PROCEEDINGS

Southern African Computer Symposium 1991

6th de Rekenaarsimposium van Suider Afrika 1991

DE OVERBERGER HOTEL, CALEDON

2 - 3 JULY 1991

SPONSORED BY

ISM

FRD

GENMIN

EDITED BY

M H Linck

DEPARTMENT OF COMPUTER SCIENCE • UNIVERSITY OF CAPE TOWN

PROCEEDINGS / KONGRESOPSOMMINGS

6th
SOUTHERN AFRICAN COMPUTER SYMPOSIUM

6de
SUIDELIKE-AFRIKAANSE REKENAARSIMPOSIUM

De Overberger Hotel, Caledon

2 - 3 JULY 1991

SPONSORED by

ISM
FRD
GENMIN

EDITED by

M H LINCK

Department of Computer Science

University of Cape Town

TABLE OF CONTENTS

FOREWORD

The 6th Computer Symposium, organised under the auspices of SAICS, carries on the tradition of providing an opportunity for the South African scientific computing community to present research material to their peers.

It was heartening that 31 papers were offered for consideration. As before all these papers were refereed. Thereafter a selection committee chose 21 for presentation at the Symposium.

Several new dimensions are present in the 1991 symposium:

* The Symposium has been arranged for the day immediately after the SACLA conference.

* It is being run over only 1 day in contrast to the 2-3 days of previous symposia.

* I believe that it is first time that a Symposium has been held outside of the Transvaal.

* Over 85 people will be attending. Nearly all will have attended both events.

* A Sponsorship package for both SACLA and the Research Symposium was obtained. (This led to reduced hotel costs compared to previous symposia)

A major expense is the production of the Proceedings of the Symposium. To ensure financial soundness authors have had to pay the page charge of R20 per page.

A thought for the future would be consideration of a poster session at the Symposium. This could provide an alternative approach to presenting ideas or work.

I would sincerely hope that the twinning of SACLA and the Research Symposium is considered successful enough for this combination survive. As to whether a Research Symposium should be run each year after SACLA, or only every second year, is a matter of need and taste.

A challenge for the future is to encourage an even greater number of MSc & PhD students to attend the Symposium. Unlike this year, I would recommend that they be accommodated at the same cost as everyone else. Only if it is financially necessary should the sponsored number of students be limited.

I would like to thank the other members of the organising committee and my colleagues at UCT for all the help that they have given me. A special word of thanks goes to Prof. Pieter Kritzinger who has provided me with invaluable help and ideas throughout the organisation of this 6th Research Symposium.

M H Linck
Symposium Chairman

SYMPOSIUM CHAIRMAN

M H Linck, University of Cape Town

ORGANISING COMMITTEE

D Kourie, Pretoria University.

P S Kritzinger, University of Cape Town.

M H Linck, University of Capè Town.

SPONSORS

ISM

GENMIN

FRD

LIST OF REFEREES FOR 6th RESEARCH SYMPOSIUM

NAME	INSTITUTION
Barnard, E	Pretoria
Becker, Ronnie	UCT
Berman S	UCT
Bishop, Judy	Wits
Berman, Sonia	UCT
Brink, Chris	UCT
Bodde, Ryn	Networks Systems
Bornman, Chris	UNISA
Bruwer, Piet	UOFS
Cherenack, Paul	UCT
Cook Donald	UCT
de Jaeger, Gerhard	UCT
de Villiers, Pieter	Stellenbosch
Ehlers, Elize	RAU
Eloff, Jan	RAU
Finnie, Gavin	Natal
Gaynor, N	AECI
Hutchinson, Andrew	UCT
Jourdan, D	Pretoria
Kourie Derrick	Pretoria
Kritzinger, Pieter	UCT
Krzesinski, Tony	Stellenbosch
Laing, Doug	ISM
Labuschagne, Willem	UNISA
Levy, Dave	Natal

MacGregor, Ken	UCT
Machanick, Philip	Wits
Mattison Keith	UCT
Messerschmidt, Hans	UOFS
Mutch, Laurie	Shell
Neishlos, N	Wits
Oosthuizen, Deon	Pretoria
Peters Joseph	Simon Fraser
Ram, V	Natal, Pmb.
Postma, Stef	Natal, Pmb
Rennhackkamp, Martin	Stellenbosch
Shochot, John	Wits
Silverberg, Roger	Council for Mineral Technology
Smit, Riël	UCT
Smith, Dereck	UCT
Terry, Pat	Rhodes
van den Heever, Roelf	UP
van Zijl, Lynette	Stellenbosch
Venter, Herman	Fort Hare
Victor, Herna	Stellenbosch
von Solms, Basie	RAU
Wagenaar, M	UPE
Wentworth, Peter	Rhodes
Wheeler, Graham	UCT
Wood, Peter	UCT

6TH RESEARCH SYMPOSIUM - 1991

FINAL PROGRAM

TUESDAY 2nd July 1991

10h00 - 13h00 **Registration**

13h00 - 13h50 **PUB LUNCH**

14h00 - 15h30 SESSION 1A

Venue: Hassner

Chairman: Prof Basie von Solms

14h00 - 14h30
"A value can belong to many types. "
B H Venter, University of Fort Hare

14h30 - 15h00
"A Transputer Based Embedded Controller Development System"
M R Webster, R G Harley, D C Levy & D R Woodward, University of Natal

15h00 - 15h30
"Improving a Control and Sequencing Language"
G Smit and C Fair, University of Cape Town

SESSION 1B

Venue: Hassner C

Chairman:Prof Roelf v d Heever

14h00 - 14h30
"Design of an Object Orientated Framework for Optimistic Parallel Simulation on Shared-Memory Computers" P Machanick, University of Witwatersrand

14h30 - 15h00
"Using Statecharts to Design and Specify the GMA Direct-Manipulation User Interface" L van Zijl & D Mitton, University of Stellenbosch

15h00 - 15h30
"Product Form Solutions for Multiserver Centres with Heirarchical Classes of Customers" A Krzesinski, University of Stellenbosch and R Schassberger, Technische Universität Braunschweig

15h30 - 16h00 **TEA**

16h00 - 17h30 SESSION 2A

Venue: Hassner

Chairman: Prof Derrick Kourie

16h00 - 16h30
"A Reusable Kernel for the Development of Control Software" W Fouché and P de Villiers, University of Stellenbosch

16h30 - 17h00
"An Implementation of Linda Tuple Space under the Helios Operating System" P G Clayton, E P Wentworth, G C Wells and F de-Heer-Menlah, Rhodes University

17h00 - 17h30
"The Design and Analysis of Distributed Virtual Memory Consistency Protocols in an Object Orientated Operating System" K MacGregor, University of Cape Town & R Campbell, University of Illinois at Urbana-Champaign

19h30 **PRE-DINNER DRINKS**

20h00 **GALA CAPE DINNER**
(Men: Jackets & ties)

WEDNESDAY 3rd July 1991

7h00 - 8h15 BREAKFAST

8h15 - 9h45 SESSION 3A

Venue: Hassner

Chairman: Assoc Prof P Wood

8h15 - 8h45
"Concurrency Control Mechanisms for Multidatabase Systems" A Deacon, University of Stellenbosch

8h45 - 9h15
"Extending Local Recovery Techniques for Distributed Databases" H L Victor & M H Rennhackkamp, University of Stellenbosch

9h15 - 9h45
"Analysing Routing Strategies in Sporadic Networks" S Melville, University of Natal

SESSION 3B

Venue: Hassner C

Chairman: Prof G Finnie

8h15 - 8h45
The Design of a Speech Synthesis System for Afrikaans" M J Wagener, University of Port Elizabeth

8h45 - 9h15
"Expert Systems for Management Control: A Multiexpert Architecture" V Ram, University of Natal

9h15 - 9h45
"Integrating Simularity-Based and Explanation-Based Learning" G D Oosthuizen and C Avenant, University of Pretoria

9h45 - 10h15 TEA

10h15 - 11h00 SESSION 4

Venue: Hassner

Chairman: Prof P S Kritzinger
Invited paper: E Coffman

11h00 - 11h10 BREAK

11h10 - 12h40 SESSION 5A

Venue: Hassner

Chairman: Prof C Bornman

11h10 - 11h40
"Efficient Evaluation of Regular Path Programs"
P Wood, University of Cape Town

11h40 - 12h10
"Object Orientation in Relational Databases"
M Rennhackkamp, University of Stellenbosch

12h10 - 12h40
"Building a secure database using self-protecting objects" M Olivier and S H von Solms, Rand Afrikaans University

SESSION 5B

Venue: Hassner C

Chairman: Prof A Krzesinski

11h10 - 11h40
"Modelling the Algebra of Weakest Preconditions"
C Brink & I Rewitsky, University of Cape Town

11h40 - 12h10
"A Model Checker for Transition Systems"
P de Villiers, University of Stellenbosch

12h10 - 12h40
"A New Algorithm for Finding an Upper Bound of the Genus of a Graph"
D I Carson and O R Oellermann, University of Natal

12h45-12h55 GENERAL MEETING of RESEARCH SYMPOSIUM ATTENDEES

Venue: Hassner

Chairman: Dr M H Linck

13h00 - 14h00 LUNCH

FINIS 6th COMPUTER SYMPOSIUM

PAPERS

of the

6TH RESEARCH SYMPOSIUM

Design of an Object-Oriented Framework for Optimistic Parallel Simulation on Shared-Memory Computers

Philip Machanick
Department of Computer Science, University of the Witwatersrand, 2050 Wits

Abstract

Parallel simulation, if it is to become a mainstream technology, must become reasonably accessible to programmers without unusual skills. Since low-cost shared memory machines are becoming an increasing possibility, a simulation package designed to perform efficiently on such machines is desirable. This paper presents previous results indicating some performance issues to be taken into account in such a simulator, and presents a design for an object-oriented package, based on features of C++, for a simulator using the optimistic model. The major finding is that C++ is suitable for such a simulation package, although there a few difficulties are identified.

keywords: *Simulation, Parallel Processing, Object-Oriented Programming*
Computing Review Categories: *I.6.1, C.1.2, D.2.2*

1. Introduction

Parallel simulation is becoming an increasingly important area, as demands for simulation increase and parallel machines become more affordable. At the same time, if it is to be useful in organizations with a relatively low budget, it should not be significantly more difficult to program than sequential simulation. The starting point for the work reported on here is an investigation of techniques for implementing software efficiently on shared-memory machines, which rely on good caching behaviour to achieve scalability. Such machines are likely to become increasingly common—and affordable—in the future, as the speed gap between affordable high-performance processors (typically RISC) and affordable memory components increases [5]. The Silicon Graphics Iris series is an existing range of machines approximating to the predicted features. There are also strong indications that Sun is to launch a machine in this class soon.

The approach taken here is to design a framework for a specific model of parallel simulation, using object-oriented techniques to present a high-level interface to the simulation programmer.

This paper builds on experience of implementation of a specific parallel simulation in the object-oriented language C++, and presents a design for a general object-oriented library for such simulations. In the previous work [4], the Argonne National Laboratories (ANL) macros [17] were translated to C++ (making minimal changes). This approach was not entirely successful in that the macros are language extensions outside of the compiler, and debugging could be difficult (for example, it was generally necessary to read the expanded macros to interpret error messages). In addition, the macros are general parallel constructs, rather than simulation constructs, and in this sense present a low-level interface to the programmer.

One of the purposes of this paper is to demonstrate that C++ is a sufficiently powerful language to achieve the required functionality without extensions (either via macros or compiler changes). Another purpose is to present a design of a general package for parallel simulation primitives in an object-oriented language, based on performance-oriented research results. Much previous work on parallel simulation has been on unconventional architectures, such as vector processors [18], the Connection Machine [6; 3], Hypercubes [14] and the BBN Butterfly [16; 22].

The rest of this paper presents some background, followed by the design. The next section outlines general development of parallel simulation, with special emphasis on the optimistic model. The following section reports previous work in implementing a parallel simulation in C++. The design follows in the next section. This is followed by a summary of related work. In conclusion, some findings are presented, along with plans for future work.

2. Parallel Simulation

In this paper, 3 models of parallel simulation are considered: *conservative*, *optimistic* and *time-stepped*. Of these, the time-stepped simulations are the least interesting from the point of view of research, because they are relatively easy to implement. The conservative and optimistic methods are both event-driven, and are essentially in competition, while the time-stepped method is usually applicable to different problems. However, some insights derived from research into a time-stepped simulation have proved useful as a basis for this work.

Time-stepped simulations generally proceed by processing a large amount of data at specific discrete time intervals. They may be implemented on a range of architectures reasonably simply. On vector machines, they are implemented by scheduling all the computations of a given type so they can be done simultaneously to maximize vectorization [18]. On a SIMD machine such as the Connection Machine, the work can be distributed reasonably easily over the whole machine [6]. On a shared-memory or distributed memory machine, they can be implemented using barriers to synchronize time-steps.

The conservative method is a distributed implementation of a conventional event-list discrete-event simulation [19]. In the conservative model, each *logical process* (LP) in the system being simulated is mapped onto a process in the program, and the event list is distributed between LPs. A communication network between the processes supplies synchronization and data transfer. Each LP lp_i has a *clock value* T_i associated with it, which is the minimum timestamp of the most recent event it has received from each of its neighbours. A process blocks if none of the events in its local event list has a timestamp earlier than T_i.

A big problem with the basic conservative method is that it can deadlock. There are various techniques for deadlock detection, such as sending null messages which have no semantic value, but serve to advance the clock. A conservative simulation can be flooded with null messages. Some techniques exist to reduce the number of null messages, such as cancelling them when another message with a higher timestamp (from the same LP) is encountered [20].

Another problem with the conservative method is that there may be considerable time lost as a process blocks waiting for its neighbours. One approach to reducing this problem is the use of *lookahead*, in which a given LP knows the latencies, service times etc. of its neighbours and can predict how far into the future they will send a message if they have not as yet processed one. Lookahead can give good results [25], but has the drawback that it generally requires application-specific knowledge [11].

The optimistic method [13] avoids deadlocks and unnecessary delays by allowing an LP to continue processing even if it is ahead of its neighbours in simulation time. Each LP operates in its own *virtual time*, and the minimum virtual time over all the LPs is called *global virtual time* (GVT). If a message called a *straggler* is received which has an earlier timestamp than one already processed, there is a *causality error*, and any invalid work has to be undone. This is done by sending out *antimessages*, cancelling the effect of erroneous messages. This cancellation is effected by rolling state back to the time prior to the cancelled message. Old state has to be maintained to make rolling back possible. Saved state which is older than GVT is no longer needed, and is referred to as a *fossil*. Fossils must eventually be reclaimed, when memory becomes scarce; this is known as *fossil collection*. One proposed relatively low-cost approach to fossil collection is to clear the dirty bit in the memory manager's page table when a whole page is known to contain fossils[1]. Such a scheme is being investigated in new work on at least one experimental operating system (though from a different starting point): the V operating system[2].

The mechanism used to realize virtual time using antimessages and rollbacks is called *time warp*.

The intention is that time warp will use up what would otherwise be idle time, or deadlock detection or avoidance, in a conservative simulation—but it is possible that the overall cost will be higher than the saving. Analytical results show that the transition between cases where timewarp is a net win and examples where it is not (when rollback costs more than is saved) is smooth [9]. The implication is that techniques to reduce the amount of rollback are likely to to push the technique close to the performance of conservative methods, even in cases more favourable for conservative simulations. Some work has already been done to limit rollback, by placing a limit on how far any LP can go ahead of GVT [23]. There is however some doubt about the value of this approach, called *moving time window* (MTW), because it is non-discriminating in holding up LPs, regardless of whether the work they are doing "ahead" of GVT will turn out to be useful or not [11].

Another variation is the replacement of the *aggressive cancellation* strategy by *lazy cancellation*, in which antimessages are only sent

[1] David Jefferson: private communication, 1991.

[2] Kieran Harty: private communication, 1991.

out when it is determined that previous computations did in fact produce erroneous results. This approach results in a performance improvement in most cases, despite the extra overhead of having to check if the state has changed before deciding to cancel [21].

A recent proposal, *temporal decomposition*, is to allow an LP to be scheduled as a different process after a specific point in virtual time. The break between two versions of the LP is handled conservatively, in the sense that the later version is only allowed to start after there is no chance of rolling back to the previous version. This strategy allows for dynamic adjustments to load balance, even in a statically load-balanced system, by allowing the LP to be assigned to different processors before and after the split [22].

One drawback of optimistic methods is the difficulty in keeping track of state which may potentially need to be rolled back, in a general programming model with arbitrary side-effects [11]. There is a case for a higher level model to support state manipulation, to ensure that state is not changed without the knowledge of the rollback mechanism. Another problem is some categories of events cannot be rolled back (for example, if they result in a state change external to the simulation, such as output to the screen). The original time warp mechanism special-cases such events [13]. The *nonretractable event* (NRE) is a useful addition to the model. Such events cannot be rolled back, and so are only scheduled when GVT has caught up with them. Aside from providing a model for events which cause external state changes, they can also be used to implement flow control, including controlling the degree of optimism [1]. In the limit, if all events are nonretractable, the simulation becomes sequential[3] .

There is considerable scope for optimizing both conservative and optimistic methods, so that both come closer to solving problems best suited to the other (e.g., Wagner's paper [25] was a rebuttal of a claim that a specific problem was only suited to the optimistic method). However, the argument that the optimistic method is less dependent on application-specific optimizations [11] is attractive.

For this reason, the new work introduced in this paper is oriented towards the optimistic method. Nonetheless, the experience being drawn on is from work with a time-stepped method; there is sufficient overlap of issues to make this valid.

[3] This is worse than conservative, since a conservative simulator uses neighbours to determine the local clock, whereas an NRE is held up until the *global* minimum clock has caught up with it. A weaker condition on when the NRE may be scheduled is possible, but is likely to lead to a requirement for the kind of mechanism found in conservative simulators to prevent deadlock.

3. Experience with a Particle-Based Simulator

This section introduces aspects of previous experience with parallel simulation, focussing on aspects relevant to the design presented in this paper. In particular, the overall program structuring techniques which are designed to achieve good caching behaviour are of relevance.

The application is MP3D, a particle-based simulation of a wind tunnel. The simulation was originally implemented on a Cray 2 [18], and the version initially used had been ported to an Encore Multimax with relatively few changes from the vector version. This Encore version has been used as a benchmark example of a program with poor caching characteristics on shared-memory machines [12]. This initial version is called MP3D–0 (actually with minimal changes from the Encore version to convert it to C++[4]).

Each timestep, particles are moved (depending on their velocity components), and paired for possible collision. A probabilistic selection function is used to determine whether the collision actually takes place. The n^2 problem of finding collision partners is reduced to $\Theta(n)$. The strategy is to divide space into unit-cube cells. After particles are moved, they are randomly paired within a cell, rather than attempting to find nearest neighbours [18]. In MP3D–0, randomization is achieved by having processors contend for particles to process, both in the move and collide phases, and by randomly ordering particles within the array in which they are stored.

There are three major weaknesses in the organization of MP3D-0. The first is the fine-grained approach, which makes poor use of caches, and the second is its data structures (following the organization originally used for the vector implementation) are in the form of several arrays, each containing a single attribute of a particle, for example, its x spatial co-ordinate. The final problem is the random positioning of particles in arrays, which makes for poor locality.

The first improvement made to MP3D was to group related data together into objects, which are padded and aligned to fit cache blocks. This modification has the effect of avoiding *false sharing*, in which the same cache block contains unrelated data items, while increasing the *prefetch* effect of large cache blocks (all the data relating to one particle may be fetched after a single cache miss). The resulting program, MP3D–1, has improved caching behaviour, but is still poor.

[4] These changes have no measurable impact on performance, though the code size is increased, owing to the larger C++ libraries.

The second improvement was to reorganize the program around the spatial locality inherent in the model. The resulting revision, MP3D–2, is a major restructuring of the program. Each processor is given a fixed area of space, containing a collection of cells called a *precinct*. Each timestep, a processor moves all its particles, interchanges those which are no longer in the same precinct with its neighbours[5] and then does collisions. Because the randomness of association between particles and processors can no longer be relied upon to ensure collision partners are randomly paired, randomization is done explicitly. Each particle contains the necessary information to find the cell it belongs to (needed once it has been moved), and each cell contains the identity of the precinct that owns it. The key aspect of this implementation is that data stays with the same processor as long as possible, thereby avoiding the high number of cache invalidations of the previous implementations.

MP3D–2, while a specific restructuring exercise, is a basis for evaluating some general techniques, and the suitability of C++ for implementing them. Some of these strategies are:

- increasing *processor affinity*—ensuring objects and groups of related objects are processed by the same processor as far as possible, rather than reassigned in a very fine-grained way
- *separation*—the forcing of unrelated data into separate cache blocks, thereby minimizing false sharing
- *co-location*—placing data which will be accessed at approximately the same time and with similar read-write characteristics (e.g., accessed by one processor) in the same cache block, thereby enhancing the prefetch effect

These strategies have been realized in MP3D–2 by two major techniques: the use of a *space directory* to support the relating of objects to the precincts (which correspond to a unit of space allocated to a processor), and the use of overloaded memory allocators, which replace the standard C++ allocators.

Simulations show that MP3D-1 has a miss ratio about 5 times lower than MP3D–0, while MP3D–2 improves over MP3D–0 by at least an order of magnitude. The significance of these changes depends on a number of factors, such as the fraction of references which are to shared memory. However, with a tendency is towards greater penalties for cache misses (up to hundreds of processor cycles), and large cache blocks [15], such changes in caching behaviour are likely to become increasingly significant. Some designs of fast uniprocessors already have cache block of 128 bytes [2], and this is a trend which is expected to continue [15].

It is possible to measure the approximate effects of the differences between the programs by measurements on machines currently available, even if they do not exactly match these characteristics.

Machines on which measurements have been made are a DEC Firefly with MicroVAX CPUs, a Firefly with essentially the same memory system but faster CVAX processors, a 4-processor Silicon Graphics SGI 4D/240S and a faster 4D/380S with 8 processors. The improved program structuring shows up in improved speedup, and more effective use of faster processors. Figures [4] supporting these claims appear in Table 1. Speedup is given relative to the uniprocessor version of the same program, as we are concerned with scalability rather than with algorithm analysis. Speedup for faster processors is given relative to the equivalent run on a slower version of the same machine.

The Silicon Graphics is a good example of the modern trend in shared-memory machines. It has fast MIPS CPUs, 16-byte cache blocks[6] and a relatively low memory bus bandwidth in relation to the speed of the processors. The Firefly is an older design. Its cache blocks are only 4 bytes (i.e., too small for false sharing to be an issue), and the purpose of the caches is more to reduce bus traffic than to reduce memory latency [24]. Despite these differences between these machines and from expected future architectures, the restructuring of MP3D produces good results on both types of machine, suggesting that the proposed techniques are reasonably general.

4. A Design

The strategy is to start with requirements for optimistic simulators, and to work towards the insights derived from the MP3D exercise (the low-level of the design).

The general goals for the design are transparency, efficiency and extensibility. The intention is to provide a general framework for optimistic simulations, incorporating the most common features, while making allowance for the addition of new features. Efficiency is to be achieved by low-level optimizations, which are hidden from the programmer, using the information-hiding capabilities of an object-oriented language.

[5] The simulation is constructed so that particles will not move more than one cell, so it is possible to determine in advance which precincts are "neighbours".

[6] Not as large as some more recent designs, but large enough for false sharing to be an issue.

machine and number of processors		MP3D–0 s	sp.r	sp.f	MP3D–1 s	sp.r	sp.f	MP3D–2 s	sp.r	sp.f
SGI 4D/240S	1	677			597			678		
	2	503	1.3		410	1.5		369	1.8	
	4	407	1.7		283	2.1		216	3.1	
SGI 4D/380S	1	544		1.2	437		1.4	569		1.2
(faster CPUs)	2	455	1.2	1.1	315	1.4	1.3	312	1.8	1.2
	4	409	1.3	1.0	268	1.6	1.1	179	3.2	1.2
	8	431	1.3		265	1.6		141	4.0	
MicroVAX Firefly	1	12839			12700			9567		
	2	8096	1.6		8389	1.5		5116	1.9	
	4	5818	2.2		5810	2.2		2732	3.5	
CVAX Firefly	1	4854		3.9	4161		3.1	3303		2.9
(faster CPUs)	2	4478	1.1	1.8	3964	1.0	2.1	1790	1.8	2.9
	4	4002	1.2	1.5	3632	1.1	1.6	1014	3.3	2.7

Table 1. Run time and speedup: 3 versions of MP3D on 4 machines
s is time in seconds, *sp.r* is speedup with relative to the uniprocessor case and *sp.f* is speedup of an equivalent run on a similar machine with faster processors

The features in the initial implementation are chosen from those which have been generally accepted, with allowance for addition of some of the less tested innovations.

State changes should generally be made through state manipulation primitives, to ensure that the rollback mechanism is reasonably simple. Nonretractable events are supplied to allow a back door for more general state changes. Although they are a relatively untried concept, a similar mechanism would in any case have to be supplied to handle input and output. In the initial model, aggressive cancellation is used, and lazy cancellation is left as an extension. Moving time window is not implemented in the initial model, because there is uncertainty as to its usefulness. There is in any case potential for investigating the claims for nonretractable events as a throttling mechanism for excessive optimism (since NREs are needed anyway).

The remainder of this section presents a high-level view of the design, the approach to implementation in C++, some C++ techniques and an outline of the resulting classes.

4.1 high-level design

The structure of the simulator is broken down into LPs. Each LP contains an input queue, containing events not yet processed, stored in timestamp order. State associated with the LP is locally stored. An event dispatcher schedules and executes events in timestamp order, and puts events to be passed to neighbours in an output queue, and makes copies of the events in the antimessages queue. When a straggler arrives, the dispatcher passes control to the rollback mechanism, which decides which antimessages should be scheduled for transmission and local execution. Those for transmission are sent to the neighbours, and those for local execution are used to determine how to roll the local state back.

The relationships between the components of an LP are illustrated in Figure 1.

LPs are grouped together, to maximize spatial and temporal locality. Each such group is assigned to a single processor, in the style of the precincts used in MP3D. Load balance is more complicated to determine in an optimistic simulator than a pessimistic or timestepped simulator. As long as an LP does not run out of internally generated events, or stick on an NRE, it will not be idle. However, it is undesirable that a single LP should get arbitrarily far ahead, and flood the network with anti-messages. A definition of load balance in terms of the distance an LP is ahead of GVT is one possibility. This is not ideal, for the same reason that moving time windows are potentially a problem: the fact that an LP is far ahead is not necessarily a problem, if all the work it is doing is valid. However, changing the load balance will not result in idling a processor unnecessarily, as happens in MTW.

One possibility for dynamic load balance is a temporal decomposition scheme. However, this is overkill on shared-memory machines, as transferring state to another processor is accomplished relatively cheaply by passing a pointer, and allowing the caching mechanism to transfer data on demand (saved state for rollbacks in particular will not be transferred if it's not needed). Something very similar to temporal decomposition can in any case be achieved by scheduling an NRE as part of the protocol for moving an LP to a new precinct.

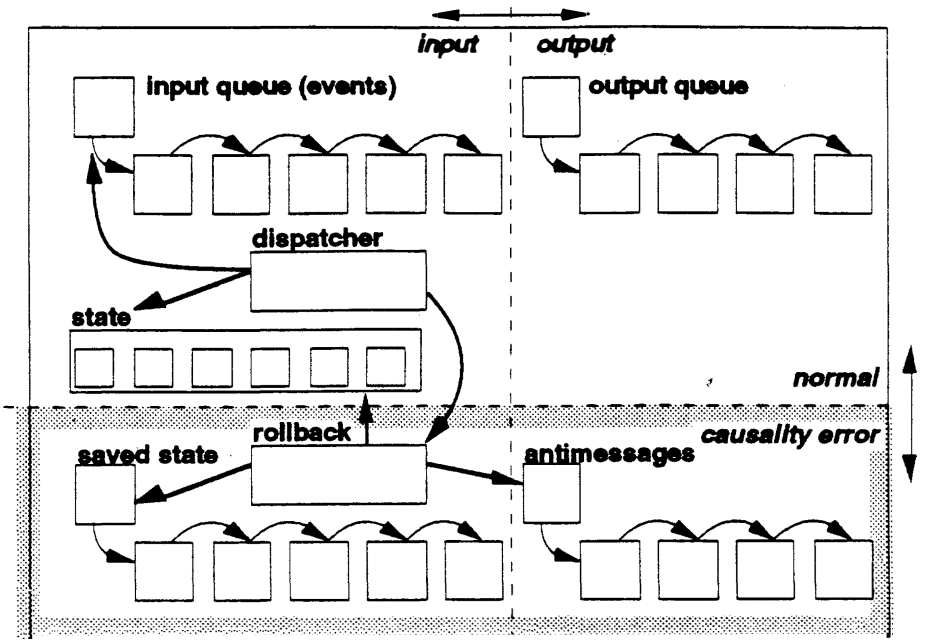


Figure 1. High-level view of a logical process
Only the dispatcher may change state, except in the case of a nonretractable event.

This NRE would in effect force the LP to wait until GVT had caught up with it before it was transferred, ensuring that no checkpointed state would need to be copied.

The details of the load balance process will be empirically determined, based on performance measurements. Figure 2 illustrates grouping of LPs in precincts, and how this may change dynamically. If a precinct is "too conservative", it will cause a lot of rollback in its neighbours (which may go too far ahead of GVT). Achieving a good load balance has to take this into account as well as simply ensuring the processors are equally loaded.

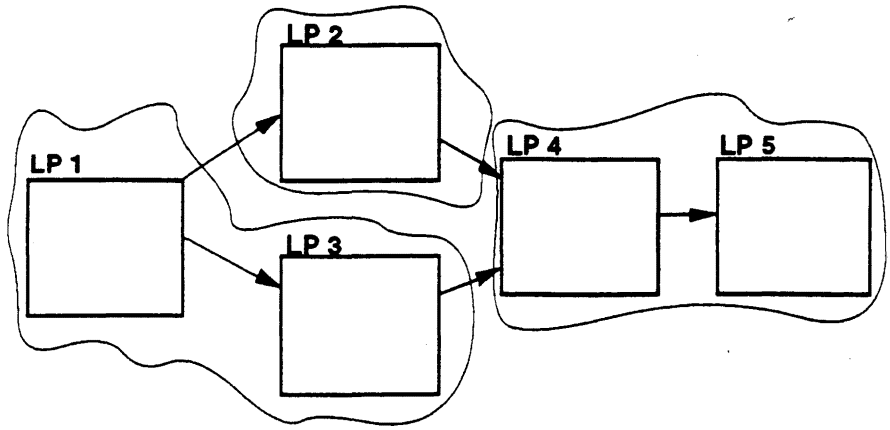
There is a general-purpose utility process, which does work such as checking load balance and periodically recomputing GVT. This process is executed by any processor which is idle, or after a new value of GVT is required (for example, to do fossil collection). This process is available to each precinct, and can be run by any of them.

4.2 some detail: the C++ approach

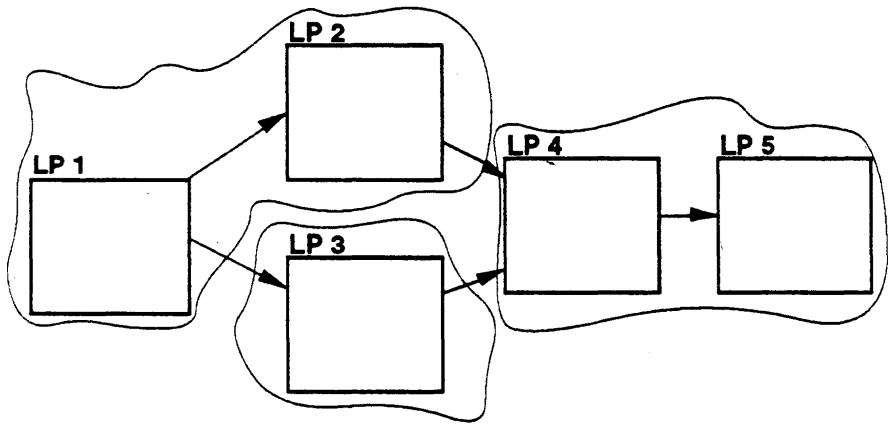
To implement these ideas in C++ (taking into account the findings of the MP3D research) it is useful to divide classes into several layers, representing the hierarchy of abstraction, from the simulation programmer's view down to machine-specific details. There is a hierarchy of classes, with inheritance links through each of these layers. An overview of the class hierarchy and its relationship to the layers is shown in Figure 3.

The layers (with examples of the classes defined at each layer) in the design are:

- *kernel*—these are the machine-specific details, at the base of the underlying threads package, such as locks, memory allocators and process creation and termination. Classes at this level include *Object*, which encapsulates low-level memory allocation and deallocation, *Lock*, which locks a specific named lock object until the end of the scope in which the Lock appears and *Process*, which launches a new process.
- *low_level_simulator*—low-level implementation details of the simulation mechanism, including time warp, message-passing between LPs and generic versions of messages. Classes at this level include *Precinct*, *LP*, *Generic_queue*, *Saved_state*, and *Antimessage*.



(a) LP1 and LP3 in the same precinct are found to be "too optimistic" whereas LP2 is "too conservative". LP4 and LP5 are in the same precinct, and appear to have the right amount of work for one processor.



(b) LP1 and LP2 now in the same precinct, and LP3 by itself.

Figure 2. Allocation of LPs to precincts and load balance

- *user_level_simulator*—higher-level view of the simulation classes. These could be used directly in simple simulations, but more typically would be used as a base for user-derived classes. Classes at this level include *Simulation_environment*, *State*, *Message* and *NRE_message*. Some of these classes are needed to implement the low_level_simulator classes (e.g., *Antimessage* is derived from *Message*).

4.3 some C++ techniques

Inheritance plays an important role in the design. One example is the implementation of anitimessages. An antimessage is a part of the *low_level_simulator* layer, and therefore cannot be redefined by the programmer. However, class *Antimessage* contains a pointer to a *Message* object, which may be a derived class created by the programmer, rather than the original class *Message*. As long as the member functions (C++ terminology for methods) needed by *Antimessage* have not be redefined in an inconsistent way, *Antimessage* will function as intended.

One of the ideas used in various parts of the design is the notion of defining[7] an object, the sole purpose of which is to generate correct initialization and termination, using the implicitly called constructors and destructors. This technique has been used for implementing tracing [7]. The general idea is an object is defined, resulting in a call of its constructor at the place of definition. At the close of the scope where the definition occurs, the destructor is automatically called. For example, in the C++ fragment:

```
{ Simulation_environment sim;
    // code to do simulation
}
```

[7] In C++, as in C, a name may be *declared*, in the style of a forward declaration in other languages such as Pascal. A full declaration is called a *definition*.

the object *sim* exists between its definition and the close of scope symbol, }. The purpose of this construct is to call the constructor for *Simulation_environment* objects, which does some global initialization. The destructor—automatically generated by the compiler—ensures termination code is called, which is useful to ensure that calling the terminating code is not forgotten by the programmer. A similar approach is used for implementing locks (though these are not expected to be used by simulation programmers).

Each class may contain *static data members* (class variables, in Smalltalk terminology). This is a useful feature for avoiding the use of global variables. In the design, this feature is used in two contexts: memory management and maintaining global state. The low-level memory management routines, defined in the kernel, have the option of taking a free space list as an argument. Where this feature is used, the free space list is stored as static data member in the class defining the specific allocator and deallocator.

Optional parameters make it possible to use the same low-level allocators, whether the programmer-supplied free space list is to be used or not.

Another useful feature is capability of overloading of the array index operator, [ ]. This makes it possible to a variable-sized 3-dimensional array, with same syntax as a statically declared one. The first two indexing operations return slices of the space directory, with decreasing dimensionality each time, and the third indexing operation returns the element desired. This capability would be useful in simulations where spatial relationships between LPs are a simple function of their positions in space, as in MP3D.

4.4 some C++ classes

This section presents an outline of the external interfaces of the classes.

The *kernel* defines the following classes:

- *Object*—defines low-level memory management, including alignment and padding to fit cache blocks.
- *Lock_data*—defines data structures to implement locks.
- *Lock*—the constructor for this class sets a lock (identified by an object of class *Lock_data*), and the destructor releases the lock at the close of the current scope. A typical example looks like this:

```
{ Lock output(output_lock);
  // code to do some output
}//lock released here
```

- *Process*—its constructor launches a thread, calling the function passed to it as a parameter, and its destructor cleans up after process termination.
- list manipulation and other generic data structure primitives, such as dynamically resizeable 3-dimensional arrays.

The *low_level_simulator* level defines the following classes:

- *Precinct*—a precinct contains a list of LPs, and a scheduler to decide which to execute next. It also knows which other precincts are its neighbours, and manages message exchange between itself and them.
- *LP*—a logical process; this relies on the semantics of *Message* and *State* to implement a specific simulation, and *Precinct* defines communication with LPs on other processors.
- *Saved_state*—contains list of changes to state, including identities of messages that caused the changes, stored as antimessages; the rollback controller is contained in this class, as is the fossil collector.
- *Anti_message*—contains a copy of the original message, plus its own timestamp.

Classes defined in *user_level_simulator* include:

- *State*—includes state change primitives and recording changes for rollback. The programmer will have to derive a new class from this one to incorporate the semantics of the state for the actual simulation problem.
- *Simulation_environment*—one object of this type is defined at the start of the program. As with *Lock*, its constructor and destructor respectively initialize the program for simulation, and clean up at the end.
- *Message*—contains a timestamp and identification of its originating LP. Uses member functions (methods in conventional object-oriented terminology) in *State* to make changes to the state when its own *execute()* member function is called.
- *NRE_message*—a nonretractable event message: may make arbitrary state changes, including doing input and output; not scheduled until the precinct detects GVT has passed its timestamp.

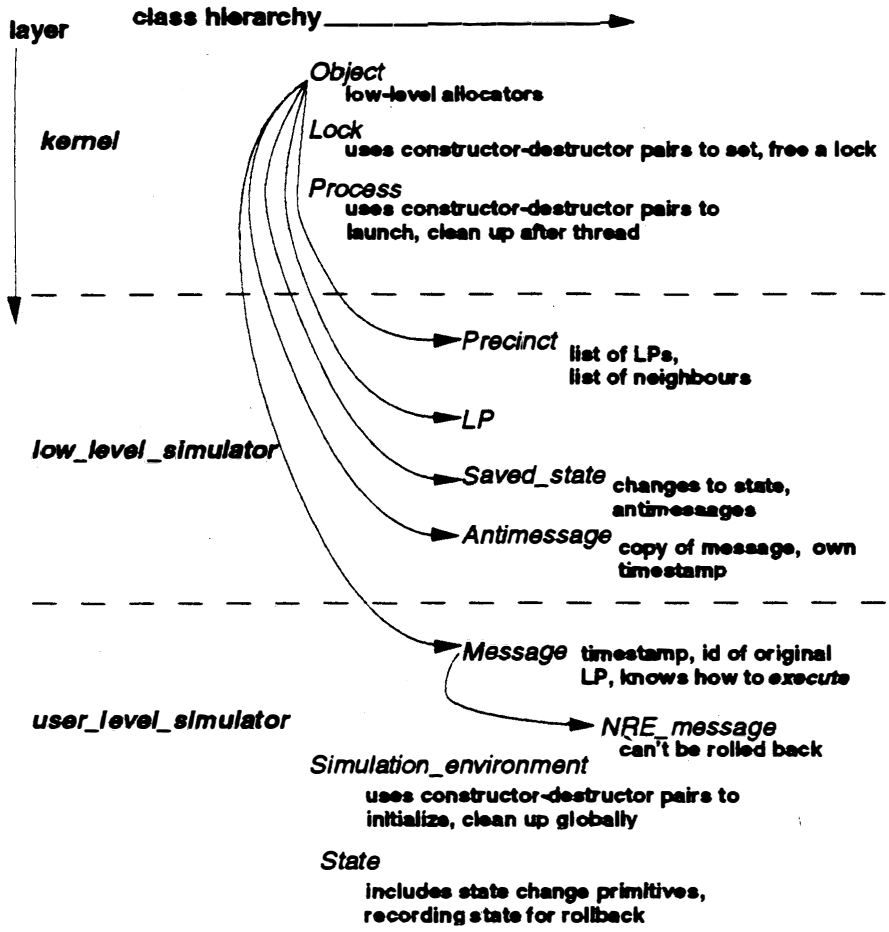


Figure 3. The class hierarchy

5. Related Work

A number of object-oriented threads packages have been implemented for shared-memory parallel machines. An example is Presto which has been implemented and evaluated on a 20-processor Sequent Symmetry [8]. Work in this category has mostly aimed at efficiency of low-level primitives, such as locks and launching threads. In this work, the emphasis is more on support for application structuring to achieve good caching behaviour.

Other work in implementation of optimistic parallel simulators on shared-memory parallel machines has not addressed the locality and caching issues raised here. Examples include BBN Butterfly implementations of optimistic simulations [11]. In terms of its programming model, the Synapse simulation toolkit has similar higher-level goals to the design presented here, except it supports the conservative model [26]. Synapse is built on Presto, and therefore does not address new implementation issues at the lower level.

In other work on structuring simulators, attention has been paid to the difficulties of efficiently vectorizing object-oriented code in C++ [10]. This is in contrast to the results on which this work is based, in which object-oriented programming is shown to be suited to cache-based architectures [4].

6. Conclusions

The design as presented here has been partially implemented (mainly the kernel level).

The goal of showing that C++ is a sufficiently powerful language for such a package has been partially met. All the required features are possible to implement, some conveniently and reasonably simply. For example, the inheritance mechanism makes it possible to hide the antimessage mechanism from the programmer, while allowing antimessages to use new classes derived from the standard *Message* class.

There are however some problems. The most serious is that the requirement that messages should not make state changes except through the *State* class, to make rolling back manageable. There is no way this rule can be enforced through the language; a preprocessor or modifications to the compiler

would appear to be the only sure ways of enforcement. Since there is a convenient mechanism for implementing general side-effects through NREs, it is possible this limitation will not cause major problems in practice.

Another deficiency is that the class hierarchy does not exactly match the logical breakdown of the problem, because some features which are logically low-level (such as antimessages) need information from higher-level constructs (messages, in this case). Although the language makes such an implementation possible, this situation makes it more difficult to describe the logical hierarchy than is desirable.

The next phase of this research is the implementation of the simulation framework described here. Experience with using it to implement simulations will be the basis for evaluating how serious these problems are.

In addition, performance measurements on both standard benchmarks and real problems will be made investigate validity of the overall program structure, especially the aspects which were successful in the MP3D exercise.

C++ has been successful as a basis for one parallel simulation exercise. The design presented here indicates that it is a promising basis for a more general package. In particular, the potential for designing a parallel simulation package that is usable without deep knowledge of the underlying technology, including parallel programming and the mechanisms of optimistic simulation, is attractive. This design is support for the view that such a framework is feasible.

7. Acknowledgements

I would like to thank David Cheriton for supporting the underlying work on which this paper is based. Henk Goosen also played a leading role in the architecture-related aspects. András Salamon made helpful comments on an earlier draft. The work on which this paper is based was supported in part by DARPA Contract N00014-88-K-0619, the Anderson-Capelli Fund and the Mellon Foundation.

References

[1] JR Agre and PA Tinker [1991], Useful Extensions to a Time Warp Simulation System, *Proceedings of the SCS Multiconference on Distributed Simulation*, January, 78–85.

[2] HB Bakoglu, GF Grohoski and RK Montoye [1990], The IBM RISC System/6000 Processor: Hardware Overview, *IBM Journal of Research and Development* **36**(1) January, 12–22.

[3] Boris Berkman and Rassul Ayani [1991], Parallel Simulation of Multistage Interconnection Networks on a SIMD Computer, *Proceedings of the SCS Multiconference on Distributed Simulation*, January, 133–140.

[4] David R Cheriton, Hendrik A Goosen and Philip Machanick [1991], Restructuring a Parallel Simulation to Improve Cache Behavior in a Shared-Memory Multiprocessor: A First Experience, *International Symposium on Shared-Memory Multiprocessing*, Tokyo (to appear).

[5] David R Cheriton, Hendrik A Goosen and PD Boyle [1991], ParaDiGM: A Highly Scalable Shared-Memory Multicomputer, *IEEE Computer* **24**(2) February (to appear).

[6] L. Dagum [1989], *A Fast Sorting Algorithm for a Hypersonic Rarefied Flow Particle Simulation on the Connection Machine*, Technical Report RIACS 89.44, NASA–Ames Research Center.

[7] Margaret A Ellis and Bjarne Stroustrup [1990], *The Annotated C++ Reference Manual*, Addison-Wesley, Reading.

[8] John E Faust and Henry M Levy [1990], The Performance of an Object-Oriented Threads Package, *ECOOP/OOPSLA Proceedings*, October, 278–288.

[9] Robert E Felderman and Leonard Kleinrock [1991], Two Processor Time Warp Analysis: Some Results on a Unifying Approach, *Proceedings of the SCS Multiconference on Distributed Simulation*, January, 3–10.

[10] DW Forslund *et al.* [1990], Experiences in Writing Distributed Particle Simulation Code in C++, *USENIX C++ Conference Proceedings*, 177–190.

[11] Richard M Fujimoto [1990], Parallel Discrete Event Simulation, *Communications of the ACM* **33**(10) October, 30–53.

[12] H A Goosen and D R Cheriton [1990], Predicting the Performance of Multiprocessor Caches, *South African Computer Journal* (2) May, 31–38.

[13] David R Jefferson [1985], Virtual Time, *ACM TOPLAS* **7**(3) July, 404–425.

[14] David Jefferson, Brian Beckman, Fred Wieland, Leo Blume, Mike DiLoreto, Phil Hontalas, Pierre Laroche, Kathy Sturdevant, Jack Tupman, Van Warren, John Wedel, Herb Younger and Steve Bellenot [1987], Distributed Simulation and the Time Warp Operating System, *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, November, 77–93.

[15] NP Jouppi [1990], Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers, *Proceedings of the 17th International Symposium on Computer Architecture*, June, 364–373.

[16] Greg Lomow, Samir Ranjan Das and Richard M. Fujimoto [1991], User Cancellation of Events in Time Warp, *Proceedings of the SCS*

Multiconference on Distributed Simulation, January, 55–62.
[17] E Lusk, R. Overbeek, *et al.* [1987], *Portable Programs for Parallel Processors*, Holt, Rinehart and Winston.
[18] JD McDonald and D Baganoff [1988], Vectorization of a Particle Simulation Method for Hypersonic Rarefied Flow, *AIAA Thermophysics, Plasmadynamics and Lasers Conference Proceedings*, June.
[19] Jayadev Misra [1986], Distributed-Discrete Event Simulation, *ACM Computing Surveys* **18**(1) March, 39–65.
[20] Bruno R Preiss, Wayne M Loucks, Ian D MacIntyre and James A Field [1991], Null Message Cancellation in Conservative Distributed Simulation, *Proceedings of the SCS Multiconference on Distributed Simulation*, January, 33–38.
[21] PL Reiher, RM Fujimoto, S Bellenot and DR Jefferson [1989], Cancellation Strategies in Optimistic Execution Systems, *Proceedings of the SCS Multiconference on Distributed Simulation*, January 112–121.
[22] Peter Reiher, Steven Bellenot and David Jefferson [1991], Temporal Decomposition of Simulations under the Time Warp Operating System, *Proceedings of the SCS Multiconference on Distributed Simulation*, January 47–54.
[23] L Sokol and B Stucky [1990], MTW: Experimental Results For a Constrained Optimistic Scheduling Paradigm, *Proceedings of the SCS Multiconference on Distributed Simulation*, January, 169–173.
[24] CP Thacker, LC Stewart and Satterthwaite [1988], Firefly: A Multiprocessor Workstation, *IEEE Transactions on Computers* **37**(8) August, 902–920.
[25] David B Wagner [1991], Algorithmic Optimizations of Conservative Parallel Simulations, *Proceedings of the SCS Multiconference on Distributed Simulation*, January, 25–32.
[26] David B Wagner [1991], The Design of an Object-Oriented Parallel Simulation Environment, *Proceedings of the SCS Multiconference on Object-Oriented Simulation*,January, 201–208.