# Virtual time for parallel simulation

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 parallel discrete event simulation in which every process represents an object in the simulation. The main problem of parallel simulation 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. The Linda programming language is considered as a tool for description of parallel running processes. The Linda programming language (or system) works in a distributed environment. The distributed environment is thought of as a set of processors which run in parallel and which do not share a common memory. The processors communicate only via communication links.*

**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 the almost same scheduling and evaluation principles as for sequential simulation. The simulation time synchronization in the sequential environment is trivial problem because of presence 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) 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.

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. 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 an timing in terms of it. The Time Warp implements virtual time.

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.

## Reducing the memory usage

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]:

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 or reduce rollbacks, but they implicitly reduce memory usage by limiting the number of future objects.

## How to describe a processes

As a possibility how to describe a process in simulated system, we propose to use a Linda language. Linda is a language that was developed at Yale University and it is copyrighted by Scientific Computing Associates, Inc. Linda, however, lacks common features of usual programming languages. Linda does not define such things as variables, statements, and a syntax of programming structures. All of these common features are provided by some language that is called a base language. The Linda system is obtained by adding the small number of Linda operators to any of the sequential languages such as Pascal or C. Linda operators are used for the creation of processes running in parallel, for communication purposes, and for a synchronization of processes. The number of Linda operators is small, and they are quite simple. So, it is easy to understand them and use them. Linda is based on an associative memory model. An elementary memory unit is called a *tuple*. The tuple is an ordered collection of fields (called elements), and it is similar to a record in the relational database theory. Each element of the tuple has a type associated with it. The type is one of the

valid types allowed in the base language. Linda defines three types of elements. *Constants* are thought of in the same meaning as constants in the base language. *Actuals* are names of variables that are used as input parameters of Linda operators. *Formals* are names of variables preceded by a question mark. Formals are used as output parameters of Linda operators.

The *tuple space* is a collection of tuples. The tuple space can contain theoretically unlimited number of copies of the same tuple. The tuple space is a global shared object, and each process has access to it.

Following rules are given for matching two tuples:

- To be a candidate for matching, the number of fields in the tuple and the their types must be the same.
- Actuals match actuals, constants match constants, and actuals match constants if they are of the same type and if they have the same value.
- Actuals match formals and constants match formals if they are of the same type (a formal has no value).

## Linda operators

Linda defines only six operators: out(tuple), rd(tuple), in(tuple), inp(tuple), eval(list_of_arguments).

- **out(tuple)** - this operator is used for putting the tuple into the tuple space. The process that performed the operation is not blocked. For example, the operation out('count', 2) will put the tuple ('count', 2) into the tuple space.
- **rd(tuple)** - the operator rd is used for reading values of elements from the tuple (placed in the tuple-space) that matches the argument. The process that executes this operation is blocked until the matched tuple is found. For example, the operation rd('count', ?x) will match the tuple generated in the previous example from the tuple space. After finishing the operation the variable x will contain the value 2.
- **in(tuple)** - this operator is similar to the rd() operator with one exception; the matched tuple is removed from the tuple space.
- **rdp(tuple), inp(tuple)** - these operators are the predicate versions of the rd() and in() operators. They are non-blocking operators. They return True if the matched tuple is found in the tuple space, otherwise they return False.
- **eval(list_of_arguments)** - this operator allows to create processes executed in parallel. An argument can be an element or a closure. *Closure* is a pair consisting of the values of all free variables defined within a function along with the text (code) of the function body. Linda evaluates all closures in parallel and the result is placed into the tuple space. For example eval('result', load(a), load(b)) will evaluate two functions load in parallel and the tuple ('result', value1, value2) will be placed

into the tuple space. The value1 is the result of the function call load(a), and the value2 is the result of the function call load(b).

## Typical simulation process in Linda language

Using of the Linda language for the simulation purposes will be shown in two examples. The model that we use consists of five blocks. Each block has several inputs and one output. The connection of blocks is shown on the following picture:

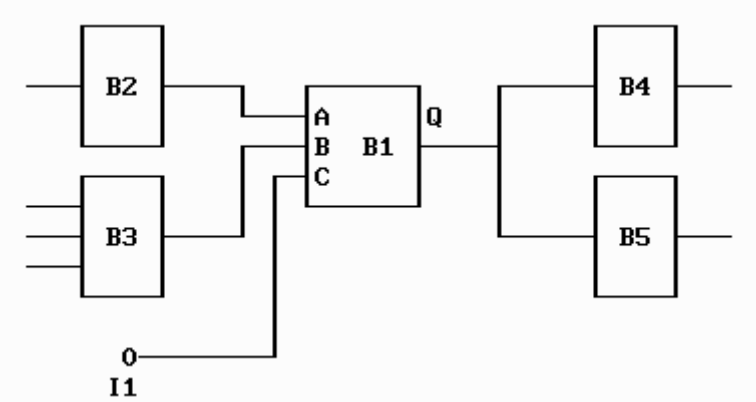


**Fig. 1 An example of simulated objects**

We will show program in Linda language which describes the B1 block. This block has three inputs A, B, C and one output Q. Value of the Q output is dependent on the input values and the model-time. In the case of continuous simulation we will use two types of tuples. The first one is keeping the current model-time, the second one is keeping the value of output of each element.

```
Tuple ("Time", Step, Time);
Tuple ("Value", "ElementName", Value);
```

```
void B1 () {
   for (;;) {
     rd ("Time", Step++, ?Time);   // Reading model time
     in ("Value", "B2", ?A);       // Reading input A
     in ("Value", "B3", ?B);       // Reading input B
     in ("Value", "I1", ?C);       // Reading input C
     Q =function (A, B, C, Time);  // Calculating output
     out ("Value", "B1", X);       // Sending output value
     out ("Value", "B1", X);       // twice - fan-out is 2
     }
   }
```

**Fig. 2 An example of object process**

The program which describes the B1 element is an endless loop. The body of the loop in each cycle reads the model time, reads the input values of the element, calculates the output value and sends it to the tuple space.

## Conclusion

The previous examples are very simple, but they show how to solve some simulation problems using the Linda language and Time Warp approach. Using this language is not only another approach to the already solved problems. It allows to speed up simulation many times because allows user-transparent division of the computational load between many processors in the distributed computing environment.

## 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

[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