Proc. 12th Workshop on Parallel and Distributed Simulation - PADS98, Calgary, Canada, June 1998,
IEEE Computer Society Press, Los Alamitos, CA, pp. 46-53

Dynamic Load Balancing in Parallel Discrete Event Simulation for Spatially Explicit Problems

Ewa Deelman*

Computer Science Department
University of California at Los Angeles
Los Angeles, CA 90095

Boleslaw K. Szymanski

Computer Science Department
Rensselaer Polytechnic Institute
Troy, NY 12180

Abstract

In this paper we present a dynamic load balancing algorithm for Parallel Discrete Event Simulation of spatially explicit problems. In our simulations the space is discretized and divided into subareas each of which is simulated by a Logical Process (LP). Load predictions are done based on the future events that are scheduled for a given LP. The information about the load of the processes is gathered and distributed during the Global Virtual Time calculation. Each LP calculates the new load distribution of the system. The load is then balanced by moving spatial data between neighboring LPs in one round of communications. In our problems, the LPs can be described as being elements of a ring from the point of view of communication. Due to the spatial characteristics, the load can be migrated only between neighboring LPs. We present an algorithm that performs the load balancing in a ring and minimizes the maximum after-balance load.

1 Introduction

Much work in load balancing for Parallel Discrete Event Simulation [5] has been based on the many-to-one process-to-processor model; therefore, process migration algorithms have been the most frequently researched. Both static [15, 16] and dynamic algorithms have been designed.

The major difference between load balancing for general parallel problems and PDES systems is the measure of a processor's load. In general problems, CPU utilization is a good measure of how much load a given processor has. In PDES, this measure is meaningless. If, for simplicity, there is one LP per processor, it is possible that a processor has a high CPU utilization but that it is continuously processing forward ahead of the others and then rolling back, thus performing no useful work. A metric often used in PDES is the number of committed events (events with timestamps smaller than the Global Virtual Time (GVT)).

In dynamic algorithms, the load is sometimes measured by the simulation advance rate [6] which is defined as the rate at which a process advances its LVT as a function of the CPU time that this process has been allocated. The goal is to make the LPs progress evenly in the virtual time. The slower LPs have proportionally more CPU time dynamically assigned to them than the faster LPs. Process migration has been proposed for cases in which the load imbalance is too large to be accommodated by CPU time slice adjustment. The load balancing is implemented by background processes running on each processor. This approach, however, relies heavily on support from the operating system.

In the Time Warp Operating System (TWOS) [10] each LP can be divided in time into several *phases*. Each phase is responsible for simulating the physical process in a different virtual time interval. When a load imbalance is detected in the system, the phases are migrated between processors. The advantage is that the phase is cheaper to migrate than an entire LP, because the phase has fewer states and messages.

Since LPs are often clustered on processors, techniques have been designed specifically for such systems [1] to enable movements of entire clusters of LPs between processors. The load of a cluster is defined as the number of events that have been processed by all the LPs in that cluster since the last load balancing calculation. The load balancing algorithm matches up the heaviest and lightest processors, and tries to migrate the clusters so that the load of the processors is approximately equal.

2 Spatially Explicit Problems

In the research presented in this paper, a different approach to dynamic load balancing is used. Our sim-

*The work presented in this paper was performed at Rensselaer.

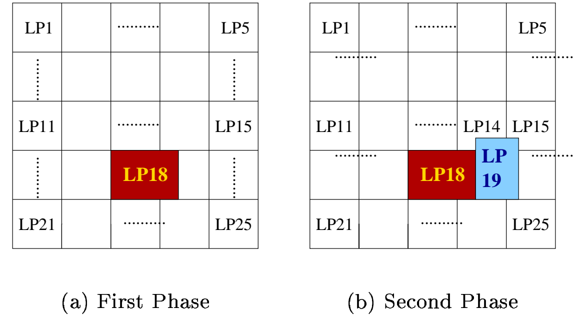
ulation system, based on Time Warp[7], is designed to simulate spatially explicit problems in which a two-dimensional space is discretized into a lattice [14]. The lattice is then partitioned and distributed among LPs. Objects present on the lattice can move around freely. Spatial problem characteristics dictate problem partitioning, processor load definition and load migration.

The simulation consists of mobile objects and events, which affect the objects and the lattice nodes. Local events affect only the state of a lattice node, while non-local events, such as Move Events, which move objects from one location to the next, affect at least two nodes of the lattice.

State saving is done incrementally [11]. When an event is processed, it is placed on a *processed event list* [3]. If the event being processed moves an object out of an area assigned to a given LP, this object and all its future events are sent to the appropriate LP. A copy of the object and the events is also placed on a *ghost list*. When a rollback occurs, events are removed from the processed list and undone. Objects that have been sent out in the time affected by the rollback are restored from the ghost list.

In our simulations, each partition of the lattice is simulated by an LP. Future information, such as the future event queue, is kept globally by the partition, whereas the past information, such as the processed event list, is kept at the corresponding lattice nodes. We have designed and implemented a method, termed Breadth First Rollback (BFR) [4]. It uses incremental state saving, to the recovery of causal relationships between events during rollback. These relationships are then used to determine which events need to be rolled back. We have found that we can obtain a nearly linear speedup with this method, when it is applied to an evenly balanced problem and a one-to-one LP-to-processor assignment [4].

Problem Partitioning Assuming the one-to-one process to processor mapping and consequently using data migration as a means of balancing the load, the issue of the partition shape for each LP remains. We have considered two standard partition shapes: rectangular and strip. The advantage of the first partitioning is the relatively small communication boundary between partitions. If the lattice is divided in both dimensions into rectangles–the size of the communicating edges is $4n/p$, where p is the number of processors and assuming an n^2 lattice. This partitioning, however, does not lend itself well to dynamic data load balancing, because the movement of space boundaries will cause irregular shapes—Figure 1(a). If the load balancing is performed by resizing the space for which a given LP is responsible, the following situation can occur: In Figure 1(a), LP_{18} acquired additional space as a result of the load migration. Therefore, if LP_{19} (see Figure 1(b)) needs to resize, it can only enlarge either to the right or to the left to maintain the partitioning shapes. If LP_{19} wants to grow in the up or down directions, the partitioning becomes irregular (LP_{14} and LP_{15} in Figure 1(b)).



(a) First Phase (b) Second Phase

Figure 1: Load Balancing with Rectangular Partitioning.

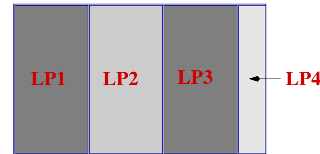


Figure 2: Strip Partitioning.

To simplify the partition shapes, the dimension in which the partition can resize is fixed. This results in a strip partitioning, where space is divided into strips in the dominant direction (Figure 2). During load balancing the strips can be easily resized. Although the size of the communicating boundaries is larger ($2n$), each LP needs to communicate with only two others. Since the space is toroidally wrapped, the LPs communicate with each other in a ring.

Load Metrics In this paper, we discuss the method that requires knowledge of loads of all the processes (which can be achieved either by load broadcast or centralized load gathering) and assumes the nearest-neighbor load migration—necessitated by the spatial characteristics. There are several metrics that can be used to determine the load of that LP. One possibility is to count the number of objects in the space assigned to a given LP. The advantage of this method is that this count can be easily obtained. However,

this metric may not give an accurate representation of the load. If, for example, there are many objects in a given space, but there are no events scheduled for these objects, then the LP will not have much work. It is also possible that an LP could contain few but "very active" objects, and it will be busy. A better approach would be to count the number of events that are scheduled for a given LP. In such case, the distribution of events in time would also be of importance. It is possible, for example, that an LP (say LP_1) has few events scheduled for the immediate future but many events scheduled for the far future; another LP (let's call it LP_2) has many events to process soon. Even if the event count is equal for the two LPs, the workload is not equal, because LP_1 will process the events far into the future and incur rollbacks, thus performing "useless" work. Meanwhile, LP_2 will be simulating immediate events. It would be more profitable to distribute the events in such a way that both LPs will work on the immediate future. In the research presented here, the future events are counted in order to determine the computational load of an LP. To differentiate between events that are scheduled to happen at different times, the events are weighted according to the distance in time of the event from the beginning of the simulation. The great advantage of counting the weights of future events instead of counting processed events, as in previously described load balancing algorithms, is that this algorithm bases its load estimate on the future of the LP rather than on its past performance. Doing so is valuable when dealing with "hot spots", parts of the simulation space in which many events are executed rapidly at some point of the simulation. In the previous approaches, the LP would have to be in the midst of the "hot spot" computation before the additional load would be discovered. However, since new events are constantly created, the future events represent only an estimate of the future load of an LP. The algorithm cannot, for example, judge if a given "hot spot" will be persistent and worthy of rebalancing or if it will dissipate quickly.

Load Migration To reduce communication overhead, it is necessary to move lattice nodes along with the objects that are present in them and together with the events scheduled for these objects and lattice nodes. An overloaded LP can "shed" layers of space in order to balance the load.

For the remainder of this paper, it is assumed that the spatial lattice of the problem is strip-divided in the horizontal dimension. Thus, the LP can send lattice columns to another LP (see Figure 3) without compromising the shape of the partition. The columns have to be moved in an organized fashion—from the outer edge toward the inside. Moving columns in any other way would result in a fragmented space and would increase the communication overhead. Although the communication cost is not explicitly included in the load calculation, it is implicit in the type of load migration (space, objects, and events versus only object migration) and in the type of space which is allowed to be moved (only contiguous space layers). We have integrated the load balancing algorithm with the centralized GVT algorithm used in SPEEDES [12]. Hence, our implementation gathers the load information together with the Local Virtual Time and the message counts of the LPs, and then distributes this information back to the processes.

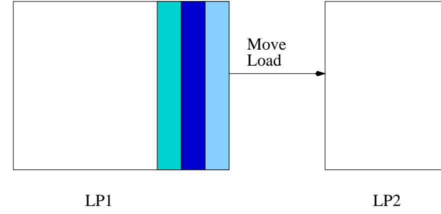


Figure 3: Moving Load between Logical Processes.

3 Dynamic Load Balancing

Each LP keeps track of the events occurring in each column of lattice nodes assigned to it (Figure 4). When an event is created, the cost of the event (defined below) is added to the load buffer with the index of the column where the event is scheduled to occur. When the event is processed, sent out, or canceled, the cost of the event is subtracted from the load buffer. The load at each LP_i is calculated as follows:

$$load_i = \sum_{L \leq j \leq R} E_j$$

where, j is the column number, L, R are the space boundaries,

$$E_j = \sum_{0 \leq k < n(j)} 2^{-time(e_j)}$$

where, $n(j)$ is the number of events in column j, e_j is the event associated with column j, and $time(e_j)$ is the event's occurrence time.

Each LP calculates its own load at the time of the GVT calculation. This information is sent along with the message counts during the GVT phase. When the GVT is calculated, the LP which initiated and concluded the calculation broadcasts the new GVT along with the loads of all the LPs in the system.

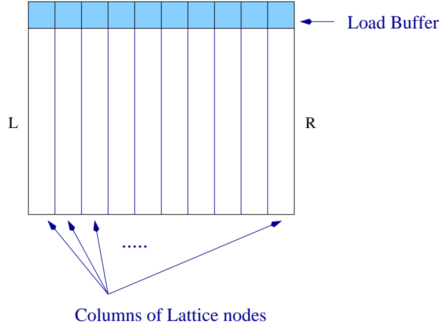


Figure 4: Counting the Computational Load of a Logical Process.

Calculation of the load distribution Each LP executes exactly the same algorithm. In order to minimize load balance overhead, only one round of communication is performed. Nearest-neighbor communications are also enforced to keep the space assigned to a given LP contiguous. Since the space is strip-divided, the topology of the LPs is a ring. The optimal post-balance load can be achieved by finding and balancing the dominant ("heaviest" in terms of load) subchain in the ring and then recursively applying this procedure to the rest of the ring. An exception is the case in which the dominant subchain is equal to the entire chain, in which case the average load balance can be achieved at each LP. The details of the algorithm are described below.

Consider a ring of n LPs: $p_0, p_1, \ldots, p_{n-1}$. Each LP p_i in the ring has the pre-balance load $a_i \geq 0$ and post-balance load b_i and each process sends data to at most two neighbors and receives data from at most two neighbors in one load balancing phase. Let x_i denote data exchanged between p_i and $p_{i\oplus 1}$ with the convention that $x_i > 0$ means that process p_i sends data and $x_i < 0$ signifies that process p_i receives data[1]. Accordingly, $-a_{i\oplus 1} \leq x_i \leq a_i$ and $b_i = a_i - x_i + x_{i\ominus 1}$.

Let $\overline{a} = \sum_{i=0}^{n-1} a_i/n = \sum_{i=0}^{n-1} b_i/n$ be the average load of the processes in the ring. The quality of the load balance can be measured by the load of the most heavily loaded processes whose progress in the simulation will be slowest. Hence, the dynamic load balance problem for PDES can be formulated as the following min-max problem:

Min-Max Problem: Given a ring of n processes, with load $a_i \geq 0$, for each pair of processes $p_i, p_{i\oplus 1}$ find such a load transfer x_i that minimizes:

$$\max_{i=1}^{n-1}(a_i - x_i + x_{i\ominus 1}),$$

[1] In the following, operations $\oplus$ and $\ominus$ denote integer addition and subtraction in modulo n arithmetic.

under the constraints:

$$(\forall_{0\leq i<n} : -a_{i\oplus 1} \leq x_i \leq a_i).$$

This problem can be solved by the $O(n^2)$ algorithm presented below [13].

Let b_{opt} denote the optimum value of the maximum, post-balance load for the ring ($b_{opt} = \max_{i=0}^{n-1} b_i$). A *chain* $c_{k,l}$ in the ring is a sequence of processes $p_k, p_{k\oplus 1}, \ldots, p_{k\oplus(l-1)}$ starting at process p_k and of length l. A chain $c_{k,l}$ is a subchain of a chain $c_{K,L}$ iff for some i, $k = K \oplus i$, $0 \leq i < L$, and $l \leq L - i$.

Definition: A load $s_{k,l}$ of a chain $c_{k,l}$, where $2 < l \leq n$ is defined as $s_{k,l} = \sum_{i=k\oplus 1}^{k\oplus(l-2)} a_i/l$.

Let $b_{max} = \max_{0\leq k<n, 2<l\leq n} s_{k,l}$. A chain $c_{k,l}$ is called maximal iff $s_{k,l} = b_{max}$ and *dominant* if it is the longest maximal chain.

Note that the load that can be transferred out of a chain, denoted here by $o_{k,l}$, is located at its end-processes (p_k and $p_{k\oplus(l-1)}$), so $o_{k,l} \leq a_k + a_{k\oplus l\ominus 1}$. Hence, it is clear that $b_{opt} \geq b_{max}$, because for the dominant chain $c_{k,l}$, $b_{opt} \geq \sum_{j=l}^{k\oplus l\ominus 1} b_j/l = \sum_{j=k}^{k\oplus l\ominus 1} a_j/l - o_{k,l} \geq \sum_{j=k\oplus 1}^{k\oplus l\ominus 2} a_j/l = b_{max}$.

Theorem: $b_{opt} = \max(\overline{a}, b_{max})$.

Below, we prove the above theorem constructively by providing an algorithm that defines valid load transfers to achieve the maximum post-balance load of $\max(\overline{a}, b_{max})$. First, we should notice that there is at most n^2 distinct subchains within a ring of size n, then by simple enumeration we can find b_{max} and the dominant chain of the ring in $O(n^2)$ steps.

By simply renumbering the nodes, if necessary, we can assume without loss of generality, that the dominant chain is $c_{1,ldom}$ with the length $ldom$. If there is imbalance, then the computation of load transfers necessary to improve load balance depends on the relation of average load $\overline{a}$ and the load of the dominant chain.

Case 1: $b_{max} \geq \overline{a}$.

The load transfers are defined as follows:

1. For processes in the dominant chain we set:

   $$x_0 = -a_1;\ x_{ldom} = a_{ldom};\ x_i = a_i - b_{max} + x_{i-1}$$

   $$\text{for } i = 1, \ldots, ldom - 1.$$

2. Setting the post-balance load to b_{max} extends iteratively beyond the right end of the chain:

   $$x_i = \max(a_i + x_{i-1} - b_{max}, -a_{i\oplus 1})$$

   while $x_i = b_{max}$, and an index of the last defined load transfer is denoted as re, so by definition $b_{re\oplus 1} = 0$.

3. If $re < n-1$, then another extension is defined iteratively through the chain's left end as:

$$x_i = \min(b_{max} - a_{i\oplus 1} + x_{i\oplus 1}, a_i)$$

$$\text{while } x_i = b_{max} \text{ for } i = n-1, n-2, \ldots .$$

   Similarly, as in the previous case, an index of the last defined load transfer is denoted as le.

   If $le > re + 2$ then the algorithm is applied recursively to the (newly created) ring $p_{re+1}, \ldots, p_{le}$ in which we change the pre-balance loads on processes $re+1$ and le to 0. Otherwise we set $x_{re} = 0$.

To avoid unnecessarily underloaded processes, we can also do a simple adjusting step for processes outside the dominant chains. Each process which sends the load may check if the post-load at the destination will become bigger than its own and then change its transfer to have the post-loads equal. This step may slightly improve the performance and is inexpensive to execute. Note, that this adjusting step preserves the correctness of the solution.

Case 2: $\overline{a} > b_{max}$.

In this case, the transfers are defined as:

$$x_0 = \min_{i=1}^{n-1}(i\overline{a} - \sum_{j=0}^{i-1} a_i);\ \ x_i = a_i + x_{i-1} - \overline{a}$$

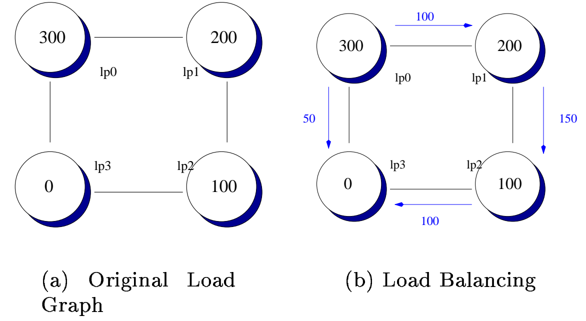
$$\text{for } i = 1, 2, \ldots n-1.$$

Which for all $0 < i < n$ gives $b_i = \overline{a}$.

The above algorithm is executed by all the LPs, so each will know how the load is being distributed. A structure indicating load movement between neighbors is maintained. Since there is only one round of load migration, a few iterations may be required to distribute the load away from a "hot spot". On the other hand, such spots might dissipate by the next load calculation due to the dynamic nature of the simulation.

Some tolerances must be allowed. That is, if the load of all LPs is within some percentage of the mean, then there is no need to perform the load balancing. The percentage of imbalance is a parameter of the simulation. The tolerance also prevents load "thrashing"—endless shifting of load between two almost evenly loaded neighboring processes.

Examples of Load Balancing Only after the necessary load movements are calculated, the sending and receiving of the loads is done. Consider the load distribution as depicted in Figure 5a. The system contains 4 LPs and the figure shows the load at each LP. The ring represents the communication connections between the LPs. During the first phase of the load calculation the loads of the LPs are gathered. The total load is 600, so the average load $\overline{a} = 150$. The dominant chain, $0-300-200-100$, is of length $ldom = 4$, so $b_{max} = (300+200)/4 = 125$. Here, the dominant chain encompasses the entire ring. Since $b_{max} < \overline{a}$, the second case applies. The following transfers are calculated: $x_0 = 100$, $x_1 = a_1 + x_0 - \overline{a} = 0 + 100 - 150 = -50$, $x_2 = a_2 + x_1 - \overline{a} = 300 - 50 - 150 = 100$, and $x_3 = a_3 + x_2 - \overline{a} = 200 + 100 - 150 = 150$. The load transfers are depicted in Figure 5b. After load migration, each LP will have the same amount of load (150).

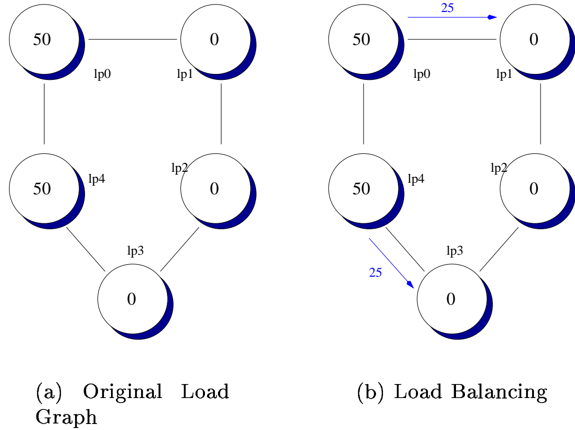


(a) Original Load Graph (b) Load Balancing

Figure 5: Example 1.

Sometimes, it is impossible to perfectly balance the load of a system in one round of communications. Figure 6a shows such a system $(50-0-0-0-50)$, for which $\overline{a} = 100/5 = 20$. The dominant chain starts at index 3 and is of length $ldom = 4$, with $b_{max} = 25$. Since $b_{max} > \overline{a}$, the first case applies. The following flows are calculated (old index 3 is now 1): $x_0 = -a_1 = 0$, $x_4 = a_4 = 0$, $x_1 = a_1 - b_{max} + x_0 = 0-25+0 = -25$, $x_2 = a_2 - b_{max} + x_1 = 50-25-25 = 0$, and $x_3 = a_3 - b_{max} + x_2 = 50 - 25 + 0 = 25$. The flows are shown in Figure 6b. The best load balance that can be achieved in this ring is $25-25-0-25-25$.

Load–Column Mapping So far, the load movement was calculated by the LPs. Now that each LP knows how much load it has to give away, it needs to calculate which columns of space it will actually send. It scans the load buffer of event costs. An error exists with the projection of a one-dimensional structure (the load buffer) onto a scalar (the load that needs to be moved). As a result, the movement of columns will generally not give an exact load balance. The algorithm should err on the side of giving too little load rather than too much.



(a) Original Load Graph

(b) Load Balancing

Figure 6: Example 2.

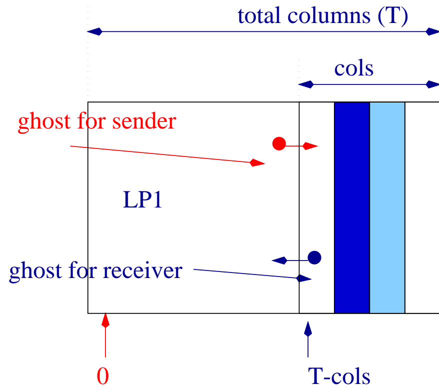


Figure 7: Moving the Load to the Right.

When sending load, columns of lattice nodes will be sent, in addition to all the objects in that space and the events associated with these objects. Since each lattice node contains a list of processed events, these lists are sent, as well. The sending LP will have a new space boundary. Let's assume that LP_1 is sending columns of space to the right, as shown in Figure 7. On the right, the new boundary for LP_1 is $T - cols - 1$, where T is the initial, total number of columns, and $cols$ is the number of columns being sent.

It is possible that there were events already processed that involved the move of an object from column $T - cols - 1$ to column $T - cols$. In that case, a new ghost has to be created and placed on the ghost list, so that when the *Move* event is undone (which will now be a move out of the local space), the object and its future events can be restored. When a ghost is created, the appropriate processed event lists and the future event queue have to be scanned to retrieve future events for the object. When the local ghost list is updated, columns of space are sent to the neighbor. Next, the objects present in that space need to be sent. There are also events in the future event queue that are not associated with an object, but rather with a lattice node. These types of events, which are scheduled to happen in the space being sent, have to be sent, as well. There might also have been some objects which moved across the new boundary between the receiver and the sender (from $T - cols$ to $T - cols - 1$). These objects have to made into ghosts for the receiving LP and then sent to the receiver. Finally, the processed events from that space are sent.

Receiving the space is straightforward: the space objects are received first, then the objects, the future events, the ghosts, and finally the processed events. There is some processing on the receiving side due to the restructuring of the space. For example, the processed events have to be placed at the appropriate lattice nodes, the objects have to be placed in the space, and the future events have to be inserted in the future event queue.

Application The application that motivated this work is the simulation of the spread of the Lyme disease. This disease is prevalent in the Northeastern United States [2, 8]. People can acquire the disease by coming in contact with a tick infected with the spirochete, which may transfer into the human's blood, causing an infection. Since the ticks are practically immobile, the spread of the disease is driven by the ticks' mobile hosts, such as mice and deer. Even though the most visible cases of Lyme disease involve humans, the main infection cycle consists of ticks and mice. We model Lyme disease at the ecological level—at the level of the mouse-tick interactions.

The mice are modeled as individuals, and ticks, because of their sheer number (as many as 1200 larvae/400m^2 [9]) are treated as "background". The space is discretized into nodes of size 20x20m^2, which represent the size of the home range for a mouse. Each node may contain any number of ticks in various stages of development and various infection status. Mice can move around in space in search of empty nesting sites. The initiation of such a search is described by the *Disperse Event*, and the moves by the *Move Event*. Mice can die (*Kill Event*) if they cannot find a nesting site or by other natural causes, such as old age, attacks by predators, and disease. Mice can be bitten by ticks (*Tick Bite*) or have ticks drop off (*Tick Drop*). From the above list of events, only the *Move Event* is non-local.

4 Load Balancing Results and Conclusions

The simulation currently runs on an IBM SP2 (we show results for up to 28 processors). The model was designed in an object oriented fashion and implemented in C++. The communications between processes use the MPI message passing library.

Several load balancing experiments were conducted. For the first two experiments, two adjacent processes were made "heavy" by assigning twice as many objects to the space belonging to these processes as were assigned to the remaining processes. Figure 8 shows the performance of the load balanced computation as compared to the unbalanced computation. A relatively small problem size of approximately 10,000 lattice nodes and 2,000 objects are used, and a tolerance of 30% was applied. For this configuration, the results show no improvement. This is the case in which the progress of each processor is limited not by the small computational load, but by communication which is proportional to the number of events occurring in the columns at the interprocessor boundaries. Shifting columns would still leave the heavy loaded columns at the boundary, just different ones than before load balancing. As a result, attempts to balance the load in such cases just incur overhead. Hence, the problem was increased to 40,000 lattice nodes and approximately 4,000 objects. The results are shown in Figure 9 for a series of tolerances ranging from 10 to 40 percent. For all the levels of tolerance used for this particular problem, the algorithm performs well. Also, for 20 or more processors, the load balanced algorithm outperforms the unbalanced version. It should be pointed out that the load imbalance decreases as the number of processors increases, because the size of the space assigned to each LP decreases (with only two LPs heavily loaded). So, in fact, the load is highly imbalanced in the 16 processor case, and the cost of the load movement exceeds the benefits of an even load distribution.

Figures 10 and 11 illustrate the results of experiments for which the load was kept constant by making 100 (25% of columns for small problem size and 12.5 % for large problem size) of the middle columns of spaces "heavy"—containing twice as many objects as other columns. Figure 10 shows the results for the small problem size. Again, for that problem size, the results are not impressive. It is interesting to note that although the load balancing does better with 8 and 12 processors, it does not perform well with 4 or 16 processors. For 4 processors, the amount of load that has to be moved might be too large, and for 16 processors, the problem size might be too small to overcome the cost of load calculation. With the large problem size (see Figure 11), the load balancing algorithm performs well from 16 to 28 processors.

We have described a dynamic load balancing algorithm which balances the load in a ring of processes with a constraint of one round of communications and nearest-neighbor communications. The algorithm was designed for and applied to a spatially explicit problem. In our implementation, each LP simulates an area of space. The load balancing is based on the future events weighted by the occurrence time, and performed by moving areas of space between neighboring LPs. Our results demonstrate, that the dynamic load balancing algorithm can be useful, but it needs to be applied discriminately. Load balancing is most worthwhile when the problem size is large. Also, given the same level of load imbalance, systems with larger numbers of processors appear to perform better, because more of the load can be moved in parallel.

The algorithm presented for dynamic load balancing generalizes to cases when the communication between neighbors is allowed for a distance $k > 1$. For $k \geq n/2$, it is always the case that $\overline{a} > b_{max}$, so the algorithm is able to achieve a perfect load balance. This version of the algorithm is therefore suitable for the initial (static) load balancing. We are currently implementing such a static technique for our application.

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.

References

[1] H. Avril and C. Tropper. The Dynamic Load Balancing of Clustered Time Warp for Logic Simulations. *Workshop on Parallel and Distributed Simulation*, pages 20–27, 1996.

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

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

[4] E. Deelman and B. K. Szymanski. Breadth-First Rollback in Spatially Explicit Simulations.

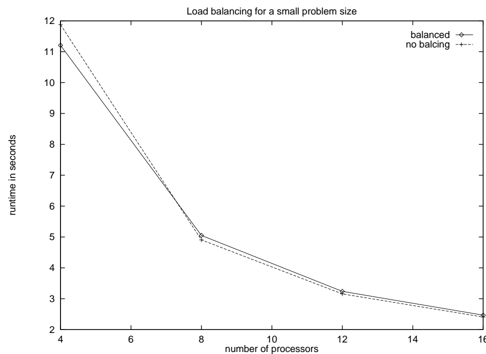


Figure 8: Small Problem and Two Heavy Processes.

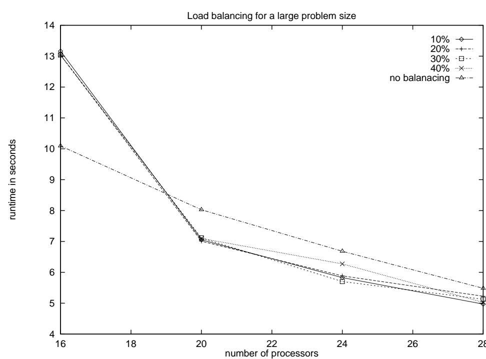


Figure 9: Large Problem and Two Heavy Processes.

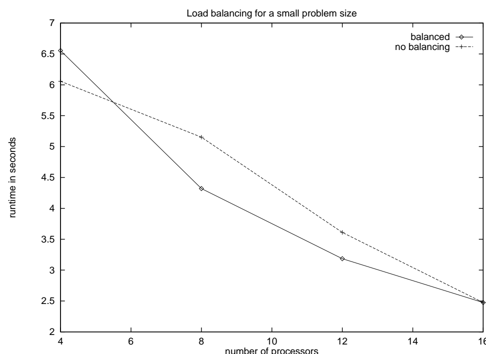


Figure 10: Small Problem and Heavy Lattice Columns.

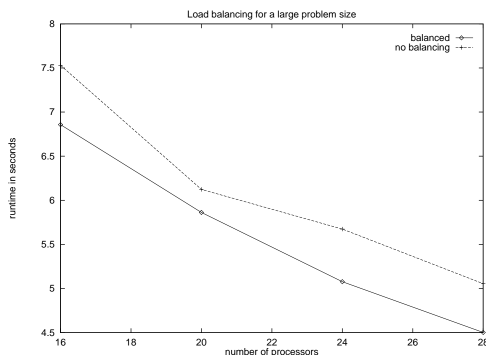


Figure 11: Large Problem Size Heavy Lattice Columns.

Workshop on Parallel and Distributed Simulation, pages 124–131, 1997.

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

[6] D. W. Glazer and C. Tropper. On Process Migration and Load Balancing in Time Warp. *Workshop on Parallel and Distributed Simulation*, pages 318–327, 1993.

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

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

[9] R.S. Ostfeld, K.R. Hazler, and O.M. Cepeda. Temporal and Spatial Dynamics of *Ixodes scapularis* (Acari: Ixodidae) in a rural landscape. *Journal of Medical Entomology*, 33:90–95, 1996.

[10] P. Reiher and D. Jefferson. Dynamic Load Management in the Time Warp Operating System. *Transactions of The Society for Computer Simulation*, 7:91–120, 1990.

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

[12] 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.

[13] B. K. Szymanski and E. Deelman. Balancing continous load on a network of workstations. *submitted to Informantion Processing Letters.*

[14] B.K. Szymanski and T. Caraco. Spatial analysis of vector-borne disease: A four species model. *Evolutionary Ecology*, 8(3):299–314, 1994.

[15] L. F. Wilson and D. M. Nicol. Automated Load Balancing in SPEEDES. *Winter Simulation Conference*, pages 590–596, 1995.

[16] L. F. Wilson and D. M. Nicol. Experiments in Automated Load Balancing. *Workshop on Parallel and Distributed Simulation*, pages 4–11, 1996.