Proc. 1997 IEEE Int. Conference on Systems, Man and Cybernetics, Orlando, FL, October 1997,
IEEE Computer Society Press, Los Alamitos, CA, pp. 2996-3001

System Knowledge Acquisition in Parallel Discrete Event Simulation*

Ewa Deelman
Computer Science Department
Rensselaer Polytechnic Institute
Troy, NY 12180

Boleslaw K. Szymanski
Computer Science Department
Rensselaer Polytechnic Institute
Troy, NY 12180

Abstract *Optimistic protocols designed for Parallel Discrete Event Simulation (PDES) rely heavily on the Global Virtual Time (GVT) calculation. Since the simulation uses large amounts of memory, the GVT is used to synchronize processes and discard obsolete system information. In this paper we present a new algorithm, the Continuously Monitored Global Virtual Time (CMGVT). System information, such as the Local Virtual Time (LVT) of each process and information about messages in transit, is appended to simulation messages. We describe and analyze three variants of our GVT algorithm: direct, indirect and transitive knowledge. The direct knowledge algorithm maintains only the local information about outstanding messages. The indirect version is augmented with the information about the knowledge of its direct neighbors. Finally, the transitive version is the most comprehensive. It keeps track of the outstanding messages sent by all processes in the system.*

1 Introduction

In Discrete Event Simulation (DES) [1], the physical system is modeled by a Logical Process (LP) consisting of the process state, the clock, and the event queue. Events are placed in the queue in the order of the time for which they are scheduled to occur. The simulation progresses as events are removed from the queue and processed. The availability and power of advanced, distributed memory, parallel platforms makes Parallel Discrete Event Simulation (PDES) attractive as a modeling technique. When the simulation is brought to a parallel or distributed architecture, the physical system is decomposed into several LPs, each with its own state, clock and event queue. The challenge in managing multiple LPs is to preserve causality between events. The two main approaches developed to solve this challenge are based on conservative and optimistic protocols [2]. The former prevents causality errors by limiting each LP's progress in time, so although causality is guaranteed [3], tight synchronization between LPs is introduced. The optimistic approach allows causality errors to occur; however, recovery requires rolling back the computation to the time just prior to such an error [4]. Once the rollback is completed, the computation is restarted.

In this paper, we concentrate on optimistic PDES. When a process at time t_1 receives an event with time $t_2 < t_1$ (this event is known as a *straggler*), it sends out *antimessages* (messages used to cancel erroneous event messages) corresponding to the messages it sent in the interval $t_2 \leq t \leq t_1$. Then, it restores its state to the time just prior to t_2 and restarts the simulation. In order to support rollback, it is necessary to save the constantly changing state information, as well as the incoming and outgoing messages. Therefore, the major drawback of the optimistic protocol is the amount of memory it requires. One way to reclaim memory is to determine which data residing in memory are no longer needed. This may be done by computing the Global Virtual Time (GVT), the minimum Local Virtual Time (LVT) of all the LPs and of the timestamps of all messages in transit. By definition, there are no events or messages in the system with a timestamp lower than the GVT, so all information (states and messages saved), that refers to times smaller than the GVT can be removed from the system.

In this paper we present a new GVT algorithm, which continuously computes a GVT estimate called the Continuously Monitored Global Virtual Time (CMGVT). Our algorithm keeps track of all messages in transit by appending information about them to event messages as well as to antimessages. This algorithm is based on the idea of the *vector* and *matrix clocks* [5], used to order events and discard obsolete system information in

*This work was supported by the National Science Foundation under Grants BIR-9320264 and CCR-9527151. The content of this paper does not necessarily reflect the position or policy of the U.S. Government—no official endorsement should be inferred or implied.

distributed systems. The challenge, however, is to be able to handle logical time going backward as well as forward, which is not supported by the vector and matrix clocks. The CMGVT distinguishes itself from other GVT algorithms since it does not require special synchronization rounds in order to calculate the GVT. The CMGVT is computed on each process based on the information constantly available to it. We present three versions of the algorithm. The difference between the three is the amount of information that is sent between the LPs. Section 4 describes the differences in detail.

2 Global Virtual Time

The major difficulty in the GVT calculation involves accounting for messages in transit. Even though all LPs might have an LVT $> t_x$, it is possible that a message with a timestamp $t_m < t_x$ has been sent but not yet received. Upon receipt of the message, the receiving LP will have to roll back to the time just prior to t_m. To keep track of messages in transit, some approaches involve acknowledging every message received while keeping track of the messages that were not acknowledged [6]. Each process keeps a list of unacknowledged messages. Upon receipt of a message, the receiving LP sends an acknowledgment. When it is received, the message is removed from the unacknowledged list. The GVT is calculated by synchronizing the LPs and taking the minimum of all local virtual times and the timestamp of all unacknowledged messages.

A centralized message tracking algorithm was proposed by Bauer [7]. In this algorithm, the processes send information messages to a central process. The messages contain information about what messages were sent and received via static communication channels. The central process combines the available information and redistributes its knowledge back to the processes. This approach, however, suffers from a communication bottleneck, since the central process can be flooded with incoming messages.

The SPEEDES system uses a practical approach to the GVT calculation [8]. When this calculation is initiated, the processes enter a *risk free* mode, in which, although they continue to process local events, they do not send any event messages; however, antimessages are sent in order to minimize impending rollbacks. During the GVT calculation phase a rollback may happen on any process, and it may cause other processes to roll back. In this phase, the LPs will not produce new events for the other processes.

3 CMGVT

Continuously Monitored Global Virtual Time (CMGVT) is designed to monitor the progress of the simulation by using locally available information as well as by keeping track of the message traffic. All messages contain a serial number. The general idea behind the algorithm is to propagate through the system the information about the LVT of the processes and about all the messages being sent and received. This is achieved by making an LP append to the event messages and antimessages its knowledge about the LVT of the LPs in the system, the number of messages that were sent by all of them, and the messages in transit. We also include indirect knowledge—the knowledge the sender has about the knowledge of the neighboring LPs about the system. We use direct and indirect knowledge to infer what messages are still in transit. Once a process receives a message from another process, it knows at least as much about the system as the sender knew at the time the message was issued.

The idea of piggy-backing system information has been used before in distributed systems. The *vector clock* [9], which consists of a vector with a size equal to the number of processes, describes the logical progress of each process in the system. This clock can be used, for example, for causal ordering of messages in a distributed environment. A *matrix clock* [5], which is represented by a $p \times p$ matrix, where p is the number of processes, describes the knowledge a particular process has about the knowledge that all the processes in the system have about each other. The matrix clock is mostly used to discard obsolete system information.

These clocks are inadequate for optimistic PDES because they rely on the assumption that logical clocks can only move forward. In optimistic PDES, the LVT can also move backward, due to rollback. There is, however, one measure of the simulation that is monotonically increasing: the number of messages being sent by each process. Our "logical clock" keeps track of the knowledge of the number of the messages sent in the system. To maintain the knowledge about the LVT of all processes and about the messages in transit, the CMGVT uses basic structures, which are maintained locally by each process: the *Message Matrix* and the *Table of Forcing Vectors*.

The Message Matrix (MM): an entry (j, k) in the matrix MM_i, belonging to LP_i, represents the knowledge that LP_i has about the knowledge LP_j had about the total number of messages that LP_k has sent. Not all the entries in the

MM are necessary. Each LP needs to record only the knowledge of it's logical neighbors (LP's with which the process is communicating). One row of the matrix must contain all the entries (row i in MM_i). This row describes the knowledge that LP_i has about the entire system. The size of the matrix is $p+(p-1)\times K$, where p is the number of processes, and K is the number of logical connections each process has to others (usually $K \ll p$, and often $K = O(1)$, particularly if the domain is a two or three dimensional space). This gives the size of the matrix $O(p)$. To simplify the explanation, we describe the algorithm for the case in which $K = p$, i.e. all processes are connected with one another.

$$MM_1 = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 5 & 2 & 1 \\ 0 & 3 & 2 & 0 \\ 0 & 2 & 1 & 1 \end{bmatrix} \quad (1)$$

The sample MM matrix (Eqn.1) is located on LP_1 (by convention we are counting processes starting from 0) and is denoted by MM_1. The system has only four LPs. The entry (1,1) with the value 5, denoted by $MM_1(1,1)$, indicates that LP_1 knows that LP_1, (in this case itself) has sent out five messages. LP_1 also knows of no messages sent by LP_0 ($MM_1(1,0)$), two messages sent by LP_2 ($MM_1(1,2)$), and one message sent by LP_3 ($MM_1(1,3)$). The MM also contains the knowledge of the owner process (here LP_1) about the knowledge that other processes had at the time of sending their last messages. Row 0 describes the knowledge of LP_0 about processes LP_0 to LP_3. Clearly, here LP_1 does not have any information about LP_0's knowledge of the system. However, the MM provides information about the knowledge of processes LP_2 and LP_3. For example, LP_1 knows that LP_2 knew about three messages sent by $LP_1(MM_1(2,1))$, two messages ($MM_1(2,2)$) sent by itself, and none sent by LP_0 ($MM_1(2,0)$) or LP_3 ($MM_1(2,3)$). Obviously, the MM does not provide any information about which messages have been accounted for. This information is maintained in a **Table of Forcing Vectors** (TFV).

A Forcing ($F(t,n,c)$) represents the basic information about a message, and is composed of three data: the time t at which the event in the message is scheduled to happen, the process n sending the message, and the current outgoing message count c on that process. Each process has a table (the TFV) indexed by the LP number. The entries of this table contain the current known logical virtual time of each LP and the *Forcing Vectors* for each process. The *Forcings* form the components of the vectors. If a forcing is placed at entry i in the table, it means the message represented by the forcing was sent to LP_i. The size of the table is $(m+1)\times p$, where m is the maximum number of forcings in a vector which is a parameter of the simulation. This gives us the size of the additional overhead introduced by the CMGVT to be $O(p)$.

$$TFV_1 = \begin{bmatrix} 0 & [F(10,1,4), F(10,2,1)] \\ 17 & \\ 19 & [F(7,1,2)] \\ 26 & [F(9,1,3)] \end{bmatrix} \quad (2)$$

A simple TFV is shown in Eqn.2. This TFV belongs to LP_1. Forcing $F(10,1,4)$ represents a message sent by LP_1 to LP_0. The message has serial number 4 and was sent at virtual time 10. LP_1 also knows that LP_2 has sent its first message to LP_0 at time 10. These messages are unacknowledged, because LP_1 has no information about LP_0 (see Eqn.1). We also see that LP_1 sent its second message to LP_2 at time 7 and the third at time 9 to LP_3. The TFV_1 also shows that LP_1 thinks that LP_2's LVT is 19 and that LP_3's is 26. We see that unless LP_2 rolls back before receiving the message from LP_1, it will have to roll back to the state prior to time 7. Theoretically, the forcing vectors can be infinite in size, but they are bounded to a finite size in our system. If the forcing vector gets too long for a certain LP, a query message is sent to it. Upon receipt of this message, the queried process sends an answer message containing up-to-date local information.

Update on Send $LP_i \rightarrow LP_j$ Every time processes communicate, the sender sends along its Message Matrix and its Table of Forcing Vectors. The table and the matrix are updated just before the send operation is performed. When LP_i sends a message to LP_j with timestamp t_s, $\mathrm{MM}_i(i,i)$ is incremented by one. A new forcing is also created—$\mathrm{F}(t_s,i,c)$, where c is the current outgoing message count ($c = \mathrm{MM}_i(i,i)$). This vector is added to the entry j in the TFV_i. The current LVT of LP_i is inserted into the TFV_i at entry i. The MM_i and the TFV_i are appended to the message being sent from LP_i to LP_j.

Update on Receive $LP_j \rightarrow LP_i$ Upon receipt of a message, the local Message Matrix and the Table of Forcing Vectors have to be updated based upon the new information received. The TFV is updated first. Updating the TFV involves checking the forcings in each vector against the Message Matrix. The local forcings are checked against the incoming Message Matrix, and the incoming vector's forcings are checked against the

local Message Matrix. Of course, the incoming forcing $F(t, j, c)$ at entry i, where c is the current message number, is automatically acknowledged since it refers to the current message. The unacknowledged forcings are entered into the local Table of Forcing Vectors at the appropriate entries, and the acknowledged forcings are discarded. Consider the TFV at process i with a non-zero vector length at entry $k \neq i \wedge k \neq j$, where LP_j is the sender. Let this entry contain a forcing $F(t_x, i, c_x)$. This means that a message with timestamp t_x and serial number c_x was sent by LP_i to LP_k. Since the forcing is still present in TFV_i, LP_i does not know if LP_k knows about that message. The question is: does LP_j have any information indicating that LP_k knows about the message (i.e., whether LP_k received the message)? To answer this question, we look at the incoming Message Matrix (MM_j). Assume that $MM_j(k, i) = c_y$, so LP_k knows of at least c_y messages that LP_i has sent. If $c_y \geq c_x$, it implies that LP_k knows of the message described by the forcing $F(t_x, i, c_x)$. This forcing can be discarded since it has been acknowledged (indirectly) by LP_k. If, however, $c_y < c_x$, it means that, to LP_j's knowledge, LP_k did not receive the message. The forcing then has to remain in the table.

The LVTs present in the TFV are updated with information brought in from the process with the most current knowledge. Such a process if identified by the number of messages which it is aware of. The more messages a process knows about, the more recent its information is. The LVT for entry k is taken from the process which knows about the most messages sent by the process LP_k.

Next, the local Message Matrix has to be updated. First, LP_i's knowledge about LP_j has to be updated. Then, LP_i's knowledge about other LPs is compared to LP_j's information about the others. The rest of the entries are also updated based upon the most current knowledge. If LP_j knows more about LP_k's knowledge than LP_i knows about it, then $MM_j(k, x) > MM_i(k, x)$. Hence, $MM_i(k, x)$ is updated to the most recent value ($MM_j(k, x)$).

Most of the work is spent on maintaining the MM and TFV data structures. Thus the GVT calculation is very simple: an LP just takes the minimum of all the LVTs and the minimum of all the forcings in the TFV. Obviously, the GVTs calculated by each of the LPs in the system need not be the same, because each LP is likely to have information about the system which is different than knowledge of other LPs. Also, it is up to each LP to decide when it wants to perform the GVT calculation. It can do so periodically, or only when on the verge of running out of memory. A more detailed explanation of the algorithm and its performance comparison with the SPEEDES algorithm is presented in [10].

4 Three Levels of Knowledge

In this paper, we investigate the impact of the amount of knowledge that is sent between the LPs on the quality of the GVT estimate. The algorithm described above contains the **TRANSITIVE** knowledge an LP has about the system. In the second version, **INDIRECT**, we include only the knowledge that each LP has about it's direct neighbors. Each LP appends the entire MM as above and the LVTs of the other LPs are also included, but only the forcings for its neighbors are added (in effect, these are the forcings representing the outstanding messages that the LP sent to its neighbors). The third version, **DIRECT**, contains the least amount of information. Only the LPs knowledge about itself is included. Here, the forcings are maintained locally by the LP, but are not sent. Only the minimum time of the forcings is sent along with the LVT information. The MM is now only a vector (row i on LP_i), representing the number of messages that the sender is aware of. Below we present the performance of these three versions.

5 Application

Our research focuses on spatially explicit problems. The system consists of a two-dimensional lattice. Each node of the lattice may contain any number of objects whose behavior we are simulating. Associated with each object are events, most of which are local to the node. The occurrence of a local event will cause a state change only at the particular node occupied by the object. We also have a non-local event, namely the "Move Event", which causes the object to move from one node of the lattice to the next. This event, obviously, affects the state of at least two nodes of the lattice. We divide the lattice into sections. Each section is assigned an LP, which is responsible for simulating all the events occurring in its section. When a "Move Event" occurs, which causes an object to move from section i to section j, LP_i, which is responsible for section i, sends a message to LP_j. This message contains the object being moved, the "Move Event", and all the future events associated with the object.

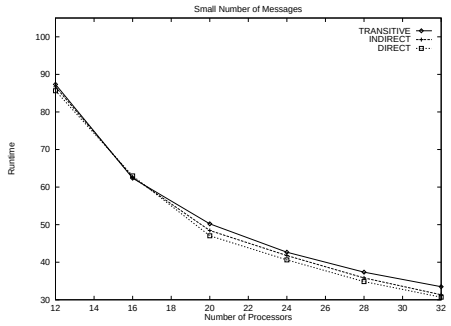

Figure 1: Runtime with a Small Number of Messages.

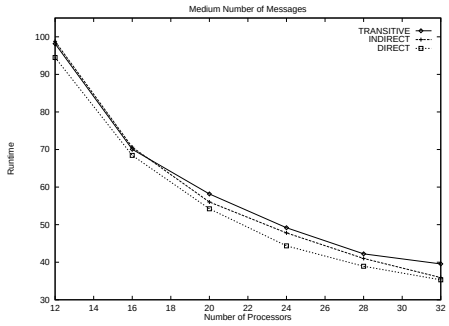

Figure 2: Runtime with a Medium Number of Messages.

We have currently implemented in our system a simulation of the spread of Lyme disease [11].

The application is programmed in C++ using MPI [12] for message passing. The simulation runs on the IBM SP2, an MIMD distributed memory machine with several processors (our configuration consists of 32). We used a strip decomposition in the dominant direction to divide the lattice among LPs. The results presented below were obtained with the division of the space into as many strips as we have processors, giving the assignment of one LP per processor. We have also investigated multiple LP-to-processor mappings [13]. We use the CMGVT algorithm to calculate the GVT at predetermined intervals of the simulated time.

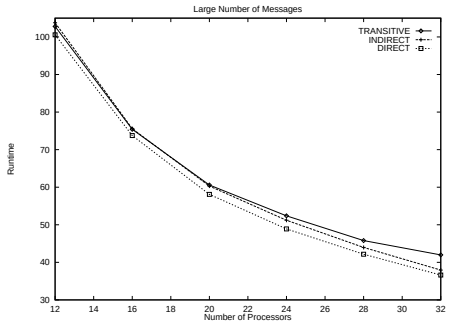

Figure 3: Runtime with a Large Number of Messages.

6 Results

First, we present the runtime differences of the three versions of the algorithm. Intuitively, since more information and message processing is contained in the TRANSITIVE version, it is expected that it will take longer to run than the INDIRECT and DIRECT versions. We tested the conjecture with three different message loads (small, medium and large). In all three cases (Figures 1,2,3) the DIRECT approach performed best, followed by the INDIRECT and TRANSITIVE versions. Below 12 processors, the algorithms performed similarly. In order to analyze how well the algorithms were able to estimate the GVT, we looked at how close the estimate was to the LVT. This difference is significant, because it is directly proportional to the amount of memory needed by an LP. When the difference is large, large amounts of memory are needed to hold state information. The following curves represent the average difference between the LVT and the GVT of the LPs of the system. Again three different message loads were considered. From Figure 4, we can see that when the LPs communicate infrequently, the TRANSITIVE version performs best. This is because the larger amount of information contained in the message allows the GVT to be estimated more accurately. When the message load increases (Figures 5 and 6) the differences become smaller.

It is interesting to note that Figures 5 and 6 indicate that the INDIRECT and DIRECT versions are not easy to compare. Sometimes the former is better, other times the latter. We believe that such varied performance is caused by the finite size of the forcing vectors. In the DIRECT method, an LP that keeps sending messages without receiving acknowledgements adds forcings to its local data structure. As a result, the forcing vector grows too long and the LP queries the recipients of the messages directly. Consequently, such an LP will receive the most up-to-date information from the recipients and will have a good GVT estimate. When the flow of messages increases, the need for update dimin-

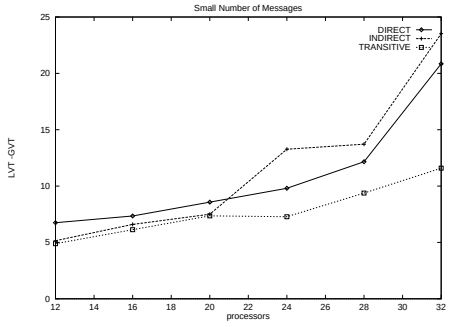

Figure 4: Small Number of Messages.

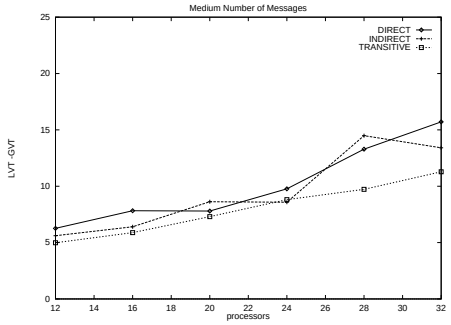

Figure 5: Medium Number of Messages.

ishes and therefore the performance is more consistent. In conclusion, when choosing a version of the CMGVT algorithm one has to take into account memory requirements. When there is enough memory available the DIRECT approach is the fastest. However, if memory is a constraint, the TRANSITIVE version will give better results.

References

[1] C.G. Cassandras. *Discrete Event Systems: Modeling and Performance Analysis.* New York, NY: Irwin Publ., 1993.

[2] R. M. Fujimoto. Parallel Discrete Event Simulation. *Communications of the ACM*, Vol. 33, No. 10, Oct. 1990, pp. 31–53.

[3] K. M. Chandy and J. Misra. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. *IEEE Trans. on Software Engineering*, Vol. 5, 1979, pp. 440–452.

[4] D.R. Jefferson. Virtual Time. *Trans. Prog. Lang. and Syst.*, Vol. 7, 1985, pp. 404–425.

[5] M. Raynal and M. Singhal. Logical Time: Capturing Causality in Distributed Systems. *IEEE Computer*, Vol. 29, Feb. 1996, pp. 49–53.

[6] G. Tel. *Topics in distributed algorithms.* Cambridge, UK: Cambridge University Press, 1991.

[7] H. Bauer and C. Sporrer. Distributed Logic Simulation and an Approach to Asynchronous GVT-Calculation. *Workshop on Parallel and Distributed Simulation*, 1992, pp. 205–208.

[8] J. S. Steinman, C. A. Lee, L. F. Wilson, and D. M. Nicol. Global Virtual Time and Distributed Synchronization. *Workshop on Parallel and Distributed Simulation*, 1995, pp. 139–148,

[9] C. Fidge. Logical time in distributed computing systems. *IEEE Computer*, Vol. 24, No. 1, 1991, pp. 28–33.

[10] E. Deelman and B. K. Szymanski. Continuously Monitored Global Virtual Time. to appear in *International Conference on Parallel and Distributed Processing Techniques and Applications*, 1997.

[11] E. Deelman, T. Caraco, and B. K. Szymanski. Parallel Discrete Event Simulation of Lyme Disease. *Pacific Biocomputing Conference*, 1996, pp. 191–202.

[12] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI.* Cambridge, MA: The MIT Press, 1994.

[13] E. Deelman and B. K. Szymanski. Simulating Lyme Disease Using Parallel Discrete Event Simulation. *Winter Simulation Conference*, 1996, pp. 1191–1198.

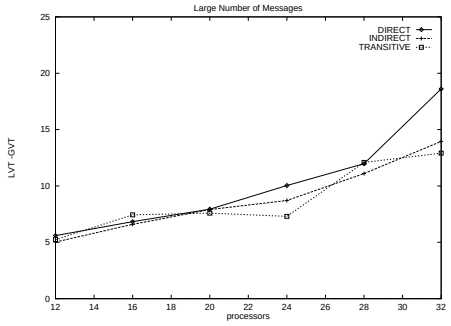

Figure 6: Large Number of Messages.