Problems of Implementation of Virtual Time for Parallel Simulation

Ing. Petr Hanáèek
Department of Computer Science and Engineering,
Faculty of Electrical Engineering and Computer Science
Technical University of Brno
Božetìchova 2, 612 66 Brno
e-mail: hanacek@dcse.fee.vutbr.cz

ABSTRACT: *The article deals with the problems of implementation of parallel discrete event simulation in which every process represents an object in the simulation. The main problem of parallel discrete event simulation is the time synchronization of the processes, running on different processors. One approach to the solution of this problem, called Virtual Time Concept, or Time Warp, is presented. In this article we will describe an approach to implementing Time Warp and some techniques that allows to implement Time Warp more efficiently.*

Keywords: parallel simulation, distributed computing, event-driven simulation, demand-driven simulation, Linda language.

Introduction

One of the major goals of the parallel discrete event simulation is to simulate the problems on the network of the processors faster than on a single processor. A number of problems is known that may impact the performance of the parallel discrete event simulation system. They include: simulation time synchronization, development of models which identify parallelism, partitioning of the problems for execution on processors, scheduling strategy to select one of many ready to run processes, and how the partitioned problem is to be allocated to processors. In this article we will deal with the first problem - simulation time synchronization.

Parallel discrete event simulation

The major issues in event-driven simulation are the scheduling of events and the evaluation of these events. For parallel execution of the simulation program are often used almost the same scheduling and evaluation principles as for sequential simulation. The simulation time synchronization in the sequential environment is trivial problem because of presence of common memory visible for all processes. The synchronization is performed after each advance of simulation time. Since the target environment for parallel execution is often loose-coupled multiprocessor architecture (e.g. network of Transputers or local area network of computers, see Fig. 1) the loose degree of coupling may result in relatively long communication delays. Big effort

should be made to minimize the overhead of synchronization in order to achieve acceptable performance.

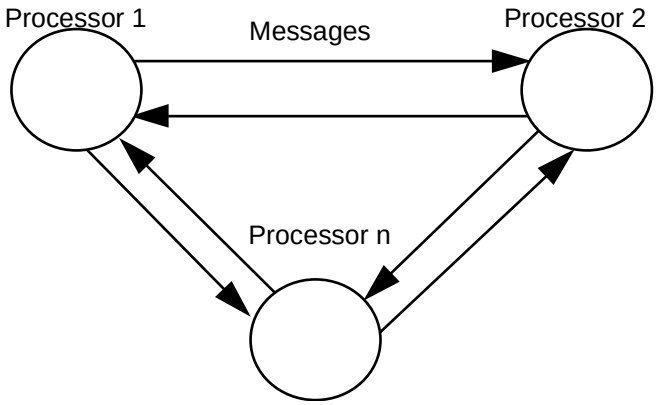

Fig. 1 Architecture for parallel simulation

To limit the overhead arising from frequent synchronization, it is desirable to increase the interval between synchronization points. One possibility that can be exploited is to enforce restrictions on which elements can be placed on a given processor, i.e. force placement of elements on the processors in such a manner that there is no need for synchronization at every time step. In that case the most of the synchronization is performed locally inside the processor an the number of synchronization points for interprocessor synchronization is reduced.

At these synchronization points the synchronization messages are exchanged among the processors (see Fig. 2). The synchronization is performed by one of the two approaches known as conservative and optimistic respectively (see [Fuj90]). Strict or *conservative* interprocessor synchronization - in which each processor waits for messages to arrive from all other processors upon which the given processor is dependent *before* beginning the next phase of computation - may lead to idle periods between successive simulation cycles which are longer than necessary. To reduce idleness, it is possible to relax the strict synchronization requirement by permitting processors to *optimistically* proceed with evaluation using currently available information, then correct any erroneous computation as messages begin arriving.

Performance studies show that both of the approaches are susceptible to some limitations. These studies indicate that conservative method fails when the application exhibits poor *lookahead* - in that case it may perform worse than sequential simulation. Accordingly, the optimistic approach becomes exposed to state saving and processing overhead especially when the application has an excessive rollbacks to the simulation system.

When the problem size and the number of processors become large, the risk for explosive cascading of rollbacks increases. This situations occurs mainly by processes that rapidly advance far in future simulation time. Cascading rollbacks dramatically decrease performance and prohibit the simulation to scale.

In following sections we will discus one approach to optimistic interprocessor synchronization for parallel simulation, called *virtual time* approach.

Virtual time

Virtual time [Jef85] and its implementation Time Warp is a method for organizing distributed systems by imposing on them a temporal coordinate system more computationally meaningful than real time, and defining all user-visible notions of synchronization and timing in terms of it. The Time Warp implements virtual time.

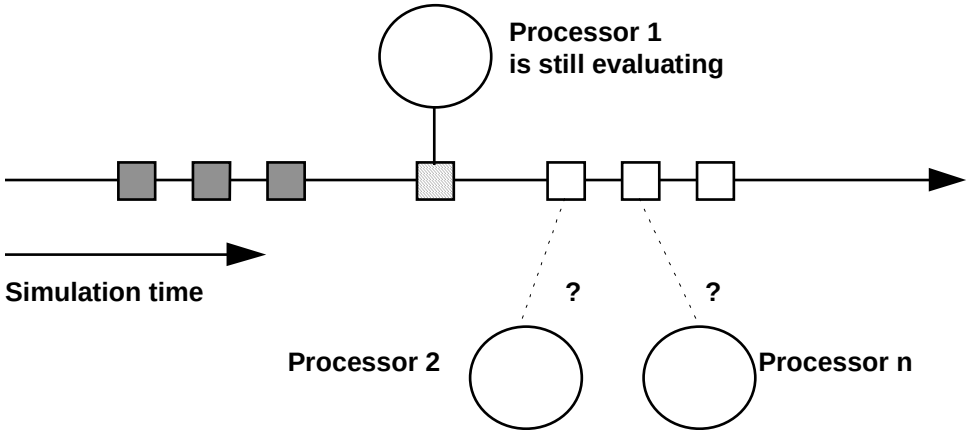

Fig. 2 Basic problem of parallel simulation

Most distributed systems (including all those based on locks, semaphores, monitors etc.) use some kind of *block-resume* mechanism to keep process synchronized. In contrast, the distinguishing feature of Time Warp is that it relies on general *lookahead-rollback* as its fundamental synchronization mechanism. Each process executes without regard to whether there are synchronization conflicts with other processes. Whenever a conflict is discovered after the fact, the offending process is rolled back to the time just before the conflict, no matter how far back that is, and then executed forward again along a revised path. Both the detection of synchronization conflicts and the rollback mechanism for resolving them are transparent to the user.

Local control mechanism

Although in the whole system is a single global virtual time, there is no global virtual time variable in the implementation. Instead, each process has its own **local virtual clock** variable that contains **local virtual time (LVT)**. The local virtual time of a process does not change during an event at that process; it changes only between events, and then only to the value in the timestamp of the next message in the input queue. At any moment some local virtual clocks will be ahead of others, but this fact is invisible to the processes themselves because they can read only their own virtual clock. Whenever a message is sent, its virtual send time is copied from the sender's virtual clock.

Each process has a single **input queue** in which all arriving messages are stored in order of increasing virtual receive time. Ideally, the execution of a process is simply a cycle in which it receives messages and executes events in increasing virtual time order. This ideal execution proceeds as long as no message arrives with a virtual receive time less then local virtual time. Whatever the reasons for the late arrival of a message with a low timestamp, the semantics of virtual time demands that incoming

messages be received by each process strictly in timestamp order. The only way to accomplish this is for the receiver to roll back to an earlier virtual time, canceling all intermediate side effects, and then to execute forward again, this time receiving the late message in its proper sequence.

Because it is impossible to wait for the „next“ message, each process executes continuously, processing in increasing virtual time receive order those messages that already arrived. All of its execution is provisional, however, because it constantly assumes that no message will arrive with a virtual receive time less than the one stamped on the message it is now processing.

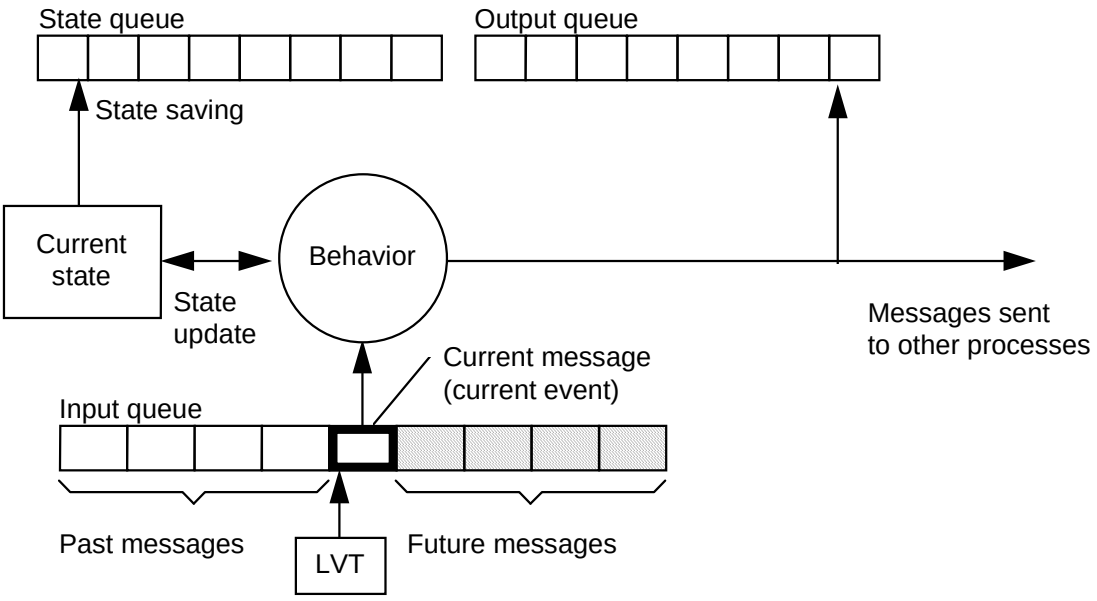

Fig. 3 Basic problem of parallel simulation

To understand the rollback mechanism, we must describe more of the structure of processes and messages. The runtime representation of a process is composed of (see also Fig. 3):

- A **local virtual clock** LVT.
- A **state**, which in general is the entire data space of the process, including its execution stack, its own variables, and its program counter.
- A **state queue**, containing saved copies of recent states of the process.
- An **input queue**, containing all recently arrived messages sorted in order of virtual receive time.
- An **output queue**, containing copies of the messages the process has recently sent, kept in virtual send time order. They are needed in case of rollback, in order to „unsend“ them.

The semantics of rollback is following:

When a message arrives at the input queue of a process with a timestamp lower than the virtual clock time, the recent work of the process is incorrect and must be undone by rollback. The first step is recover the state of the process to the state, saved in state queue. The second step is undone the effect of incorrect messages sent. This is done by sending an antimessage for every incorrect message sent. For every message there exists an **antimessage** that is exactly like it in format and concept except one field, its *sign*. Whenever a message and its antimessage occur in the same queue, they immediately *annihilate*. If message and antimessage annihilate and message was not performed, nothing is done. But if message and antimessage annihilate and message was performed, there is need for secondary rollback on another process.

This antimessage protocol is extremely robust, and works correctly under all possible circumstances. There is no possibility of deadlock (simply because there is no blocking). There is also no possibility of the „domino effect“ (i.e., a cascading of rollbacks far into the past); the worst case is that *all* processes in the system roll back to the same virtual time as the original one did, and then proceed forward again.

Memory management schemes for Time Warp

The huge memory usage is one of the problems of the optimistic approach. Some schemes were developed that reduce the memory usage. We will present three examples of reducing of the memory usage [Sam89]. Time Warp consumes memory by storing three types of objects: state vectors in the state queue, messages in input queue and messages in output queue. We can classify memory management schemes in Time Warp into two types:

- Schemes that reduce average memory usage but cannot necessarily reclaim memory „on demand“, when the system runs out.

- Schemes that can reclaim memory „on demand“.

Following are the examples of schemes that reduce average memory usage:

1. *Incremental State Save*. When state size is large and only a small portion of the state is modified by an event, only the change is recorded rather than making a copy of the entire state. This reduces both space usage an copying time. However when a rollback occurs, some time must be spent to recover an old state from a series of recorded changes.

2. *Infrequent State Save*. State saving frequency can be reduced to suit the memory available in the system. This, however, has a certain performance penalty as some correct computations must be executed that would not be required if state were saved more frequently. Also, there is a tradeoff because infrequent state saving precludes fossil collection of some past events.

3. *Limited Optimism*. Different variations of the optimistic approach have been developed that limit the degree to which processes can advance ahead of others. Some of these bound all the processes within a time window, and some try to

control the spread of erroneous computation as quickly as possible. These schemes were suggested primarily to reduce rollbacks, but they implicitly reduce memory usage by limiting the number of future objects.

Schemes that can reclaim memory „on demand“ are quite complicated and their description is out of scope of this article. More information about these schemes can be found e.g. in literature [Sam89].

Conclusion

The aim of the article was to present some problems with implementation of the Time Warp for parallel or distributed simulation. The main problem presented is a local control mechanism itself. This mechanism is the heart of the efficient Time Warp system. The problems of memory management was presented as well.

References

[Fuj90] Fujimoto, R.: Parallel Discrete-Event Simulation, Comm. of ACM vol. 33, No. 10, 30-53

[Jag91] Jagannathan, S.: Optimizing Analysis for First-Class Tuple-Spaces, in Advances in Languages and Compilers for Parallel Processing, Pitman Publishing, London 1991, ISBN 0953-7767

[She91] Shekhar, K., H, Srikant, Y., N.: Linda Sub System on Transputers, in Transputing 91, IOS Press 1991

[Han92] Hanáèek P., Pøikryl P.: The Linda System in a Distributed Environment -- the Experimental Implementation, SOFSEM'92, Ždiar, Magura, 22.11. - 4.12.1992, 4 strany

[Car91] Carriero N., Gelernter D.: Tuple Analysis and Partial Evaluation Strategies in the Linda Precompiler, in Advances in Languages and Compilers for Parallel Processing, Pitman Publishing, London 1991, ISBN 0953-7767

[Han93] Hanáèek P.: Parallel Simulation Using the Linda Language, MOSIS'93, Olomouc 1.6.-4.6. 1993

[Han96] Hanáèek P.: Virtual Time for Parallel Simulation, MOSIS'96, Zábøeh na Moravì, 1996

[Sam89] Das, S. R., Fujimoto, R. M.: A Performance of the Cancelback Protocol for Time Warp, NSF grant CCR-8902362, 1989

[Jef85] Jefferson, D. R.: Virtual Time, ACM Transactions on Programming Languages and Systems, 7(3):404-425, July 1985

[Jef90] Jefferson, D. R.: Virtual Time II: The Cancelback protocol for storage management in distributed simulation, Proc. 9th Annual ACM Symposium on Principles of Distributed Computation, pages 75-90, August 1990