximal interaction between the LPs compared to the case when we set the probabilities of all directions to be equal. However there may be substantial interaction as the threads may cover the whole LP-grid. This scheme will probably lead to a small number of rollbacks because of the many LP interactions.
- B This is the same configuration as above, but using slightly different probabilities: interaction between the threads will be even less common. So threads may run more out of pace possibly leading to less frequent, but larger rollbacks.
- C If the initial message have all been sent in the same direction (e.g. East to West or vice versa) then we get long LP threads which rarely interact leading to the same observations as in scheme A. However, while scheme A may lead to a 2-dimensional area of interacting LPs in one thread, this scheme leads to long 1-dimensional threads. If initial message are sent both horizontally and vertically we get the same behaviour as for scheme E (see below), but to a much smaller extent.
- D Because of the very local computation we can expect the most number of rollbacks here and thus the largest turnaround time. However because of the uniform distribution of

the initial messages and the constant time increment the probability for out of pace execution may be small. This may not be the case in a heterogeneous computing environment like the cluster of workstations if some of the machines are faster than others.

E We will probably get the smallest number of rollbacks and turnaround time here, as this scheme allows for maximal interaction between the LPs in a more tightly coupled way.

F Because a fraction of the messages are sent to Self there will be less interaction between the LPs than in the previous scheme. However one may still expect better performance and less rollbacks than in all the other schemes (expect for scheme E).

It is expected that the greatest impact on performance will come from *parameter 6* the ratio of computational grain to interprocessor communication cost as is generally observed for any parallel program. Besides the general effect of communication delay on parallel programs, TW performance might degrade due to rollbacks if communication is delayed too much.

When using faster communication one can expect to get longer rollbacks as the erroneous computation can spread faster than in the case of slower communication if the amount of work per event is not sufficiently high. Anti-messages on the other hand will also propagate faster through the system, and thus slow down erroneous computations faster. However delayed communication may delay positive events from reaching LPs in time so that the LVT cannot be updated fast enough relative to the progress of the other LPs. This may substantially increase the probability for rollbacks. To see which effect is greater we will investigate the rollback length in events and the fraction of committed events.

Results of the APHOLD experiments are given in section B.2 on page 73. All experiment results are discussed in the following chapter.

# Chapter 7

## Discussion and conclusions

In this chapter we will discuss the experiments from the previous chapter and their outcomes. The communication library experiments will be discussed first, followed by a discussion of the APSIS tests. The chapter is concluded with suggestions for possible future work.

### 7.1 Discussion

Except for the PVM tests on the Sun machines one can fit two linear functions through each graph of the communication test results given in section B.1. This is probably because the libraries run out of buffer space when using larger messages. The fitted function running through the set of smaller messages has been used, in order to determine the communication delay for the libraries; the delay is given by the value of the fitted function for message size 0. We use the set of smaller messages because the APSIS message size itself is 156 bytes. Table 7.1 summarises the measured delays for each type of machine and communication library.

Table 7.1: *Communication timings for small messages*

<b>Machine</b>	<b>Communication library</b>	<b>Delay (in msec)</b>
DAS-node	PVM	0.08
	MPI	0.08
Sun/Sparc	PVM	0.35
	MPI	0.48

PVM and MPI on the DAS show almost identical results. This is probably caused by the fact that these libraries are built on top of Panda and that they deliver equivalent Panda performance to the user. PVM is always better than MPI on the Sun machines. When using small messages (like in APSIS), message communication times will mainly depend on the latency of the communication library used. Figure B.5 on page 69 shows the communication

time distribution for messages sent on the Sun machines using PVM. For 156 bytes we can see that sending this message costs around 0.36 milliseconds, while the latency for PVM on the Suns is 0.35 milliseconds. One can also observe that for message sizes up to around 750 bytes, the communication time does not increase significantly. The other machine/communication library combinations show equivalent timings. The maximal user data size per APSIS message could be safely set to a higher value (up to 750 bytes) as the communication time will not increase substantially, while the user is allowed to define larger events. PVM was used for the set of APHOLD experiments, as it shows the lower delay.

We will now discuss the APHOLD results on the DAS using PVM. The probability for rollbacks in an APHOLD run on the DAS is small if the APHOLD parameters are set to the default values given in table 6.1 on page 52. Because of this set of parameters LPs will evolve through simulation time in more or less the same pace. However on the Suns this might not be the case if some machines are slower than others, e.g. because a lot of other (non-APHOLD Unix) processes are running on the same machine which is executing one of the APHOLD processes. If this is the case, the APHOLD process execution may be repeatedly and randomly interrupted and/or it may be slower than execution of the other LPs.

The first APHOLD parameter to be tested was the **number of LPs**,  $N_{lp}$ . Figure B.9 on page 73 shows that the turnaround time,  $T$ , decreases relatively fast with increasing  $N_{lp}$  — this is true for  $N_{lp}$  up to 16 LPs, after which the decrease is somewhat slower. The number of processed events per LP,  $E_p$ , also decreases relatively fast with increasing  $N_{lp}$ . This indicates that up to 16 LPs parallelism is well exploited, after which it is less and less exploited. This is probably a result of the rollback behaviour. The number of rollbacks,  $N_{rb}$ , and rolled back events,  $E_{rb}$ , increase relatively slowly with increasing  $N_{lp}$ , while  $E_p$  keeps decreasing relatively fast. Because of the behaviour of  $E_{rb}$  and  $E_p$  we see a relatively large decrease in the fraction of committed events,  $F_c$ , which decreases down to as much as 71%. The  $F_c$  decrease follows from the fact that by definition  $F_c = \frac{E_p - E_{rb}}{E_p}$ . So while increasing the number of LPs does not lead to much more rollbacks,  $F_c$  does decrease substantially because of the relatively large decrease in  $E_p$ .

Another interesting behaviour can be observed for  $N_{lp}$  which are prime. At prime  $N_{lp}$  one can observe a sudden sharp decrease in  $F_c$  and an increase in  $N_{rb}$  and  $E_{rb}$ . This behaviour is the strongest for  $N_{lp} = 37$ , the largest tested prime. Recall that the LPs are put in a 2D-torus topology and that for prime  $N_{lp}$ , the LPs are put along a 1D-ring instead of in a mesh. Because each LP will then communicate with only 2 different other LPs — the East and West neighbours — and itself, the LPs will be less tightly coupled, while the probability for an incoming event causing a new event to be locally scheduled will be 0.6 (using the default APHOLD settings except for  $N_{lp}$ ). The two neighbours will be selected with probability of only 0.2 each. The more LPs are used the higher the probability for cascading rollbacks as the probability for out of pace execution will be higher: there will be a larger delay for LVT

updates to get across the entire LP grid from one end of the grid to another.

Finally, if we look at the relative efficiency,  $E(p) = \frac{T_1}{pT_p}$ , which is plotted in figure B.13 on page 77 one can observe a speedup for all  $N_{lp} \leq 45$ : we have an efficiency of 58% in the worst case (45 LPs). Regarding  $T$ , one can conclude that for the chosen APHOLD defaults increased parallelism with increasing  $N_{lp}$  is more important than the increase in rollback probability. However, one can observe a significant impact on  $T$  by the chosen communication topology: a ring-topology leads to much more rollbacks than a mesh-topology. Rollbacks in the ring-topology have significantly more impact on  $T$ .

The second APHOLD parameter which has been tested was the **initial message population**,  $N_{msg}^{init}$ . When  $N_{msg}^{init}$  is increased one can observe an increase in  $T$  (figure B.10 on page 74). For small values of  $N_{msg}^{init}$  (1 up to 10),  $F_c$  is decreasing down to an  $F_c$  of 87%. It then starts to increase again up to an  $F_c$  of 99% and higher.  $F_c$  is 1 for  $N_{msg}^{init} = 1$  as this is the only message which can increase the LVT at each LP: an LP has to wait until it receives a message before it can increase its LVT. As more messages become available more LPs can increase their LVT at the same time. However, at first there are too few events available to exchange LVT information frequent enough, resulting in out of pace execution and relatively fast increasing rollback probability. Then, as more events become available LPs can update their LVT more frequent and exchange more events with other LPs leading to more tightly coupled LPs which run more and more in pace resulting in smaller rollback probability.  $N_{rb}$  and  $E_{rb}$  increase slowly after  $N_{msg}^{init}$  of 10. The equivalent shapes of  $N_{rb}$  and  $E_{rb}$  suggest that while the number of rollbacks increases, albeit slow, the number of rolled back events per rollback is small. However, at  $N_{msg}^{init} = 4$  one can observe a sudden slightly sharper increase in  $T$  and  $E_{rb}$  and decrease in  $F_c$ . As  $N_{rb}$  increases relatively smoothly around this value, one may conclude that at  $N_{msg}^{init} = 4$  relatively more events have been rolled back per rollback. Although  $F_c$  is increasing after  $N_{msg}^{init}$  of 10, this has little effect on  $T$ ; the extra time needed to process the extra events and caused by the extra communication evidently dominates the fact that a smaller and smaller fraction of the events are rolled back. This may be different if an exponential time increment is used as one might expect rollbacks in such a system to emerge more often, having a much larger impact on the execution time.

If we look at the time spent processing each message thread, which is plotted in figure B.14 on page 77 we see a decrease with increasing  $N_{msg}^{init}$  — and because the number of processed events increases linearly (figure B.10) the same behaviour may be expected for the mean processing time per event. So, more initial messages lead to more parallelism being exploited.

We see that, with the chosen APHOLD defaults, the increase in turnaround time is mainly determined by the extra work and communication associated with the increase in  $N_{msg}^{init}$ . One can also note an increase in the level of parallelism used.

The third examined APHOLD parameter was the **movement function**,  $F_{move}$ . In figure B.11 on page 75 we can see that  $F_{move} = E$  has a low  $T$  compared to the other values, with

little variance in execution times. It is followed by F with slightly higher T. The other function values have higher T and a slightly higher variance in execution times. However for movement function A a very large maximum can be observed. As the median is much closer to the minimum, this maximum is probably caused by one LP in one of the experiment runs. Movement function E has as expected the lowest T. It has also the highest  $F_c$  and although it has the highest  $N_{rb}$ , it had rollbacks which were much shorter than at the other function values, as its  $E_{rb}$  is slightly smaller than for the other function values. Although rollback probability is small due to the chosen APHOLD defaults, we do see effects of  $F_{move}$  on the fraction of committed events and the number of rollbacks, albeit small; ordered in increasing  $N_{rb}$  we have D, C, B, A, F, E if we look at the medians. However, much more important is  $E_{rb}$  as this actually determines time lost due to erroneously processed events. Ordered by increasing medians we get C, A, E, D, B and F, although E has the smallest variance. Ordered by increasing  $F_c$  we have A, B, C, D, E and F and ordered by increasing  $F_c$ -variance we have F, B, A, C, D and E. We expected the worst performance for D, but found none: evidently the LPs hardly ever communicated as D had the smallest amount of rollbacks and rollback lengths. This is also suggested by the fact that on average only 25 rollbacks have occurred, while  $1.6 \times 10^4$  events were rolled back: rollbacks were rare, while the rollback lengths were very large (around 640 events per rollback); the other movement functions show a much smaller number of events per rollback. It also seems that the message threads of A and B have had a lot of interactions as the performance was relatively good (except for one LP in one run). After distributing the initial messages, APHOLD chooses the source of the initial messages from the neighbouring LPs with equal probability. Taking the lack of performance penalties into account for movement scheme C, more penalties might be introduced if the initial message source was set to the same value for all initial messages, e.g. East or West.\* In that case,  $F_{move} = C$  would lead to only East/West movements of messages, causing a number of LP subgroups who hardly ever interact, while the LPs inside a subgroup are reasonably coupled.

So, except for C we got the expected rollback behaviour as described in section 6.3 on page 51, although the differences were small, probably caused by the choices made for the default PHOLD parameters settings.

The forth and last examined APHOLD parameter was the **computational grain**,  $C_{grain}$ . We found hardly any change of average rollback behaviour when this quantity was varied (figure B.12 on page 76). Only the variance of the measured quantities decreased with increasing  $C_{grain}$ . We did find a sudden sharper increase in T for  $C_{grain} = 10^3$ . One can also observe a sudden but small decrease in  $F_c$  and also small increase in  $E_{rb}$ . Because of the relatively small mentioned increases and decreases, the sudden increase in T is probably not caused by the rollback behaviour for this set of experiments.

---

\* However, the lack for performance penalties might be caused by the default APHOLD parameter settings which imply reduced rollback probability

Looking at  $F_{grain} = \frac{T}{C_{grain}}$  we first see that overhead like communication and TW operations get smaller per grain size as  $C_{grain}$  is increased. However at  $C_{grain} = 10^4$  the overhead increases again. This increase is not caused by the rollback behaviour as  $E_p$  is more or less the same,  $F_c$  is higher and  $T_{rb}$  is lower for the higher values of  $C_{grain}$  compared to the lower values; the increase in  $F_{grain}$  cannot be explained by the measurements of figure B.12.

We can see that for **all experiment** very little time is lost for performing the rollbacks which have happened: almost all show that the total time spent in performing the rollbacks is less than 7 seconds, while the turnaround time is much larger (up to  $10^3$  to  $10^4$  larger). For example, the rollback time is only about 1 percent of the turnaround time even when  $F_c$  is as low as 71% (figure B.9 on page 73 voor 37 LPs). We also found an increasing execution efficiency if  $N_{lp}$  and  $N_{msg}^{init}$  were increased and an optimum value for  $C_{grain}$  around  $10^4$ , although  $C_{grain}$  had little effect on rollback behaviour.

## 7.2 Future work

The following might be investigated in the future. Using an exponentially distributed time increment and a communication library with low performance in order to increase rollback probability. Testing APSIS with other benchmarks/sample simulation programs in order to verify the findings above. Testing other communication libraries (interrupt driven) active messages instead of FM, etc. and compare different APHOLD runs using different communication libraries in order to explicitly determine the fitness of a communication library for APSIS. More movement function parameters should be tested (e.g. "Self 0.1", "Self 0.01", "Right 0.9999", "Self .5 Right .5", etc.) and testing the movement function with less events (e.g. 36 processes, message population of only 6 messages) or much more processes, so that the LP interaction will only be determined by the chosen movement scheme. Of course the APHOLD experiments on the workstations should still be performed.



# Appendix A

## APHOLD

```
int main(int argc, char *argv[]){
    TIMER(start_t_init);

    /* Init Time Warp */
    nr_procs = tw_init(NUM_LINKS, &argc, &argv);
    my_pid = tw_pid();

    phold_init(my_pid, argc, argv);

    /* Create torus topology and start processing events */
    if(tw_2dtorus_create(topology[nr_procs].dimx, topology[nr_procs].dimy)){
        /* Distribute initial events over time and space */
        scheduleInitialEvents(my_pid, nr_procs, MessagePopulation);
        TIMER(end_t_init);

        /* Main event loop: process/schedule events */
        TIMER(start_t_main);
        mainEventLoop(nr_procs, my_pid, SimulationStopTime);
        TIMER(end_t_main);

        TIMER(start_t_final);
        /* Done: cleanup */
        tw_2dtorus_free();
        tw_finalize();
    } else {
        phold_finalize(my_pid);
        tw_finalize();
        exit(EXIT_FAILURE);
    }
    TIMER(end_t_final);
    saveTimings(my_pid);
    phold_finalize(my_pid);
    exit(EXIT_SUCCESS);
    return 0;
}
```

Figure A.1: APHOLD pseudo code: the main program.

```

void mainEventLoop(int nprocs, int my_pid, vtime maxGVT){
  tw_state_save(&QEsent, sizeof(QEsent));
  while( tw_rcv(&event, sizeof(event)) > -1 ){           /* while events pending */
    if(tw_gvt() < maxGVT){
      if(DEFAULTEVENT == event.type){
        /* determine next event */
        event.src = my_pid;
        event.dest = nextEventLocation(my_pid, event.src);
        event.vt = nextEventOccurenceTime(tw_lvt());

        /* send next event */
        tw_send(event.dest, event.vt, &event, sizeof(event));
        processEvent();
      }
      /* else consume event (and don't schedule any new event) */
    } else if(!QEsent){
      event.src = my_pid;
      event.type = QUITEVENT;
      tw_send(SOUTH, tw_lvt(), &event, sizeof(event));
      tw_send(NORTH, tw_lvt(), &event, sizeof(event));
      tw_send(EAST, tw_lvt(), &event, sizeof(event));
      tw_send(WEST, tw_lvt(), &event, sizeof(event));

      tw_state_save(&QEsent, sizeof(QEsent));
      QEsent = TRUE;

      tw_state_save(&lastgvt, sizeof(lastgvt));
      lastgvt = tw_gvt();
    }
    /* else consume event (and don't schedule any new event) */
  } /* end while */
}

```

Figure A.2: APHOLD pseudo code: the event processing loop.

```

void processEvent(){
  static double x = M.E;

  /* Save x and random generator state */
  tw_state_save(&x, sizeof(x));
  tw_state_save(distribution[compGrainDid].state, distribution[compGrainDid].statesize);

  work_amount = (int) rint(compGrainDistr(compGrainDid));

  for(i = 0; i < work_amount; i++)
    x = 10*sin(x);
}

```

Figure A.3: APHOLD pseudo code: computational grain function.

```

vtime nextEventOccurenceTime(vtime lvt){
  /* Save random generator state */
  tw_state_save(distribution[timeIncrementDid].state, distribution[timeIncrementDid].statesize);
  while( (delta_t = timeIncrementDistr(timeIncrementDid) < 1);
  return lvt + (vtime) delta_t;
}

```

Figure A.4: APHOLD pseudo code: the time increment function.

```

int nextEventLocation(int me, int src){
  /* Save uniform random generator state and get next uniform distributed nr. */
  tw_state_save(distribution[UniformDid].state, distribution[UniformDid].statesize);
  unif_val = UniformDistr(UniformDid);

  /* Choose message direction with probabilities given in global variable dirPr[] */
  for(dir = 0, sum_Pr = 0; dir < NRDIRS; dir++)
    if((sum_Pr ≤ unif_val) && (unif_val < sum_Pr+dirPr[dir]) )
      break;
    else
      sum_Pr += dirPr[dir];

  /* Convert dir to pid */
  if(Self == dir)
    return me;

  if(src == me){ /* and dir != Self */
    src = LastNeighbour; /* recall previously saved message src != me, see also initEvents() */
  }else{ /* src != me; save source of last message from neighbour */
    tw_state_save(&LastNeighbour, sizeof(LastNeighbour));
    LastNeighbour = src;
  }

  if(Forward == dir)
    dest = (NORTH == src)?SOUTH:(SOUTH == src)?NORTH:(EAST == src)?WEST:EAST;
  else if(Back == dir)
    dest = src;
  else if(Right == dir)
    dest = (NORTH == src)?WEST:(SOUTH == src)?EAST:(EAST == src)?NORTH:SOUTH;
  else /* Left */
    dest = (NORTH == src)?EAST:(SOUTH == src)?WEST:(EAST == src)?SOUTH:NORTH;

  return dest;
}

```

Figure A.5: APHOLD pseudo code: movement function.

```

void scheduleInitialEvents(int my_pid, int nprocs, int total_events){
  /* uses low level tw comm primitives: */
  /* blocking, asynchronous send and blocking receive */
  extern void async_send(int, void *, int, int);
  extern int async_rcv(int, void *, int, int);

  /* Determine the initial events */
  if(my_pid == 0) {
    int *events; /* storage for nr of events per process */

    /* Determine distribution of events over LPs */
    maxPid = nprocs-1;
    events = calloc(sizeof(int), nprocs);
    for(i = 0; i < total_events; i++)
      ++events[nextInitEventLocation(maxPid)];

    /* Send each proc its amount of events */
    my_events = events[0]; /* LP_0 */

    for(i = 1; i < nprocs; i++)
      async_send(i, &(events[i]), sizeof(int), TW_USER1);

    /* Determine occurrence time of init events */
    tmpT = calloc(sizeof(vtime), total_events);
    for(i = 0; i < total_events; i++)
      tmpT[i] = nextInitEventOccurenceTime();

    /* Send each proc the initial simulation time of its events */
    if(0 != my_events){
      initT = calloc(sizeof(vtime), my_events); /* LP_0 */
      memcpy(initT, tmpT, my_events*sizeof(vtime)); /* LP_0 */
    }

    for(i = 1, startIdx = events[0]; i < nprocs; i++){
      if(0 != events[i]){
        async_send(i, &tmpT[startIdx], events[i]*sizeof(vtime), TW_USER2);
      }
      startIdx += events[i];
    }

    free(events);
    free(tmpT);
  } else {
    async_rcv(0, &my_events, sizeof(int), TW_USER1);
  }
}

```

Figure A.6: APHOLD pseudo code: scheduling the initial events; first part.

```

if(0 != my_events){
    initT = calloc(sizeof(vtime), my_events);
    async_recv(0, initT, sizeof(vtime)*my_events, TW_USER2);
}
}

/* Send initial events to myself */
/* Select each src with equal probability from neighbours, */
/* required for correctness of nextEventLocation() */
event.type = DEFAULTTEVENT;
event.dest = my_pid;

event.creator = my_pid; /* accounting info used to track messages */

delta = 1.0/(1.0 * NUM_LINKS); /* nr. of chann.: North, East, West, South */
for(i = 0; i < my_events; event.id = i, i++){
    /* Select next destination with equal probability */
    unif_val = UniformDistr(UniformDid);
    if(unif_val < delta)
        event.src = SOUTH;
    else if(unif_val < 2*delta)
        event.src = EAST;
    else if(unif_val < 3*delta)
        event.src = NORTH;
    else /* [3*delta, 4*delta] */
        event.src = WEST;

    tw_send(my_pid, initT[i], &event, sizeof(event));
}

if(0 != my_events){
    free(initT);
} else{ /* init LastNeighbour with random src */
    unif_val = UniformDistr(UniformDid);
    if(unif_val < delta)
        event.src = SOUTH;
    else if(unif_val < 2*delta)
        event.src = EAST;
    else if(unif_val < 3*delta)
        event.src = NORTH;
    else /* 4*delta */
        event.src = WEST;
}

LastNeighbour = event.src; /* global variable used in nextEventLocation() */
}

```

Figure A.7: APHOLD pseudo code: scheduling the initial events; second part.

```

vtime nextInitEventOccurenceTime(void){
    /* Save random generator state */
    tw_state_save(distribution[initialTimeDid].state, distribution[initialTimeDid].statesize);
    while( (t = initialTimeDistr(initialTimeDid)) < 0); /* minimum init vt = 0 */
    return (vtime) t;
}

```

Figure A.8: APHOLD pseudo code: determining the initial time distribution.

```
int nextInitEventLocation(int maxPid){
  /* Save random generator state */
  tw_state_save(distribution[initialSpaceDid].state, distribution[initialSpaceDid].statesize);
  /* LP should be in [0 .. maxPid] */
  do{
    LP = (int)rint(initialSpaceDistr(initialSpaceDid));
  }while( (LP < 0) || (LP > maxPid) );
  return LP;
}
```

Figure A.9: APHOLD pseudo code: distributing the initial events over the LPs.

# Appendix B

## Experiment results

### B.1 Communication libraries experiments

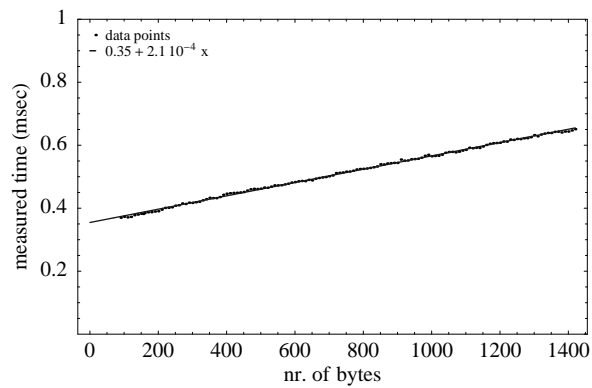


Figure B.1: Median time against message size for PVM on Sun machines.

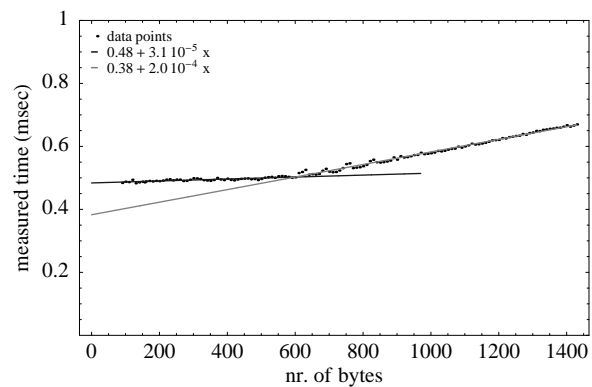


Figure B.2: Median time against message size for MPI on Sun machines.

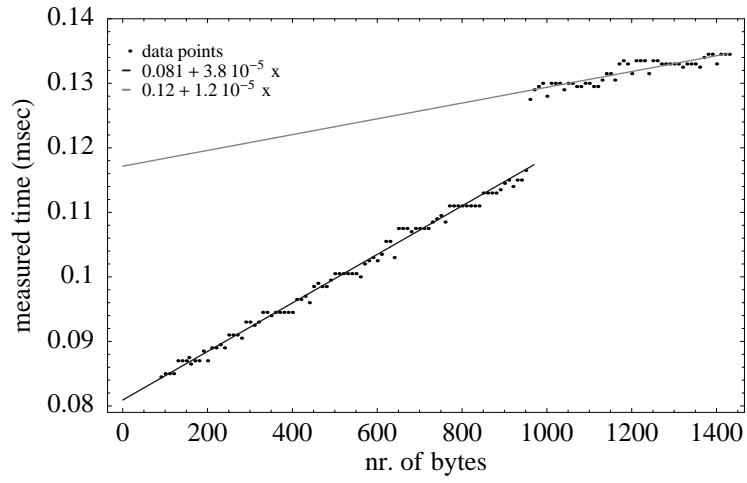


Figure B.3: Median time against message size for PVM on DAS nodes.

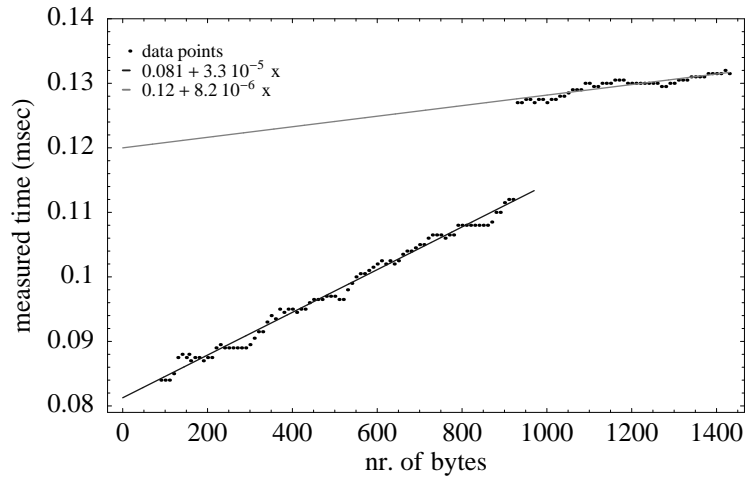


Figure B.4: Median time against message size for MPI on DAS nodes.

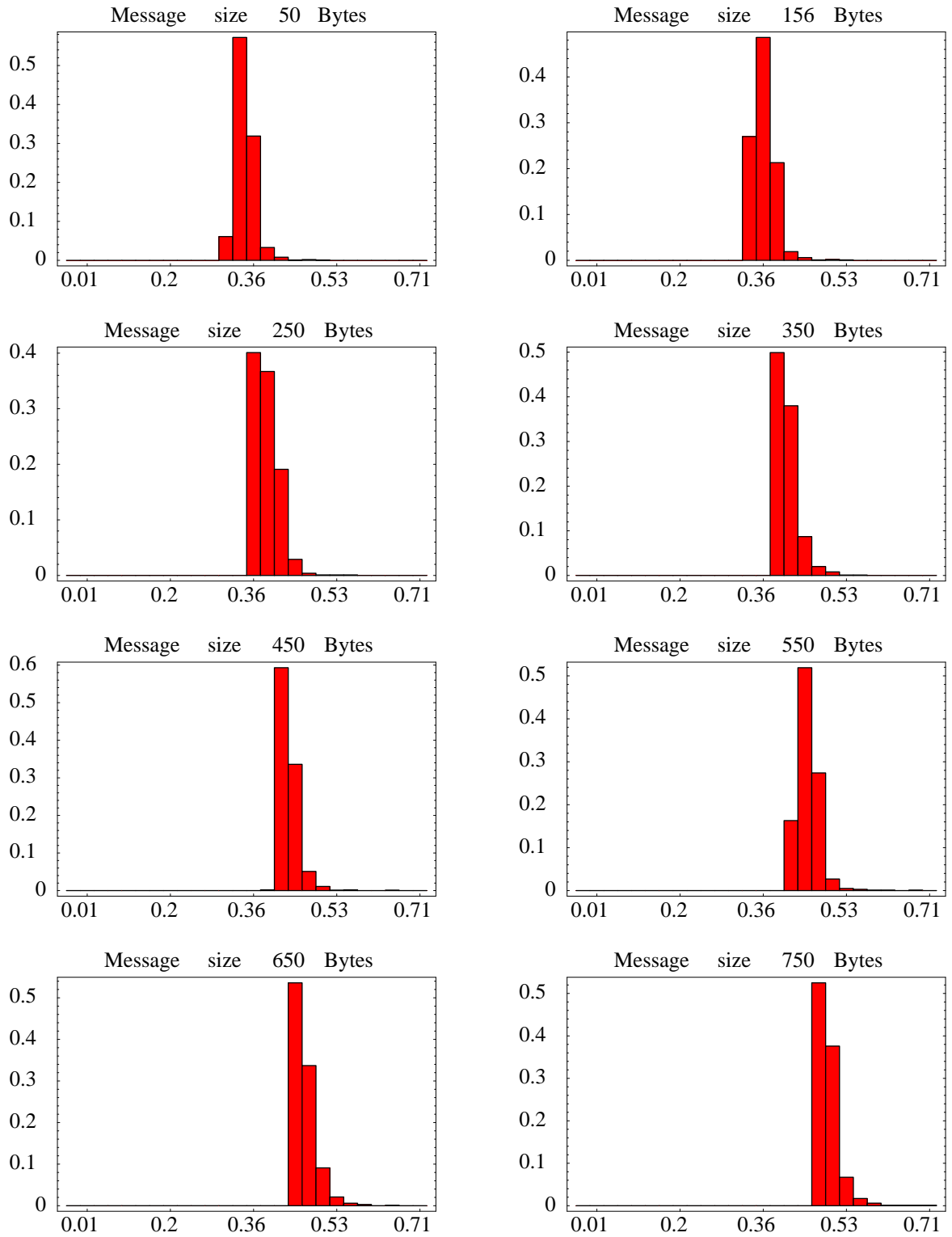


Figure B.5: PVM communication on Sun machines: frequencies (vertical axis) against measured communication times (horizontal axis, in msec).

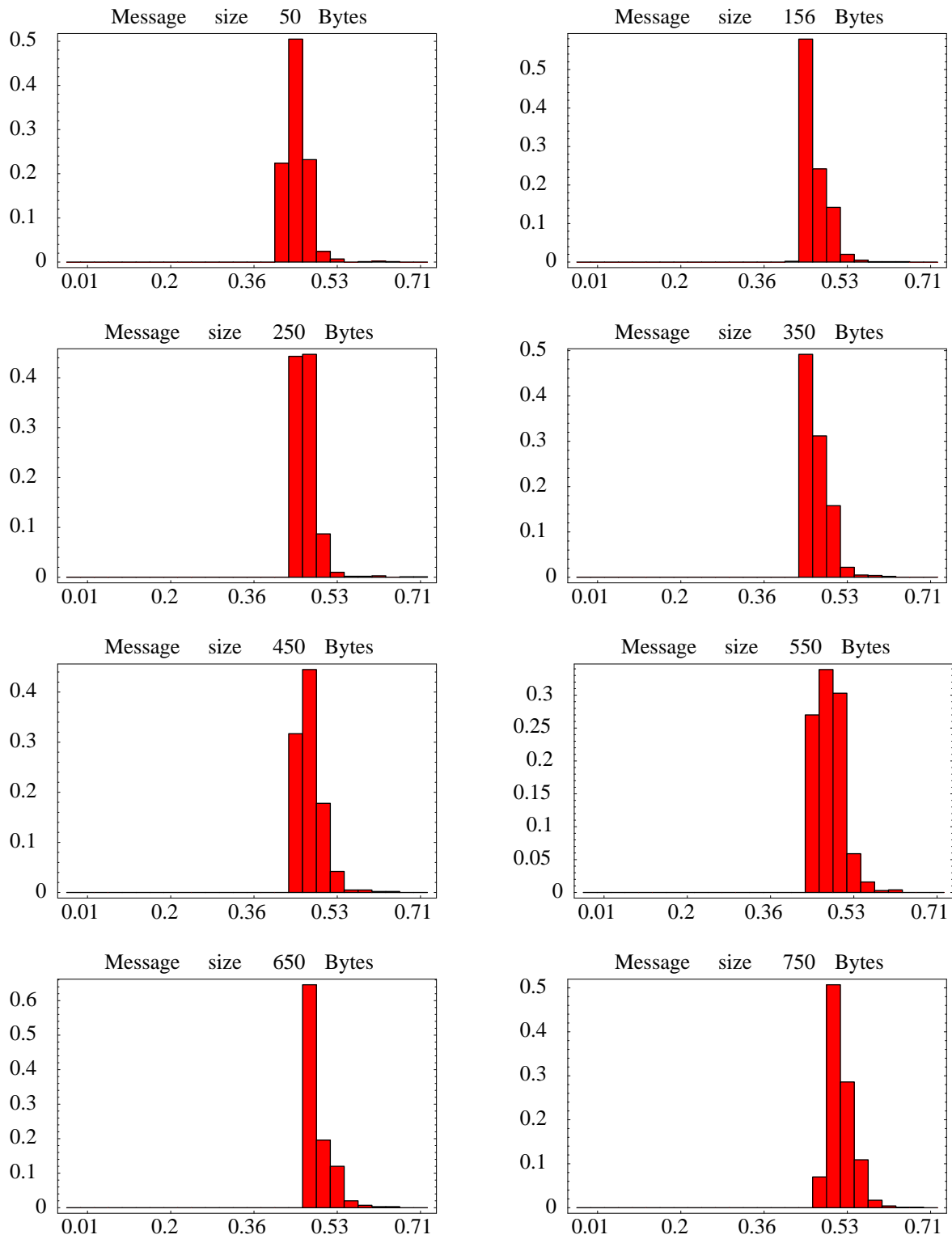


Figure B.6: MPI communication on Sun machines: frequencies (vertical axis) against measured communication times (horizontal axis, in msec).

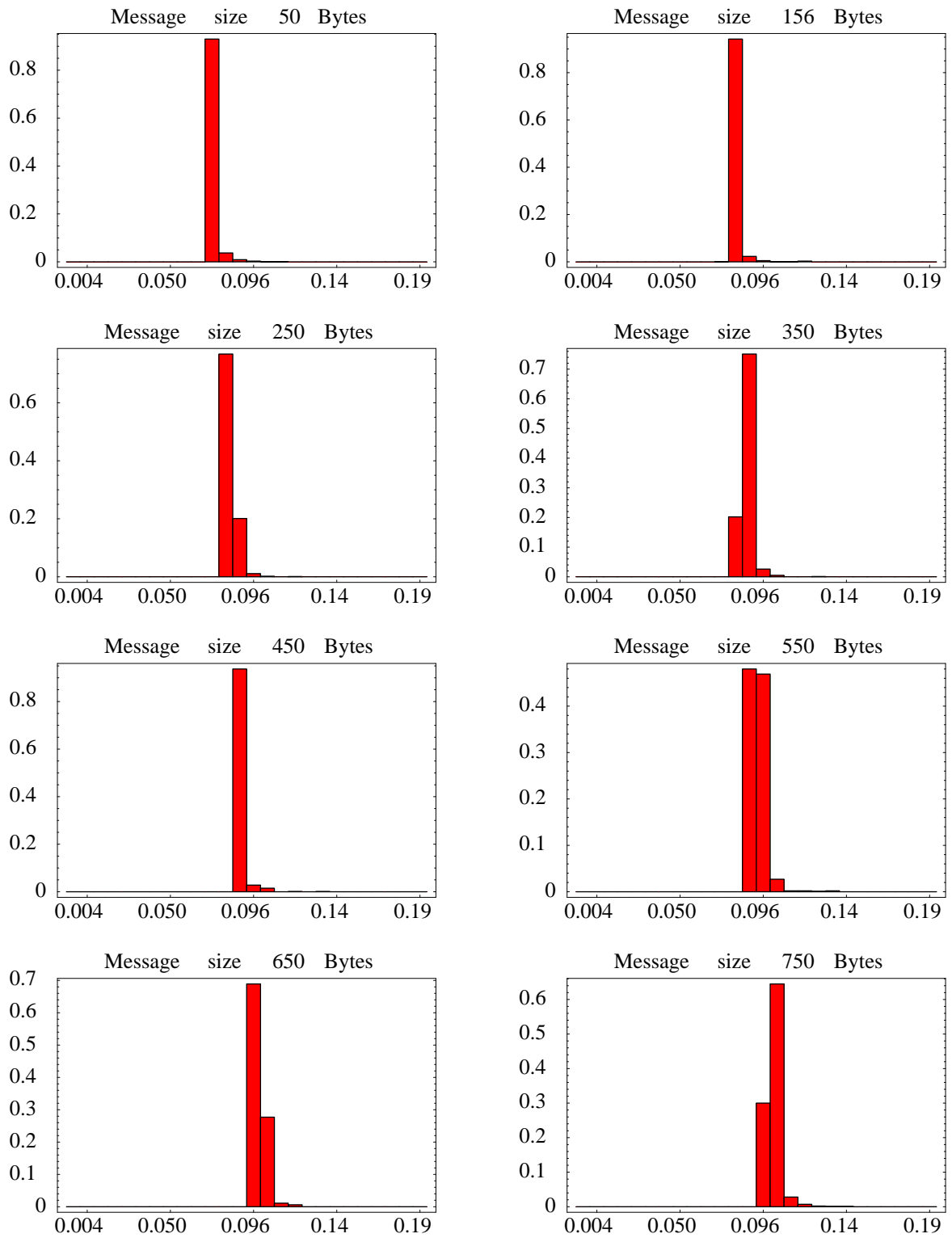


Figure B.7: PVM communication on DAS nodes: frequencies (vertical axis) against measured communication times (horizontal axis, in msec).

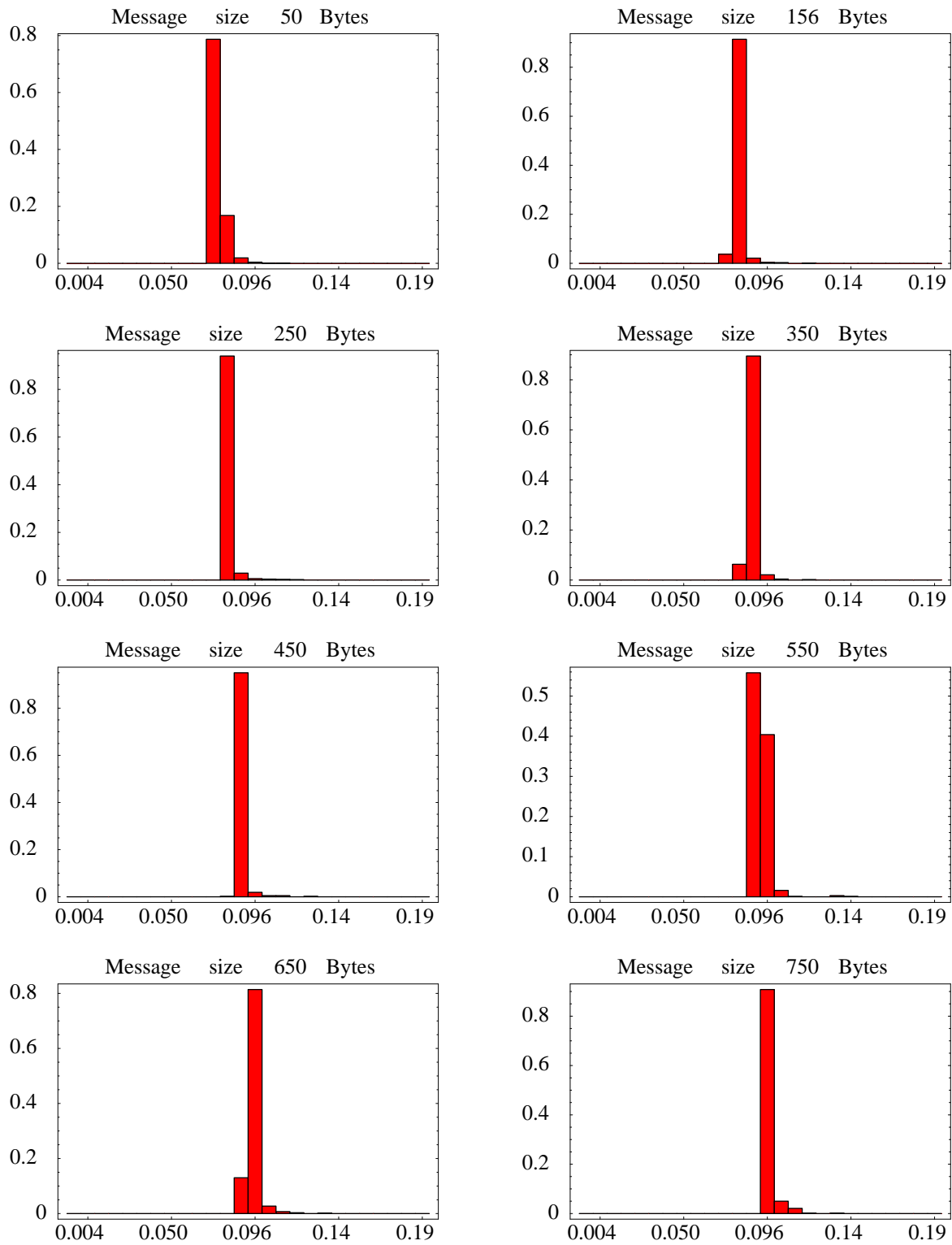


Figure B.8: MPI communication on DAS nodes: frequencies (vertical axis) against measured communication times (horizontal axis, in msec).

## B.2 PHOLD experiments

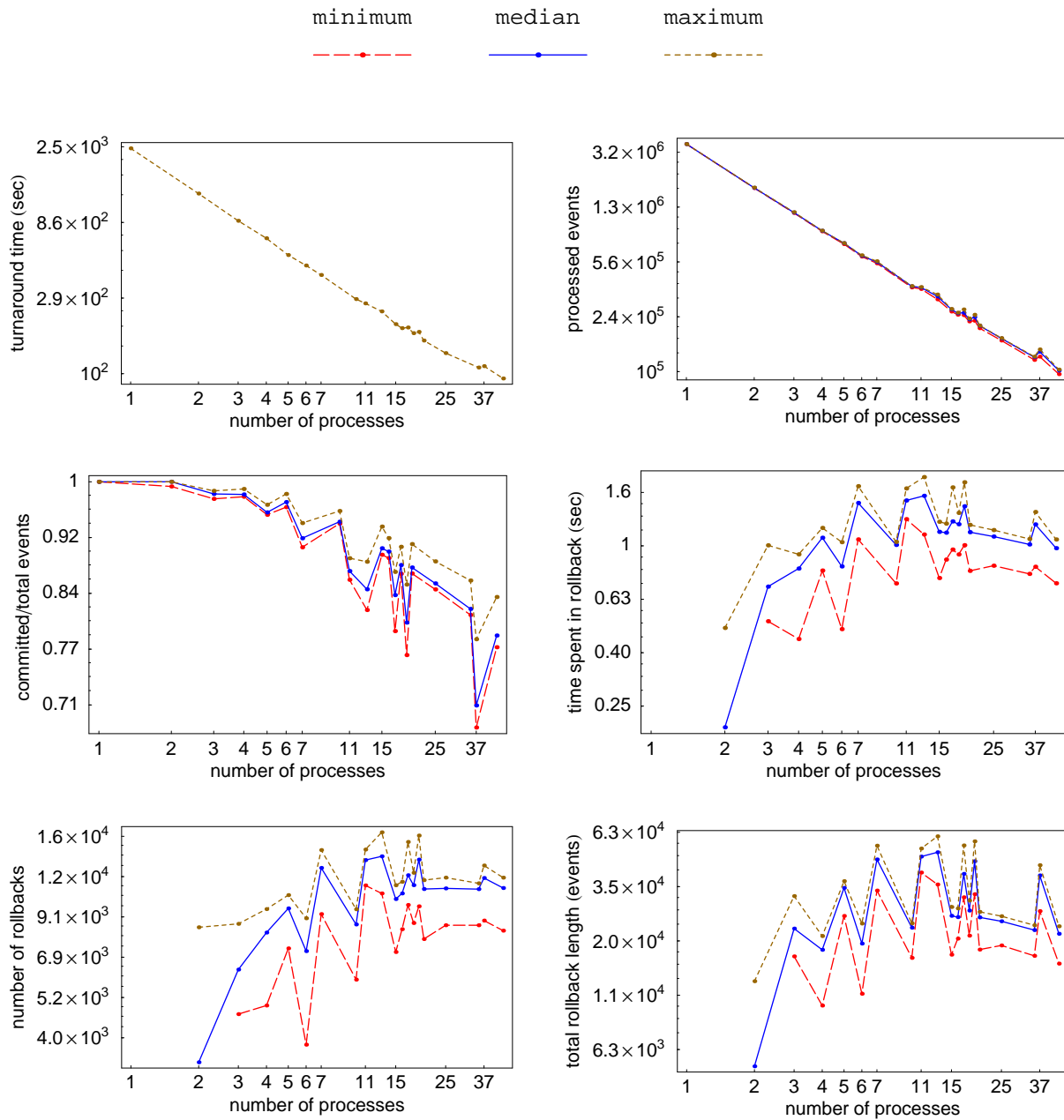


Figure B.9: Variation of the number of processes on the DAS using PVM (log-log plots, values of 0 have been omitted).

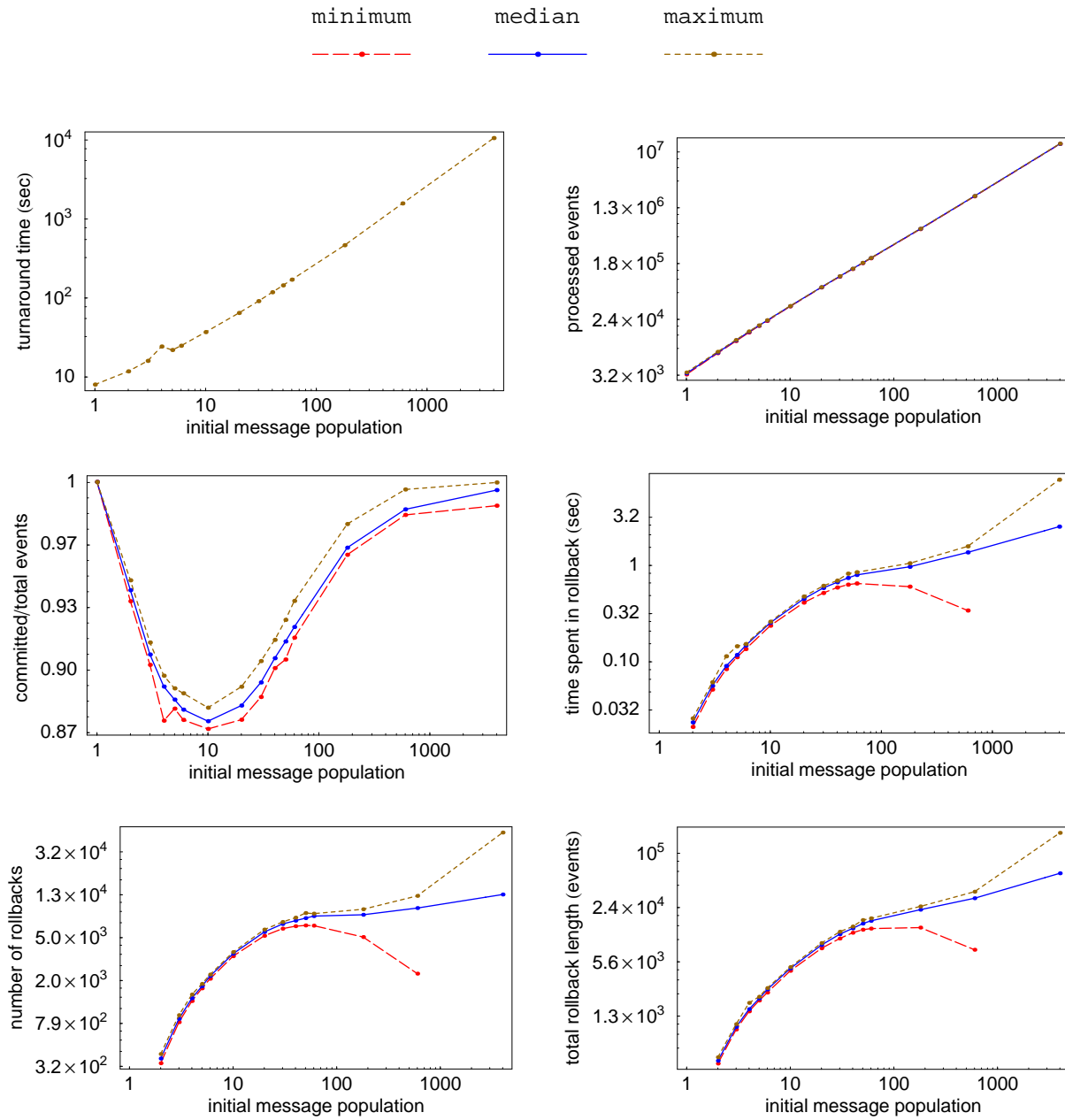


Figure B.10: Variation of the initial message population on the DAS using PVM (log-log plots, values of 0 have been omitted).

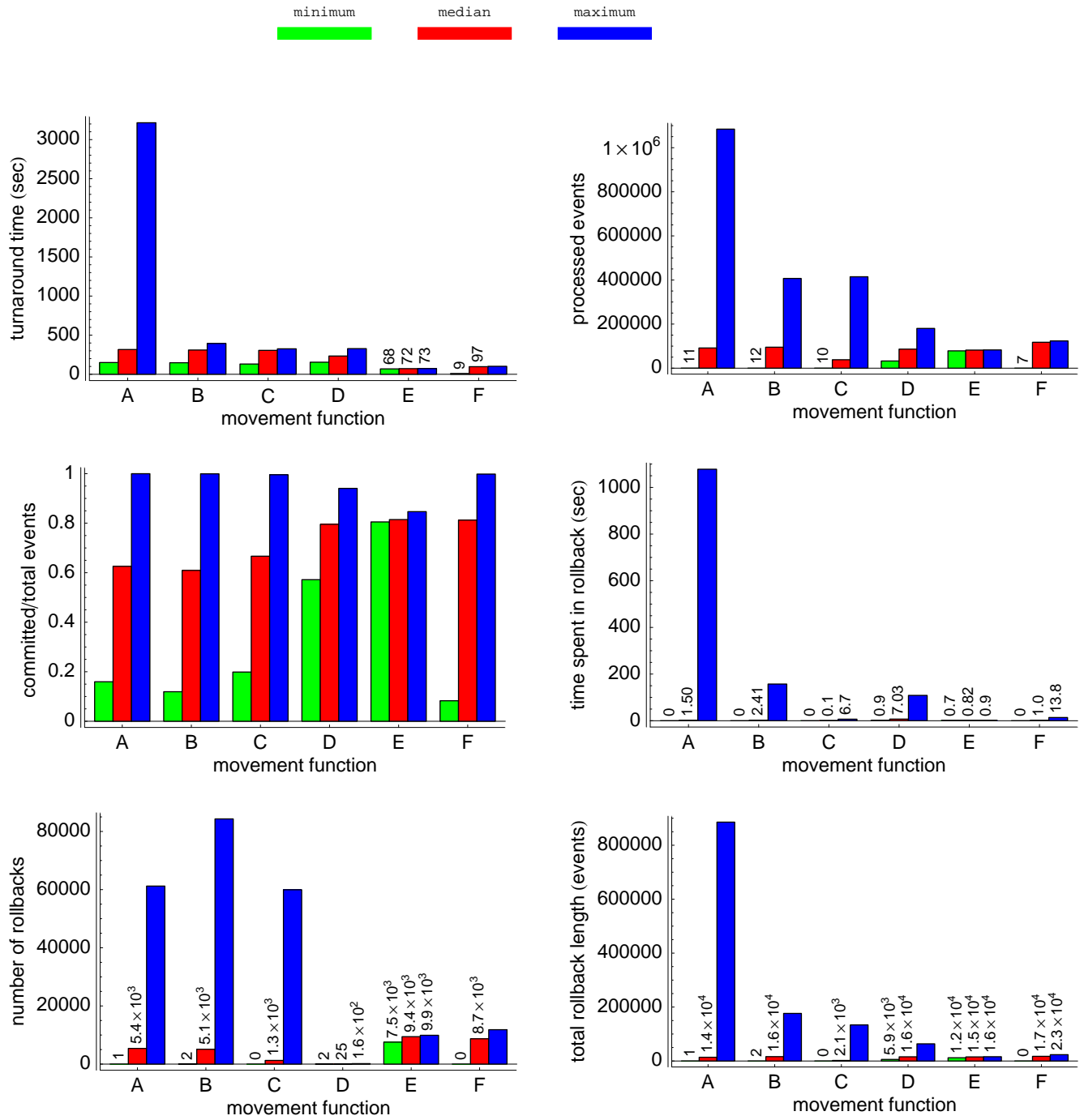


Figure B.11: Variation of the movement function on the DAS using PVM.

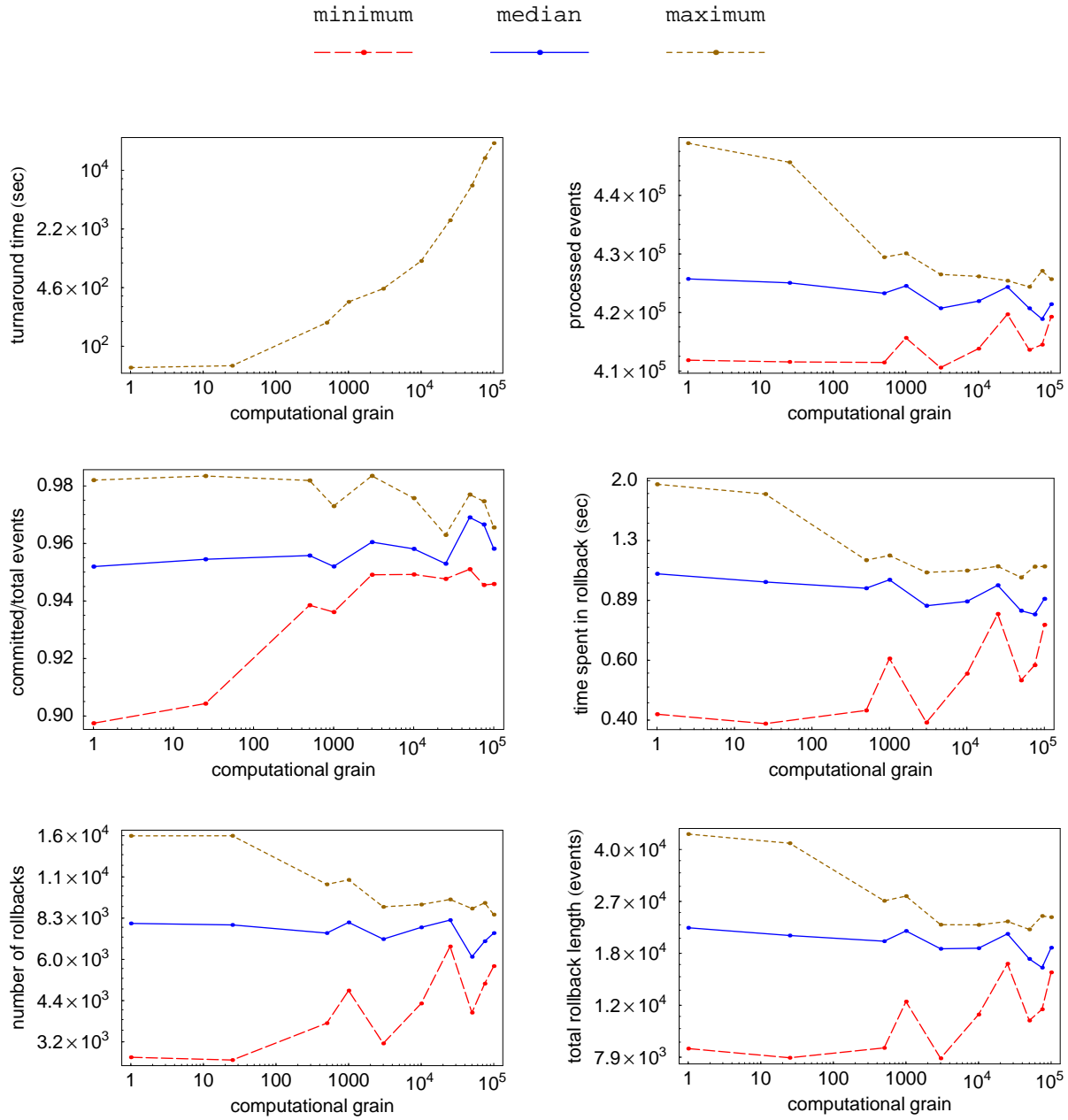


Figure B.12: Variation of the computational grain on the DAS using PVM (log-log plots).

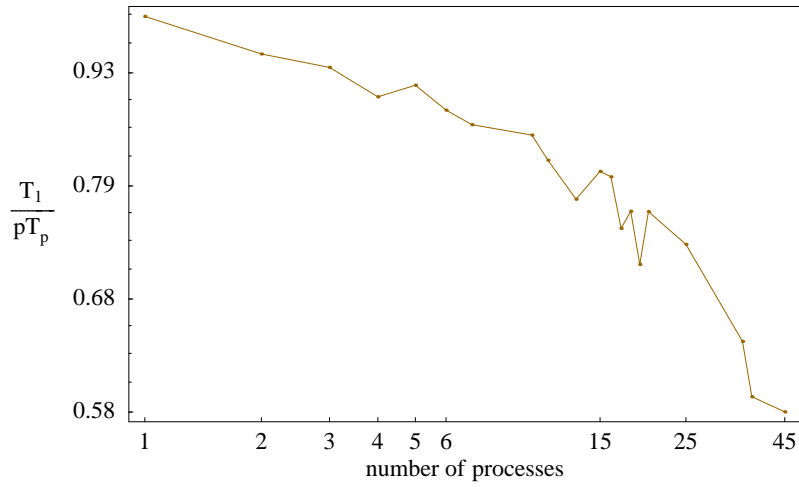


Figure B.13: Normalised turnaround time when varying the number of LPs on the DAS using PVM (log-log plot).

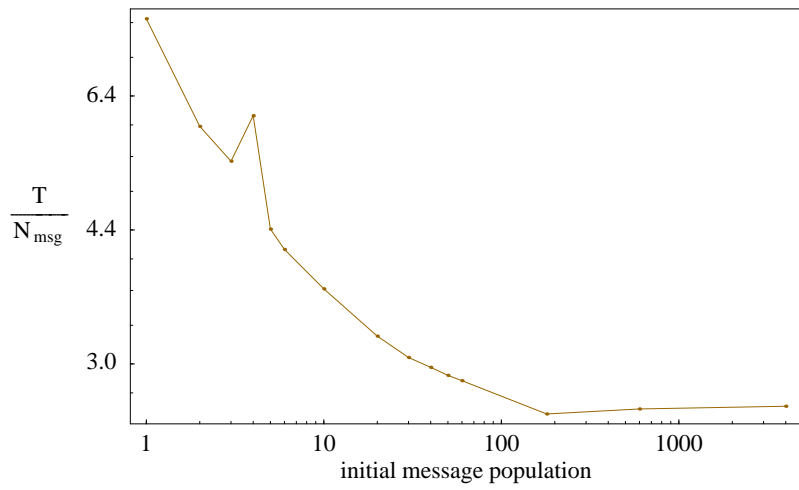


Figure B.14: Normalised turnaround time when varying the number of initial messages on the DAS using PVM (log-log plot).

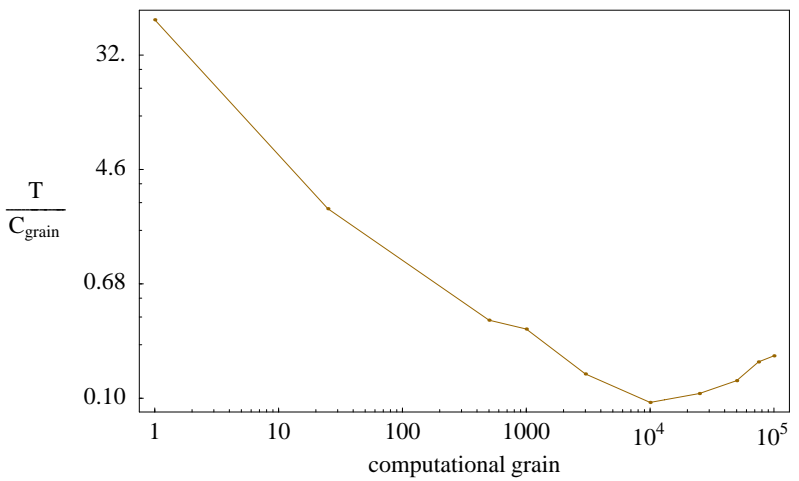


Figure B.15: Normalised turnaround time when varying the computational grain on the DAS using PVM (log-log plot).



# Bibliography

- [1] Herbert Bauer and Christian Sporrer. Distributed logic simulation and an approach to asynchronous GVT-calculation. In *ACM/SCS/IEEE Workshop on Parallel and Distributed Simulation (PADS)*, volume 24, pages 205–209, January 1992.
- [2] Sharon Calor. Cellular Automaton Models for Population Dynamics. Master's thesis, Faculty of Science, University of Amsterdam, Amsterdam, the Netherlands, March 12 1998.
- [3] Samir R. Das and Richard M. Fujimoto. Adaptive memory management and optimism control in Time Warp. In *ACM Transactions on Modeling and Computer Simulation*, volume 7, pages 239–271, January 1997. Available from <http://www.acm.org/pubs/articles/journals/tomacs/1997-7-2/p239-das/p239-das.pdf>.
- [4] A. Ferscha. *Parallel and Distributed Simulation of Discrete Event Systems*. Institut für Angewandte Informatik, University of Vienna, 1995. Available from <http://www.ani.univie.ac.at/~ferscha/E-PAPERS/handbook.ps.gz>.
- [5] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, Technical report. University of Tennessee, Knoxville, June 1995. Available from <http://www.mcs.anl.gov/mpi/mpi-report.ps>.
- [6] C. D. Carothers, R. M. Fujimoto and P. England. The effect of communication overheads on Time Warp performance: An experimental study. In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation (PADS '94)*, pages 118–125, July 1994.
- [7] R. M. Fujimoto. Discrete event simulation. In *Communication of the ACM*, volume 33, pages 31–52, October 1990.
- [8] R. M. Fujimoto. Performance of Time Warp under synthetic workloads. In *Proceedings of the SCS Multiconference on Distributed Simulation*, volume 22, pages 23–28, January 1990.
- [9] R. M. Fujimoto. Parallel and distributed simulation. In *Proceedings of the 1995 Winter Simulation Conference*, pages 118–125, 1995.

- [10] Özalp Babaoglu and Keith Marzullo. *Distributed Systems*, chapter Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms, pages 54–93. Addison-Wesley, second edition edition, 1996.
- [11] F. Gomes. A survey of GVT algorithms. 1993. Available from <http://www.cpsc.ucalgary.ca/~gomes/PAPERS/GVT.ps>.
- [12] Benno Overeinder, Bob Hertzberger and Peter Sloot. Parallel discrete event simulation. In W.J. Withagen, editor, *The Third Workshop Computersystems*, Faculty of Electrical Engineering, Eindhoven University, pages 19–30, Eindhoven, The Netherlands, May 1991. Available from [http://www.science.uva.nl/research/scs/papers/archive/Overeinder91\\_1.ps.gz](http://www.science.uva.nl/research/scs/papers/archive/Overeinder91_1.ps.gz).
- [13] Kai Hwang. *Advanced Computer Architecture, parallelism, scalability, programmability*. McGraw-Hill Inc, 1993. ISBN 0-07-031622-8.
- [14] D. R. Jefferson. Virtual time. In *ACM Transactions on Programming Languages and Systems*, volume 7, pages 404–425, July 1985.
- [15] S. Pakin, V. Karamcheti, and A. Chien. Fast messages: Efficient, portable communication for workstation clusters and massively-parallel processors. In *IEEE Concurrency*, volume 5, pages 60–73, April-June 1997. Available from <http://www.cs.nyu.edu/vijayk/papers/fm-pdt.ps>.
- [16] G. A. Geist, J. A. Kohl and P. M. Papadopoulos. PVM and MPI: A comparison of features. In *Calculateurs Paralleles*, volume 8, May 30 1996. Available from <http://www.epm.ornl.gov/pvm/PVMvsMPI.ps>.
- [17] Tim Rühl, Henry Bal, Raoul Bhoedjang, Koen Langendoen and Gregory Benson. Experience with a portability layer for implementing parallel programming systems. Technical report, University of Amsterdam, 1996. Available from [ftp://ftp.cs.vu.nl:/pub/amoeba/orca\\_papers/pdpta96.ps.Z](ftp://ftp.cs.vu.nl:/pub/amoeba/orca_papers/pdpta96.ps.Z).
- [18] M. Lauria and A. Chien. MPI-FM: High performance MPI on workstation clusters. In *Journal of Parallel and Distributed Computing*, volume 40, pages 4–18, January 1997. Available from <http://www-csag.cs.uiuc.edu/papers/jpdc97-normal.ps>.
- [19] S. Pakin, M. Lauria, and A. Chien. High performance messaging on workstations: Illinois fast messages (FM) for Myrinet. In *Supercomputing '95*, San Diego, California,, 1995. Available from <http://www-csag.ucsd.edu/papers/myrinet-fm-sc95.ps>.
- [20] C. D. Carothers, R. M. Fujimoto, Y. B. Lin and P. England. Distributed simulation of PCS networks using Time Warp. In *Proc. International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunications Systems*, pages 2–7, 1994.

- [21] C. D. Carothers, Y. B. Lin and R. M. Fujimoto. A re-dial model for personal communications services networks. In *Proceedings of the IEEE 45th Vehicular Technology Conference (VTC '95)*, pages 135–139, July 1995. Available from <http://www.cc.gatech.edu/grads/c/Chris.Carothers/PAPERS/vtc-95.ps>.
- [22] Yi-Bing Lin and Paul A. Fishwick. Asynchronous parallel discrete event simulation. In *IEEE Transactions on systems, man and cybernetics*, volume 26, pages 397–412, July 1996. Available from <ftp://ftp.cis.ufl.edu/cis/tech-reports/tr95/tr95-005.ps>.
- [23] Yi-Bing Lin and Edward D. Lazowska. A study of Time Warp rollback mechanisms. In *ACM Transactions on Modeling and Computer Simulation*, volume 1, pages 51–72, January 1991.
- [24] Jon B. Weissman Lisa M. Sokol and Paula A. Mutchler. MTW: an empirical performance study. In *Proceedings of the 1991 winter simulation conference*, pages 557–563, December 1991. Available from <http://www.acm.org/pubs/articles/proceedings/simulation/304238/p557-sokol/p557-sokol.pdf>.
- [25] V. K. Madiseti and D. A. Hardaker. Synchronization mechanisms for distributed event-driven computation. In *ACM Transactions on Modeling and Computer Simulation*, volume 2, pages 12–51, January 1992.
- [26] Malolan Chetlur, Nael Abu-Gazaleh, R. Radhakrishnan and P. A. Wilsey. Optimizing communication in Time-Warp simulators. In *Proceedings of the 12th workshop on Parallel and distributed simulation*, pages 64–71, May 26-29 1998. Available from <http://www.acm.org/pubs/articles/proceedings/simulation/278008/p64-chetlur/p64-chetlur.pdf>.
- [27] Al Geist, Adam Beguelin, Jack Dongorra, Weicheng Jiang, Robert Mancheck and Vaidy Sunderam. *PVM A Users' Guide and Tutorial for Networked Parallel Computing*. The MIT press, Cambridge, Massachusetts, London, England, 1994. Available from <ftp://netlib2.cs.utk.edu/pvm3/book/pvm-book.ps>.
- [28] J. Misra. Distributed discrete event simulation. In *Computing Surveys*, volume 18, March 1986.
- [29] D. M. Nicol and X. Liu. The dark side of risk (what your mother never told you about Time Warp). In *Proceedings of the 11th workshop on Parallel and distributed simulation*, pages 188–195, 1997. Available from <http://www.acm.org/pubs/articles/proceedings/simulation/268826/p188-nicol/p188-nicol.pdf>.
- [30] B. J. Overeinder and P.M.A. Sloot. Application of Time Warp to parallel simulations with Asynchronous Cellular Automata. In A. Verbraeck and E.J.H. Kerckhoffs, editors,

- European Simulation Symposium 1993, Society for Computer Simulation International*, pages 397–402, Delft, The Netherlands,, October 1993. Available from [http://www.science.uva.nl/research/scs/papers/archive/0vereinder93\\_1.ps.gz](http://www.science.uva.nl/research/scs/papers/archive/0vereinder93_1.ps.gz).
- [31] P. S. Pacheco. *A User's Guide to MPI*. University of San Francisco, San Francisco,, March 30 1998. Available from <ftp://math.usfca.edu/pub/MPI/mpi.guide.ps>.
- [32] M. Lauria, S. Pakin and A. Chien. Efficient layering for high speed communication: Fast messages 2.x. In *Proceedings of HPDC-7'98*, pages 28–31, Chicago, Illinois, July 1998. Available from <http://www-csag.ucsd.edu/papers/hpdc7-lauria.ps>.
- [33] L. Barriga, R. Rönngren and R. Ayani. Benchmarking parallel simulation algorithms. In *Proc. of the IEEE First Int. Conf on Algorithms and Architectures for Parallel Processing (ICA PP-95)*, pages 611–620, Brisbane, Australia, April 1995.
- [34] P. M. A. Sloot. Micro module 1: Modelling and Simulation. In the lecture notes of the course *Parallel Wetenschappelijk Rekenen en Simulatie*, pages 1–41, Department WINS, University of Amsterdam, Amsterdam, The Netherlands, 1995.
- [35] P. M. A. Sloot. Notebook 2: Population Dynamics. In the lecture notes of the course *Simulating and Modelling*, Department WINS, University of Amsterdam, Amsterdam, The Netherlands, 1995.
- [36] Jeff S. Steinman. *Discrete-event simulation and the event horizon*. Jet Propulsion Laboratory, California Institute of Technology, Pasadena, US, 1992. Available from <http://www.ca.metsci.com/speedes/papers/eventhorizonpart1.pdf>.
- [37] Jeff S. Steinman. Breathing Time Warp. In *Proceedings of the 1993 workshop on Parallel and distributed simulation*, pages 109–118, San Diego, CA USA, May 16-19 1993. Available from <http://www.acm.org/pubs/articles/proceedings/simulation/158459/p109-steinman/p109-steinman.pdf>.
- [38] UNIX. *manual page drand48(3C)*, January 22 1993. `erand48()` manual online on machine `carol.wins.uva.nl`.
- [39] Kees Verstoep. *The distributed ASCI Supercomputer (DAS)*, September 17 1999. WWW pages <http://www.cs.vu.nl/~versto/DAS>.
- [40] M. Snir, S. Otto, S. Huss-Lederman, D. Walker and J. Dongarra. *MPI: The Complete Reference*. The MIT press, Cambridge, Massachusetts, London, England, 1996.
- [41] Frederick Wieland. The threshold of event simultaneity. In *Proceedings of the 11th workshop on Parallel and distributed simulation*, pages 56–59, 1997. Available from <http://www.acm.org/pubs/articles/proceedings/simulation/268826/p56-wieland/p56-wieland.pdf>.