Automatic BSP Time Warp

Mauricio Marín
Computing Department, University of Magellan
Casilla 113-D, Punta Arenas, CHILE
E-mail: mmarin@ona.fi.umag.cl

ABSTRACT

Optimistic parallel discrete event simulation has been demonstrated to be an efficient alternative for the simulation of various classes of large scale systems. In particular, it is well-known that organizing the event processing task as a sequence of iterations delimited by the barrier synchronization of the processors is a convenient strategy to follow since it is possible to impose certain global order to an inherently prone-to-unstability form of simulation. Thus an inmediate concern is to ensure a proper control of the amount of work allowed to be done by the processors at any given period of the simulation in a way which is not costly in terms of computation and communication. This paper describes an automatic method for performing such task.

1. Introduction

Discrete-event simulation is a widely used technique for the study of dynamic systems which are too complex to be modelled realistically with analytical and numerical methods. Amenable systems are those that can be represented as a collection of state variables and whose values may change instantaneously upon the occurrence of events in the simulation time. It is not difficult to find real-life systems with associated simulation programs which are computationally intensive enough to consider parallel computing, namely *parallel discrete-event simulation* (PDES) [1], as the only feasible form of computation. Parallelism is introduced by partitioning the system into a set of concurrent objects called logical processes (LPs). Events take place within LPs, their effect is the change of LP states, and LPs may schedule the occurrence of events in other LPs by sending event messages to them.

Once the LPs are placed onto the processors, one is faced with the non-trivial problem of synchronising the occurrence of events that take place in parallel. The Time Warp (TW) synchronisation protocol [2] solves this problem by optimistically processing the occurrence of events as soon as they are available at the LPs and locally re-executing this process whenever the simulation of earlier events is missed in one or more LPs. This is in constrast to the more restrictive approach followed by conservative protocols [6] in which events are only simulated when there is certainty that no earlier events can take place in the LPs. This implies to explicitly define (and maintain) the communication topology among LPs and/or to define lower bounds on the times of any possible next event to arrive from downstream LPs.

In the TW protocol, the simulation time may advance at differing time intervals in each LP. Whenever a LP receives a "straggler" message carrying an event e with timestamp in the "past", the LP is "rolled back" into the state just before the occurrence of e and the simulation of the LP is resumed from this point onwards. That is, all the events with timestamps later than e's timestamp are re-simulated. The LPs save periodically their states to support roll-backs. In addition, any change propagated to other LPs as a result of processing erroneous events is corrected with similar roll-back method. To this end, special (anti-)messages are sent to the affected LPs in order to notify them of the previous sending of erroneous messages. During simulation, a quantity called global virtual time (GVT) is calculated periodically. Its value is such that no LP can be rolled-back earlier than it, and thereby the processed events and anti-messages with timestamps less than GVT are discarded to release memory. GVT steadily advances forward regardless the amount of roll-backs.

This paper focuses on the efficient realisation of TW simulations which proceed in iterations delimited by the barrier synchronization of processors. A number of researchers have already shown that introducing points of global synchronisation has possitive effects in efficiency and stability. However, these simulations can still present unstable behaviour when the amount and length of roll-backs tend to increase unboundly. This paper describes a method for preventing such a behaviour so that efficient performance is kept steady at low cost in running time. Empirical studies presented in [3, 5] show that the basic TW simulation algorithm [4] we use is more efficient than other optimistic approaches and has competitive performance in the simulation of systems specially suited for conservative simulation.

Without loss of generality, we assume that the bulk-synchronous parallel (BSP) model of computing [8] is employed to perform the TW simulations. In this model both computation and communication take place in bulk before the next point of global synchronisation of processors. A BSP program is composed of a sequence of *supersteps*. During each superstep, the processors may only perform computations on data held in their local memories and/or send messages to other processors. These messages are available for processing at their destinations by the next superstep, and each superstep is ended with the barrier synchronisation of the processors. The model fits well with the kind of simulations we aim for, and it can be easily implemented in todays parallel architectures as facilities for performing message sending and barrier synchronisation are commonly available. Moreover, instead of general purpose communication libraries such as PVM or MPI, a special purpose BSP library can be employed to write efficient C++ parallel programs.

2. The synchronisation method

The simulation is carried out by executing a continuos sequence of event processing supersteps [4]. In each superstep, each processor simulates the occurrence of events in accordance with messages received at the start of the current and/or previous superstep(s). This simulation is effected sequentially by using a processor event-list. In addition, for each processor, there is an (adaptive) upper limit to the number of events allowed to take place in each superstep. Moreover, newly-generated and rolled-back events are treated identically: they are all kept in their respective processor event-lists so that they are selected for processing in chronological timestamp order. This means that (**i**) upon rolling back a LP, all the events that become "unprocessed" are re-inserted into the processor event-list, and (**ii**) these events might not take place in the same superstep of the roll-back.

In a superstep, straggler events arriving from other processors may cause the insertion of rolled-back events into the event-list of a given processor. However, the upper limit to events per superstep remains the same regardless of the number of rolled-back events in the processor. The net effect of these stragglers is thus the actual reduction of the speed of simulation time advance of the processor since it now has to cause the chronological occurrence of a comparatively larger number of rolled-back events per superstep. This in turn leads to its relative synchronisation in the simulation time: any processor executing events too far in time is expected to receive a significant amount of stragglers in the following supersteps which will bring this processor back to synchronisation with the other processors. This is so because our scheme tends to emulate a global event-list which gives priority to the processing of the least timestamped events in the entire system. Rollback thrashing is avoided in this way.

In general, the larger the upper limits to events, the larger may be the number of stragglers, roll-back overheads, and number of times that events are re-simulated before being committed. However, the processors cannot advance arbitrarily far into the simulated future. On the other hand, the smaller the upper limits, the smaller may be the overheads due to roll-backs and event re-simulations but also the average GVT advance per superstep is slower and thereby the total number of barrier synchronisation of processors that must be executed is larger. Clearly there is some optimal value for these limits, and these values may change dynamically during the simulation.

The key to achieving good performance under this scheme is to (**i**) set suitable upper limits to events so that near optimal GVT progression per superstep is achieved at low overheads, and (**ii**) make these limits adaptive to follow any significant change in the evolution of the simulation model. In the following section we propose a method for automatically determining the values of these upper limits for any simulation model.

3. The control method

Our approach is based on the idea of using information from a sequential simulation in order to determine the proper tuning of the parallel simulation. During the actual Time Warp simulation, we execute a code associated with the sequential simulation of an ideal BSP asynchronous protocol which synchronises the parallel simulation of the same simulation model (e.g., a BSP Time Warp simulation where no roll-backs ever take place). We call this protocol *oracle*.

To complete the execution of the simulation model, the oracle processes some average number of events per superstep in each processor. We obtain these averages (one per processor) from the oracle simulation, and set them as the upper limits to events per superstep (K's; one per processor) of the actual Time Warp simulation. We calculate new averages at regular intervals of supersteps. The averages from the previous interval become the K values for the current interval. This provides the required adaptiveness to the scheme. Initially we set $K = D_p$.

The oracle simulation is effected by executing in each LP an *asynchronous* superstep counter code which is driven by event messages carrying superstep counts from other LPs. To this end, we maintain one superstep counter SStep[i] for each LP i and each event message e carries an integer $e.s$ indicating the minimum superstep at which this event may take place in the parallel simulation being synchronised by the oracle protocol (not the actual Time Warp simulation). We say that these superstep counters describe the superstep advance of a virtual BSP machine. For each message e that is received at a given LP i we execute the code

$$\textbf{if}\ \ e.s > \text{SStep}[i]\ \ \textbf{then}\ \ \text{SStep}[i] := e.s\,,$$

and each new event e^* scheduled by the LP i in another LP j carries the superstep count $e^*.s = \text{SStep}[i] + 1$ if j is located in other processor or $e^*.s = \text{SStep}[i]$ if j is a co-resident LP. In addition, the counters SStep[...] are considered part of the states of their respective LPs, and thereby their values are corrected by roll-backs.

As the Time Warp mechanism ensures that all the events with timestamps less than GVT have taken place in strict chronological order in each LP, the LP states (counters) just before GVT provide an accurate indication of the

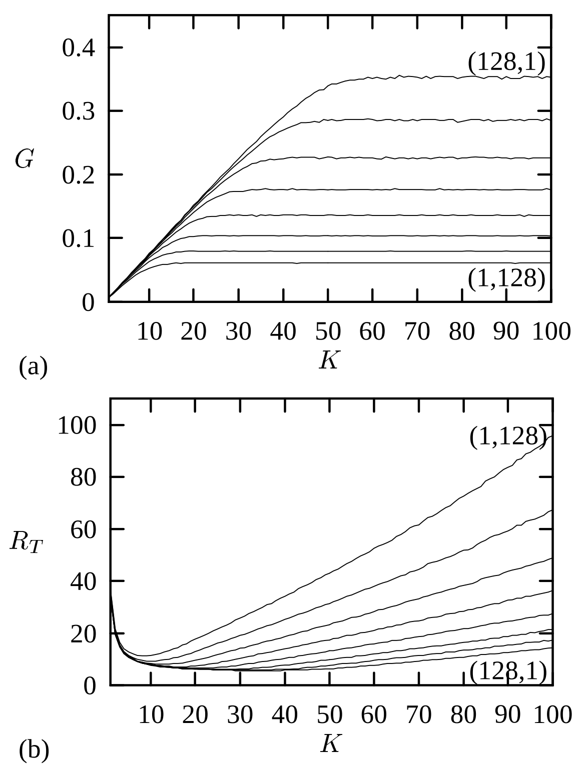

Figure 1. G= GVT advance per superstep, R_T= total running time (sec.), K= upper limit for number of events per superstep. Each curve is data for a different relation between the number of LPs per processor (D_p) and the number of events per LP (D). Each curve is the pair (D_p, D) with values (128,1), (64,2), ..., (1,128). The random event time increments X are generated by using exponential distribution with mean one.

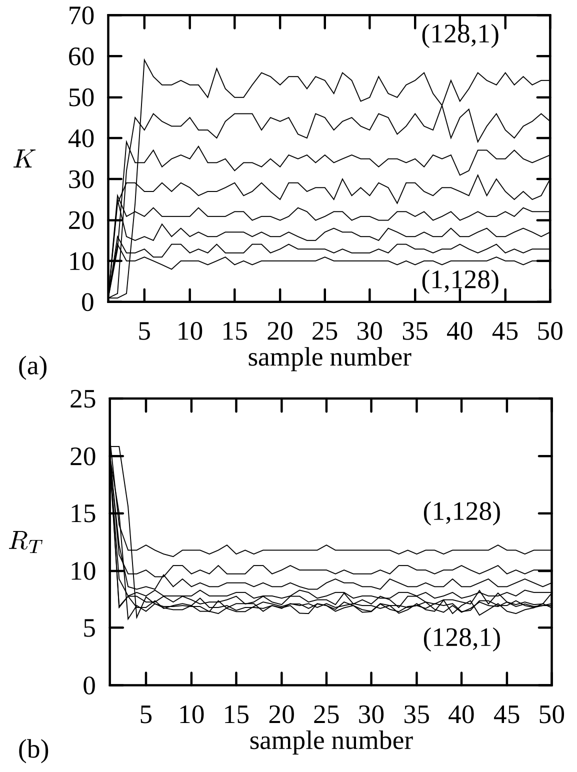

Figure 2. Notation: K= upper limit for number of events per superstep, R_T= total running time (sec.) of separate runs. Each curve is data for a different relation between the number of LPs per processor (D_p) and the number of events per LP (D). Each curve is the pair (D_p, D) with values (128,1), (64,2), ..., (1,128). The random event time increments are generated using exponential distribution with mean one. Initially $K = 1$, and sampling is made every N_S=50 supersteps.

supersteps executed by the virtual BSP machine up to GVT. At this point, the total number of supersteps executed by each processor of the virtual machine is the maximum of the superstep counters maintained in its co-resident LPs.

The averages which define the K values are calculated at the end of intervals of N_S supersteps. The quantity N_S is defined so that at the start of the *last* superstep s of a given interval, a new GVT value is available. The fossil collection procedure is executed at this superstep s. In this way, the particular K to be used by a processor p during the following interval of N_S supersteps is calculated as K= number of events that are committed at superstep s in processor p divided by the increment in oracle supersteps during the current interval in processor p.

These calculations are made locally, in each processor, after completing the fossil collection procedure. The increment in oracle supersteps is calculated as the difference between the maxima of LP's superstep counter values just before GVT at supersteps s and $s - N_S$ (the last superstep of the previous interval).

Figure 1 shows PHold simulations for a case in which the value of K is set up manually. The figure shows the points in which a certain K produces the optimum performance in GVT advance per superstep (part a) and that optimal GVT produces also an optimal running time (part b). In figure 2.a we show the effectiveness of this oracle based scheme. Each simulation starts with $K = 1$ and the figure shows that the system rapidly and automatically finds itself a suitable operational K (these values should be compared with the values of figure 1.a). The sampling interval is $N_S = 50$ supersteps and the figure shows that the method is able to extract near-optimal GVT advance per superstep using a quite small N_S (note that real-life simulations can easily require the execution of 10^4 or more supersteps).

Figure 2.b shows the total running time if the complete simulation were performed using each oracle K shown in the figure 2.a (i.e., separate runs). This figure shows that near-optimal running time is achieved with PHold (comparison with minima of figure 1.b).

Figure 3.a shows the adaptiveness of the scheme. In this case, the timestamp increments of events are scheduled with exponential distribution during 2/3 of the simulation period, and with biased distribution during 1/3 of

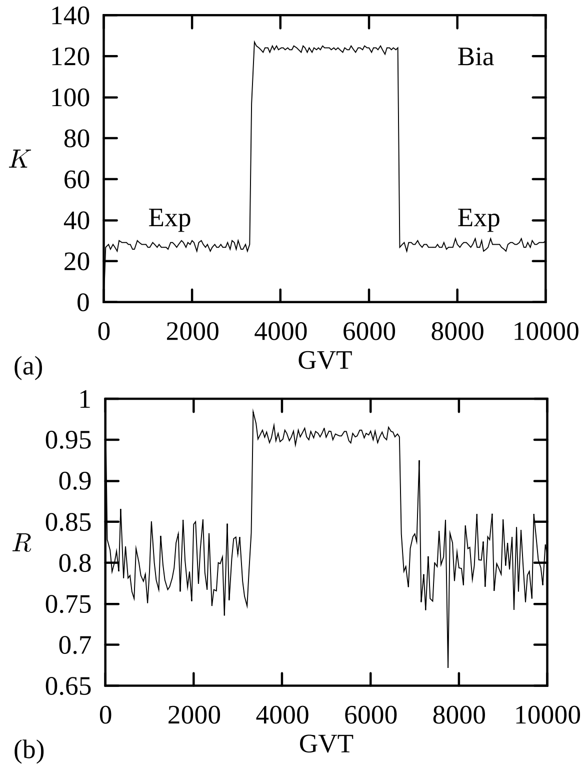

Figure 3. (a): K= upper limit to the number of events per processor per superstep, GVT= global virtual time, and experiment with $(D_p = 16, D = 8)$ where 2/3 of the virtual time is with exponential distribution and 1/3 with biased distribution ($N_S = 50$). Figure (b): R= (total number of committed events)/(total number of simulated events), for similar experiment of (a).

this period (center). Figure 3.b shows the ratio number of committed events to number of simulated events. This is a measure of the degree of optimism of the simulation.

We emphasise that the oracle simulation is carried out *locally*, at processor level. That is, no global computations are required since the oracle algorithm we execute in each LP is driven only by the asynchronous message passing process among LPs.

3.1 Sequential event-list sizes

Driving parallel simulation only by the oracle statistics does not necessarely lead to efficient performance. Consider, for example, a system of n LPs with one LP per processor where LP i send messages to LP $i+1$, LP 1 is a source LP and LP n is a sink LP. In this case, the optimal number of supersteps is n and the oracle indicates that in superstep i the LP i processes all its events while all the other LPs process none! The result is a sequential simulation with the overheads of a parallel one. Systems like this have the feature that no cycles are formed and thereby the oracle superstep counters are not repeatedly incremented during the simulation (they instantaneously achieve their final values i as soon as the processors i receive the first message from processors $i-1$). At processor level, this case is easily detected from the oracle simulation since the increment of supersteps is zero across sampling intervals. Thus if a processor detects no increment in the oracle supersteps during the interval of N_S supersteps defined above, we adopt the strategy of simulating one sequential event-list in each superstep. Actually for these particular systems there is no impediment to process two or more sequential event-lists per superstep (if enough memory is available) since the simulation becomes a simple pipelining process.

However, in general, it may happen that a combination of cases takes place. That is, some processors receive no messages that increment their local oracle superstep counters and some others do. Thus our approach is to let the processors decide *locally* whether to process a sequential event-list per superstep or process the average number of events produced by the oracle based on the observed superstep increments of the oracle simulation.

The size L_E of the piece of sequential event-list located in a given processor can be calculated from fossil events by using the excelent formula proposed in [7]: $L_E = S_E/\Delta_G$ where S_E is the sum of the differences [ReceiveTime $-$ SendTime] found in the fossil events, and Δ_G is the GVT increment during the interval of N_S supersteps (the send and receive times are the simulation times at which an event is generated and takes place respectively). This formula is derived by modelling the event scheduling process as a queue with infinite servers (G/G/∞). The average number of active servers is the above L_E, and for a G/G/∞ queue it is defined as the ratio of the arrival rate of events to the service rate of events. If n events are scheduled during Δ_G, then the arrival rate is n/Δ_G and the average service rate of each server is n/S_E. Figure 4.a shows both K and L_E values for a given processor during a PHold simulation (the event population is doubled in the second half of the simulation). Figure 4.b is related to the next subsection.

3.2 Flow control

In our synchronisation scheme it is possible that some processors set their upper limits to events in accordance with the averages produced by the oracle and some others with sequential event-list sizes. Long term producer-consumer imbalances might arise among these processors leading to memory exhaustion. A simple example is a system where $P-1$ processors send messages each other as in PHold, and the P-th processor send messages to the group but it does not receive messages from them. In this case, the P-th processor sets the size of the piece of sequential event-list located in it as its event limit, and it may certainly cause a memory stall in the group of $P-1$ processors which are operating with comparatively smaller event limits. Though much less likely, a similar situation could occur if all the processors use upper limits given by the oracle simulation.

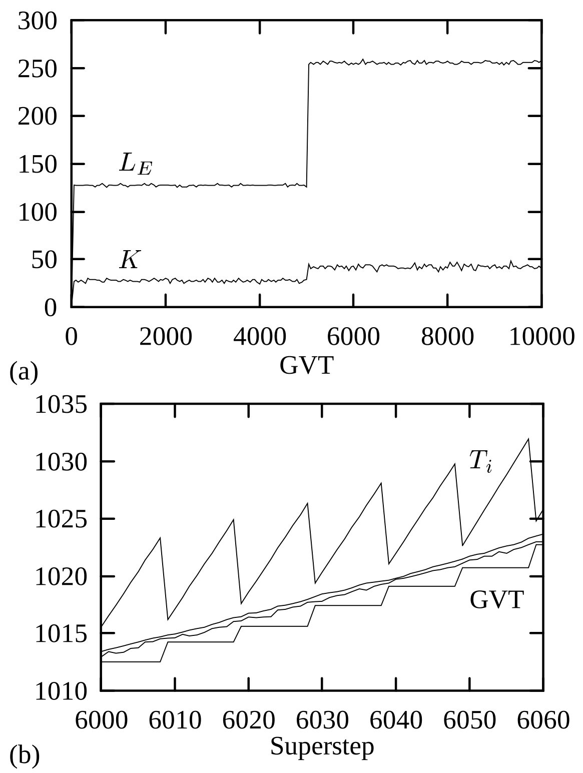

Figure 4. (a) This figure shows two curves, where L_E indicates the size of the piece of sequential event-list (formula $L_E = S_E/\Delta_G$) and K the oracle's upper limits to events in a given processor (the work-load is similar in each processor). Experiments with $(D_p = 16, D = 8)$ where the number of scheduled events is suddenly doubled when GVT becomes greater than 5000 units of simulation time. Figure (b) shows four curves (y-axis is simulation time). From top to bottom the curves show the values of time barriers T_i in a given processor i, timestamp of the last event processed in each superstep, timestamp of the first event processed in the superstep, and GVT advance. Experiments with $(D_p = 16, D = 8)$.

However, it is not difficult to see that for any simulation model where particular patterns in its communication topology are eliminated by distributing the LPs uniformly at random over the processors (which is the sensible approach to load balancing many systems), the probability of memory exhaustion due to producer-consumer imbalance becomes quite low. Nevertheless, some form of flow control is needed in a system intended to be general purpose.

Note that the producer-consumer imbalance arises whenever the event scheduling rate is higher than the rate at which the events are committed. Thus the straightforward approach to avoid memory exhaustion under this imbalance is to set some upper limit to the number of events that a processor may have scheduled at any time. We use *local* time barriers T_i to perform flow control at processor level: at any superstep, only the events with timestamps less than T_i are allowed to take place in the processor i. We emphasise that these local time barriers are not the primary tool to synchronise the LPs in the simulation time (we also emphasise that each processor works with its own locally defined time barrier). In other words, the proposed method uses time barriers only when it is necessary to adaptively rein in the advance of "fast" processors. For example, figure 4.b presents empirical results for PHold where increasing time barrier values of a given processor are shown along with the time of the first and last event simulated in the processor. The figure shows that in PHold there is no producer-consumer imbalance and thereby the time barriers do not restrain the advance in simulation time of the processors.

To define the time barrier T_i, a processor i assumes locally that all other processors are simulating one sequential event-list in each superstep. This means that, at best, exactly one entire global sequential event-list is being committed per superstep. Thus it is not necessary to schedule more events than this maximum rate. Let δ_g be the GVT increment since the last fossil collection, and let n_i be the number of fossil collected events. The quantity $\delta_i = \delta_g/n_i$ is the average time increment between events belonging to the piece of sequential event-list located in processor i. Let L_E be the size of this piece of event-list and define $\Delta_i = \delta_i \cdot L_E$.

Under maximum event processing rate the processor i simulates L_E events and advances Δ_i units of simulation time per superstep (on average). Thus at any superstep s, the processor i should not be allowed to advance beyond, say, $\text{GVT}_s + \Delta_i$ where GVT_s is an estimation of GVT at the specific superstep s. Every time a new value of GVT is available, the local time barrier T_i for this superstep is set to $T_i = \text{GVT} + (n_g + 1)\,\Delta_i$ with n_g being the number of supersteps required to complete the GVT min-reduction. For the supersteps that follow GVT we set $T_i = T_i + \Delta_i$. In this way, the time barriers are incremented following the observed GVT advance which restrains the advance of "fast" processors.

Note that a sudden increase in the time of the pending events in the entire system may make the current Δ_i values comparatively too small. And this values could be so small that the GVT increment between GVT calculations becomes zero, namely a deadlock takes place. This can be avoided by adopting the rule that between GVT calculations at least n_v events must be processed with n_v being the number of supersteps elapsed between two GVT values. As T_i is gradually incremented across supersteps, this requirement is easily met by most simulation models. Alternatively, if the GVT increment becomes zero then the T_i values may be set to infinity and the upper limits to events may be set to one. This causes a "fresh-start" of the simulation which is a sensible approach to follow when recent past information is no longer useful to predict the near future since a dramatic change in the evolution of the simulation model has taken place.

4. Conclusions

We have presented an automatic method for controlling the amount of optimistic execution allowed in bulk-synchronous Time Warp simulations. Figure 5 shows the points in the TW simulation in which the actions of the proposed method are executed. It can be seen that the method has low cost in running time since very few computations are required in each protocol event. Communication is also low as a single integer is carried in each event message. The proposed method is adaptive since it uses information from the recent past to predict the operation of the Time Warp engine during the near future. The asynchronous nature of the computations associated with the method allows them be made locally, at processor level, and thereby it has efficient performance.

In each superstep, the events are simulated chronologically in each processor and the upper limits are not modified as a result of roll-backs. The combination of these two factors tends to emulate a global event-list which gives preference to the processing of the least timestamped events in the whole system. This effectively avoids roll-back thrashing since "fast" LPs are not given enough processing time to advance far into the simulation future.

Acknowledgement This work has been partially funded by research project FONDECYT 1010723.

References

[1] R.M. Fujimoto. "Parallel discrete event simulation". *Comm. ACM*, 33(10):30–53, Oct. 1990.

[2] D.R. Jefferson. "Virtual Time". *ACM Trans. Prog. Lang. and Syst.*, 7(3):404–425, July 1985.

[3] M. Marín. "An Empirical Assessment of Optimistic PDES on BSP". In *10th SCS European Simulation Symposium*, Oct. 1998.

[4] M. Marín. "Time Warp On BSP Computers". In *12th SCS European Simulation Multiconference*, June 1998.

[5] M. Marín. "An evaluation of conservative protocols for bulk-synchronous parallel discrete event simulation". In *14th SCS European Simulation Multiconference*, May 2000.

[6] J. Misra. "Distributed discrete-event simulation". *Computing Surveys*, 18(1):39–65, March 1986.

[7] K.S. Panesar and R.M. Fujimoto. "Adaptive flow control in Time Warp". In *11th Workshop on Parallel and Distributed Simulation (PADS'97)*, pages 108–115, 1997.

[8] L.G. Valiant. "A bridging model for parallel computation". *Comm. ACM*, 33:103–111, Aug. 1990.

```
$K$ = Upper limit to events per superstep (init $K = D_p$).
$O_I$ = Oracle's supersteps increment in the processor.
$L_E$ = Size of the piece of sequential event-list in
        the processor.
$N_C$ = Total number of fossil-collected events
        (init $N_C = 0$).
$T$ = Time barrier for flow control (init $T = \infty$).
$\Delta$ = Time increment per superstep (init $\Delta = \infty$).
$n_g$ = Supersteps required to perform a
        GVT min-reduction.
$n_f$ = Number of supersteps between fossil collections.
$n_e$ = Number of simulated events.

(1) Event $e$ takes place at time $e.t$
    with $n_e \leq K$ and $e.t \leq T$: $n_e = n_e + 1$.

(2) End of superstep: $T = T + \Delta$; $n_e = 0$.

(3) New GVT value: $T = \text{GVT} + (n_g + 1)\,\Delta$.

(4) Fossil collection: $N_C = N_C + \text{NumFossilEvents}()$.

(5) Update $K$ and $\Delta$:
    $O_I = \text{OracleSStepIncrement}()$;
    $L_E = \text{EventListSize}()$;
    if $O_I \neq 0$
        then $K = N_C / O_I$;
        else $K = L_E$;
    $\Delta = (\text{GVTincrement}()/N_C)\,L_E$;
    $N_C = 0$.

(6) GVT increment is zero:
    $K = D_p$; $n_e = N_C = 0$; $T = \Delta = \infty$;
    GlobalSync(GVT).
```

Figure 5. Method's activities in each processor.