Proc. Int. Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97),
Las Vegas, NV, June 30-July 3, 1997, Vol. I, pp. 1-10

Continuously Monitored Global Virtual Time

Ewa Deelman
Computer Science Department
Rensselaer Polytechnic Institute
Troy, NY 12180
deelmane@cs.rpi.edu

Boleslaw K. Szymanski
Computer Science Department
Rensselaer Polytechnic Institute
Troy, NY 12180
szymansk@cs.rpi.edu

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). Unlike others, this algorithm allows processes to calculate the GVT based on the local information constantly available to each process. System information, such as the Local Virtual Time (LVT) of each process and information about messages in transit, is appended to simulation messages. We present experimental data that show the performance of the CMGVT algorithm.*

Keywords: distributed simulation, discrete event simulation, virtual time, parallel processing

1 Introduction

Many systems are of a discrete nature. They change their state at discrete points in time due to the occurrence of events [1]. Discrete Event Simulation (DES) models the behavior of such a system using the following mechanism: Events are placed in an event 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. Computationally, the physical system is modeled by a Logical Process (LP) consisting of the process state, the clock, and the event queue. The availability and power of advanced, distributed memory, parallel platforms makes Parallel Discrete Event Simulation (PDES) attractive as a modeling technique. Parallel machines are capable of speeding up the computation, and, by taking advantage of their large collective memory, they are able to simulate increasingly complex problems. 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 error [4]. Once the rollback is completed, the computation may safely be restarted.

When a process at time t_1 receives an event with time $t_2 < t_1$ (this event is known as a *straggler*), it rolls back as follows. First, the process 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 mes-

sages. 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. This process of memory reclaiming is referred to as *fossil collection*.

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 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. Although this algorithm involves some communication overhead, we demonstrate that it works well in our simulations. We also compare the performance of the CMGVT against a well known GVT algorithm used in SPEEDES [6].

2 Global Virtual Time Algorithms

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 [7]. Each process keeps a list of unacknowledged messages. When an LP sends a message, it puts it on the list. Upon receipt of a message, the receiving LP sends an acknowledgment to the sender. When the acknowledgment is received, the sender of the message removes it 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.

Another approach used in GVT calculations is to keep lists of messages sent and received [8]. These lists can be globally gathered and their difference determined. Then the minimum timestamp of the messages in transit and minimum of all local virtual times can be computed. Unfortunately, the lists and the messages carrying them can get very long.

If messages carry sequence numbers, they can be acknowledged by sending the highest consecutive message number received [9]. For example, assume LP_1 sends messages numbered from 1 to 50 to LP_2. Assume also that, when LP_2 receives a control message that signals the start of the GVT calculation, it already received messages numbered from 1 to 45. In such a case, LP_2 needs only to send an acknowledgment to message 45 informing LP_1 that messages 46 to 50 are still in transit. Based on this information, LP_1 can compute the minimum local time as the minimum timestamp of unacknowledged messages.

A centralized message tracking algorithm was proposed by Bauer [10]. 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, suf-

fers from a communication bottleneck, since the central process can be flooded with incoming messages.

It is easy to show that, without stopping the computation, it is impossible to calculate the true Global Virtual Time, because, by the time any process has collected global statistics, the state of the processes might have changed. Hence, the value obtained during the GVT calculation is often referred to as the EGVT (estimate for the GVT). The SPEEDES system uses a practical approach to the GVT calculation [6]. 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. However, these rollbacks would occur anyway. On the other hand, in this phase, the LPs will not produce new events for the other processes. We have implemented this algorithm in our system and used it as a basis of comparison with our CMGVT.

3 Continuously Monitored Global Virtual Time

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. We assume that the messages arrive in the order that they were sent. This assumption holds true for most message-passing environments, and it is certainly satisfied in our system. We do not send out any acknowledgment messages. 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* [11], 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. Additional data structures needed to maintain the knowledge about the LVT of all processes and about the messages in transit, are described below.

The CMGVT uses two basic structures, which are maintained locally by each process: the *Message Matrix* and the *Table of Forcing Vectors*.

3.1 Main Data Structures

The **Message Matrix** (MM): an entry (j, k) in the matrix MM_i, belonging to LP_i, repre-

sents 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 with). 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). 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 have. 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 knows of 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. It is possible that an LP receives information about an incoming message before it receives the message itself.

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

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 the 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.

At the beginning of the simulation the MM has all entries equal to zero, the LVT for every entry in the table is set to the simulation start time, and no forcings are present. When

a process sends or receives an event message or antimessage, it has to update the MM and TFV.

3.2 Update on Send $LP_i \rightarrow LP_j$

1. MM$_i$(i,i)++
2. add F(t_s,i,MM$_i$(i,i)) to TFV$_i$[j]
3. TFV$_i$[i].lvt = current lvt of LP_i

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, MM$_i$(i,i) is incremented by one. A new forcing is also created—F(t_s,i,c), where c is the current outgoing message count (c = 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.

3.3 Update on Receive $LP_j \rightarrow LP_i$

Updating the *Forcing Vector:*

For every entry k in the TFV_i

1. for every Forcing $F(t,n,c)$ in Forcing Vector ($TFV_i[k]$)
   if ($MM_j(k,n) \geq c$)
   remove F from $TFV_i[k]$
   (message acknowledged, info given by sender)
2. for every new Forcing $F(t,n,c)$ in Forcing Vector ($TFV_j[k]$)
   if ($MM_i(k,n) < c$)
   add F to $TFV_i[k]$
   (message not acknowledged, info provided by receiver)

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.

Updating the *Local Virtual Times:*

1. $TFV_i[i].lvt$ is unchanged since the receiver knows its lvt best.
2. $TFV_i[j].lvt = TFV_j[j].lvt$, update the sender's lvt based on the sender's information.
3. for all other entries in the TFV_i (if sender knows about k more than the receiver)
   if ($MM_j[j,k] > MM_i[i,k]$)
   $TFV_i[k].lvt = TFV_j[k].lvt$

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.

Updating the Message Matrix:

1. **Update what i knows about j:** $MM_i(i,j) = MM_j(j,j)$
2. **Update what i knows about others based on what j knows about others:** $\forall_{0 \le k < p \wedge k \neq i}$ $MM_i(i,k) = max(MM_i(i,k), MM_j(j,k))$
3. **Update the knowledge that i has about the knowledge of others:** $\forall_{0 \le k,l < p, \wedge k \neq i}$ $MM_i(k,l) = max(MM_i(k,l), MM_j(k,l))$

Next, the local Message Matrix has to be updated. First, LP_i's knowledge about LP_j has to be updated (step 1 above). Then, LP_i's knowledge about other LPs is compared to LP_j's information about the others (step 2). The rest of the entries are also updated based upon the most current knowledge (step 3). 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 ($MM_j(k,x)$). Steps 1-3 are distinct purely for a clarity of description. In the implementation all three steps can be collapsed into one.

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.

To demonstrate the algorithm in action, consider the following scenario. In each of the following tables, the first column represents the simulation step on each LP, the second represents the LVT, and the last represents an activity (e denotes event processing, $x \rightarrow y$ indicates a message sent from x to y, and $x \leftarrow y$ represents x receiving a message from y). The timestamp of each event sent in the message is equal to the local LVT.

LP_0

step	LVT	action
0	0	e
1	2	0→2
2	3	0→1
3	4	e
4	9	0←2

LP_1

step	LVT	action
0	0	e
1	1	e
2	3	1→0
3	5	1←0
4	6	1←2

LP_2

step	LVT	action
0	0	e
1	10	2→0
2	15	2→1
3	20	2←0
4	2	e

Let's look at what is happening on LP_1:
At step $t = 0$ no messages have been sent and the local virtual times are initialized to 0 (Eqn.3).

$$MM_1 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, TFV_1 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \quad (3)$$

At $t = 2$, after sending a message to LP_0, the "1" in the matrix (Eqn.4) shows one message sent by LP_1 and the *forcing* F(3,1,1) at entry 0 represents the message sent by LP_1 to LP_0 at time 3 (and the fact that it is LP_1's first message sent).

$$MM_1 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}, TFV_1 = \begin{bmatrix} 0 & F(3,1,1) \\ 3 & \\ 0 & \end{bmatrix} \quad (4)$$

At step $t = 3$, LP_1 receives a message from LP_0. That message contains MM_0 and TFV_0 (Eqn.6). They correspond to the MM_0 and

TFV_0 at step 2 on LP_0. Entry (0,0) in MM_0 shows that LP_0 has sent two messages. Since these messages are not acknowledged, they are represented by the two forcings in the TFV_0. The local forcings (in TFV_1, Eqn.5) are compared to the incoming knowledge (MM_0, Eqn.6). $F(3,1,1)$ is checked against entry $MM_0(0,1) = 0$. Since it indicates that LP_0 does not know of the first message sent by LP_1, the forcing remains in TFV_1. Now the forcings in TFV_0 are compared to the knowledge that LP_1 has about the system (MM_1). $F(3,0,2)$ is automatically removed, since it represents the current message. $F(2,0,1)$ is placed in TFV_1 because entry $MM_1(2,0) = 0$ shows that LP_2 knows of no messages sent by 0. The LVTs are also updated based on the most recent information. Since the message came from LP_0, $TFV_1[0].lvt$ is set to $TFV_0[0].lvt = 3$. MM_1 is just the maximum of coefficients of MM_0 and MM_1. $newMM_1$ shows that LP_1 knows that LP_0 sent out two messages. This gives LP_1 the updated TFV ($newTFV_1$) and updated MM ($newMM_1$) (Eqn.7).

$$MM_1 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad TFV_1 = \begin{bmatrix} 0 & F(3,1,1) \\ 5 & \\ 0 & \end{bmatrix} \quad (5)$$

$$MM_0 = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad TFV_0 = \begin{bmatrix} 3 & \\ 0 & F(3,0,2) \\ 0 & F(2,0,1) \end{bmatrix} \quad (6)$$

$$newMM_1 = \begin{bmatrix} 2 & 0 & 0 \\ 2 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

$$newTFV_1 = \begin{bmatrix} 3 & F(3,1,1) \\ 5 & \\ 0 & F(2,0,1) \end{bmatrix} \quad (7)$$

At $t = 4$, after receiving a message from LP_2, MM_1 is as in Eqn.7, and TFV_1, TFV_2 and MM_2 are shown in Eqn.8. Updates of MM_1 and TFV_1 are performed. $F(3,1,1)$ is checked against $MM_2(0,1) = 0$ and $F(2,0,1)$ against $MM_2(2,0) = 0$. Both forcings stay since neither LP_0 nor LP_2 know of these incoming messages. The incoming forcings are compared against the local MM; for example $F(10,2,1)$ is compared against $MM_1(0,2)$ and as a result it remains unacknowledged. $F(15,2,2)$ is the current message and therefore is automatically acknowledged. The LVT in the $TFV_1[2].lvt$ is updated to the most recent information of 15. MM_1 contains the maximum of coefficients of MM_1 and MM_2. The updated structures are shown in Eqn.9.

$$MM_2 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 2 \end{bmatrix} \quad TFV_1 = \begin{bmatrix} 3 & F(3,1,1) \\ 6 & \\ 0 & F(2,0,1) \end{bmatrix}$$

$$TFV_2 = \begin{bmatrix} 0 & F(10,2,1) \\ 0 & F(15,2,2) \\ 15 & \end{bmatrix} \quad (8)$$

$$newMM_1 = \begin{bmatrix} 2 & 0 & 0 \\ 2 & 1 & 2 \\ 0 & 0 & 2 \end{bmatrix}$$

$$newTFV_1 = \begin{bmatrix} 3 & F(3,1,1) & F(10,2,1) \\ 6 & & \\ 15 & F(2,0,1) & \end{bmatrix} \quad (9)$$

Looking at Eqn.9, we can see that the LVTs provided by the LPs are not sufficient to calculate the GVT—the forcings need to be taken into consideration. In the described scenario, the GVT is 2, because LP_2 has a forcing for that time.

4 Spatial Complexity

Obviously, only forcings for communicating processes are ever created. If we have a forcing $F(t,x,c)$ at entry y in the TFV_i, it can be removed only when we have information about how many messages LP_x knows that LP_y has sent ($MM_i(x,y)$). This implies that only entries indicating communicating processes are needed in the MM (K entries in each row, where K is the connectivity of a process). Additionally, we need the entry MM(x,x) to maintain the number of messages sent by an LP and to be able to update what the receiver knows about the sender. Additionally the row i in matrix MM_i is needed to be able to update the

LVT information. Since we do not have the full matrix to use when comparing the Message Matrices, we assume non-existent entries to be 0. The LVT update is then as described in the previous section. In summary, the Message Matrix is of size $K \times (p-1)+p$. K is the connectivity of an LP, the number of neighbors an LP has. In the case of two-dimensional spatial problems, K is up to 8. The size of the TFV is $(m+1) \times p$, where m is the maximum number of forcings in a vector which is a parameter of the simulation. Hence, an overhead of the CMGVT algorithm amounts to $(K+m+2) \times p - K = O(p)$ memory locations.

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 that 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. We have currently implemented in our system a simulation of the spread of Lyme disease [12].

The application is programmed in C++ using MPI [13] 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 [14].

To support rollback, we use incremental state saving [15]. Using full state saving would be too expensive in our simulation, because the state of the system would have to be saved after each event or at some time interval. This approach could be efficient if the events affected the majority of the state variables. Our events, however, affect at most two lattice nodes. Incremental state saving stores only the parts of the state that are actually affected by the occurrence of an event. To this end, every event keeps a copy of the state variables it has changed. A processed event is placed on a *processed event* list. When an event causes a message to be sent out (for example the "Move Event"), the event is placed on the *message list*. When an object is removed from the lattice, due to a move or a deletion, the object and all its future events are placed on the *ghost list*. Together, the three lists contain all the information needed to roll back the simulation. When a rollback for time t_r occurs at time t, first all the antimessages canceling the messages sent out in the time interval [t_r,t] are sent. Then, events that happened after t_r are removed from the *processed event* list and are undone (the state variables are restored). If an event caused the removal of an object, that object and all its future events are restored from the *ghost list*. The simulation then restarts at time t_r.

We use the CMGVT algorithm to calculate the GVT at predetermined intervals of real time. When the GVT is computed, fossil collection deletes the obsolete information from the *processed event*, *message*, and *ghost* lists.

6 Results and Conclusions

We present the performance of the CMGVT algorithm in comparison to the algorithm used in SPEEDES described in section 2. In SPEEDES, the GVT is calculated by flushing out the messages out of the system and then exchanging LVT information among the processes. The CMGVT does not use any synchronization rounds, instead it relies on the information locally available. In the results presented in this paper, the computational and communication load is distributed equally among processes. Since the mapping of processes to processors is one-to-one, below we use the term processors instead of processes.

The first Figure 1 shows the runtime for both algorithms with a relatively small problem size (48,000 node lattice and 12,000 mice). The performance of the two methods is similar, but there are slight differences. The speedup curves (Figure 2) show that SPEEDES algorithm performs slightly better for a small data set. When the number of processors is increased beyond 20, the performance of both algorithms degrades because the communication to computation ratio increases (the size of space assigned to each process decreases).

Larger problem sizes are needed to compare the performance of the algorithms with a higher number of processors. Figure 3 shows the runtime for SPEEDES and the CMGVT with about 200,000 nodes and almost 50,000 mice. The graph shows that the SPEEDES algorithm performs better than the CMGVT up to 20 processors. However when the number of processors increases to 24, 28 and 32, the runtime of the CMGVT continues to decrease, whereas the SPEEDES runtime starts to level off. This shows that the increased message size due to the additional Message Matrix and the Table of Forcing Vectors has a smaller detrimental effect on performance than the increased synchronization time imposed by the larger number of processors.

It is important to mention the role of the amount of messages sent in the system. The above results were obtained for a system in which processes exchange messages with each other throughout the simulation. This is typical of the spatially explicit problems which we are researching. However it is possible to design a system in which processes do not communicate frequently. In such a systems, the SPEEDES algorithm will perform better. Flushing the system of outstanding messages is faster. In the case of the CMGVT, due to the lack of incoming information, the processes have to query each other in order to be able to update the local data structures. The querying process might therefore slow down the progress of the simulation.

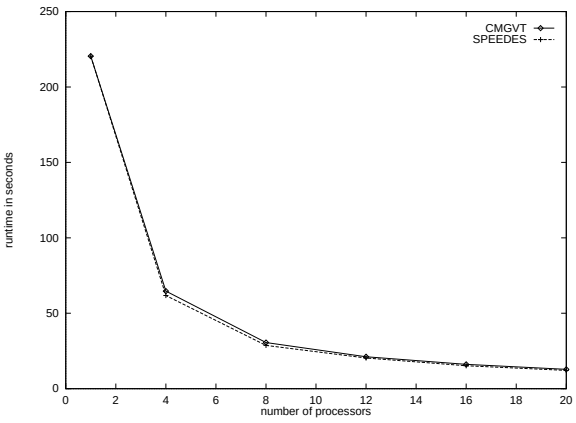

Figure 1: Runtime for a small data set

In conclusion, the SPEEDES GVT algorithm is suitable for systems where the processes communicate infrequently. When the processes are communicating with each other often, the CMGVT is a appropriate algorithm to use, especially when the number of processors used is above 20.

Acknowledgments

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.

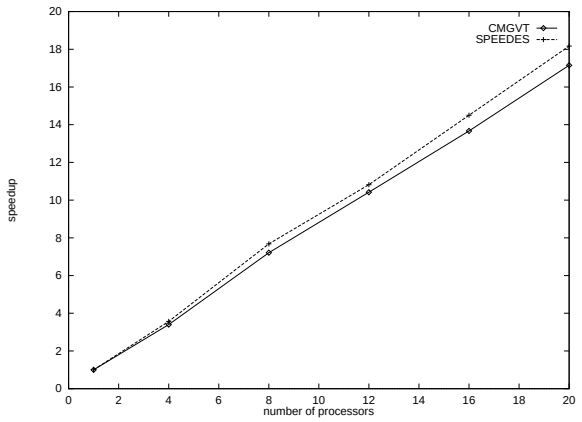

Figure 2: Speedup for a small data set

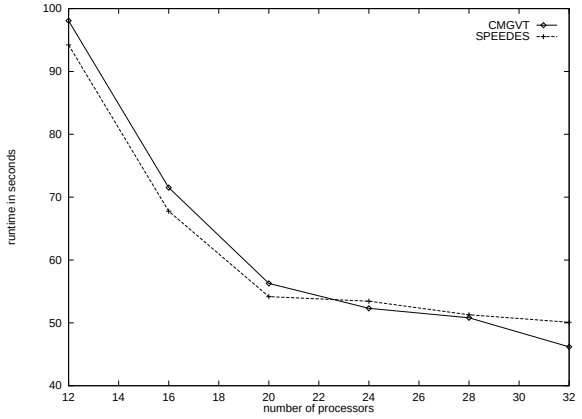

Figure 3: Runtime for a large data set

References

[1] C.G. Cassandras. *Discrete Event Systems: Modeling and Performance Analysis.* 1993.

[2] R. M. Fujimoto. Parallel Discrete Event Simulation. *Communications of the ACM*, 33(10):31–53, 1990.

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

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

[5] M. Raynal and M. Singhal. Logical Time: Capturing Causality in Distributed Systems. *IEEE Computer*, page 49, 1996.

[6] 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*, pages 139–148, 1995.

[7] G. Tel. *Topics in distributed algorithms.* Cambridge University Press, 1991.

[8] T. Lai and T. Yang. On distributed snapshots. *Inform. Process. Lett.*, 25:153–158, 1987.

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

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

[11] C. Fidge. Logical time in distributed computing systems. *IEEE Computer*, pages 28–33, 1991.

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

[13] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI.* The MIT Press, 1994.

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

[15] J. S. Steinman. Incremental State Saving in SPEEDES using C++. *Winter Simulation Conference*, pages 687–696, 1993.