Parallel and Distributed Discrete Event Simulation

Carl Tropper

NOVA

PARALLEL AND DISTRIBUTED DISCRETE EVENT SIMULATION

CARL TROPPER
EDITOR

Nova Science Publishers, Inc.
New York

Senior Editors: Susan Boriotti and Donna Dennis
Coordinating Editor: Tatiana Shohov
Office Manager: Annette Hellinger
Graphics: Wanda Serrano
Editorial Production: Jennifer Vogt, Matthew Kozlowski, Jonathan Rose, Ron Doda and Maya Columbus
Circulation: Ave Maria Gonzalez, Indah Becker and Vladimir Klestov
Communications and Acquisitions: Serge P. Shohov
Marketing: Cathy DeGregory

Library of Congress Cataloging-in-Publication Data
Available upon request.
ISBN 1-59033-377-2

PARALLEL AND DISTRIBUTED DISCRETE EVENT SIMULATION

APPLICATIONS OF PARALLEL AND DISTRIBUTED DISCRETE EVENT SIMULATION

CARL TROPPER

Discrete-event simulation has long been an integral part of the design process of complex engineering systems and the modelling of natural phenomena. Many of the systems which we seek to understand or control can be modelled as digital systems. In a digital model, we view the system at discrete instants of time, in effect taking snapshots of the system at these intstants. For example, in a computer network simulation an event can be the sending of a message from one node to another node while in a VLSI logic simulation, the arrival of a signal at a gate may be viewed as an event.

Each event in a discrete-event simulation has a timestamp associated with it. When an event is processed, it is possible that new events are generated as a consequence of this processing. These new events have larger timestamps, obtained by adding a simulation time advance to the timestamp of the event which it had prior to processing. The events in the simulation are stored in a heap and are processed in order of the lowest timestamp first.

Digital systems such as computer systems are naturally susceptible to this approach. However, a variety of other systems may also be modelled this way. These inclued transportation systems such as air-traffic control systems, epidemological models such as the spreading of a virus, and military war-gaming models.

As the systems and phenomena we want to model increase in size and complexity, the memory demands of these simulations increase and it becomes increasingly difficult to obtain acceptible execution times. The circuits which we want to simulate now contain hundreds of millions of gates. A detailed simulation of the Internet would make inordinate demands on a workstation. In order to accomodate the grwoing need for the simulation of larger models, it became necessary to make use of distributed and parallel computer systems to execute the simulations. In the early 90's parallel machines were made use of while more recently the use of clusters of workstations (Beowulf, Myrinet) are utilized as they represent a more cost-effective approach then parallel machines. In addition, shared memory multiprocessors are now much more cost-effective.

Like any other distributed program, a distributed simulation is composed of processes which communicate with one another. The communication may occur via shared memory or via message passing. The processes in a distributed simulation are each intended to simulate a portion of the system being modelled and are referred to as Logical Processes (LPs). An LP creates events, sends events to other LPs and receives events from other LPs in the course of a simulation. Associated with each LP are input queues used to store messages from other LPs.

The advance of time in a distributed simulation poses an intriguing problem because of its inherant lack of global memory. The events of a distributed simulation are spread among the processors and consequently time is advanced in each process independantly of the other processes. This is accomplished via the notion of Local Simulation Time (LST) which is maintained by each LP. When an event is processed at an LP, the LST takes on the value of its timestamp. The LP processes events from its input queues by selecting the event bearing the smallest timestamp from all of its queues.

The treatment of time in a distributed system is of fundamental importance to the understanding and building of distributed systems. [16] contains a discussion of the nature of time in a distributed system for the interested reader. It was necessary to develop synchyronization strategies for the distributed and parallel programs which execute distributed simulations. As mentioned before, in a uniprocessor events are stored in a heap and simulated in the order of the smallest timestamp first. Since the events of a distributed simulation are spread among the processors it is possible to execute events out of their correct order. This happens if an event with a smaller timestamp then an event which has already been processed arrives at a processor from another processor. In a military simulation, it matters in which order the events "aim the gun" and "fire the gun" are executed. The central problem of Distributed simulation is the development of synchronization algorithms which are capable of maintaining causality and which do so with minimal overhead. The interested reader should consult the proceedings of the IEEE Workshop on Parallel and Distributed Simulation (PADS) and a recent book which describes the field [10].

Two primary approaches to synchronization algorithms have been developed, the *conservative* and *optimistic* classes of algorithms. A conservative algorithm is characterized by its blocking behavior. In a conservative simulation, if one of the input queues at an LP is empty the LP blocks awaiting a message from another LP. This behavior exacts a price, however, in the form of increased execution time and the possible formation of deadlocks. A deadlock forms if a group of LPs is arranged in the form of a cycle such that each of the LPs is awaiting a message from its predecessor in the cycle. More generally, a deadlock occurs if the LPs are arranged in the form of a knot. Hence it becomes necessary to either find means to avoid deadlocks or to detect and break them. There are a plethora of algorithms for each approach. We describe several algorithms.

In the null message approach [9], each time an LP sends a message to a neighboring LP, it also sends a message (the null message) to its other neighbors containing the timestamp of the message, thereby causing the neighbors to advance their LST's. As a consequence, all of the LPs are aware of the earliest simulation time that a message can arrive at an empty input queue. The LP can use this information to avoid blocking by comparing the smallest timestamp in each of its input queues to this value. If the smallest timestamp in all of the input queues is samller then this value then it can process the associated event. Otherwise it must block. The drawback of this approach is clearly the large number of messages which must be sent; as a consequence much work has been done to avoid the sending of a large number of null messages. [13] and [14] are efforts in this direction. Another form of deadlock avoidance is a time window approach in which tiem windows are successively generated with the property that the events contained in these windows are safe to process [1].

The deadlock detection and breaking approach relies upon algorithms to detect deadlocks. For example, in [19], a knot detection algorithm is used to detect a deadlock, and the event bearing the minimal timestamp in the knot is detected as well. This event is safe to execute.

The other major class of synchroniztion algorithm is known as optimistic. Time Warp [4] is the prime example in this category. In optimistic algorithms LPs maintain one input queue and all of the events which arrive from other LPs are stored in the queue. The events are processed without any concern for the arrival of events with smaller timestamps. If such a "straggler" eventarrives at an LP, the LP restores its state just prior to the arrival of the straggler and continues with the simulation from that point, a process known as rolling back. The LP must maintain checkpoints of previous states in order to roll back. In addition, it is necessary to to cancel messages which were produced subsequent to the straggler as they may well be incorrect. In order to do this, each LP maintains an output queue in which it stores copies of messages which it has already sent. Upon the arrival of a straggler, the LP sends these copies to the same LPs which received the original messages. If the message and its copy meet in an input queue, the two messages cancel (anihilate) one another. For this reason, the copy is known as an anti-message. If the anti-message arrives after the original message has been processed, the destination LP rolls back to the time of the anti-message and sends out its own anti-messages. Clearly, the memory demands imposed by storing copies of LP's states and maintaining anti-messages in the output queue are onerous. Hence techniques to reduce the amoutn of memory used by Time Warp are fundamental to its success. One important technique involve the computation of the smallest simulation time to which any LP in the simulation may roll back, known as the Global Virtual Time (GVT). If we define the LVT as the timestamp of the last message processed at an LP, then the GVT is the minimum of (1) the LVT values of all of the LPs in the simulation and (2) the minimum timestamp of all of the events which have been sent but which have not been processed at a given point in real time. Since no LP can roll back prior to the GVT, all of the memory allocated prior tothe GVT may be released. Many algorithms for computing the GVT have been developed [7]. Another technique to reduce the amount of memory employed is to periodically checkpoint the states of an LP, instead of after every event.

In a shared memory multiprocessor the use of direct cancellation [12] is used to control the use of memory. Here pointers are used to link the descendants of an event to the event, thereby eliminating the need for anti-messages. Recently, a direct cancellation technique for distributed memroy environments has been developed [17]. Memory management techniques are also used to reclaim space from LPs so that a stalled simulation may be allowed to continue [5, 6, 12].

In recent years the emphasis in the field has changed from developing efficient synchronization techniques to the application of these techniques to real world problems. In the area of computer networks, an on-going effort is simulation of the Internet [8]. The intention of the project is to provide a realistic test-bed for the development of new Internet protocols and to locate performance problems. A detailed distributed simulation of the Internet provides an environment in which experimental conditions can be controlled and in which it is possible to replicate experiments. To date, it has been necessary to experiment on the actual network itself.

The simulation of VLSI circuitry provides another fruitful area for the application of distributed simulation. A simulation environment for VHDL circuitry is described in [15]. A similar project for Verilog is underway.

Military simulations are benefitting from advances in distributed simulation. An important application is the uniting of existing simulations of different aspects of combat together, e.g. tank warfare, close air-support and infantry warfare. The inclusion of live exercises into this environment is also being pursued. The field of distributed interactive simulation (DIS) is devoted to these advances. A number of conferences are devoted to this area, among them the IEEE Real Time and Distributed Interactive simulation conference.

Yet another area is the simulation of large manufacturing environments, such as the production of VLSI wafers.

The chapters in this book are representative of the advances in these fields. In "The Development of Conservative Superstep Protocols for Shared Memory Systems", Gan et al provides strong evidence for the utility of (conservative) distributed simulation of a wafer fabrication process. The performance of several different synchronous conservative protocols are compared on a number of models based on data sets supplied by Sematech. Sematech is a consortium of semiconductor manufacturing companies that does research for its members in semiconductor manufacturing. In the "Implementation of a Virtual Time Synchronizer for Distributed Databases on a Cluster of Workstations", Boukerche et al study the performance of a distributed database system implemented on a network of workstations and synchronized by an optimistic protocol. In "Applying Multilevel Paritioning to Parallel Logic Simulation", Subramaninan et al describe a partitioning algorithm for logic simulation and examine its performance making use of the optimistic simulation kernel which lies at the heart of the VHDL simulator referred to above. In "Self-Organizing Criticality in Optimistic Simulation of Correlated Systems". Overeinder et al examine the relationship between the rollback behavior in Time Warp and hte physical complexity of Ising Spin systems. Ising Spin models are used to simulate the magnetization of ferro-metals. Finally, Moradi et al study the DOD's High Level Architecture (HLA) in the context of a simulation of an air-traffic control system.

REFERENCES

[1] TURNER, S. AND XU, M., *Performance Evaluation of the Bounded Time Warp Algorithm*, Proceedings of the SCS Multi-conference on PADS, volume 22, pages 117-126, 1992.

[2] DAS, S. AND FUJIMOTO, R., *An Adaptive Memory Management Protocol for Time Warp Parallel Simulation*, Proceedings of ACM SIGMETRICS Conference on Measure-

ment and Modeling of Computer Systems, volume 22, pages 201-210, 1994.

[3] R. FUJIMOTO, *Time Warp on a Shared Memory Multiprocessor*, Transactions of the Society for Computer Simulation, Vol. 6, No. 3, pp. 211-239, July 1989.

[4] D.A. JEFFERSON, *Virtual Time*, ACM Transactions on Programming Languages and systems, Vol. 7, No. 3, pp. 404-425, July 1985.

[5] D.A. JEFFERSON, *Virtual Time II: The Cancelback Protocol for Storage Management in Time Warp*, Proc. 9th Annual ACM Symposium on Principles of Distributed Computing, pp 75-90, ACM, 1990

[6] Y-B LIN, E. LAZOWSKA, *Reducing the State Saving Overhead for Time Warp Parallel Simulation*, T-R 90-02-03, Dept. of Computer Science and Engineering, Univ. Washington, Seattle, WA, 1990

[7] MATTERN, F., *Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation*, Journal of Parallel and Distributed Computing, 18: 423-434, 1993.

[8] D. NICOL, J. CROWE, A. OGIELSKI, *Modelling of the Global Internet*, Computer Science and Engineering, vol1, no1, Jan-Feb 1999, pp. 42-50.

[9] K.CHANDY, J.MISRA, *Distributed Simulation:A Case Study in the Design and Verification of Distriuted Programs*, IEEE Trans. Software Eng S-5, Sept. 1979, pp. 440-452

[10] R. FUJIMOTO, *Parallel and Distributed Simulation Systems*, Wiley Interscience, 2000

[11] H. AVRIL, C. TROPPER, *Clustered Time Warp and Logic Simulation*, Proceedings of the 9th Workshop on Parallel and Distributed Simulation,1995, pp112-119.

[12] S. DAS, R. FUJIMOTO,K. PANESAR, D. ALLISON, M. HYBINETTE *GTW: A Time Warp System for Shared Memory Multiprocessors* Proceedings of the 1994 Winter Simulation Conference, 1994

[13] J. MISRA, *Distributed discrete event simulation*, ACM Computing Survey 18,1 ,March 1986, pp 39-65

[14] W. CAI, E. LETERTRE, S.J. TURNER,*Dag Consistent Parallel Simulation: a Predictable and Robust Conservative Algorithm*, Proc. 11th Workshop on Parallel and Distributed Simulation (PADS97), pp 178-181, Lockenhaus, Austria, June 1997

[15] P. WILSEY,ET AL, *Analysis and Simulation of Mixed Technology VLSI Systems*, Special Issue of Journal of Parallel and Distributed Computing, to appear, April 2002

[16] L. LAMPORT, *Time, Clocks, and the ordering of events in a distributed system*, Communications of the ACM, vol21(7), pp. 558-565.

[17] J-L. ZHAO, C. TROPPER, *The Dependence List in Time Warp*, Proc. Workshop on Parallel and Distributed Simulation (PADS01), to appear, Los Angeles, California, May 2001

[18] D. E. MARTIN, R. RADHAKRISHNAN, D. M. RAO, M. CHETLUR, K. SUBRAMANI, AND P. A. WILSEY, *Analysis and Simulation of Mixed-Technology VLSI Systems*, Journal of Parallel and Distributed Computing (in press).

[19] A, BOUKERCHE, C, TROPPER, *Parallel Simulation on the Hypercube Multiprocessor*, Distributed Computing, Springer-Verlag, vol.8, no.4, pp 181-191.

CONTENTS

THE DEVELOPMENT OF CONSERVATIVE SUPERSTEP PROTOCOLS FOR SHARED MEMORY MULTIPROCESSOR SYSTEMS

BOON-PING GAN[†], YOKE-HEAN LOW[†], WENTONG CAI[‡], STEPHEN J. TURNER[‡], SANJAY JAIN[†], WEN JING HSU[‡], AND SHELL YING HUANG[‡]

Abstract. This paper summarizes our work involving the successive refinements of a conservative superstep protocol. The refined protocols are known as the SST, SLS, and FET protocols. Each of the refinements improves the safetime bound, which in turn reduces the number of supersteps. In general, this helps to reduce the synchronization overhead and thus the execution time. However, a change in the number of supersteps can also introduce load imbalances within supersteps into the simulation. This observation becomes apparent with the FET protocol in particular. The performance of the refined protocols is compared through several semiconductor supply-chain simulation models. The results from the experiments strongly indicate the feasibility of using parallel discrete-event simulation techniques in the simulation of large scale systems such as a supply-chain.

1. Introduction. Parallel discrete-event simulation (PDES) has been a well-researched area for many years. An important area in PDES has been the design and development of new synchronization protocols. In PDES, a simulation is decomposed into logical processes (LPs) that can be simulated in parallel. PDES synchronization protocols are used to ensure that parallel execution of these LPs does not violate the causality constraint, i.e. events are strictly processed in timestamp order. PDES synchronization protocols can be broadly classified into two categories, conservative and optimistic. Conservative protocols [7, 4] strictly enforce the causality constraint by allowing the simulation to progress up to a safetime. A safetime is a time guarantee that no events with a lower timestamp will be received from the upstream LPs, and is normally computed based on events received from the upstream LPs. Unlike the conservative approach, optimistic protocols [8, 12] allow events to be executed without considering if executing those events will result in any causality violation. Each LP in this case will proceed to execute each incoming event as it arrives. If an event message arrives and has a timestamp that is lower than the ones that have been processed, the LP must correct this error by undoing those events that have been executed.

The focus of this paper is on the conservative protocol. Conservative protocols can be further sub-divided into two classes: synchronous [4] and asynchronous [7]. In the synchronous approach, LPs progress in supersteps. Each LP must wait for all the other LPs to complete their current superstep before the next superstep can proceed. In the asynchronous approach, each LP progresses in the simulation independently as long as there exist events that are

[†]Gintic Institute of Manufacturing Technology, 71 Nanyang Drive, Singapore 638075

[‡]Centre for Advanced Information Systems, School of Computer Engineering, Nanyang Technological University, Singapore 638798

safe to process. In this paper, we will provide a summary of our work in successively refining a synchronous conservative superstep protocol. A separate paper describing our work in using the asynchronous approach in a conservative synchronization protocol can be found in [9]. We will compare the performance of the different improvements made to the conservative superstep protocol using a semiconductor supply-chain simulation model.

The rest of the paper is organized as follows. Section 2 explains the original superstep protocol described in [4]. Section 3 describes the three refined superstep protocols that improve the safetime bound calculation. Section 4 discusses three important issues for the superstep protocols. In section 5, the semiconductor supply-chain simulation model used in the experiments will be explained. The performance of these protocols when running the supply-chain simulation model will also be presented and compared. Section 6 provides an overview of other related work carried out by researchers in the PDES community. Section 7 concludes this paper and outlines future research directions for our project. The proof of correctness of the four versions of the superstep protocol is given in the appendix.

2. Original superstep protocol. We will first describe the original conservative superstep protocol based on that presented in [4]. The conservative superstep protocol proceeds in a series of supersteps, where each superstep is followed by a barrier synchronization. Each superstep consists of two phases, the computation phase and the communication phase. This is analogous to the bulk synchronous parallel (BSP) model first proposed in [22]. The communication phase involves the exchange of event messages between the LPs. In the computation phase, each LP will compute a safetime for itself for the current superstep. The safetime is usually computed based on the events received from the upstream LPs. The LP can then simulate, in the current superstep, all events in its event-list with timestamp less than or equal to its safetime without violating any causality constraint.

The algorithm for the original conservative protocol is shown in Figure 2.1. In the algorithm, each communication link between two LPs is implemented by two buffers, which are priority queues, to take advantage of the shared memory architecture. The sender LPs will insert external events to one buffer while the receiver LPs will receive events from the other. The two buffers are swapped at the end of the superstep in the main simulation loop. With this approach, the conflict in accessing the same buffer between the sender and the receiver LPs is avoided. This eliminates the need for buffer locking.

The calculation of SafeTime (Part2(A) in Figure 2.1) uses both local and global information, and is set to be the maximum of LP_i.InClock and the global simulation time (GST). Intuitively, the LP_i.InClock provides the local information while GST constitutes the global information.

To calculate LP_i.InClock, a *link clock* is kept for each input link of LP_i (e.g. in the pseudo-code in Figure 2.1, LP_i.clock[k] holds the link clock value for the input link from LP_k to LP_i). It keeps track of the timestamp of the

```
// PART 1: Global initialization
Initialize the links between LPs and each LP's state.
GST = 0;
for all initial event e caused by InitialState do
  // assume event e scheduled for $LP_i$
  OrderInsert(e@e.TimeStamp, $LP_i$.event_q);
endfor

// PART 2: Executed by every LP, say $LP_i$.
while (GST < $\infty$) do

  // (A) calculate SafeTime
  $LP_i$.Inclock = $\infty$
  for each $LP_k$, s.t. $LP_k$ has a directed link to $LP_i$ do
    if(InBuff[$LP_k$][$LP_i$] $\neq$ empty) // update the link clocks
      $LP_i$.clock[k]=LastElementTime(InBuff[$LP_k$][$LP_i$])
      Merge InBuff[$LP_k$][$LP_i$] into $LP_i$'s event-list, i.e. $LP_i$.event_q
    endif
    $LP_i$.InClock=min($LP_i$.InClock, $LP_i$.clock[k]);
  endfor
  SafeTime = max($LP_i$.InClock, GST);
  $LP_i$.out = $\infty$; // Tracks the smallest time of all external events sent out by
     $LP_i$

  // (B) simulate all safe events
  while (FirstElementTime($LP_i$.event_q) $\leq$ SafeTime) do
    e = RemoveFirstElement($LP_i$.event_q); // dequeue an event and process it
    $LP_i$.local_time=e.TimeStamp;
    $LP_i$.state = Simulate(e);
    for all InternalEvent ie caused by Simulate(e) do // enqueue new internal
     events
      OrderInsert(ie@ie.TimeStamp, $LP_i$.event_q);
    endfor
    for all ExternalEvent ee caused by Simulate(e) do //output external events
      // external event ee has timestamp equal to $LP_i$.local_time
      OrderInsert(ee@$LP_i$.local_time, OutBuff[$LP_i$][$LP_j$]) where ee is from $LP_i$
     to $LP_j$
      $LP_i$.out = min($LP_i$.out, ee.TimeStamp);
    endfor
  endwhile

  // (C) calculate smallest timestamp of any event in this LP
      if ($LP_i$.event_q $\neq$ empty) then $LP_i$.MinTime =
    FirstElementTime($LP_i$.event_q);
  else $LP_i$.MinTime = $\infty$; end if
  $LP_i$.MinTime = min($LP_i$.out, $LP_i$.MinTime);

  // (D) global reduction to calculate new GST after all LPs reach
    barrier
  barrier_begin();
  GST = min_reduce($LP_i$.MinTime);
  Swap InBuff and OutBuff
  barrier_end();

endwhile
```

FIG. 2.1. *Original Superstep Parallel Simulation Protocol*

last message received on the input link in the last superstep. The timestamp of the last message is readily available when the input link is implemented using priority queues ordered by the timestamp of events. LP_i.InClock is defined as the minimum value of all input link clocks. Thus, from the current superstep onwards, no event can arrive on an input link with a timestamp that is smaller than the LP_i.InClock value. As the calculation of LP_i.InClock assumes messages for communication links are received in non-decreasing timestamp order, all external events generated by an LP have timestamp equal to its local simulation time.

GST is defined as the minimum of the simulation time of all events waiting in the event-list, and the timestamp of all events that have just been generated in the current superstep. At the end of a superstep (Part 2(C) of the pseudocode in Figure 2.1), each LP computes the minimum timestamp event that it knows about, by taking the minimum timestamp of events in its event-list, and those events that it generated in the current superstep. This minimum value is stored in the MinTime field. GST can then be computed by doing a global min-reduction of all the MinTime's. Therefore, GST is the smallest timestamp of any event in the whole system and so no event with a smaller timestamp than GST can be generated.

With the GST and LP_i.InClock values, SafeTime of the current superstep can then be computed. Each LP can proceed to simulate events with timestamp smaller than or equal to its SafeTime (Part 2(B) in Figure 2.1). The simulation terminates when GST reaches infinity (beginning of Part 2). This happens when there are no more events in the system. The algorithm is proved to be safe and deadlock free in the appendix.

3. Refinement of the original protocol. In this section, we discuss the three refined protocols which are improved versions of the original conservative superstep protocol. In each case, the improvement lies in the way the safetime of each LP is computed. In addition, the refined protocols all exploit lookahead information in the safetime computation. The original protocol does not use this lookahead information since external events are timestamped with the sender LPs' local simulation time. A technique called event pre-sending (to be discussed in Section 4) is used to further enhance the lookahead value for the three refined protocols. This has a significant impact on the performance of the protocols.

3.1. Sender-Simulation-Time protocol (SST). In this section, we will discuss the Sender-Simulation-Time (SST) protocol. The improvement to the safetime calculation is derived by noting that if there is a link from LP_k to LP_i, the original algorithm uses the link clock in the safetime calculation. However, we know that the timestamp of future events from LP_k must be greater than or equal to LP_k's local simulation time. Therefore, the local simulation time would give a more relaxed bound for LP_i's safetime calculation than the link clock used in the original protocol.

Figure 3.1 shows the algorithm for the modified conservative superstep protocol. The SST algorithm has essentially the same structure as the original algorithm. The two major differences are in Part 2(A), where the SafeTime is computed, and Part 2(B), where external events are pre-sent with timestamp equal to their occurence time.

In Part 2(A), for each LP_k that has a directed link to LP_i, LP_i computes the value

```
// PART 1: Global initialization
Initialize the links between LPs and each LP's state.
GST = 0;
for all initial event e caused by InitialState do
  // assume event e scheduled for $LP_i$
  OrderInsert(e@e.TimeStamp, $LP_i$.event_q);
endfor

// PART 2: Executed by every LP, say $LP_i$.
while (GST < $\infty$) do

  Merge events from all InBuff[$LP_k$][$LP_i$] for all $LP_k$ connected to $LP_i$, into
     $LP_i$'s event-list

  // (A) calculate SafeTime
  SafeTime = $\infty$
  for each $LP_k$, s.t. $LP_k$ has a directed link to $LP_i$ do
    SafeTime = min (SafeTime, $LP_k$.local_time + LookAhead[$LP_k$][$LP_i$])
  endfor
  SafeTime = max(GST, SafeTime)
  $LP_i$.out = $\infty$; // Tracks the smallest time of all external events sent out by
     $LP_i$

  // (B) simulate all safe events
  while (FirstElementTime($LP_i$.event_q) $\leq$ SafeTime) do
    e = RemoveFirstElement($LP_i$.event_q); // dequeue an event and process it
    $LP_i$.local_time=e.TimeStamp;
    $LP_i$.state = Simulate(e);
    for all InternalEvent ie caused by Simulate(e) do // enqueue new internal
     events
      OrderInsert(ie@ie.TimeStamp, $LP_i$.event_q);
    endfor
    for all ExternalEvent ee caused by Simulate(e) do //output external events
      Insert(ee@ee.TimeStamp, OutBuff[$LP_i$][$LP_j$]) where ee is from $LP_i$ to $LP_j$
      $LP_i$.out = min($LP_i$.out, ee.TimeStamp);
    endfor
  endwhile

  // (C) calculate smallest timestamp of any event in this LP
  if ($LP_i$.event_q $\neq$ empty) then $LP_i$.MinTime =
     FirstElementTime($LP_i$.event_q);
  else $LP_i$.MinTime = $\infty$; end if
  $LP_i$.MinTime = min($LP_i$.out, $LP_i$.MinTime);

  // (D) global reduction to calculate new GST after all LPs reach
     barrier
  barrier_begin();
  GST = min_reduce($LP_i$.MinTime);
  Swap InBuff and OutBuff
  barrier_end();
endwhile
```

FIG. 3.1. *Sender-Simulation-Time Protocol*

SafeTimeBound$_k$ = LP_k.local_time + LookAhead[LP_k][LP_i]. LP_k.local_time is the current simulation time of LP_k. LookAhead[LP_k][LP_i] represents the minimum advancement in simulation time required by LP_k to generate an external event for LP_i. The value SafeTimeBound$_k$ thus provides a time guarantee to LP_i that the next external event LP_k sends to LP_i will have timestamp no less than SafeTimeBound$_k$. LP_i then computes STB = the minimum of all its SafeTimeBound$_k$ values. The SafeTime of LP_i is taken to be the maximum of STB and GST.

In Part 2(B), each LP can simulate all events with timestamp smaller or equal to its SafeTime. Since the SafeTime calculation in this algorithm does not rely on the information on the link clocks, external messages need not be constrained to have their timestamps equal to the sender LP's local time. This allows external messages to be pre-sent with timestamp equal to their occurence time. The priority queue between LPs in the original algorithm can also be replaced by an unordered list and the OrderedInsert operation by a simple Insert. Any pre-sent external events received by an LP are held in the LP's event-list (ordered by timestamp of events) and are executed only when the LP's SafeTime becomes greater than or equal to the event's timestamp. The correctness of this protocol is proven in the appendix.

3.2. Sender-Last-SafeTime protocol (SLS). In this section, we describe the second improvement made to the conservative superstep protocol. We will refer to this algorithm as the Sender-Last-SafeTime (SLS) algorithm in subsequent sections.

We first note that at the end of each superstep n, each LP_k has simulated all events e that satisfy the condition $t_e \leq LP_k.S_n$, where t_e is the timestamp of event e and $LP_k.S_n$ is the safetime of LP_k for superstep n. The local time of LP_k is now set to the timestamp of the last event executed in this superstep. Since the safetime $LP_k.S_n$ for LP_k is a guarantee that any event it receives in the next superstep will have timestamp at least equal to or greater than $LP_k.S_n$, we can effectively take the safetime of LP_k as its local time at the end of a superstep and use this value to compute the safetime of its individual receiver LPs at the beginning of the next superstep.

Figure 3.2 shows the SLS algorithm. In this algorithm, we use a variable *LastSafeTime* to keep track of the safetime of each LP in the previous superstep. In Part 1, this variable is initialized to 0. Since each LP starts off with a local time of 0, we can use the individual lookahead values in the links to compute an initial safetime for each LP.

In the SLS algorithm, the only difference from the SST algorithm presented in section 3.1 is the computation of SafeTime in Part 2(A). For each LP_i, it computes a SafeTimeBound$_k$ value for each LP_k that has a directed link to it. However, instead of using the local time of LP_k, we now use the previous SafeTime of LP_k to compute SafeTimeBound$_k$. We set SafeTimeBound$_k$ to be the sum of LP_k.LastSafeTime and LookAhead$[LP_k][LP_i]$.

In Part 2(C), the GST computation is unchanged. However, in addition to the GST computation, each LP has to store the SafeTime value computed in the beginning of the current superstep in the LastSafeTime variable. The algorithm is proven to be correct in the appendix.

3.3. Future-Event-Time protocol (FET). The Future-Event-Time (FET) protocol is a further refinement of the SLS protocol, in which the safetime bound calculation is relaxed further. The FET protocol relaxes this bound by computing the safetime based on the sender LP's first safe event time in the current superstep instead of the sender LP's safetime at the last superstep. The