Programming Research Group

BULK SYNCHRONOUS PARALLEL ALGORITHMS FOR OPTIMISTIC DISCRETE EVENT SIMULATION

Radu Calinescu

PRG-TR-8-96

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD

Bulk Synchronous Parallel Algorithms for Optimistic Discrete Event Simulation

Radu Calinescu

April 1996

Abstract

The optimistic approach to parallel discrete event simulation (PDES) has led to a number of algorithms capable of fully exploiting the inherent parallelism of discrete event systems. On the other hand, these parallel algorithms, as well as most implementations of the Time Warp mechanism were designed to suit a specific parallel architecture, therefore suffering from lack of portability. This paper proposes the bulk synchronous parallel (BSP) model as a target platform for the design of portable parallel algorithms for optimistic simulation. After an overview of the main directions in PDES, the paper describes the Time Warp mechanism, presenting the most important issues related to optimistic simulation. A class of BSP algorithms for GVT computation is introduced and analysed in terms of the the BSP cost model. Then, two BSP algorithms for optimistic PDES are discussed; the first algorithm aims at avoiding recursive rollbacks in aggressive-cancellation Time Warp implementations, while the second one is a BSP variant of filtered rollback.

Keywords: Bulk Synchronous Parallel Computers, Optimistic Parallel Simulation, Discrete Event Dynamic Systems, General Purpose Parallel Computing.

Contents

1 Introduction **3**

2 A Model for Parallel Discrete Event Simulation **7**

3 Optimistic Parallel Discrete Event Simulation **9**

3.1 The Time Warp mechanism 9
3.2 Memory management in optimistic parallel simulation 11
3.3 Algorithms for GVT computation 13
3.4 Variants of the basic optimistic PDES algorithm 14
3.4.1 Lazy cancellation/lazy reevaluation versus aggressive cancellation 15
3.4.2 Optimistic time windows 15
3.4.3 The filtered rollback approach 16
3.4.4 Other optimistic approaches 17

4 The Bulk Synchronous Parallel Model **17**

5 BSP Approaches to Optimistic PDES **19**

5.1 A class of BSP algorithms for GVT computation 19
5.1.1 Several instances of the class 23
5.1.2 The analysis of the algorithm 23
5.2 An optimistic BSP algorithm for recursive-rollback avoidance 24
5.3 Limiting the degree of optimism in BSP discrete event simulations: a BSP filtered rollback algorithm 28

6 Conclusions and Further Work Directions **29**

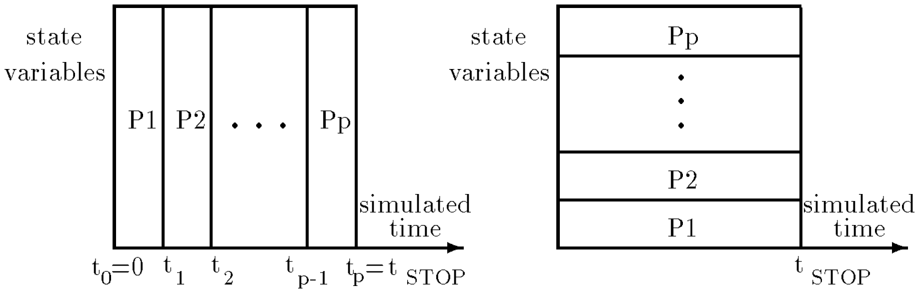
1 Introduction

Many systems of huge practical importance are modelled as discrete event dynamic systems, i.e., as systems that change their status at discrete moments in time. Possible examples range from flexible manufacturing systems and computer systems to logic circuits and communication networks. Unfortunately, all these systems share as a common characteristic a complexity that makes them unsuitable for both analytical and numerical analysis. Therefore, the only technique able to assess the capabilities of such systems is discrete event simulation (DES). The DES paradigm was issued in the early 1970s, when Fishman [1] stated the governing principles of discrete event simulation, and the first simulation languages [2,3] and packages [4] were developed. However, the sequential approach to DES—although based on a very efficient algorithm [5]—has been unable to provide but medium-sized system simulations. In order to overcome this limitation, parallel approaches to DES have been considered since early 1980's. Most parallel discrete event approaches reported so far fall [6–9] into one of the following classes.

Automatic parallelism detection The attempts belonging to this category use a parallelizing compiler to detect parallelism in the sequential simulation code. As the structure of the problem is completely neglected, only a reduced speedup is attainable in this case.

Parallel runs This class comprises approaches that are based on separate simulation runs concurrently performed on the available processors. Since the runs are totally independent, no communication is involved, so maximum efficiency is to be expected. Therefore, this is always the best choice whenever possible; unfortunately, in most situations this is not the case, either because the parameters used in a given execution of the simulator are based on the results of the previous runs, or because the space requirements of an execution of the simulator exceed the memory available to any single processor.

Function distribution The approaches falling into this category distribute the various routines that compose the sequential simulator (e.g., event list handling routines, random number generators, etc.) among different processors. This strategy has the advantage of incurring minimal code changes in the sequential code, and yields a deadlock-free parallel simulator. Nevertheless, it suffers from two severe limitations: it is not able to balance the workload among processors, and the number of processors it may use is limited by the number of routines in the sequential simulator (i.e., the



(a) time-parallel decomposition (b) space-parallel decomposition

Figure 1: Domain decomposition approaches to parallel discrete event simulation.

parallelization is not scalable).

Farm processing The event list that forms the basis of a sequential DES is organised in the applications belonging to this forth class as a work pool managed by a master processor. Whenever one of the other processors (called *slaves* or *workers*) becomes idle, the master schedules the earliest event in the pool for execution on that processor. These approaches provide a good workload balance at the expense of a large communication overhead, being therefore suitable only for parallel systems with high communication bandwidths. Moreover, this technique is limited to those simulations for which independent successive events do exist and can be identified by the master processor.

Domain decomposition This class groups the most attractive approaches to parallel DES; they are all based on the view of a DES as the computation of certain state variables of the system across the simulated time interval. According to this view, the target of the simulation is to fill-in the corresponding space-time graph [8]. Clearly, this may be done in parallel by assigning different parts of the graph to different processors. There are two possible types of partitioning (figure 1). Using a parallel computer with p processors P1, P2, ..., Pp, the first choice is to compute in parallel the values of the state variables over different intervals $[t_{i-1}, t_i]$, $0 < i \leq p$, of the simulated time and is called *time-parallel decomposition*, while the second is to partition the system into p subsystems (*space-parallel decomposition*) and to simulate them in parallel. The approaches based on the time-parallel decomposition of the space-time graph exhibit a theoretically unlimited degree of parallelism, therefore arousing a great interest. However, they seem to be hopelessly restricted to the simulation of systems for which the *state-*

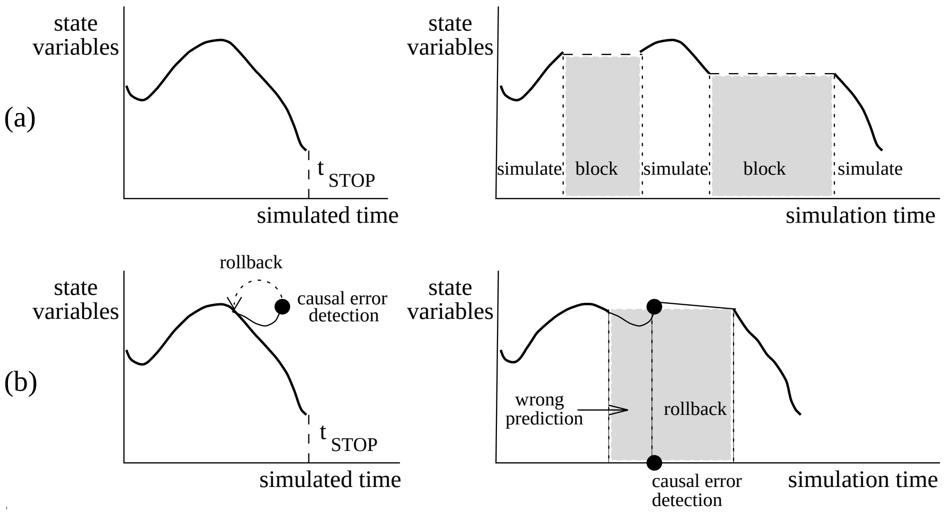


Figure 2: Conservative (a) vs. optimistic (b) space-parallel simulation.

matching problem (i.e., the problem of ensuring that the state of the system at the end of the interval $[t_{i-1}, t_i]$ matches the state at the beginning of the interval $[t_i, t_{i+1}]$) can be efficiently solved. The usual way to tackle this problem is to assume a possible initial state for each interval $[t_{i-1}, t_i]$, and then to perform the simulation; for any interval for which the assumed initial state differs from the final state of the previous interval, the simulation results are correspondingly adjusted by re-simulating the system over that interval (possibly changing the state at the end of the interval). This procedure is reiterated until a complete state matching is achieved. In order to avoid "too many" reiterations (i.e., to solve efficiently the state-matching problem) the system must have the property that the final state of an interval is seldom dependent on its initial state [8], which is clearly not the case for a generic discrete event system.

On the contrary, since it exploits the intrinsic parallelism of the system, the space-parallel decomposition is applicable to any discrete event simulation. This quality, along with its robustness and flexibility, has determined the consideration of the space-parallel simulation as the most promising approach to PDES. A great deal of research work has been pursued in this compelling area, leading to many parallel algorithms for DES; surveys of the field can be found in [7–9]. As different, inter-dependent parts of the system are concurrently simulated, all these parallel algorithms have to deal with the causality errors that may occur if the local simulations are unsynchronised. The easier (but more restrictive) way to solve the problem is to temporarily block part of the simulation whenever there is a risk for a causality error to appear—this is exactly what the so-called *conservative* algorithms do (figure 2) at the expense of incurring deadlocks. The alternative to this policy is employed by the *optimistic* algorithms, which perform a prediction of the future behaviour of the system whenever "safe" simulation is impossible (and a conservative algorithm would block). If the prediction proves to be accurate, it is accepted; otherwise, the prediction is discarded and the simulation is rolled back to the latest simulation time for which the simulation results are known to be correct. As shown in figure 2, the optimistic simulation will outperform the conservative one if and only if the time consumed for wrong predictions and rollbacks is less than the time for which the corresponding conservative simulation is blocked. Although no proof has been provided, the general opinion is that the optimistic approaches do lead to higher performance in most cases [7,8,10]. As this paper only deals with space-parallel decomposition approaches to PDES, any further reference to parallel discrete event simulation must be interpreted as indicating this class of approaches.

Many valuable conservative [11–14] and optimistic [7,10,15,16] algorithms for PDES have been developed so far. However, all the simulators based on these algorithms were designed to suit a specific parallel architecture, either a shared memory machine [17–19], a message-passing parallel computer [20,21], or a workstation network [22]. Although efficiently running on the architecture for which they were designed, these simulators suffer from the same drawback as most parallel applications: lack of portability.

A first attempt to overcome this limitation by using the bulk synchronous parallel (BSP) model [23,24] as a unifying framework for the design, analysis and implementation of conservative PDES algorithms is presented in [25,26]. In this paper we intend to provide an alternative to these approaches by discussing the most important issues related to the optimistic BSP simulation of discrete event systems and presenting two BSP algorithms for optimistic PDES.

The rest of the paper is organised as follows. Section 2 describes the model used by the PDES algorithms. Then, a brief survey of the optimistic PDES paradigm is provided in section 3; the overview includes a presentation of the Time Warp mechanism (i.e., the mechanism that forms the basis of any optimistic simulation), a discussion of the memory management problem, as well as a review of several variants of the basic optimistic algorithm. The BSP programming and cost model is introduced in section 4. Then, in section 5, the BSP model is considered as a target platform for optimistic PDES. After presenting a class of BSP algorithms for global virtual time (GVT) computation, this section introduces two algorithms for optimistic BSP simulation. The first algorithm uses a recursive-rollback avoidance technique for the reduction of rollback overheads, while the second algorithm achieves the same goal by limiting the degree of optimism. Finally, a number of conclusions and further work directions are presented in section 6.

2 A Model for Parallel Discrete Event Simulation

The goal of a discrete event simulation algorithm is [27,28] to reproduce the behaviour of a system that can be modelled as a (non-empty) directed graph whose vertices are *physical processes* or PPs, and whose edges are unidirectional channels between pairs of PPs. Three types of PPs can be distinguished: physical processes without any incoming edge (*sources*), physical processes with no outgoing edge (*sinks*), and PPs that have both an incoming and an outgoing degree greater than 0 (*servers*).

In this representation, physical processes stand for components of the real system, while edges model the inter-dependencies among these components. Each PP has a state and an internal *event queue* (or *message queue*) storing appropriate codifications of the events in which the corresponding component of the system would be involved in a real situation. Each such codification (also called an *event* or a *message*) has assigned a *timestamp* whose value specifies the moment when the event would occur in the real system. The processing of an event from the internal queue of a process PP_i may lead to one or more of the following effects: the state of PP_i is updated; new events are scheduled and inserted in the event queue and/or old events are discarded; new events are sent to some PP_j's for which an edge $< \mathrm{PP}_i, \mathrm{PP}_j >$ exists in the system and are inserted in their queues. Finally, the following restriction must apply: for every cycle of PPs, there exists a constant $\epsilon > 0$ and at least one PP whose inputs at any time $t \geq 0$ do not influence its outputs until time $t + \epsilon$. This condition ensures that the system is [27] *predictable* (i.e., the output of every PP up to any time t can be computed given the initial set of PP states).

In order to concurrently simulate a system with the above specification over a finite time interval, each PP is simulated by a corresponding *logical process* or LP with the structure shown in figure 3a. Beyond the state vector (s) and the internal event queue (eq), each LP comprises a buffer for each incoming or outgoing edge (ib_1, ib_2, ..., ob_1, ob_2, ...), as well as a *local* (*virtual*) *clock* that measures local (virtual) simulation time (lvt). As pointed out in figure 3, the structure of the logical system strongly resembles

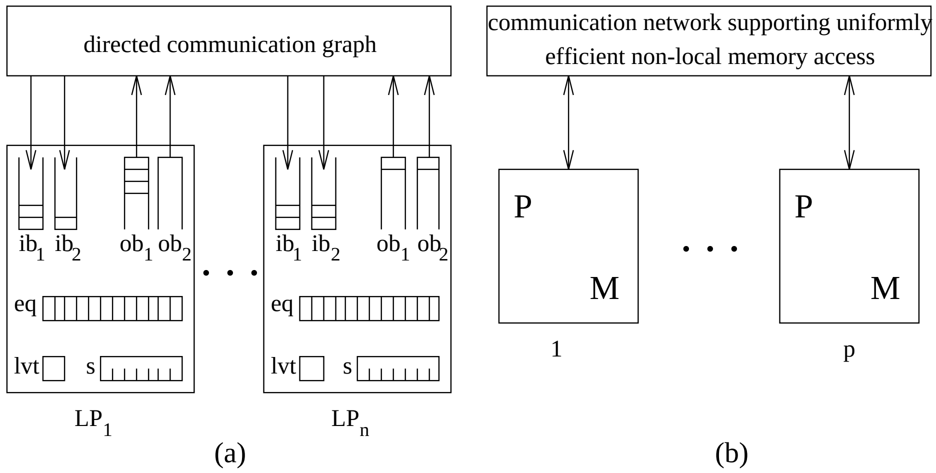


Figure 3: A strong resemblance exists between the structure of the logical system (a) and that of a BSP computer (b).

the structure of a BSP computer [23,24], with the LPs and their internal data structures standing for processor-memory pairs (P/M), and the system's communication graph standing for the communication network of the BSP computer. This resemblance indicates that mapping the logical system onto a BSP computer (i.e., building a BSP simulator) would be a natural approach.

Once the local virtual times are initialised to 0, the simulation proceeds in parallel, each LP iteratively executing [27] the following basic steps: receive messages from other LPs; select and process appropriate events from the local event queue updating lvt to the value of their timestamps; send messages to other LPs. The simulation terminates when all local virtual times reach the end of the simulation time interval (t_{STOP}). The simulation results are guaranteed to be correct as long as [8] each LP processes the events in its queue in nondecreasing timestamp order (the *local causality constraint*). As indicated in section 1, there are two ways to respect this constraint: one may either restrict LPs from processing any event that might break this rule, or may allow LPs to process "unsafe" events as well if an appropriate mechanism exists for recovering from causal errors. Such a mechanism, called the *Time Warp*, is presented in the next section, followed by a number of algorithms for BSP optimistic DES in section 5; those interested in BSP algorithms based on the former approach are referred to [25,26].

3 Optimistic Parallel Discrete Event Simulation

3.1 The Time Warp mechanism

The Time Warp mechanism introduced in [10] represents the synchronisation protocol used by the optimistic PDES algorithms. During an optimistic simulation, each LP tries to obey the local causality constraint by processing the messages from its internal queue in nondecreasing timestamp order and accordingly advancing its *lvt*, until eventually the queue becomes empty and *lvt* is set to $+\inf$. However, this does not prevent an LP from receiving a message whose timestamp is less than the value of its *lvt* (i.e., a message in the past of the local simulation). It is exactly the occurrence of these *straggler* events that the Time Warp protocol must handle by rolling back the simulation to a moment before the timestamp of the straggler. In order to restore the state of the LP before the timestamp of the straggler and to "unsend" all the wrongly sent messages, the Time Warp requires that each LP keeps a record of its past behaviour. This requirement is fulfilled by adding two new elements to the internal structure of each LP (figure 4):

- A state queue (*sq*) containing saved copies of the LP's state, together with the values of the *lvt* when the savings took place; a new copy of the state vector *s* is added to *sq* on a regular basis (e.g., after each event processing, for certain values of the *lvt*, etc.). Of course, the more often the state queue is updated, the smaller the rollbacks. Nevertheless, saving the state after each event processing (which is the best one can do) may also mean important space overheads.

- An output queue *oq* keeping a copy for each message sent by its owner LP. To facilitate the annihilation of erroneously sent messages, each event in the system is assigned a *sign* field, which is fixed when the event is created to one of the values '+' or '−'. Excepting the messages created to be inserted in an output queue—which are "negative" copies of messages sent to other LPs (or *antimessages*)—all other messages have the *sign* field set to '+'.

Finally, since rollbacks involve reprocessing of events, the events in the input queues are not discarded after processing.

The simulation takes place as follows. The n LPs are uniformly distributed among the p processors of the parallel machine, and the simulation progresses smoothly as long as no LP receives an event in the past of its simulation time. However, when such a straggler event does appear, the Time Warp mechanism is triggered for the corresponding LP, leading to the following (local) effects:

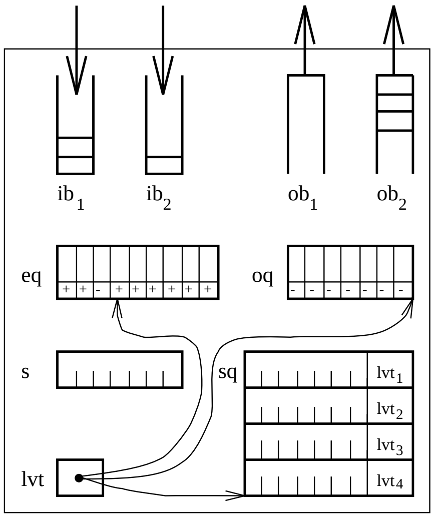


Figure 4: The structure of an optimistic LP

- A state whose associated *lvt* field is less than or equal to the timestamp of the straggler is identified in the state queue and restored into *s* (this state may be the initial state). All the states from *sq* that have later *lvt* values than this one are discarded.
- The local virtual time of the LP is set to the value of the restored state's *lvt*.
- All antimessages from the output queue with a timestamp greater than the new *lvt* value are sent to their destinations and discarded from *oq*.

These three steps ensure that the LP that broke the local causality constraint is brought to a state identical to the one it would have been if the straggler event had appeared in right time. As both messages and antimessages are transmitted among LPs, clear rules must be stated for their manipulation; these rules [9,10] are presented in table 1.

Although the Time Warp mechanism is extremely robust and works correctly under all possible circumstances [10], the local virtual times may decrease during the simulation, so they cannot be used to measure its overall progress. However, there is another parameter that can be used to assess the progress of the simulation at global level; this parameter is called the *global virtual time* (GVT). GVT at (real) time t is defined as the minimum over the set of all local virtual times at time t and over the set of the timestamps of all the unprocessed messages in the system at time t. GVT has a number of very useful provable properties:

τ	type of message: positive message	type of message: antimessage

$\geq lvt$	if the antimessage is in eq then
	discard both messages (annihilation)
	else
	insert message in eq
		if the message is in eq then
		discard both messages (annihilation)
		else
		insert negative message in eq (it is kept in eq but never processed)

$< lvt$	if the antimessage is in eq then
	discard both messages (annihilation)
	else
	insert message in eq
	rollback the simulation to τ or earlier
		if the message is in eq then
		discard both messages (annihilation)
		rollback the simulation before τ
		else
		insert antimessage in eq

Table 1: The rules governing the message manipulation in an optimistic parallel simulation (τ represents the timestamp of the processed message).

- GVT never decreases and represents the lower bound for the time to which any LP could ever roll back;
- With the exception of one state, any information from an LP's state queue, event queue or output queue that is older than GVT will never be used again and may be discarded to free memory;
- If there is enough memory, GVT must eventually increase, and can be used to identify the termination of the simulation.

Nevertheless, its definition make GVT computation very difficult, if possible at all; what one usually uses instead of GVT is an estimate of it. A number of algorithms for the computation of GVT estimates are briefly presented in subsection 3.3.

3.2 Memory management in optimistic parallel simulation

As due to the requirement to keep a history of the simulation memory consumption may become a problem, memory management represents an important issue in optimistic PDES. The "normal" way of lowering the memory overheads is to use GVT to identify and discard obsolete history recordings; all the following items will never be used by any possible rollback and can be disposed of in an operation called *fossil collection*:

- all but one state older than GVT in each state queue;
- all messages older than GVT in the output queues;

- all messages in the event queues whose timestamps are less than GVT.

Clearly, the more frequently and accurately GVT is computed, the most efficient the fossil collection is; however, it is worth noticing that frequent GVT computation could be very expensive in terms of communication overheads.

Another strategy that may help in reducing the memory overheads is the so-called *incremental state saving*. This strategy works well when the difference between the state vectors after and before an event processing is often a sparse vector; in this case, only the non-zero elements of the difference vector have to be saved, thus reducing the amount of memory allocated for old state storing. Notwithstanding the merits of this strategy, one must be aware of the computational overheads incurred by the re-computation of an old state, should such a state be needed for a rollback. A similar reasoning apply for the *interleaved state saving* technique (i.e., saving the state infrequently), excepting that in this case the overheads are due to the rollbacks that may have to be done too far in the past.

Nevertheless, the strategies described so far can limit memory consumption, but none of them is able to prevent memory exhaustion. Several mechanisms dealing with this last problem are described as follows. The first mechanism is to recover space by simply returning unprocessed messages from the event queue to the processes that sent them [10]. Of course, the LP that receives such a message has to roll back to a state previous to that in which it was when it originally sent the message. The messages selected by this *sendback* protocol should be those that are farthest in the future of the local simulation, as they are most likely to be incorrect.

Another technique for recovery from memory exhaustion is described in [29]. In this paper, the author proposes that the item with the largest timestamp from the event queue, output queue and state queue taken together is discarded whenever a memory overflow appears. Then, depending on the type of the item selected for disposal, one of the following operations is performed:

- if the item is a state from the state queue, it is simply discarded;
- if the item is a message from the output queue (an antimessage), it is sent, discarded from the output queue, and the state before the sending of the original message is restored; as this item is the one with the largest timestamp for the considered LP, there are chances that the antimessage will annihilate the corresponding message in the receiver's event queue, further increasing the amount of free memory;
- if the item is a message in the input queue, it is sent back to its original sender like in the Jefferson's approach in [10].

A generalisation of the previous approach is described in [30], where the author describes a protocol called *cancelback*, protocol that discards in a similar way the item with the largest timestamp in the whole system. This protocol is clearly suitable for implementation on parallel computers for which such an item may be efficiently identified (e.g., shared memory computers).

Finally, in [31], an *artificial rollback* is presented as a "natural" solution to memory overflows: the same procedure used to recover from causality errors is also artificially invoked to release memory occupied by saved states and antimessages. However, the choice of an appropriate LP and local virtual time for the artificial rollback could be rather difficult.

3.3 Algorithms for GVT computation

As emphasised throughout this section, the use of GVT in memory management and termination detection makes GVT computation an important issue for optimistic PDES. Clearly, the definition of GVT as a property of an instantaneous global snapshot of the simulation means that the computation of the actual GVT would require a global synchronisation of the simulation, which is unacceptably expensive. Therefore, an estimate of the actual GVT is used in practice. Usually, the value of this estimate is somewhere between the GVT corresponding to the moment when the estimation is triggered and the GVT corresponding to the moment when the computation of the new estimate is accomplished; what is really important is that no estimate must ever be greater than the actual value of GVT.

The basic algorithm for GVT computation is presented in [32]: a GVT *manager processor* is in charge of periodically starting a GVT computation by broadcasting a "GVT-start" message to the LPs in the system. When an LP receives such a message, it computes its local GVT estimate (i.e., the minimum of its lvt, of the timestamps of its unprocessed events, and of the timestamps of the messages it sent but for which it has not received an acknowledgement from their destination LPs), and sends it toward the GVT manager in a min-reduction fashion. Finally, the GVT computation is completed with the broadcast of the new estimate by the GVT manager. The cost of a GVT computation is obviously $O(t_{broadcast})$, where $t_{broadcast}$ is the cost of a broadcast-like operation on the machine performing the simulation.

The main disadvantage of the basic algorithm is the requirement to acknowledge the reception of each single message; this practically means that the message traffic is doubled. In order to reduce this overhead, a modified acknowledgement protocol is proposed in [33]. In this modified protocol, each LP assigns an identifier to every message it sends to another LP. The identifiers are provided by a number of internal counters, one for each output channel. Once a new message (marked with the value of the corresponding internal counter) was sent, the counter is increased by one. In this way, when a new GVT estimate is to be computed and message-reception acknowledgements are needed, each LP is able to detect the identifier i of the earliest message it has not received yet on each of its incoming channels. It therefore can acknowledge the receipt of all the messages with identifiers less than i by returning the value $i-1$ to the LP that sends messages through that channel (clearly, if the messages are always received in the sending order, $i-1$ is simply the identifier of the last received message). Thus, the communication overheads due to message acknowledgements can be substantially reduced.

A completely different GVT computation algorithm based on the observation that many LPs do seldom contribute to the increase of the GVT estimate is described in [34]. In this new algorithm (called *passive response GVT* or *pGVT*), a new estimation of the GVT is *LP-initiated*; the algorithm works as follows. Each LP stores besides the value of the current GVT estimate information on the GVT progress. This information is used to identify the moment when a change in the value of the local GVT estimate is likely to affect the global GVT value; when such a moment appears, the value of the local estimate is sent to the GVT manager, which possibly broadcasts a new global GVT estimate and new GVT history information. As the decision to inform the GVT manager about changes in local GVT estimates is taken in a way that favours delayed LPs, this algorithm may successfully decrease the communication overheads.

To conclude, there are two important problems related to GVT computation; the first is the requirement to acknowledge the receipt of messages, and the second is the overhead incurred by the min-reduction/broadcasting operations. However, whereas the second problem is intrinsically related to the GVT computation itself and is therefore justified, the former problem seems to be an unexpected burden. We shall see in section 5 how one can get rid of this burden by using the BSP programming model.

3.4 Variants of the basic optimistic PDES algorithm

Many variants of the basic algorithm for optimistic parallel simulation have been devised either to improve the time performance of the Time Warp mechanism, or to lighten its memory requirements. This subsection describes several of the most relevant approaches proposed so far.

3.4.1 Lazy cancellation/lazy reevaluation versus aggressive cancellation

The Time Warp mechanism presented in subsection 3.1 is based on an aggressive cancellation policy: whenever a rollback does occur, all antimessages related to that rollback are immediately sent to their destinations. However, it may be possible that some of the messages that will be cancelled by these antimessages are regenerated during the re-simulation of the rollbacked interval. A milder cancellation policy called *lazy cancellation* has been proposed [29] to take advantage of this possibility. In lazy cancellation, an antimessage is not sent at once, but only when after re-performing the simulation until lvt reaches the timestamp of the antimessage, the original message is not regenerated. It has been noticed [35] that using lazy cancellation one can overpass the upper performance bound imposed by the critical (dependency) path of the system. This is due to the fact that it is theoretically possible to obtain appropriate results while using incorrect input data (e.g., there is a 50% chance to correctly evaluate the expression '$a > b$' while using an arbitrary pair (a', b') instead of (a, b)).

Nevertheless, lazy cancellation has its own disadvantages; beyond the memory overhead involved by the storing of possibly obsolete antimessages, it also allows erroneous messages to spread farther than the aggressive cancellation would permit. However, although evidence based on best/worst case analysis [36] proves that in certain situations each of the two cancellation protocols may significantly outperform the other one, the practical experiments have suggested that lazy cancellation tends to lead to slightly better results in most cases.

A similar technique that aims to exploit state insensitiveness to straggler events was introduced in [37]. In this approach, all the states that would have been discarded from the state queue in an aggressive rollback are maintained with the hope that one of them will be matched during the re-simulation of the rollback interval (*lazy reevaluation*). If such a match does occur, then the simulation is instantly advanced to the local virtual time before the rollback, thus eliminating the re-simulation of correct states. The main drawbacks of this approach include memory overheads and complication of the Time Warp code.

3.4.2 Optimistic time windows

One of the most vehement criticisms of the optimistic simulation of discrete event systems is related to the risk of performance degradation with the spread of erroneous computations too far in the simulation's future. In order to prevent this from happening, many approaches have proposed a limitation of the amount of optimism in PDES. Probably the most known approach of this kind is the *Moving Time Window* (MTW) protocol [38].

In the MTW algorithm, only the events whose timestamps are in an interval $[T, T+W]$, where T is the earliest timestamp of an event in the simulation and W is the size of the "window", are considered for execution. Usually, T is taken to be the current GVT estimate, while W is fixed for a given simulation; however, the size of the window may be dynamically adjusted in a variant of the original algorithm. The main limitation of MTW is that correct computations may also be hindered by the restriction to process only events falling within the simulation window. Moreover, it is usually difficult to establish W, unless the distance between successive event occurrences is fairly constant in comparison with the window size.

3.4.3 The filtered rollback approach

An approach similar to MTW is the *filtered rollback* described in [15]. The filtered rollback is an extension of the *conservative bounded lag* algorithm originally introduced in [14]. In this algorithm, a so-called bounded lag restriction is imposed to the parallel simulation: only events whose timestamps differ by at most B (where $B > 0$ is the parameter of the bounded lag restriction) are processed concurrently. After computing the minimum propagation delay matrix (i.e., the matrix whose element $d(i,j)$ represents the minimum time after which an event processed at LP_i may affect LP_j), the simulation proceeds through a number of iterations in a synchronous fashion. Such an iteration comprises the following steps, each followed by a synchronisation barrier:

- the *simulation floor* is computed as the timestamp of the earliest event in the simulation and is broadcasted to all LPs;
- for each LP_i, a lower bound $\alpha(i)$ of the earliest time when the history of LP_i can be affected by other LPs is computed based on local virtual times, on the information in the minimum propagation delay matrix, and on the value of the bounded lag restriction parameter B;
- all the events whose timestamps are within a B distance of the simulation floor and are less than the corresponding $\alpha(i)$ bound are processed.

The filtered rollback algorithm is obtained directly from the bounded lag algorithm by relaxing the computation of the $\alpha(i)$ bounds. Indeed, by replacing $\alpha(i)$ with an estimate of it, a certain degree of optimism is added to the original algorithm, while lightening the overheads involved by an exact computation. Moreover, by correspondingly tuning the computation of the $\alpha(i)$ bounds [15], the rollbacks can be "filtered" in an appropriate way, leading to an efficient optimistic scheme for PDES.

3.4.4 Other optimistic approaches

There are many other algorithms that aim to improve one or another aspect of the Time Warp mechanism. For instance, in [39], a "*wolf calls*" protocol is used to quickly stop the spread of incorrect computations whenever a straggler message arrives at a given process LP_i. This is achieved by broadcasting special control messages to all LPs that might have been reached by the incorrect computation (i.e., to all LPs belonging to the *influence sphere* of LP_i). However, there is a significant risk that such a strategy will also block correct computations and that unnecessary communication overheads will occur due to the overestimation of the influence spheres.

Another approach to ensure a fast cancellation of erroneous computations is proposed in [18]. Here, a *direct cancelling* technique based on pointers set in a shared memory from any event e_i to each event e_j scheduled by e_i is used to rapidly chase the incorrect events when a straggler message is identified. Nevertheless, the applicability of this protocol is restricted to shared memory parallel architectures.

Finally, in [40] the authors suggest a limitation of the optimism to the processing of "unsafe" events at LP level: no possibly incorrect events are sent among LPs. In this way, rollbacks may occur only locally, and no antimessages are ever to be sent.

4 The Bulk Synchronous Parallel Model

The existence of a standard model is the only way to fully impose parallel computing as a viable alternative to sequential computing. The BSP model proposed in [23] and further developed in [24,41] provides such a unifying framework for the design and programming of general purpose parallel computers. A bulk-synchronous parallel computer consists of:

- a set of processor-memory pairs;
- a communication network for point-to-point message delivery;
- a mechanism for efficient barrier synchronisation of all processors or of a subset of processors.

No specialised broadcasting or combining facilities are available. The performance of a BSP computer is fully characterised by the quadruple $< p, s, l, g >$, where

- p is the number of processors;
- s represents the processor speed, i.e., the number of basic operations executed on local data per second;
- l represents the minimal number of time steps between successive synchronisation operations, or the *synchronisation periodicity*;
- g is the ratio between the total number of local operations performed by all processors in one second and the total number of words delivered by the communication network in one second.

The parameter l is a measure of the latency of the network, while the parameter g is related to the time required to complete a so-called *h-relation*, i.e., a routing problem when each processor has at most h packets to send to various processors in the network, and where at most h packets are to be received by each processor; practically, g is the value such that gh is an upper bound for the number of time steps required to perform an h-relation.

A BSP computation consists of a sequence of supersteps; in each superstep, the processors can execute operations on locally held data and/or initiate read/write requests for non-local data. However, the non-local memory accesses initiated during a superstep take effect only when all the processors reach the barrier synchronisation that ends that superstep.

In order to analyse the complexity of a BSP algorithm, one has to take into account the complexity of the supersteps composing the algorithm. The cost of a superstep S depends on the synchronisation cost (l), on the maximum number of local computation steps executed by any processor during S (w), and on the maximum number of messages sent/received by any processor during S (h_s, respectively h_r):

$$cost(S) = l + w + g \max\{h_s, h_r\}. \tag{1}$$

Equivalent results within a multiplicative constant can be obtained adopting other expressions for the cost of a superstep, for instance $\max\{l, w+gh_s, w+gh_r\}$ [42] or $\max\{l, w, gh_s, gh_r\}$ [41].

The expression of the superstep cost shows that the performance of any BSP algorithm depends not only on the problem size and on the number of processors, but also on the BSP parameters l and g. Moreover, as the same implementation of a BSP algorithm can be executed on different target machines, the two parameters can be used [24] to identify the characteristics of the target machine and to dynamically tune the BSP program for best results. Thus, if g approaches 1, the BSP computer closely resembles a shared memory parallel system. If, on the other hand, g has a high value, approximately g operations on locally held data must be performed for every non-local memory access such that the communication overheads do not dominate the computation costs. As concerns the synchronisation periodicity l, it should also not dominate the cost (1), so a certain degree of *parallel slackness* is required to compensate for high values of l (i.e., the program must be written for a number of virtual processors exceeding p).

5 BSP Approaches to Optimistic PDES

A number of bulk synchronous parallel algorithms for conservative simulation, as well as several general considerations related to BSP discrete event simulation (including a solution for the inter-LP communication) have been presented in [25,26]. However, the optimistic simulation algorithms involve some additional problems. For instance, due to the bulk synchronism of a BSP program, a rollback occurrence into a cycle of LPs may lead to antimessages chasing their positive counterparts for the rest of the simulation (figure 5). Whereas the correctness of the Time Warp mechanism still applies, the simulation performance may drastically decrease in such a case. A BSP algorithm that prevents this kind of recursive rollbacks is presented in subsection 5.2. This algorithm assigns an identifier to each event transmitted between two LPs and, based on the values of these message identifiers and on recent rollback histories maintained at certain “key” LPs, detects and breaks recursive rollbacks.

A BSP variant of the filtered rollback algorithm is outlined in subsection 5.3. However, unlike the original algorithm, its BSP version computes the simulation floor (which is identified with the current GVT estimate) at the same time with the actual simulation (i.e., during the same supersteps). The important problem of GVT computation is approached separately in subsection 5.1. A class of BSP algorithms employing a GVT manager processor is devised, and the complexity of different members of the class is analysed in terms of the BSP cost model.

5.1 A class of BSP algorithms for GVT computation

As pointed out in subsection 3.3, one of the most important overheads incurred by GVT computation is due to the requirement to acknowledge message reception; even for the modified acknowledgement protocol, these

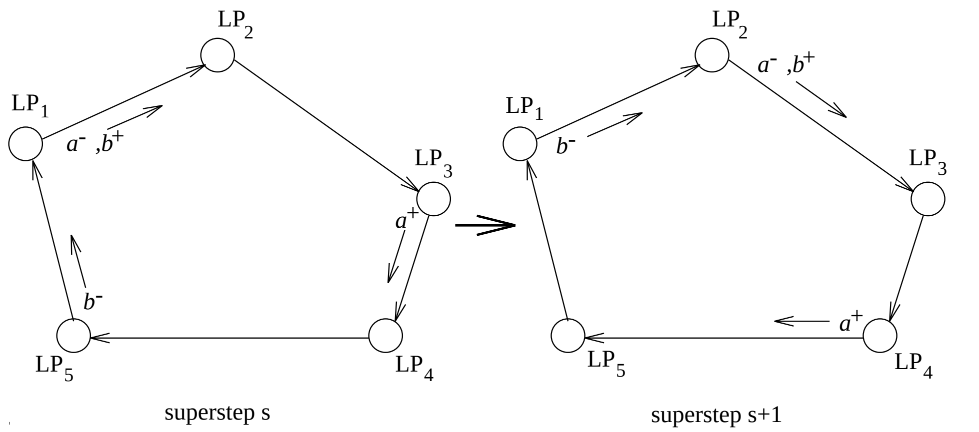


Figure 5: Rollback occurrences into a cycle of LPs may lead to antimessages (e.g., a^-, b^-) chasing their positive counterparts (i.e., a^+, b^+) for the rest of the simulation.

overheads are only reduced, not eliminated. However, since all the messages sent during a superstep are guaranteed to be delivered at the synchronisation barrier that ends that superstep, no acknowledgements are required in a BSP optimistic simulation. If the original GVT computation algorithm is considered as a basis, this elimination of acknowledgement messages means a 50% reduction in the overall message traffic, and a simplification of the GVT computation scheme at the same time.

In this subsection, a class of BSP algorithms that trigger the computation of a new GVT estimate every $r \geq 1$ supersteps is presented. From the point of view of these algorithms, the $p > 1$ processors on which the simulation is performed are viewed as organised in a complete q-tree, $q \geq 2$ (i.e., a tree whose non-leaf vertices have exactly $q \geq 2$ sons, and whose leaves are all on the same level). Clearly, this means that $p = (q^k - 1)/(q - 1)$ for some integer $k \geq 2$.

Each GVT computation (also called a GVT *epoch*) takes $2k - 1$ supersteps; during the first k supersteps, a new GVT estimate is computed by performing a min-reduction onto the complete q-tree, while the last $k - 1$ supersteps are used to broadcast the GVT estimate (and to use it locally in memory management, termination detection, etc.). This is a description of what happens during each of the $2k - 1$ supersteps of a GVT epoch:

- in the first superstep, the leaf processors estimate the GVT over the set of LPs they simulate, and send this value to their parents in the q-tree;

```
if [sstep_no − k + 1 + ⌊log_q((q − 1)myid + 1)⌋] mod r = 0 then
  if ⌊log_q((q − 1)myid + 1)⌋ = k − 1 then               /* leaf processor      */
    compute g = gvt_estimate(myid)
  else                                                  /* non-leaf processor  */
    compute g = min{gvt_estimate(myid)} ∪ {i : 0..q − 1 • son_gvt[i]}
  endif
  if ⌊log_q((q − 1)myid + 1)⌋ ≠ 0 then                   /* non-root processor  */
    bsp_store(father(myid), g, &son_gvt[son_no(myid)], sizeof(g))
  else                                                  /* root processor (P0) */
    for i = 1 to q do
      bsp_store(i, g, &gvt, sizeof(g))
    endfor
    use gvt = g
  endif
endif
if [sstep_no − k + 1 − ⌊log_q((q − 1)myid + 1)⌋] mod r = 0 then
  if ⌊log_q((q − 1)myid + 1)⌋ ≠ k − 1 then               /* non-leaf processor  */
    for i = 1 to q − 1 do
      bsp_store(first_son(myid) + i, g, &gvt, sizeof(g))
    endfor
  endif
  use gvt
endif
```

Figure 6: The GVT computation algorithm; the various functions are defined as follows:

- $gvt\_estimate(myid)$ = the minimum of the local virtual times and of the timestamps of the earliest unprocessed events corresponding to the LPs simulated by processor P_{myid};
- $father(myid) = (q^{l-2} - 1)/(q - 1) + \lfloor (myid - (q^{l-1} - 1)/(q - 1))/q \rfloor$ (the index of the father of node $myid$);
- $son\_no(myid) = (myid - (q^{l-1} - 1)/(q - 1)) \mod q$ (the relative index of node $myid$ among its father's sons);
- $first\_son(myid) = (q^{l} - 1)/(q - 1) + q(myid - (q^{l-1} - 1)/(q - 1))$ (the index of the first son of node $myid$),

where $l = 1 + \lfloor \log_q((q - 1)myid + 1) \rfloor$ represents the level of node $myid$ in the q-tree, and $myid$ has adequate values in each case (e.g., it is improper to compute $father(0)$). The **bsp_store** primitive is that of the Oxford BSP Library [43].

$sstep\_no$	processors involved in the GVT computation	operation(s)	remarks
s	leaf processors, i.e. processors whose $myid$ satisfies $\frac{q^{k-1}-1}{q-1} \leq myid < \frac{q^{k}-1}{q-1}$ or: $\lfloor \log_q((q-1)myid+1) \rfloor = k-1$	$\uparrow$	
$s+1$	$\lfloor \log_q((q-1)myid+1) \rfloor = k-2$	**min**,$\uparrow$	$s = sstep\_no - k + 1 +$ $\lfloor \log_q((q-1)myid+1) \rfloor$
$\vdots$			
$s+k-2$	$\lfloor \log_q((q-1)myid+1) \rfloor = 1$ (i.e., P1, P2, ...,Pq)	**min**,$\uparrow$	
$s+k-1$	$\lfloor \log_q((q-1)myid+1) \rfloor = 0$ (i.e., P0)	**min**,$\downarrow$,**use**	
$s+k$	$\lfloor \log_q((q-1)myid+1) \rfloor = 1$ (i.e., P1, P2, ...,Pq)	$\downarrow$,**use**	
$\vdots$			$s = sstep\_no - k + 1 -$ $\lfloor \log_q((q-1)myid+1) \rfloor$
$s+2k-3$	$\lfloor \log_q((q-1)myid+1) \rfloor = k-2$	$\downarrow$,**use**	
$s+2k-2$	$\lfloor \log_q((q-1)myid+1) \rfloor = k-1$	**use**	

Table 2: The $2k-1$ supersteps of a GVT epoch starting at superstep s, $s \bmod r = 0$; the operations are: $\uparrow$= send local GVT estimate to parent; **min**=compute the minimum of the q estimates received in the previous superstep and of the local GVT estimate; $\downarrow$=send global GVT estimate to the q sons; **use**=use new GVT estimate for memory management.

- in the supersteps 2, 3, ..., $k-1$, all the processors that received q GVT estimates in the previous superstep compute the minimum of these estimates and of their local GVT estimate, sending the result to their parent;
- in the superstep k, the root processor P0 (which has just received the q estimates from its sons) computes the global GVT, sends it to processors P1, P2, ..., Pq, and uses it for local memory management, etc.;
- in the supersteps $k+1$, $k+2$, ..., $2k-2$, all the processors that received GVT in the previous superstep send it to their sons, and use its value locally;
- finally, in the superstep $2k-1$, the leaf processors (which have just received the new GVT value) end the GVT epoch using the new estimate locally.

Clearly, two (or more) successive GVT epochs overlap if $r < 2k-1$. The algorithm is presented in figure 6; an explanation of the strategy used to identify each of the $2k-1$ supersteps is provided in table 2.

5.1.1 Several instances of the class

a. For $q = 2$, $r = 2\log_2 p$, GVT is computed in a binary-tree fashion in $2\log_2 p - 1$ supersteps, and a new GVT epoch is triggered as soon as the previous one is completed.

b. For $q = 2$, $r = 1$, GVT is computed in a pipelined binary-tree fashion, each superstep yielding a new GVT estimate. In any superstep, each processor is involved in both a min-reduction operation for the computation of a new GVT estimate, and in the broadcasting of the current GVT.

c. For $q = p-1$, $r = 2$, the GVT epochs comprise two supersteps: in the first superstep, processors P_1, P_2, ..., P_{p-1} send their local estimates to P_0, while in the second P_0 returns a new GVT value; then, the next GVT computation is triggered in the third superstep.

5.1.2 The analysis of the algorithm

If a min-reduction is underway during a given superstep s, it leads to an extra cost

$$\Delta cost(s) = \left(\frac{2n}{p} + q\right) + gq \qquad (2)$$

where n represents the problem size (i.e., the number of LPs in the simulated system). The first term in equation (2) stands for the computational cost ($2n/p$ elements must be compared to obtain the local GVT estimate, then another q comparisons are required by the min-reduction operation), while the second term is the cost of the q-relation. Since the algorithm is executed at the same time with the actual simulation, no synchronisation overheads occur. In a similar way, the extracost that applies to a superstep s involved in the broadcasting phase is:

$$\Delta cost(s) = gq. \tag{3}$$

Any superstep s of the simulation is in exactly one of the following situations:

- s does not belong to any GVT epoch; $\Delta cost(s) = 0$;
- one or more GVT broadcasts, but no min-reductions are underway during s; $\Delta cost(s) = gq$;
- one or more GVT min-reductions, or both min-reductions and broadcasts are underway during s, but no processor is involved in both operations; $\Delta cost(s) = (2n/p + q) + gq$;
- there is at least one processor involved in a GVT broadcast and in a GVT min-reduction during s; $\Delta cost(s) = (2n/p + q) + 2gq$.

It is worth noticing that the communication overheads do not increase when GVT epochs overlap; indeed, the only consequence is that the corresponding O(q)-relation becomes less incomplete. Therefore, the complexity of the entire algorithm for the computation of a single GVT estimate is:

$$\log_q p\left(\frac{2n}{p} + q\right) + 2gq \log_q p \left[+2l \log_q p\right], \tag{4}$$

where the synchronisation cost was added in square brackets for completeness. In choosing an appropriate value for the parameters q and r, one must take into account both the cost expressed in (4) and the requirements of the specific simulation (e.g., less frequent or longer GVT computations may lead to unacceptable memory overheads, etc.).

5.2 An optimistic BSP algorithm for recursive-rollback avoidance

The algorithm presented in this section is an extension of the Time Warp algorithm with a protocol for recursive-rollback avoidance. The new protocol is based on the fact that in an optimistic BSP simulation, all the events sent in a given superstep are processed by their destinations in the following superstep, so if a recursive rollback is to ever occur for a given LP, it will occur within a number of supersteps bounded by the size of the largest cycle that includes the considered LP from the occurrence of the original rollback. Using this property, the recursive-rollback avoidance protocol attaches an identifier to each message, and, by maintaining temporary rollback information at key points along cycles of LPs, succeeds to identify and discard messages that are chased for annihilation and would lead to recursive rollbacks if not cancelled. The new algorithm is presented in figure 7; it works as follows:

- Each message transmitted between two LPs is assigned a message identifier $msg\_id = lp : no : instance$; the three fields of the identifier are set:
  - if the message is generated as a result of processing another message, to the values of the same fields in the message whose processing generated the new message, incrementing the $instance$ field by one if the new message is positive and an item with the same identifier exists in the rollback history queue (the structure of this queue is described below);
  - otherwise (e.g., in case of a message generated by a source LP), lp to the identifier of the generating LP, no to the value provided by an LP-internal counter (incremented by one after every usage), and $instance$ to 0.
- Each server LP maintains a *rollback history queue* to which new items of the form $< msg\_id, ss\_no >$ are added for each generated antimessage if the LP has more than one outgoing edge (as it may belong to many cycles) or only for the generated antimessages with no correspondent in the event queue (antimessages with new identifiers launched into the system), otherwise. The items in the rollback history queue are kept until they become obsolete, namely until their ss_no field which is decremented by one every superstep reaches 0. According to the remark at the beginning of this subsection, ss_no is initialised always to the size of the largest cycle including the considered LP, or to the number of LPs in the system if the structure of the system is unknown.

A simulation example is shown in figure 8; here a, b, c, d are events whose timestamps are in the relation $timestamp(a) < timestamp(b) < timestamp(c) < timestamp(d)$. The whole system is a queueing network

```
start_superstep
for all LPs simulated by processor myid do
  /* insert new messages in the event queue, checking for recursive rollbacks */
  for all messages m in the input buffers do
    if msg_id(m) is in the rollback history queue then
      discard m
    else
      insert m in the event queue
    endif
  endfor
  /* process messages, using a modified version of the rollback protocol */
  for all messages m in eq do
    if m is a straggler message then
      rollback
      send antimessages, adding a new item to the rollback history queue:
        –for each antimessage, if the current LP has more than one outgoing edge
        –only for those antimessages with no correspondent in the event queue
         (i.e., for which an antimessage with the same identifier does not exist
         in the event queue), otherwise
      increase by one the instance fields of the message identifiers corresponding
      to messages recheduled for execution and with no correspondent in the
      event queue before the rollback
    else
      process m
    endif
  endfor
  /* eliminate obsolete items from the rollback history queue */
  for each item < msg_id, ss_no > in the rollback history queue do
    ss_no = ss_no − 1
    if ss_no = 0 then
      discard < msg_id, ss_no >
    endif
  endfor
endfor
/* call the GVT computation routine */
compute_GVT
end_superstep
```

Figure 7: The recursive-rollback avoidance protocol.

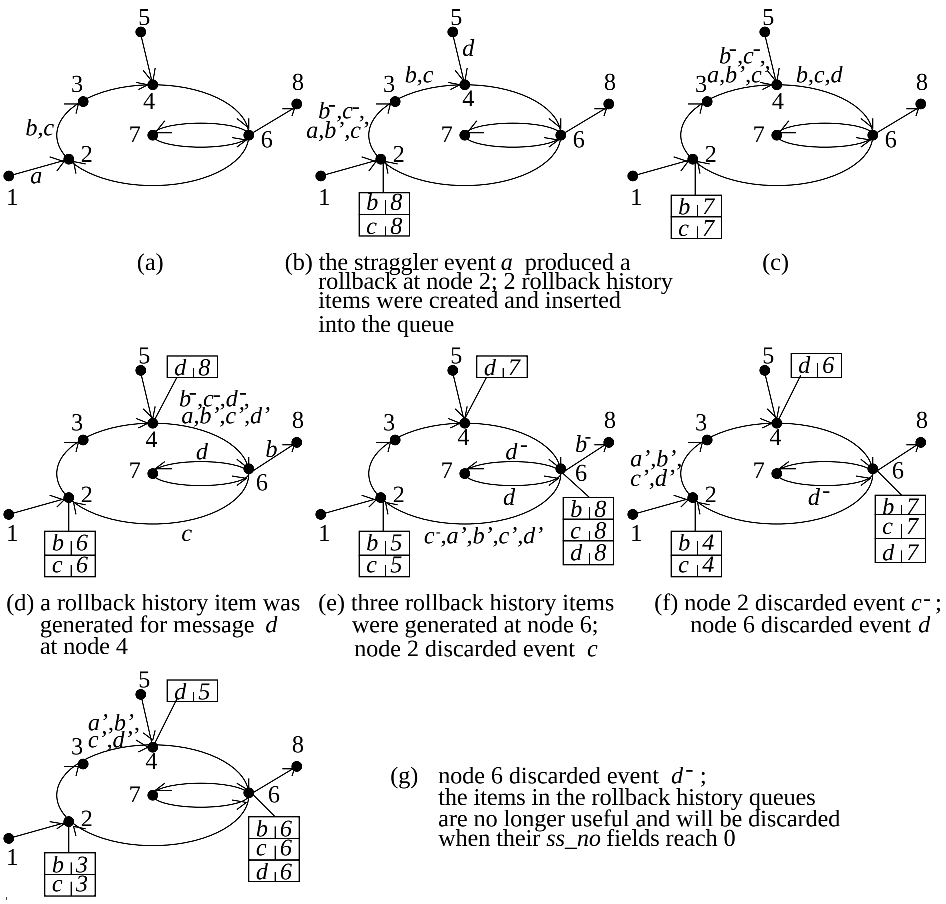


Figure 8: A simulation example for the recursive-rollback avoidance protocol; only non-empty rollback history queues are shown.

that preserves the relations among the timestamps of the four events while passing (and possibly modifying) them from one LP to the next. Although for the sake of simplicity the events are denoted by their "name" (and their antimessages by their name followed by a '$^-$'), each of them represents in fact a complex structure with the form $< msg\_id, sign, timestamp, from\_LP,$ $to\_LP, content >$. The messages followed by a prime sign (e.g., a', b', etc.) differ from the original messages (i.e., a, b, etc.) in that they have the $instance$ field of their identifiers one unit greater.

The main advantage of the new protocol is that it avoids recursive rollbacks while unrestricting the optimism of the Time Warp mechanism. The memory overheads are similar to those corresponding to the lazy cancellation

compute the minimum propagation delay matrix
initialise gvt; lvt_i, $0 \leq i < n$
<u>while</u> $gvt < t_{STOP}$ <u>do</u>
  start_superstep
  call the GVT computation procedure (with $q = p - 1$, $r = 1$)
  estimate $\alpha(i)$, the upper bound of the earliest time when the history of LP_i can be affected by other LPs
  perform the actual simulation, using $B/2$ as the bounded lag restriction parameter for the first superstep
  send information for the estimation of the $\alpha(i)$'s in the next superstep to neighbour LPs
  end_superstep
<u>endwhile</u>

Figure 9: The BSP filtered rollback algorithm. The algorithm uses $B/2$ as the bounded lag restriction parameter for the first superstep in order to allow GVT to progress in each superstep rather than every two supersteps. For details concerning the computation of the $\alpha(i)$ estimates, the reader is referred to [15].

approach, although more limited as only a reduced quantity of obsolete information is maintained by some "key" LPs in their rollback history queues, instead of preserving all output antimessages.

5.3 Limiting the degree of optimism in BSP discrete event simulations: a BSP filtered rollback algorithm

Another strategy for restricting erroneous computations from propagating too far in the future of the simulated time is to restrict the optimism of the Time Warp protocol. An BSP approach based on such a strategy and on the filtered rollback algorithm introduced in [15] and described in section 3 is outlined here. Nevertheless, the BSP algorithm differs from the original filtered rollback in the way the simulation floor (considered to be the current GVT estimate) is computed. Indeed, the BSP variant of the filtered rollback uses the GVT computation algorithm devised in subsection 5.1 (with $q = p - 1$ and $r = 1$) to compute the floor at the same time (i.e., during the same supersteps) with the actual simulation. The algorithm is illustrated in figure 9 (for details concerning the filtered rollback algorithm see subsection 3.4.3).

Like the original filtered rollback algorithm, its BSP variant provides a solution for optimistic (bulk) synchronous simulation of asynchronous discrete event systems; the degree of optimism may be varied by tuning the $\alpha(i)$'s (and the value of B) from unrestricted optimism ($\alpha(i) = +\inf$, $B = +\inf$) to no optimism at all ($\alpha(i)$'s exactly computed).

6 Conclusions and Further Work Directions

A class of BSP algorithms for GVT computation was presented and analysed, and two BSP algorithms for optimistic PDES were discussed in this paper. The use of the BSP model as a target model for these approaches made possible the design of generic, portable parallel algorithms. Moreover, since one of the strongest points of the BSP model is its cost model, the BSP approach to optimistic simulation permits an easier and more realistic analysis of the new algorithms, which represents an important objective of the current research on PDES [8].

Nevertheless, the effectiveness of the two BSP optimistic PDES algorithms must be assessed by testing their implementations based either on the Oxford BSP Library [43], or on a BSP programming language (e.g., GL [44], Opal [45], BSP++ [46]). Further work must also be dedicated to the design of a BSP algorithm for LP-initiated GVT computation, and to the improvement of the BSP filtered rollback protocol by using a less costly variant of the GVT computation algorithm (e.g., one with a smaller q).

References

[1] Fishman G.S., *Principles of Discrete Event Simulation*, John Wiley, New York, 1978.

[2] Markowitz H.M. et al., *SIMSCRIPT, A Simulation Programming Language*, Prentice Hall, 1963.

[3] Pristker A.A.B., *The GASP IV Simulation Language*, John Wiley, New York, 1974.

[4] Birtwistle G.M. et al., *DEMOS: A System for Discrete Event Simulation*, Macmillan Press, New York, 1979.

[5] Zeigler B.P., *Theory of Modelling and Simulation*, John Wiley, New York, 1976.

[6] Righter R., Walrand J.C., Distributed simulation of discrete event systems. In: *Proceedings of the IEEE*, vol. 77, no. 1, Jan. 1989, pp. 99–113.

[7] Fujimoto R.M., Parallel discrete event simulation. In: *Communications of the ACM*, vol. 33, no. 10, October 1990, pp.30–53.

[8] Fujimoto R.M., Parallel simulation of discrete event systems. In: Cohen G., Quadrat J.-P. (Eds.), *Lecture Notes in Control and Information Sciences 199*, Proc. 11th Int. Conf. on Analysis and Optimisation of Systems, Discrete Event Systems, Sophia-Antipolis, June 15-16-17, 1994, Springer Verlag, 1994, pp. 419–428.

[9] Ferscha A., Tripathi S.K., Parallel and distributed simulation of discrete event systems, Technical Report CS-TR-3336, Dept. of Computer Science, Univ. of Maryland, 1994.

[10] Jefferson D.R., Virtual time. In: *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 3, July 1985, pp. 404–425.

[11] Chandy K.M., Misra J., Distributed simulation: A case study in design and verification of distributed programs. In: *IEEE Trans. Software Engineering*, vol. SE-5, no. 5, Sept. 1979, pp. 440–452.

[12] Chandy K.M., Misra J., Asynchronous distributed simulation via a sequence of parallel computations. In: *Communications of the ACM*, vol. 24, no. 11, Nov. 1981, pp. 198–205.

[13] Bagrodia R.L. et al., A message-based approach to discrete-event simulation. In: *IEEE Trans. on Software Engineering*, vol. SE-13, no. 6, June 1987, pp. 654–665.

[14] Lubacevski B.D., Efficient distributed event-driven simulations of multiple-loop networks. In: *Communications of the ACM*, vol. 32, no. 1, Jan. 1989, pp. 111–131.

[15] Lubacevski B.D., Shwartz A., An analysis of rollback-based simulation. In: *ACM Trans. on Modeling and Computer Simulation*, vol. 1, no. 2, Apr. 1991, pp. 154–193.

[16] Reynolds P.F., Jr., A spectrum of options for parallel simulation, Technical Report IPC-TR-88-007, Institute for Parallel Computation, School of Engineering and Applied Science, University of Virginia, Charlottesville, VA 22901, 1988.

[17] Reed D.A. et al., Parallel discrete-event simulation using shared memory. In: *IEEE Transactions on Software Engineering*, vol. 14, no. 4, Apr. 1988, pp. 541–553.

[18] Fujimoto R.M., Time Warp on a shared memory multiprocessor. In: *Trans. of the Soc. for Computer Simulation*, vol. 6, no. 3, July 1989, pp. 211–239.

[19] Konas P., Pen-Chung Y., Synchronous parallel discrete-event simulation on shared memory multiprocessors. In: *Proceedings of the 1992 SCS Western Simulation MultiConference and Distributed Simulation*, 20-22 Jan. 1992, Newport Beach, California, pp. 12–21.

[20] Cai W., Turner S.J., An algorithm for distributed discrete-event simulation: The "carrier null message" approach, in *Distributed Simulation*, Proceedings 1990 SCS Multiconference on Distributed Simulation, January 90, pp. 3–8.

[21] Alonso J.M. et al., Conservative parallel discrete-event simulation in a transputer based multicomputer. In: Grebe R. et al., *Transputer Applications and Systems '93*, IOS Press 1993, pp.636–650.

[22] Groselj B., Tropper C., The distributed simulation of clustered processes. In: *Distributed Computing* (1991), vol. 4, pp. 111–121.

[23] Valiant L.G., A bridging model for parallel computation. In: *Communication of the ACM*, vol. 33, August90, pp. 103–111.

[24] McColl W.F., BSP Programming. In: Blelloch G., Simon I. (eds.), *Proc. 13th IFIP World Computer Congress*, vol. I, Elsevier, 1994, pp. 539–546.

[25] Calinescu R., A BSP approach to discrete-event simulations. In: *Distributed vs. Parallel: Convergence or Divergence?*, Proceedings PPECC Workshop '95, Abingdon, UK, 14-15 March 1995, pp. 31–36.

[26] Calinescu R., Bulk synchronous parallel algorithms for conservative discrete event simulation. To appear in: *Journal of Parallel Algorithms and Applications*, vol. 11, no. 1–2.

[27] Misra J., Distributed discrete-event simulation. In: *Computing Surveys*, vol. 18, no. 1, March 1986, pp. 39–65.

[28] Bagrodia R. et al., A unifying framework for distributed simulation. In: *ACM Transactions on Modeling and Computer Simulation*, vol. 1, no. 4, Oct. 91, pp. 348–385.

[29] Gafni A., Rollback mechanisms for optimistic distributed simulation systems. In: Unger B., Jefferson D.R. (Eds.), *Proceedings of the SCS*

Multiconference on Distributed Simulation, vol. 19, no. 3, 1988, pp. 25–29.

[30] Jefferson D.R., Virtual Time II: The cancelback protocol for storage management in Time Warp. In: *Proceedings 9th Annual ACM Symposium on Principles of Distributed Computing*, ACM Press, 1990, pp. 75–90.

[31] Lin Y.-B., Preiss B.R., Optimal memory management for Time Warp parallel simulation. In: *ACM Transactions on Modeling and Computer Simulation*, vol. 1, no. 4, Oct. 1991, pp. 283–307.

[32] Samadi B., Distributed simulation: Performance and analysis. Ph.D. dissertation, Dept. of Computer Science, UCLA, Los Angeles, 1985.

[33] Lin Y.-B., Lazowska E., Determining the global virtual time in a distributed simulation. In: *Proceedings 1990 Int. Conf. on Parallel Processing*, 1990, pp. III.201–III.209.

[34] D'Souza L.M. et al., pGVT: An algorithm for accurate GVT estimation. In: *Proceedings 8th Workshop on Parallel and Distributed Simulation*, IEEE Computer Society Press, 1994.

[35] Jefferson D.R., Reiher P., Supercritical speedup. In: Rutan A.H. (Ed.), *Proceedings 24th Annual Simulation Symposium*, New Orleans, Louisiana, USA, April 1-5, 1991, IEEE Computer Society Press, 1991, pp. 159–168.

[36] Reiher P.L. et al., Cancellation strategies in optimistic execution systems. In: *Proceedings of the SCS Multiconference on Distributed Simulation*, vol. 22, no. 1, Jan. 1990, pp. 112–121.

[37] West D., Optimizing Time Warp: Lazy rollback and lazy re-evaluation, M.Sc. thesis, University of Calgary, 1988.

[38] Sokol L.M. et al., MTW: A strategy for scheduling discrete simulation events for concurrent execution. In: *Proceedings of the SCS Multiconference on Distributed Simulation*, vol. 19, no. 3, July 1988, pp. 34–42.

[39] Madisetti V., et al., Wolf: A rollback algorithm for optimistic distributed simulation systems. In: *Proceedings of the 1988 Winter Simulation Conference*, Dec. 1988, pp. 229–305.

[40] Dickens P.M., Reynolds P.F., Jr., SRADS with local rollback. In: *Proceedings of the SCS Multiconference on Distributed Simulation*, vol. 22, no. 1, Jan. 1990, pp. 161–164.

[41] McColl W.F., General purpose parallel computing. In: Gibbons A. M., Spirakis P. (eds.), *Lectures on Parallel Computation*. Proc. 1991 ALCOM Spring School on Parallel Computation, volume 4 of Cambridge International Series on Parallel Computation, Cambridge University Press, Cambridge, UK, 1993, pp. 337–391.

[42] Gerbessiotis A.V., Valiant L.G., Direct bulk-synchronous parallel algorithms. In: *Journal of Parallel and Distributed Computing*, vol. 22, no. 2, August 1994, pp. 251–267.

[43] Miller R., Reed J.L., The Oxford BSP Library: Users' Guide Version 1.0, Oxford Parallel Technical Report, Oxford University Computing Laboratory, 1994.

[44] McColl W.F., GL: An architecture independent programming language for scalable parallel computing. Technical Report 93-072-3-9025-1, NEC Research Institute, Princeton, NJ, 1993.

[45] Knee S., Program development on BSP machines using Opal. In: *Distributed vs. Parallel: Convergence or Divergence?*, Proceedings PPECC Workshop '95, Abingdon, UK, 14-15 March 1995.

[46] Lecomber D., Object-oriented programming with BSP++. In: *Distributed vs. Parallel: Convergence or Divergence?*, Proceedings PPECC Workshop '95, Abingdon, UK, 14-15 March 1995.