Proc. 1996 Winter Simulation Conference, San Diego, December 1996, pp. 1191-1198

SIMULATING LYME DISEASE USING PARALLEL DISCRETE EVENT SIMULATION

Ewa Deelman
Boleslaw K. Szymanski

Department of Computer Science
Rensselaer Polytechnic Institute
Troy, NY 12180

Thomas Caraco

Department of Biological Sciences
State University of New York at Albany
Albany, NY 12222

ABSTRACT

Lyme Disease affects many people in the northeastern United States. One of the most important mechanisms that sustains the epidemic is the interaction between white-footed mice (*Peromyscus leucopus*) and deer ticks (*Ixodes scapularis*). When mice move around in their territory they carry diseased ticks to new locations. Our system simulates the different developmental stages of the tick through its active spring/summer period. The system uses the optimistic protocol for Parallel Discrete Event Simulation. In this paper, we present the model of the spread of the disease. We describe how we parallelize the problem, and we sketch a new global virtual time algorithm used in our system. We present performance benefits resulting from a parallel platform.

1 BIOLOGICAL OVERVIEW

Lyme Disease is caused by a spirochete (*Borrelia burgdorferi*) which is most commonly present in ticks (Barbour and Fish 1993, Miller *et al.* 1990). When an infected tick feeds on an animal or human, the spirochete may be transferred into the host's blood stream, causing an infection. Since the ticks are practically immobile, the spread of the disease is driven by their mobile hosts, such as mice and deer. An infected tick might infect a mouse, which in turn may infect an uninfected tick at a new location.

The life cycle of ticks encompasses several stages over a period of two years. The eggs hatch in the summer and become larvae. If a larva is successful in feeding on a host, it molts into a nymph. The nymphs that survive the winter, and that are able to find a blood meal during the spring, molt into adults. Adult female ticks deposit eggs. Ticks are assumed to hatch uninfected. The ticks in their larval and nymphal stages prefer to feed on mice whereas the adults prefer deer. Ticks that are unable to find a host die off. Since the mouse-tick interaction is at the heart of the Lyme disease cycle, our simulation is currently modeling only mice and ticks (Figure 1). The duration of the simulation is 180 days, starting in the late spring. This time is the most active for the ticks and mice. Mice, during that time, are looking for nesting sites and may carry ticks a considerable distance (Ostfeld *et al.* 1996).

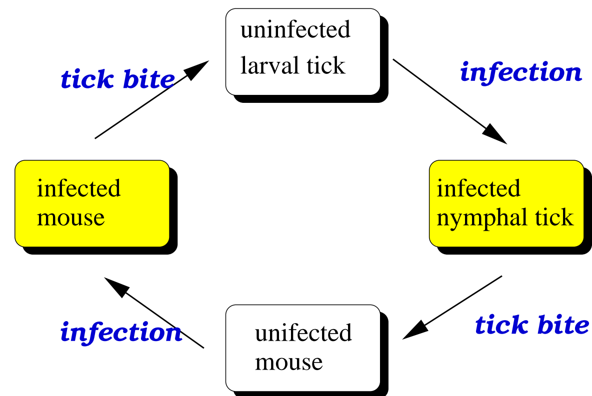

Figure 1: Infection Cycle

2 SIMULATION MODEL

Our goal is to understand the mouse-tick interaction at the lowest level, so we treat mice as individuals. We have chosen to use discrete event simulation (DES), because it lends itself well to individual-based modeling (Deelman, Caraco, and Szymanski 1996). It allows one to model the behavior of an individual through events that comprise the individual's life history. DES is also appropriate because of the temporal and spatial aspects of the physical system. We might have an area where there are no ticks or mice, and therefore no events. Time-step simulation methods might unnecessarily have to check the "empty" areas for activity. Also, DES can progress through simu-

lated time quickly during episodes where the simulation objects are practically inactive, such as during the winter months.

Mice, as mentioned, are treated as individuals. We discretize the space, which results in a two-dimensional lattice, wrapped in both directions. The node size is assumed to be the size of a mouse home range ($400\mathrm{m}^2$). Ticks and mice can be present at any node of the lattice. Ticks, because of their sheer number (as many as 1200 larvae/$400\mathrm{m}^2$, Ostfeld *et al.* 1996), are viewed as background. Ticks are immobile. Mice move as a result of the occurrence of a *Move Event*. Other events associated with mice are:

1. *Disperse Event*, the mouse starts looking for a nesting site. This event triggers a new *Move Event*.

2. *Kill Event*, the mouse is removed from the simulation.

3. *Tick Bite*, when a mouse is in an area occupied by ticks, the mouse will be bitten by a tick with a predefined probability. The type of bite (larval or nymphal) depends on the type of ticks present in the area. When this event occurs, a number of ticks are removed from the location and placed on the mouse. This event creates the event *Tick Drop*, since after a blood meal, a tick drops off. If the mouse moves out of the area before the ticks drop off, the ticks will be deposited at the new location during the *Tick Drop* event.

4. *Tick Drop*, the ticks that are present on the mouse drop off. We assume that ticks molt just before the drop. Larvae that fed on the mouse drop off as nymphs. If the type of ticks on the mouse were nymphs, they drop off as adults. Also, a new *Tick Bite* of the same type (larval or nymphal) is triggered.

When a mouse starts moving in a certain direction, it will continue moving in the same direction until it settles at a nesting site. With each move, the mouse's survival probability diminishes. If a mouse cannot find an empty site within a certain number of steps, it dies. A mouse can also die of other causes, which are modeled by the random selection of the life span for each individual. Figure 2 shows a diagram of possible events associated with a mouse.

Although ticks are not modeled as individuals, their densities are updated at the time mouse events occur. It is possible that there are ticks at the lattice nodes where no mice are present. In this case the update of densities is done periodically. For simplicity, we update these densities during the *fossil collection*

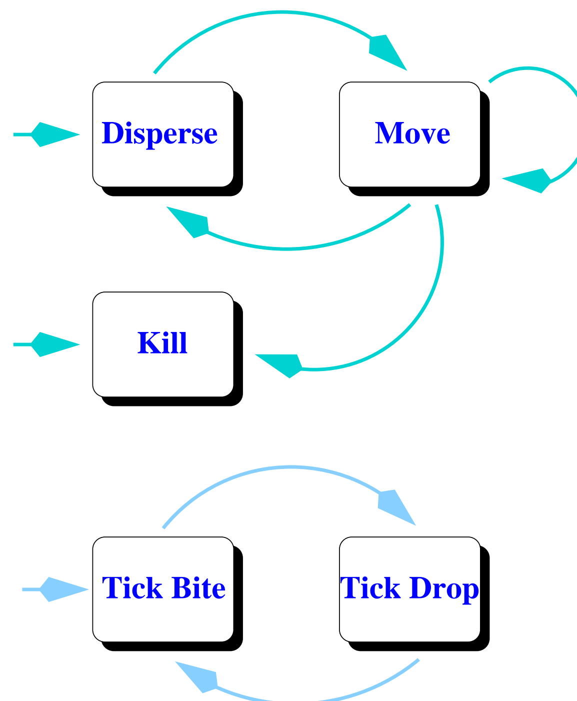

Figure 2: Possible Events for a Mouse

described below. We do not count all individual tick bites, since studies show that there can be as many as five individual larval bites per day. Accordingly, we combine multiple bites into one: ten larval bites or five nymphal bites at one time.

At the beginning of the simulation we have only nymphs that have over-wintered. They are then questing nymphs. At about the 90th day eggs hatch, larval ticks enter the simulation, and the number and type of ticks at each spatial node are updated. When a mouse is bitten by ticks, the number of ticks at the lattice node where the mouse is located is decreased by the number of ticks that bit the mouse. When the ticks drop off the animal, the tick densities at the lattice node are increased. We also make the assumption that when the mouse dies, the ticks (if any) present on the mouse die as well. We assume that mice are bitten by ticks as long as there are enough questing ticks on the node of the lattice. This assumption implies that there is a threshold for both larval and nymphal ticks below which we do not "notice" any new bites.

3 IMPLEMENTATION ISSUES

The model is implemented on a parallel platform, the IBM SP2, a MIMD machine (below, we present the results of runs on up to 16 processors). The implementation is written in C++ to take full advantage

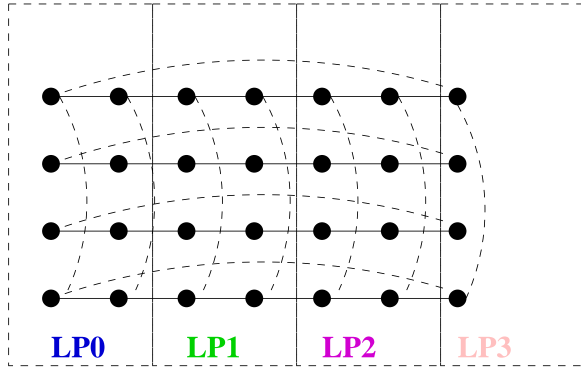

Figure 3: Strip Decomposition for 4 *LP*s

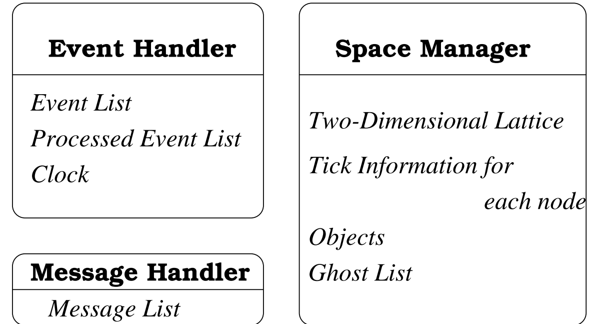

Figure 4: Components of an *LP*

of the object-oriented nature of DES. In Parallel Discrete Event Simulation (PDES), the physical system is divided into several physical processes. The behavior of each physical process is simulated by a Logical Process *LP* (Fujimoto 1990). In general, an *LP* consists of a state, event queue, and clock. The simulation progresses as events are removed from the queue and are processed. The event causes a state change, and the simulation clock advances. The MPI message passing library (Gropp, Lusk, and Skjellum 1994) is used for communication between *LP*s. We use a strip decomposition in the dominant direction to divide the space (Figure 3). The number of strips is equal to the number of processors. We assign one *LP* per strip. We chose this decomposition as the starting point for the system. As the research progresses, we will add other spatial decompositions.

Each *LP* is composed of three logical components: the *Event Scheduler*, which is responsible for processing events, the *Message Handler*, which carries out the communication between *LP*s, and the *Space Manager*, which oversees the movement of objects on the lattice (Figure 4). The *Space Manager* indicates to the objects if the space is occupied or not. Based on this information, the individual makes a decision whether to continue to move or to stay and, in the case of a mouse, to nest. The *Space Manager* also checks if the object is moving outside of the area assigned to the *LP*. If it does, control is given to the *Message Handler*, which sends the object, the *Move Event*, and all the future events associated with the object to the appropriate *LP*.

3.1 Optimistic Protocol Aspects

The major challenge in PDES is to make sure that causality between events is preserved. There are two major approaches to the problem: conservative (Chandy and Misra 1979) and optimistic (Jefferson 1985). In the conservative approach, causality errors are avoided by making sure that no *LP* processes an event for time t unless all events in the system with a timestamp less than t are already processed. In the optimistic approach, causality errors are allowed to occur. When an event with timestamp t_x is received by an *LP* with a clock of $t_c > t_x$, the *LP* has to restore its state to that just prior to time t_x and restart the computation from there. An event causing a rollback is known as a *straggler*. Obviously, the state of an *LP* has to be saved frequently. The more frequently the state is saved, the more memory is used; however, if the state is saved infrequently, the cost (in terms of time) of the rollback grows.

Our parallel simulation uses the optimistic protocol. Since the state of an individual *LP* consists of several lattice nodes (each with a number of individuals) and "background" information, it is prohibitively expensive in this application to save the state of the *LP* after each event, especially so because a single event affects at most only two lattice nodes (the *Move Event*). Therefore, we are using incremental state saving, described by Steinman (Steinman 1993). Each simulation event is augmented with information about the subset of state variables it has changed.

When an event is processed, we place it on the *processed list.* If the event causes a message to be sent to another *LP*, as in the case of a *Move Event* between two lattice strips, we place the message on the *message list.* We use aggressive cancellation; therefore, when a rollback occurs, *antimessages* are sent immediately to minimize the progress of the erroneous computation on other *LP*s. *Antimessages* are used to cancel the original messages that were sent between the logical time to which we have to roll back and the current logical time. The set of such messages is easily determined by scanning the *message list.* Next, we remove events from the *processed list* and "undo" them. When an event is undone, the data which were

changed during its processing are restored. This is possible because each event is augmented with this information.

Processing a *Tick Bite* event involves calculating the number and type of ticks that will bite the mouse based on the number and type of ticks present at the lattice node where the mouse resides. We "place" that number and type of ticks on the mouse. If there are some infected ticks present in the biting sample, the mouse, if not already infected, will become infected with the spirochete. In order to be able to "undo" the bite, the *Tick Bite* event "remembers" the infection status of the mouse before the bite, as well as the number and type of ticks present at the lattice node. When the *Tick Bite* is undone, the ticks present on the mouse are returned to the lattice node in their original infection status. If the mouse was originally uninfected, it will be returned to a healthy state.

If a *Move Event* which involved a move from one *LP* to another is undone, we have to put back the object at the old location and place all the events that were sent out with it on the event list. This is done with the support of the *ghost list*. When an event causes the object to be removed from an *LP*'s space, for example, due to a move or death, the object and all its future events are placed on the *ghost list*. When the events are undone, it is very easy to restore the object and its events from that list. The original move to a new *LP* is cancelled during the *antimessage* phase previously mentioned.

3.2 Global Virtual Time

A very important part of optimistic protocols is the *global virtual time* (*gvt*) calculation. The *gvt* is the minimum time of all the *local virtual times* (*lvt*) of all the *LP*s and of all the timestamps of the messages in transit (Jefferson 1985). Since there are no events in the system with a time smaller than the *gvt*, all information that refers to events that happened before the *gvt* can be removed from memory. This property is very important because optimistic simulations use large amounts of memory to save state information necessary to support rollback. When reclaiming memory (also called *fossil collecting*), we can remove "old" events from the *processed list*, "old" messages from the *message list*, and "old" objects and their events from the *ghost list*.

The challenge of the *gvt* calculation is to capture information about the messages in transit. In our system, the simulation messages are augmented with information that the sender has about all the *LP*s. This knowledge includes not only the known *lvt*s of all *LP*s, but also basic information about messages believed to be still in transit. We have designed a new *gvt* algorithm for our simulation system. It not only calculates the *gvt* but also gives us a good view of how the simulation progresses throughout the entire system. Our *Continuously Monitoring Global Virtual Time* (*CMGVT*) algorithm relies on two basic properties of our system: messages are received in the order in which they were sent, and each simulation message has a serial number. The first property holds in most parallel systems. The second only requires that an extra tag be attached to each message.

Each message and *antimessage* can be simply represented with four quantities: sender id, receiver id, serial number and timestamp. We call this representation of a message a *forcing*, since in optimistic simulations, a message might "force" a process to roll back. To indicate which messages are in transit, we combine two data structures: one quantitative—the *Message Matrix* (MM), which holds the knowledge an *LP* has about the messages sent by itself and its neighbors plus that *LP*'s knowledge of other processes' knowledge of messages sent by itself and its neighbors, and the second qualitative–the *Table of Forcing Vectors* (TFV), which indicates which messages are still unacknowledged. The messages in the table are described in terms of *forcings*. A sample MM and TFV are presented in Eq.(1) and Eq.(2) respectively. The data structures represent those of LP_1, and the system consists of four *LP*s.

$$MM_1 = \begin{bmatrix} 0 & 0 & 0 \\ 6 & 0 & 2 \\ 3 & 0 & 2 \\ 2 & 0 & 1 \end{bmatrix} \qquad (1)$$

The columns of the MM refer to LP_1 (column 0, counting from 0) and its neighbors (here in the strip decomposition, each *LP* has two neighbors): LP_0 (column 1) and LP_2 (last column). The rows of the MM represent the knowledge each of the four *LP*s has about the number of messages LP_1 and LP_1's neighbors have sent. For example, the entry (1,0)=6 in Eq.(1) shows that LP_1 knows that LP_1 has sent out 6 messages. Entry (1,2)=2 shows that LP_1 knows that LP_2 has sent out 2 messages. The matrix also contains the knowledge other *LP*s have. Entry (2,0)=3 shows that LP_2 knows of 3 messages sent by LP_1. Since row 0 is 0, LP_1 does know anything about LP_0's knowledge. Also, because column 1 is 0, LP_1 does not know of any other *LP* that has any knowledge of LP_0's messages.

$$TFV_1 = \begin{bmatrix} 0 & [F(5,1,4), F(7,1,5), F(10,2,1)] \\ 15 & \\ 20 & [F(11,1,2)] \\ 36 & [F(10,1,3)] \end{bmatrix} \quad (2)$$

The TFV is used to keep track of all the messages still unacknowledged as well as the latest known *lvt* of all the *LP*s in the system. The table is indexed by the *LP* id. The first element in each row is the *lvt*. In Eq.(2) LP_1's view of the system is as follows: LP_1 is at time 15, LP_2 at 20, LP_3 at 36, and there is no knowledge about LP_0. The second entry in the table is the vector of forcings. When a message is sent to LP_x, a forcing $F(t,s,c)$, where t is the timestamp of the message, s is the sender's id, and c is the serial number of the message, is entered in row x. From Eq.(2) we see that LP_1 has sent two messages to LP_0: one at time 5 with serial number 4 and the other at time 7 with serial number 5. To LP_1's knowledge there is also a message sent by LP_2 to LP_0 in the system ($F(10,2,1)$). From Eq.(2) also notice that the first message sent by LP_1 ($F(t,1,1)$) was acknowledged, since it is not present in the TFV_1.

The *gvt* is easily calculated just by looking at the TFV. The *gvt* is simply the minimum of the *lvt*s present in the table and of the timestamps present in the forcings. In the above example, the *gvt* is 0 since there is no information about the progress of LP_0.

When a message is sent, the MM and TFV are appended. When a message is received, the incoming information is compared with local data and the receiver updates its own information. The sender might, for example, indicate that an LP (LP_x) is aware of m messages sent by LP_y. If the receiver has a forcing for a message n sent by LP_y to LP_x, and $n \leq m$, then that forcing can be deleted, since that message is indirectly acknowledged by LP_x. Details of the algorithm and the proof of correctness are presented in Deelman and Szymanski 1996. Similar algorithms have been developed previously for systems in which logical time is monotonically increasing (Raynal 1996). We extended the idea of a matrix clock to be able to account for the time going backward as well as forward.

The $CMGVT$ distinguishes itself from other *gvt* algorithms (Steinman *et al.* 1995) since it does not require special synchronization rounds in order to calculate the *gvt*. The $CMGVT$ is computed by each process based on the information currently available to it. The *gvt* can be calculated when needed, for example, when an *LP* is about to run out of memory, or at some predetermined intervals. Here, we update the *gvt* each time the difference between the *lvt* and the *gvt* goes beyond a threshold value GVT_DIS which is an adjustable parameter of the simulation. Obviously, as in most *gvt* algorithms, we calculate only an estimate of the *gvt*, since, to get an actual value, the simulation would have to be suspended. It is interesting to note that each *LP* can have a different estimate of the *gvt*, based on the information it has received from other *LP*s. A problem might arise when some *LP*s do not communicate often enough. In this case, if one *LP* has not received any new information from another, it just queries the noncommunicating *LP*. Even though at first glance it seems expensive to send additional information required by this protocol along with the simulation messages, we are able to achieve good results.

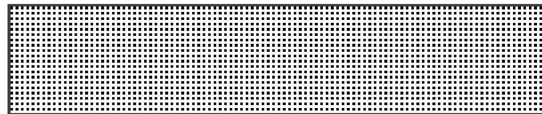

Figure 5: Mice on Day 1

4 RESULTS

Initially we divide the space into as many sections as we have processors. We calculate a new *gvt* when the simulation gets ahead of the previous *gvt* by 10 days. For all the simulations we have the following initial conditions: mice are placed on every other node of the lattice (Figure 5) and ticks are positioned in a 20-node wide "band" in which every fourth row is populated with ticks, 15% of which are infected with the spirochete. Figure 6 shows the infected ticks at the beginning of the simulation. These initial conditions are interesting, because we can see the spread of the diseased ticks to areas previously devoid of them. The spread is caused by mice carrying diseased ticks to new locations and infected mice infecting uninfected ticks.

Figure 7 shows the progress of the disease on day 86 of the simulation. The initial "lines" of ticks are becoming "blurry" as the mice pick up infected ticks and move them to new locations. Figures 8 and 9 show the final configuration of mice and ticks, respectively. The data points at the top of Figure 9 represent infected ticks that the mice carried from the lower part of the figure. This is possible because the lattice wraps around its edges.

We are able to achieve good speedup for small data sets: 2,400 lattice nodes with 800 mice initially. The

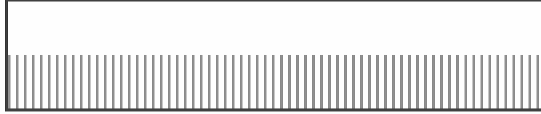

Figure 6: Infected Ticks on Day 1

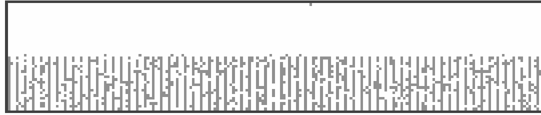

Figure 7: Infected Ticks on Day 86

results are shown in Figure 10. The speedup grows with the number of *LP*s for up to 10 processors. With 12 processors the communication overhead becomes large, decreasing the overall performance.

When the lattice size is increased to 32,000 nodes and 8,000 mice, with the same distribution of mice and ticks, the speedups are less impressive (Figure 11). This is caused by rollbacks whose cost is proportional to the size of the lattice for which each *LP* is responsible. With 4 processors, the lattice size per *LP* is large—8,000 nodes. When a rollback occurs, all the events that happened in the affected time in all of the 8,000 nodes have to be rolled back.

We investigate several methods of reducing the possibility and cost of rollbacks.

First, we increase the number of strips into which the lattice is divided, thus decreasing the area assigned to each *LP*. The processes are mapped by the job scheduler of the IBM SP2; therefore, a process is not aware if processes with which it exchanges data are run on the same processor as it is running. Hence, it always calls interprocess communication for such exchanges, slowing the execution. The results of dividing the problem into as many as 20 *LP*s on up to 16 processors are presented in Table 1. The best times for 4 processors are achieved when each processor has 5 *LP*s. Still, there is no speedup (the sequential time is 227 sec.). The speedup with 8 processors and 16 *LP*s is very small, around 1.6. The speedup with 12 processors is best when 20 *LP*s per processor are used,

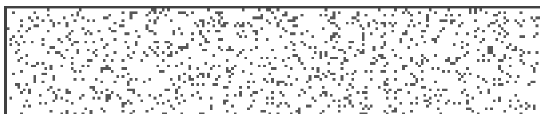

Figure 8: Mice on Day 180

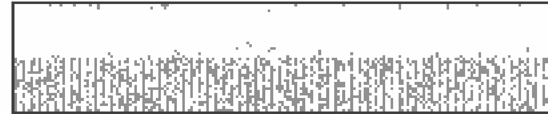

Figure 9: Infected Ticks on Day 180

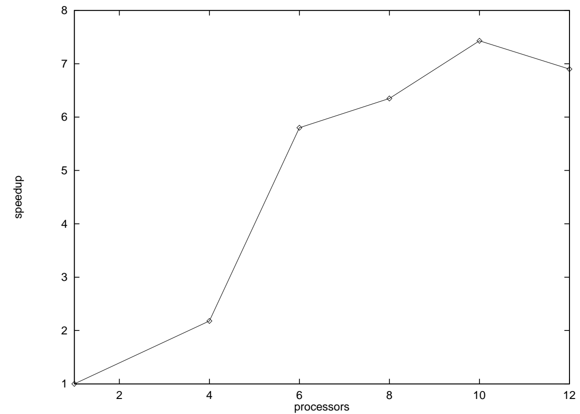

Figure 10: Speedup for Small Data Set

and is equal to 2.2. With 16 processors the speedup improves slightly to 2.8.

Another way to decrease the impact of rollbacks is to curb the optimism by allowing each *LP* to process events only within a limited time into the future. Toward this end, a process is allowed to advance only by 20 or 30 days ahead of the average *lvt* of others. Each process knows this value from the information sent by our *CMGVT* protocol. The results are shown in Table 2. The performance of this method was better than in the previous case only for 16 processors, with the best speedup of 3.7 for the 30-day time cap.

We also tried to combine both the use of multi-

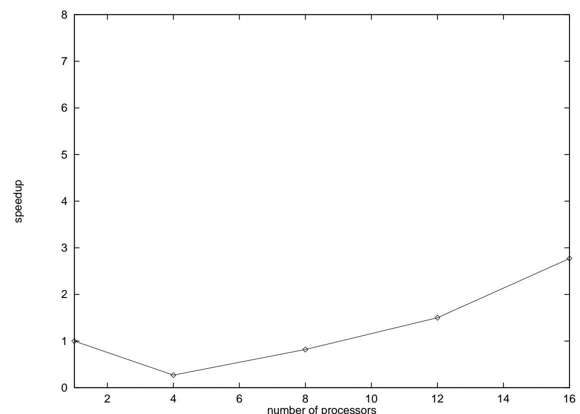

Figure 11: Speedup for Large Data Set

Table 1: Runtime in Seconds for Multiple LPs per Processor

Processors	Number of LPs 8	12	16	20
4	502.8	510.07	289.96	226.85
8	275.76	280.4	139.7	249.29
12	-	149.42	116	103.7
16	-	-	82	98.16

Table 2: Curbing the Optimism (time in sec.)

Processors	20 Days	30 Days
4	769.27	936
8	177.46	224
12	122	128.48
16	> 200	61.79

ple LPs per processor and curbing the optimism to processing only up to 30 days ahead of the average *lvt*. The results are shown in Table 3. The combined method resulted in an overall improvement over each of the component methods with the most significant improvement for 12 processors.

Table 3: Multiple LPs and Curbed Optimism (time in sec.)

Processors	Number of LPs 8	12	16	20
4	492.98	379.37	359.6	242.55
8	224.0	215.23	118.17	188.7
12	-	128.48	88.81	100.27
16	-	-	61.79	74.49

We checked if we were indeed reducing the number of rollbacks by using multiple LPs and reducing optimism. The dramatic results of a typical run on 8 processors are shown in Figure 12. With the increase of the number of LPs, the average number of rollbacks per LP decreases significantly. However, the number of rollbacks is not the only measure of performance. For example, the average number of rollbacks for 20 LPs is smaller than for 16 LPs , but the runtime is higher. This is because an increase in the number of LPs per processor intensifies the contention for the CPU, which slows the entire simulation. We have also noticed that the average number of rollbacks decreased as we curbed the optimism of the simulation. For 8 processors, 8 LPs, and an optimism cap of 30

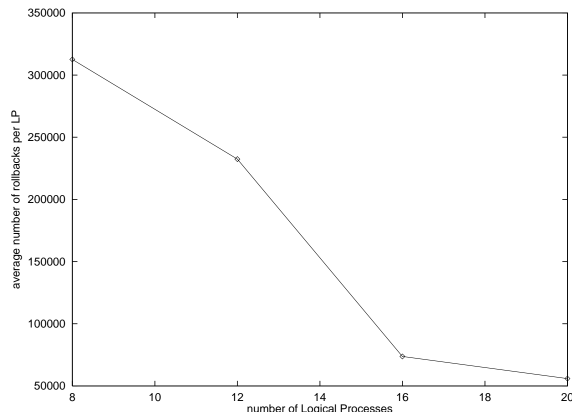

Figure 12: Average Number of Rollbacks for 8 Processors

days, we had on the average 136,355 rollbacks per LP; for a 20 day cap, the average was 92,770.

We investigated if the size of the messages had an adverse effect on the performance of the simulation. Our messages have the overhead of adding the *gvt* information. The size of the *Message Matrix* is $C_m p$, where C_m is the connectivity of an LP (up to 8 for spatially explicit two-dimensional problems), and p is the number of LPs. The size of the *Table of Forcing Vectors* is $c_t p$, where c_t is the maximum size of a *Forcing Vector* and is set by the system (the results presented here use $c_t = 10$). Thus, the additional message size is $C_m p + c_t p = O(p)$.

To see how the message size affects the performance of the simulation, we have increased the messages by an additional $.5(C_m + c_t)p$. The sender packs the message buffer up to its maximum size, but the receiver just reads out the original information. The results are shown in Figure 13. Increasing the message size had an unexpected effect: it actually improved the performance of the simulation! It reduced the number of rollbacks and the number of messages sent between LPs. This improvement might be caused by slowing down LPs that send many messages, or slowing down LPs that get ahead of others, thus allowing the slower LPs to catch up. These results encourage us to investigate methods for controlling the progress of the simulation in order to achieve better performance.

ACKNOWLEDGMENTS

This work was supported by the National Science Foundation under Grant BIR-9320264. 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.

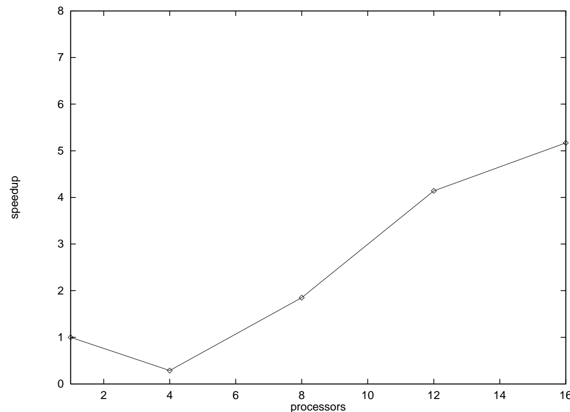

Figure 13: Speedup for Large Messages

REFERENCES

Barbour, A. and D. Fish. 1993. The biological and social phenomenon of Lyme disease. *Science* 260:1610-1616.

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

Deelman, E., T. Caraco and B. K. Szymanski. 1996. Parallel Discrete Event Simulation of Lyme Disease. In *Pacific Symposium on Biocomputing* 191-202.

Deelman, E. and B. K. Szymanski. 1996 Continuously Monitored Global Virtual Time. *Department of Computer Science Technical Report 96-18*, Rensselaer Polytechnic Institute.

Fujimoto, R. M. 1990. Parallel Discrete Event Simulation. *Communications of the ACM* 33:31-53.

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

Jefferson, D. R. 1985. Virtual Time. *Trans. Prog. Lang. and Syst.* 7:404-425.

Miller, G. L., R. B. Craven, R. E. Bailey and T. F.Tsai. 1990 The epidemiology of Lyme disease in the United States 1987-1998. *Laboratory Medicine* 21:285-289.

Ostfeld, R. S., K. R. Kirsten and O. M.Cepeda. 1996. Temporal and Spatial Dynamics of *Ixodes scapularis* (Acari: Ixodidae) in a Rural Landscape. *Journal of Medical Entomology.* 33:90-95.

Raynal, M., and Mukesh Singhal. 1996. Logical Time: Capturing Causality in Distributed Systems. *IEEE Computer* 49.

Steinman, J. S. 1993. Incremental State Saving in SPEEDES using C++. In *Proceedings of the 1993 Winter Simulation Conference.* 687-696.

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

AUTHOR BIOGRAPHIES

EWA DEELMAN is pursuing her Ph.D. in the Department of Computer Science at Rensselaer Polytechnic Institute. She received a B.A. in Mathematics at Wells College and a M.Sc. in Computer Science at SUNY New Paltz. Her research interests include designing algorithms for parallel scientific computation and building tools and compilers for parallel platforms. Her Ph.D. work focuses on new paradigms for Parallel Discrete Event Simulation and their use in biological applications. email: deelmane@cs.rpi.edu, http://www.cs.rpi.edu/∼deelmane

BOLESLAW K. SZYMANSKI is a Professor of Computer Science and a cofounder of the Scientific Computing Research Center at Rensselaer Polytechnic Institute, Troy, NY. He received a Ph.D. in Computer Science from the National Academy of Science in Warsaw, Poland, in 1976 and was a post-doctoral fellow at Aberdeen University in Aberdeen, UK. His research interests include language and compiler issues in large scale computing systems; analysis, design and verification of distributed and parallel algorithms; and simulation and modeling of computer and ecological systems. email: szymansk@cs.rpi.edu, http://www.cs.rpi.edu/∼szymansk

THOMAS CARACO is an Associate Professor of Biological Sciences at the University at Albany, State University of New York. His research interests include the effects of behavior on population dynamics, and the landscape ecology of epidemics. Dr. Caraco is an editor of *Evolutionary Ecology* and an advisory editor for *Behavioral Ecology and Sociobiology.*