Computer Science and Information Systems

Rekenaarwetenskap en Inligtingstelsels
Guest Contribution

This guest contribution is a slightly edited report to the Foundation for Research and Development (FRD) drawn up by Ed Coffman. Ed was an FRD-sponsored guest at the 6th South African Computer Research Symposium. The report was not originally intended for general distribution. Rather, it was specifically compiled for the FRD and its staff. I am therefore grateful to both the FRD and the author for agreeing to its publication in SACJ. I believe that it contains several incisive observations that merit further thought and discussion amongst South African computer scientists. (Editor)

Impressions of Computer Science Research in South Africa

E. G. Coffman, Jr.

AT&T Bell Laboratories, USA

In commenting on the cross section of computer science research in South Africa, I will use the classification in the table of contents of the "Summary of Awards: Fiscal Year 1989," a document published recently by the US National Science Foundation. Of the 5 categories, I will treat Numeric and Symbolic Computation as inappropriate for the discussion below. In this category I noted no research in the computer science setting in South Africa. It is also common in the US and elsewhere to place this effort in other departments, e.g., departments of mathematics or applied mathematics.

Of the remaining categories I found South Africa to be strongest in software systems and engineering, to have a substantial investment in computer systems and architecture, and to be weakest in computer and computation theory.

The coverage in software systems and engineering (SSE) was broad, topical, and similar in scope to that in US universities. Technology transfer and the corresponding relations with industry seemed to be in place or developing along promising lines. I comment in passing that this was rather surprising to me. In the US the development of SSE within university departments has lagged behind almost all other disciplines of computer science. A primary problem has been the insatiable appetite of industry for all Ph.D. graduates in the SSE field.

The investment in parallel processing, computer networks, and distributed computing appears sound, although I expected to see a greater emphasis on mathematical foundations (see my remarks below), particularly in the parallel algorithms area. Given current resources, South African institutions are doing remarkably well in computer science research. But computer science is a fundamentally important course of study, beginning at an early age and extending through graduate Ph.D. research; I take this as sufficiently obvious that I need not dwell on justifications. With this in mind, and with the necessary resources in hand, South Africa should, in my opinion, expand and consolidate its computer science research effort, increase its visibility in the international arena, and correct the rather thin distribution of graduate research among universities.

I can see much of this proceeding along present lines, but I would strongly recommend a concerted development in computer and computation theory (CCT), education and research; this is mainstream computer science and forms the basis for virtually all other fields of study within computer science. It is by no means absent in South Africa curricula, but it appears to be underrepresented in advanced studies and Ph.D. level research.

At the graduate level CCT is heavily mathematical. I understand that mathematical foundations are supplied by mathematics departments in certain cases. This is not ideal, but workable and it is justified by limited resources. However, it is important that mathematics departments not regard this as a mere service; faculty will have to make a major commitment to theoretical computer science, publishing in its leading journals (e.g. SIAM Journal of Computing, Journal of the ACM, Journal of Algorithms, Algorithmica, Journal of Computer and Systems Sciences, Theoretical Computer Science, etc.), and providing the supervision of theses sponsored by computer science departments and leading to degrees in computer science. I would also encourage active participation in the international computer science "theory" societies and their meetings; two highly prestigious examples of the latter are the annual Symposium on the Theory of Computing and the Foundations of Computer Science conference.

Returning to the thin distribution of computer science research, I would make the following point. If the current situation is only a stage of development - i.e., if further resources (both human and financial) can be counted on to bring at least a few of the departments to a critical mass - then little needs to be said beyond the earlier remarks. Critical mass is hard to define, but calls for adequate, expert coverage of mainstream computer science research. In view of the breadth of this research, 8-10 Ph.D. full-time-equivalent faculty would seem to be barely adequate; with the usual clumping of faculty in specific research areas, more would be expected. South Africa has a talent base such that there is little doubt that such departments would achieve a much wider international recognition.
On the other hand, if resources remain fixed at current or even slightly retrenched levels, then I would recommend consolidation to achieve the same goals on a smaller scale. Within a university this can often be done by establishing interdisciplinary, degree-granting laboratories or institutes of computer science, which bring together the computer science efforts located in various departments other than computer science, such as electrical engineering, industrial engineering, business/-management science, mathematics, and operations research. The idea is to enjoy the advantages (opportunity, synergy, awareness, etc.) to both students and faculty of reasonably large computer science programs. There are many examples of such intramural laboratories in North America and Europe.

This approach could also be considered among universities within a confined geographical area, admitted with greater difficulty perhaps. The Institute of Discrete Mathematics and Computer Science connecting Princeton University, Rutgers University, AT&T Bell Laboratories, and Bell Communications Research is a possible model. Examples in South Africa might consist of universities and research institutions on the Reef or those in the Western Cape (just to mention those with which I'm a little familiar).

As a final comment, I should note that my impressions have been based on limited information which may not give a representative picture. I am sure that my reactions will be appropriately discounted where I have been off target.

---

Editor's Notes

Prof John Schochot has graciously accepted to be SACJ's subeditor for papers relating to Information Systems. Authors wishing to submit papers in this general area should please contact him directly. I look forward working with John, and to a significant increase in IS contributions in future.

The hand of the new production editor, Riel Smit, will be clearly evident in this issue. Those papers not prepared in camera-ready format by the authors themselves were prepared by him in TEX. He will be announcing revised guidelines for camera-ready format in a future issue. If you use TEX or one of its variations, Riel would be happy to provide you with a styles document to SACJ format.

At last some Department of National Education committee has decided that SACJ should now be on the list of approved journals. This places it amongst the ranks of some 6800 other journals. These include not merely a number of ACM and IEEE Transactions but also such journals as Ostrich, Trivium, Crane Bag, Koers, Mosquito News, Police Chief, Connoisseur, Lion and the Unicorn, About the House and Ohio Agricultural Research and Development Center Department Series ESS. You will recall that in 1990 this same committee decided that, if judged on its own merits, SACJ did not deserve to be on the illustrious list. In the absence of other evidence, we must assume that the sole reason for its revised decision is that SACJ's predecessor, Questiones Informaticae, was there. (I have a secret suspicion that the committee liked that name.)

It is my understanding that for official purposes, all journals on this list are regarded as equally meritorious, and all of them are more meritorious than any conference proceedings. What does all of this mean?

The momentous implication of the committee's deliberations is that the State will not give your institution a single cent for anything that you publish in SACJ. Instead, the State and your institution will scrupulously keep a score of the annual number of publications that count - but actually don't - because someday they might! And to encourage your enthusiastic participation in this Alice in Wonderland exercise, your institution might actually give you some of the standard subsidy funding that the State should have provided according to its own formulae, but didn't.

You will not be allowed to use this money to buy yourself a car - not even a casual meal. You may only use it to finance activities that are provably directed towards producing more papers in approved journals. The great consolation, of course, is that you will not be required to pay income tax on this money. The only tax involved will be the VAT component when you spend it in an approved manner. As a good computer scientist who enjoys recursion, my vote would be that all such revenue collected by the State should be earmarked to be placed in the pay packets of committee members who decided that SACJ should be approved.

If you publish in these approved journals with sufficient regularity and enthusiasm you will almost deserve to be regarded as a researcher. What you additionally need to do, is to ensure that you befriend and impress at least three overseas referees. You then apply to the FRD for official recognition as a researcher, and if they are sufficiently impressed, they will give you more of the non-taxable kind of money that you need to spend on research to publish in approved journals.

Derrick Kourie
Editor
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 are a few difficulties 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, 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 $lpi$ 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 be useful or not [11].

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

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. 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++).

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.

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 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.

---

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.

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

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".
Table 1. Run time and speedup: 3 versions of MP3D on 4 machines

<table>
<thead>
<tr>
<th>machine and number of processors</th>
<th>MP3D-0</th>
<th></th>
<th></th>
<th>MP3D-1</th>
<th></th>
<th></th>
<th>MP3D-2</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>s</td>
<td>sp.r</td>
<td>sp.f</td>
<td>s</td>
<td>sp.r</td>
<td>sp.f</td>
<td>s</td>
<td>sp.r</td>
<td>sp.f</td>
</tr>
<tr>
<td>SGI 4D/240S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>677</td>
<td>1.3</td>
<td>1.7</td>
<td>597</td>
<td>1.5</td>
<td>2.1</td>
<td>678</td>
<td>1.8</td>
<td>3.1</td>
</tr>
<tr>
<td>2</td>
<td>503</td>
<td>1.3</td>
<td>1.7</td>
<td>410</td>
<td>1.5</td>
<td>2.1</td>
<td>369</td>
<td>1.8</td>
<td>3.1</td>
</tr>
<tr>
<td>4</td>
<td>407</td>
<td>1.3</td>
<td>1.7</td>
<td>283</td>
<td>2.1</td>
<td>2.1</td>
<td>216</td>
<td>1.8</td>
<td>3.1</td>
</tr>
<tr>
<td>SGI 4D/380S (faster CPUs)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>544</td>
<td>1.2</td>
<td>1.4</td>
<td>437</td>
<td>1.4</td>
<td>1.4</td>
<td>569</td>
<td>1.2</td>
<td>1.2</td>
</tr>
<tr>
<td>2</td>
<td>455</td>
<td>1.2</td>
<td>1.4</td>
<td>315</td>
<td>1.4</td>
<td>1.4</td>
<td>312</td>
<td>1.2</td>
<td>1.2</td>
</tr>
<tr>
<td>4</td>
<td>409</td>
<td>1.3</td>
<td>1.4</td>
<td>288</td>
<td>1.6</td>
<td>1.6</td>
<td>179</td>
<td>3.2</td>
<td>1.2</td>
</tr>
<tr>
<td>8</td>
<td>431</td>
<td>1.3</td>
<td>1.4</td>
<td>265</td>
<td>1.6</td>
<td>1.6</td>
<td>141</td>
<td>4.0</td>
<td>1.2</td>
</tr>
<tr>
<td>MicroVAX Firefly</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>12839</td>
<td></td>
<td></td>
<td>12700</td>
<td></td>
<td></td>
<td>9567</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>8096</td>
<td>1.6</td>
<td></td>
<td>8389</td>
<td>1.5</td>
<td></td>
<td>5116</td>
<td>1.9</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5818</td>
<td>2.2</td>
<td></td>
<td>5810</td>
<td>2.2</td>
<td></td>
<td>2732</td>
<td>3.5</td>
<td></td>
</tr>
<tr>
<td>CVAX Firefly (faster CPUs)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>4854</td>
<td>3.9</td>
<td></td>
<td>4161</td>
<td>3.1</td>
<td></td>
<td>3303</td>
<td>2.9</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4478</td>
<td>1.1</td>
<td></td>
<td>3964</td>
<td>1.0</td>
<td></td>
<td>1790</td>
<td>1.8</td>
<td>2.9</td>
</tr>
<tr>
<td>4</td>
<td>4002</td>
<td>1.2</td>
<td></td>
<td>3632</td>
<td>1.1</td>
<td></td>
<td>1014</td>
<td>3.3</td>
<td>2.7</td>
</tr>
</tbody>
</table>

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 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.

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).

---

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

**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 antimeessages queue. When a straggler arrives, the dispatcher passes control to the rollback mechanism, which decides which antimeessages 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 antimeessages. 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.

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.
(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.

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.

**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.

- **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`,

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`. `Message` and `NRE_message`. Some of these classes are needed to implement the low_level_simulator classes (e.g., `Antimessage` is derived from `Message`).

Figure 2. Allocation of LPs to precincts and load balance
some C++ techniques

Inheritance plays an important role in the design. One example is the implementation of antimessages. 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 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:

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

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.

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:

```cpp
{ 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

---

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.
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 may allow 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 to 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.

References


Acknowledgements: the author 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.
Notes for Contributors

The prime purpose of the journal is to publish original research papers in the fields of Computer Science and Information Systems, as well as shorter technical research papers. However, non-refereed review and exploratory articles of interest to the journal's readers will be considered for publication under sections marked as a Communications or Viewpoints. While English is the preferred language of the journal papers in Afrikaans will also be accepted.

Typed manuscripts for review should be submitted in triplicate to the editor.

Form of Manuscript

Manuscripts for review should be prepared according to the following guidelines.

• Use double-space typing on one side only of A4 paper, and provide wide margins.

• The first page should include:
  - title (as brief as possible);
  - author's initials and surname;
  - author's affiliation and address;
  - an abstract of less than 200 words;
  - an appropriate keyword list;
  - a list of relevant Computing Review Categories.

• Tables and figures should be on separate sheets of A4 paper, and should be numbered and titled. Figures should be submitted as original line drawings, and not photocopies.

• Mathematical and other symbols may be either handwritten or typed. Greek letters and unusual symbols should be identified in the margin, if they are not clear in the text.

• References should be listed at the end of the text in alphabetic order of the (first) author's surname, and should be cited in the text in square brackets. References should thus take the following form:

Manuscripts accepted for publication should comply with the above guidelines, and may be provided in one of the following formats:

• in camera-ready format.
  A page specification is available on request from the editor, for authors wishing to provide camera-ready copies. A styles file is available from the editor for Wordperfect, T\text{EX} or L\text{ATEX} documents.

Charges

Charges per final page will be levied on papers accepted for publication. They will be scaled to reflect scanning, typesetting, reproduction and other costs. Currently, the minimum rate is R20.00 per final page for camera-ready contributions and the maximum is R100.00 per page for contributions in typed format.

These charges may be waived upon request of the author and at the discretion of the editor.

Proofs

Proofs of accepted papers will be sent to the author to ensure that typesetting is correct, and not for addition of new material or major amendments to the text. Corrected proofs should be returned to the production editor within three days.

Note that, in the case of camera-ready submissions, it is the author's responsibility to ensure that such submissions are error-free. However, the editor may recommend minor typesetting changes to be made before publication.

Letters and Communications

Letters to the editor are welcomed. They should be signed, and should be limited to about 500 words.

Announcements and communications of interest to the readership will be considered for publication in a separate section of the journal. Communications may also reflect minor research contributions. However, such communications will not be refereed and will not be deemed as fully-fledged publications for state subsidy purposes.

Book reviews

Contributions in this regard will be welcomed. Views and opinions expressed in such reviews should, however, be regarded as those of the reviewer alone.

Advertisement

Placement of advertisements at R1000.00 per full page per issue and R500.00 per half page per issue will be considered. These charges exclude specialized production costs which will be borne by the advertiser. Enquiries should be directed to the editor.
Contents

GUEST CONTRIBUTION

Impressions of Computer Science Research in South Africa
E.G. Coffman, Jr. ....................................................... 1

RESEARCH ARTICLES

An Implementation of the Linda Tuple Space under the Helios Operating System
PC Clayton, EP Wentworth, GC Wells & FK de-Heer-Menlah ......................... 3

Modelling the Algebra of Weakest Preconditions
C Brink and I Rewitzky ................................................ 11

The Design and Analysis of Distributed Virtual Memory Consistency Protocols in an Object Oriented OS
KJ McGregor and RH Cambell ...................................... 21

An Object Oriented Framework for Optimistic Parallel Simulation on Shared-Memory Computers
P Machanik ............................................................. 27

Analysing Routing Strategies in Sporadic Networks
SW Melville ............................................................. 37

Using Statecharts to Design and Specify a Direct-Manipulation User Interface
L Van Zijl & D Mitton ............................................... 44

Extending Local Recovery Techniques for Distributed Databases
HL Viktor & MH Rennhakkamp ................................... 59

Efficient Evaluation of Regular Path Programs
PT Wood ............................................................... 67

Integrating Similarity-Based and Explanation-Based Learning
GD Oosthuizen & C Avenant .................................. 72

Evaluating the Motivating Environment For IS Personnel in SA Compared to the USA. (Part I)
JD Cougar & DC Smith ........................................... 79

TECHNICAL NOTE
An Implementation of the Parallel Conditional
U Jayasekera and NCK Philips .................................. 85

COMMUNICATIONS AND REPORTS

Book Review .......................................................... 87
The CSP Notation and its Application to Parallel Processing
PG Clayton .......................................................... 90