Design and Implementation of Dynamic Load Balancing Algorithms for Rollback Reduction in Optimistic PDES

FALGUNI SARKAR* and SAJAL K. DAS†

Department of Computer Science, University of North Texas, P.O. Box 311366, Denton, TX 76203-1366

(Received 26 May 1998)

In an optimistic parallel simulation, logical processes (lps) proceed with their computation without any constraints. However, if the computing requirements of different lps are not balanced or if the processors are not homogeneous, some lps may lag behind in simulation time while others surge forward. In other words, if the simulation clocks of different lps are not progressing at the same rate, cascading rollbacks may occur nullifying the potential benefit of an optimistic parallel discrete event simulation (PDES). Hence it is necessary to balance the computational load on different lps in such a way that their local simulation clocks advance almost at the same rate. In this paper, we propose two algorithms for dynamic load balancing which reduce the number of rollbacks in an optimistic PDES system. Our first algorithm is based on the load transfer mechanism between lps; while the second algorithm, based on the principle of evolutionary strategy, migrates logical processes between several pairs of physical processors. We have implemented both of these algorithms on a cluster of heterogeneous workstations and studied their performance. The experimental results show that the algorithm based on the load transfer is effective when the grain size is greater than 10 milliseconds. The algorithm based on the process migration yields good performance only for grain sizes of 20 milliseconds or larger. In both of these cases the speed up ranges mostly between 1 and 2 using four processors.

Keywords: Genetic algorithms, GVT computation, load balancing, performance study, process migration, time warp

1. INTRODUCTION

Simulation is an important tool for studying and modeling the behavior of many complex systems

such as VLSI logic circuits, telecommunication networks, and battlefield simulation [29]. Most of these applications, however, require enormous amount of computing power which may not be

* Current address: Wireless Networks, Nortel, P.O. Box 833871, Richardson, TX 75083-3871.

† Corresponding author. Tel.: (940) 565-4256, Fax: (940) 565-2799, e-mail: {fsarkar@nortel.com, das@cs.unt.edu}

provided by a conventional sequential computer. Parallel discrete event simulation (PDES) aims to solve a computation-intensive simulation problem by partitioning it into modules and running them concurrently on different processors, such that a significant speedup is achieved.

The physical system in a PDES is modeled as a set of independent logical processes (called lps) which interact among themselves at discrete time points by message passing. Each of these logical processes is mapped onto a physical processor. Logical links between the processes are embedded in the underlying interprocessor communication network topology of the multiprocessor architecture. This helps in maintaining a modular view of the simulation model and provides a natural means for exploiting parallelism. Figure 1 shows an example of mapping 16 lps into a completely connected multiprocessor system of four processors. The communication between lps is represented by a directed graph, called *lp-digraph*, in which every node is associated with an lp and there is a directed edge from one node to another if message transfer takes place between the corresponding lps. Figure 2 shows an example of an *lp-digraph*.

In a simulation system there is usually a strict ordering among events based on their time of

FIGURE 1 A mapping of lps to physical processors.

FIGURE 2 An *lp-digraph*.

occurrences. This ordering must be preserved in order for the simulation results to be correct and consistent. Synchronization problems in PDES occur due to out of sequence execution of these events. There are two major approaches to solve this synchronization problem. In a *conservative* simulation [5, 7], every lp computes the time T for the earliest message arrival into the lp, and the simulation process is advanced upto time T . Conservative simulation guarantees correct simulation at any point of time but lps may remain idle trying to ensure proper synchronization among themselves. It is also susceptible to *deadlock*, which can be avoided in several ways, for example, with the introduction of *null messages* [8], such that no arrival will occur with a timestamp less than that of the actual message. On the other hand, deadlock may be allowed to occur which is then detected and eventually broken. Misra [25] described a special algorithm in which a special marker message is used to detect a deadlock. Some of the other approaches for deadlock detection and recovery are due to Chandy and Misra [8] and Grošelj and Tropper [17].

In an *optimistic* (or Time Warp) parallel simulation [19], every lp proceeds with its simulation (*i.e.*, receives and processes the event messages from its own input queue) disregarding the

possibility of out of order execution of events. Every lp maintains its own local clock known as the *local virtual time* (lvt). Each lp saves its state and messages, which is called *checkpointing*. In case an lp receives a *straggler* message with timestamp less than its local clock value, the process execution is interrupted and a *rollback* action takes place. A rollback consists of restoring the process to the appropriate states saved earlier, and sending cancelation notices (using *antimessages*) for messages produced by the rollback portion of the computation. Thus there is some amount of wasted computation associated with every rollback. Although an optimistic simulation can result in high parallelism, but too many rollbacks may degrade the system performance severely without achieving significant speedup. A message is called transient if it has been sent but not yet received. *Global virtual time* (GVT) is defined as the minimum of all local virtual times and the timestamp on all transient messages. No system can rollback to a state prior to GVT and hence all system states saved prior to GVT can be discarded in an optimistic parallel simulation. Efficient GVT computation mechanisms is another important aspect of optimistic parallel simulation.

For surveys of different approaches and analysis of parallel and distributed simulation, refer to Fujimoto [14]; Misra [25]; Nicol and Fujimoto [29]; Ferscha [13] and LeMaster *et al.* [20].

The number of rollbacks in an optimistic simulation is due to a large number of factors such as the difference in service times for various types of events, differences in event generation rate, communication delay between processors, multiprocessor architecture, and so on. However, the effect of all these are reflected in the rate of progress of simulation at different lps. If the simulation clocks of different lps are not progressing at a similar rate, some of the lps may lag behind in simulation time and in turn send messages to the ones far ahead in their simulation clock values. This situation causes a large number of rollbacks. Therefore, it is desirable to keep the

local clocks of all the lps in synchrony, which motivates this paper.

Any global load balancing scheme will need to collect data from each of the processors in the system. Since in a PDES environment the lps communicate their *lvts* while computing GVT, it is natural to invoke the load balancing algorithm with GVT computation piggybacking on the communication pattern used. In this paper, we describe two dynamic load balancing techniques for reducing the total number of rollbacks in optimistic PDES systems. Both of our algorithms piggyback on the existing GVT computation routines and hence do not incur a large amount of additional overhead. The basic idea is to keep the simulation clock at different lps in synchrony (*i.e.*, to keep the rate of progress of simulation at different lps similar), which helps reduce the number of rollbacks. We have also implemented the proposed algorithms on a cluster of workstations. The experimental results show that our load balancing approaches are effective (in the sense that the number of rollbacks are actually reduced compared to the situation where no load balancing is used) only when the grain size of the computation at the lps is kept within a certain range, and also when the algorithmic steps are simple and fast. The first algorithm transfers the load across lps, while the second algorithm is based on the concept of evolutionary strategy (genetic algorithm) and uses process transfer. Since no efficient mechanism exists as of today for process transfer, we have only mimicked the process transfer in our simulation system.

The rest of the paper is organized as follows. Section 2 summarizes the related work on rollback reduction techniques and reviews the existing load balancing methods in the context of PDES. Section 3 proposes two new load balancing algorithms based on the load transfer and process transfer. Section 4 discusses the implementation details of our algorithms, and specific constraints faced during the implementation phase. Section 5 presents and analyzes the experimental results, while Section 6 concludes the paper.

2. RELATED WORK

2.1. Rollback Reduction Techniques

Whenever an out of order computation is detected in an optimistic PDES, three phases of additional computation are introduced. Consider a simulation system which does not save every system state, and suppose it receives a straggler message with time stamp T_2 at virtual time T_3 , such that $T_2 < T_3$. But since every state of the system is not saved, the process needs to restore the system state at T_2 before resuming execution from T_2 . Suppose $T_1 \leq T_2$ is the time when a checkpoint was taken just before T_2 . Then during the *rollback* phase, the computation performed during the virtual time interval $[T_2, T_3]$ is undone. In the *coast forward* phase, the state at T_2 is restored by performing the same set of computation which were performed during the virtual time period $[T_1, T_2]$, after restoring the state at T_1 . Then comes the *normalforward execution* phase when the system is advanced to time T_3 again from T_2 . Thus, there is a considerable overhead associated with every rollback operation.

Lubachevsky [23] introduced the concept of *filtered* rollbacks, which postpones the execution of certain events optimistically to keep the number of rollbacks in an optimistic PDES under control. He also presented an analytical evaluation using barrier synchronization scheme, and studied *cascading* rollbacks for a simulation system with self feeding loop. A bounded time warp algorithm was designed by Turner and Xu [39] in order to reduce the number of rollbacks, especially in the simulation of telephone switching networks.

Prakash and Subramanian [32] presented a method of filtering events to prevent cascading rollbacks. This is a modified optimistic algorithm and each message carry a list of conditions to be satisfied in order for the message to eventually survive simulation. Each node also maintains a list called the rollback list of straggler events, and after receiving a message it checks the rollback list for

possible violation of any condition. Otherwise the event is processed.

Remote process execution to keep simulation clocks of different logical processes in synchrony is due to Butler and Wallentine [6]. Bagrodia and Liao [1] have shown how the detection of artificial rollbacks can reduce the overhead due to state saving and coast forward phase. Prasad and Naqib [30] studied the effect of a global event queue in rollback reduction, by experimenting on an 8-processor SGI machine.

2.2. Load Balancing in PDES

Although the area of PDES is being actively researched in the past decade, the problem of scheduling and load balancing in the context of parallel simulation has received relatively less attention. Load balancing issues in the general setup of parallel and distributed processing has been a widely studied problem [36]. The goal is to achieve improved system performance by minimizing the response time and maximizing the throughput or resource utilization by means of distributing the computation load among processors. There are three important aspects of load balancing—*load distribution mechanism*, *load information management* and *load balancing strategy* [24]. Load distribution mechanism is concerned with issues like who should invoke the load balancing algorithm and how often should the load balancing take place. Load information management decides what information should be managed, how should it be represented and how often should it be collected. Since the development of an optimal strategy for load balancing is known to be an NP-hard problem in general, approximate algorithms as well as heuristics have been proposed in the literature for suboptimal solutions. A load balancing strategy decides which algorithm should be used.

Load balancing in the context of PDES possesses some unique characteristics and it is different from the general approaches followed in distributed systems. This is primarily due to the

synchronization constraints or dependencies between the lps. For example, in an optimistic parallel simulation, it is not only enough to keep the computation balanced among the lps in order to avoid a large number of rollbacks. It is also desirable to distribute the load in such a fashion that the simulation speed becomes comparable at different lps. More precisely, the load should be distributed such that the local virtual clock values of different lps at any time are more or less the same. For an excellent survey on various load balancing strategies in parallel discrete event simulations, refer to Boukerche and Das [3]. In the following let us summarize the techniques proposed for conservative and optimistic simulations.

2.2.1. Conservative Simulation

The limited research that has been done on load balancing in PDES has focused mainly on the conservative approaches. Nicol and Reynolds [28] proposed a statistical approach to dynamic partitioning for a homogeneous system of processors. Hagerer and Lang [18] presented a heuristic (based on set partitioning) for balancing the workload in a parallel programmable event-flow computer. Nandy and Loucks [27] designed an iterative improvement algorithm for partitioning and mapping in a conservative parallel logic simulation environment. They attempted to minimize the communication overhead as well as distribute the workload among the processors evenly. They also reported a parallel version of their partitioning algorithm and implemented on a multicomputer consisting of 8 INMOS transputers [26]. Deo, Medidi and Prasad [12] have developed parallel battlefield simulators on the BBN Butterfly GP-1000 machine and Intel's iPSC/2 multicomputer. They addressed the communication overhead problem by processor allocation strategies that suit the underlying architecture which leads to dynamic load balancing for the BBN butterfly machine. Boukerche and Tropper [4] used simulated annealing with adaptive search schedule to

find good (sub-optimal) partitions. They conducted experiments in hypercube machine to study the effect of conservative synchronized parallel simulation making use of null messages. Making use of the Chandy-Misra null message protocol, Boukerche and Das [2] recently proposed an elegant dynamic load balancing scheme which assumes no compile time knowledge about the workload parameters. This approach is based upon a process migration mechanism and the notion of CPU-queue length, which indicates the workload of each processor. Two variants of the algorithm are implemented on an Intel Paragon A4 multicomputer and tested with queueing network simulations of tori. A reduction of 30–40% in synchronization overhead is observed when compared with the static partitioning algorithm proposed in [4].

2.2.2. Optimistic Simulation

As mentioned earlier, the area of load balancing in optimistic PDES is yet to receive significant attention. Reiher *et al.* [33, 34] showed how temporal load balancing offers better performance in Time Warp Operating System (TWOS), which runs a single job at a time and the goal is to complete the job as quickly as possible rather than maximizing the utilization of the hardware. TWOS simulation consists of objects representing parts of the simulation system. These objects are further divided into phases which are responsible for an object's behavior for some time interval of virtual time. Load balancing is achieved by migrating phases from one processor to another. The cost of TWOS is higher memory requirements due to state duplication. Any phase will need a copy of the previous phase's last state as the input. In case of a rollback, the copy of the state of a phase may need to be transmitted from one processor to another. Their load balancing is based on *effective utilization*, which is defined as the proportion of work that can not be rolled back. The strategy is to keep the effective utilization at all the nodes close to the average.

Schlagenhaft *et al.* [35] described a simulation technique on a network of workstations where the number of basic simulation elements (such as gates in digital simulation) may be very high. Their approach detects strongly connected components in the circuit and combines them into disjoint clusters called corollas reducing the cut cost. These corollas are combined into clusters, and several clusters are put in a partition and assigned to each simulation process. During load balancing clusters are moved from processors to processors. An obvious advantage of this approach is that during a rollback the whole partition need not rollback. Their load balancing algorithm is based on the concept of *integrated virtual time* (IYT) which is analogous to average virtual time of simulation per cluster for every simulation step. *Virtual time progress* (VTP) during an interval is defined as the ratio of the change in IVT to the length of the interval. Since there is a cost associated with a cluster movement, the predicted VTP is compared to this cost to decide if a cluster needs to be moved. Sporrer and Bauer [37] used a two-level partitioning strategy for digital circuits to achieve load balance and to reduce rollbacks in the simulation of VLSI circuits, thus leading to higher speed up.

Das and Sarkar [11] proposed a scheme to reduce rollbacks in optimistic simulation by means of process transfer. This approach is based on the computation of *global virtual time* (GVT). The authors probabilistically analyzed the performance of their scheme using a random walk model, and derived conditions under which the scheme is effective.

2.2.3. Contributions of this Paper

In this paper, we propose two dynamic load balancing schemes for optimistic PDES, using the rate of progress of GVT as the load information to be managed. Different load distribution schemes have been used for these strategies. Sections 3.3 and 3.4 provide details of these schemes. The algorithm in Section 3.3 is based on load transfer between processors. Our approach is

different from that of Reiher and Jefferson [33], who assume that the behavior of the objects is fairly regular over a long period of time so that an estimate based on earlier computations fairly accurately represents the current effective utilization. They also assume that there is a single job in the system and hence to avoid processors' idle cycle time they use effective utilization. But our system, a network of workstations, is shared by many users and hence a load balancing based on the progress of the virtual time will increase the system throughput. We only assume that the rate of progress of the simulation clock at an lp does not change drastically between two consecutive load balancing invocations.

The algorithm described in Section 3.4 is based on transferring a process from one node to another. The approach in [35] has an overhead associated with the VTP computation and each lp is assumed to be a conglomeration of clusters. In our scheme, on the other hand, each lp is a single process and hence the entire lp is moved during load balancing. So it is more general at the higher cost of process migration.

3. PROPOSED LOAD BALANCING STRATEGIES

In a parallel discrete event simulation, there are several logical processes (lps) which represent various components of the system to be simulated. A subset of lps is mapped onto a physical processor in a multiprocessor architecture. Let us assume that the simulation system has n lps denoted by $L_1, L_2, \dots, L_n$. Let E_i denote the local event queue of L_i , and $t(e)$ stand for the timestamp of an event e .

Before proceeding further, let us define a few terminology and describe the problem scenario.

3.1. Terminology and Notations

The *local virtual time*, $lvt_i(\tau)$, of an lp L_i at real time τ is defined as the minimum among timestamps of all the events in E_i . That is, $lvt_i(\tau) =$

$\min\{t(e_j)|e_j \in E_i\}$. If the system adopts preemptive scheduling policy, then $\text{lv}_i(\tau)$ also measures the local simulation clock value, SC_i , of L_i at time τ .

If there is a delay in message delivery in the multiprocessor network, the local virtual time should take this into account. Let δ be the time to deliver a message between two lps, L_i , and L_j . If at any point of time the lvts at both lps are the same, then a message sent by L_i , will reach L_j after δ time units and may cause rollback.

A message, m_{ij} , sent from L_i to L_j is said to be *transient* if it is yet to be received by L_j . Let $t(m_{ij})$ be the timestamp of the event represented by the transient message m_{ij} .

The *global virtual time*, $\text{GVT}(\tau)$, of the system at time τ is defined as the minimum among the local virtual times at all the lps and the timestamps of all the transient messages. In other words, $\text{GVT}(\tau) = \min\{\min\{\text{lv}_i(\tau) \text{ for all } i\}, \min\{t(m_{ij})|L_i, \text{ communicates with } L_j \text{ for all } i, j\}\}$.

3.2. Problem Scenario

Broadly, load balancing algorithms can be classified into two categories—static and dynamic. In *static* load balancing, the load (computation and/or communication) values are estimated before the task execution and tasks are distributed among processors based on these estimates. However, the system behavior often changes dynamically. It is more natural to adopt a dynamic load balancing strategy, though it may be more costly. In a *dynamic* load balancing scheme, as computation progresses the load function is estimated at different points of time and tasks are transferred across processors to achieve a balance. In the following subsections we propose dynamic load balancing strategies for optimistic parallel discrete event simulation systems.

Ideally, any load balancing scheme designed for optimistic PDES should be such that if there is no major change in the simulation characteristics, then the following two properties hold, where $\text{EXP}(X)$ denotes the expected value of a statistical parameter X and $\text{Var}(X)$ is its variance.

- $\text{EXP}(\text{lv}_i(\tau) - \text{GVT}(\tau)) \leq c_1$, for all values of τ , where c_1 is a constant and the expected value is taken over the entire set of n lps.
- $\text{Var}(\text{lv}_i(\tau) - \text{GVT}(\tau)) \leq c_2$, for all values of τ , where c_2 is a constant and the variance is computed over all lps.

If an lp, say L_x , needs to transfer a certain amount of load (a part of the functionalities) to another lp, say L_y , the physical transfer of load across the system may be accomplished in three different ways.

1. Send a fraction of load from L_x to L_y by means of remote procedure execution.
2. Partition L_x into two different lps, say L_{x_1} and L_{x_2} , and migrate L_{x_2} to the processor handling L_y .
3. When the number of lps running on a physical processor is more than one, transfer an lp from one physical processor to another to achieve the desired load transfer.

Now the method to be adopted in a given application depends on the system characteristics. If the cost of message transfer is not very high or the number of available links in the architecture is high, one might adopt the first method. On the other hand, say in VLSI logic simulation, each lp may correspond to a gate and there are a large number of lps per processor. In this case, the third method can be used since it is easier to transfer an lp from one physical processor to another.

3.3. Load Balancing Based on the Rate of Progress of Simulation

In most optimistic simulation systems, a GVT computation algorithm already exists which uses a binary tree embedded in the logical processor network. However, it is not necessary for this tree to be a binary tree. In fact, any spanning tree of the underlying lp-digraph can be used for GVT computation although the actual time required may be larger in the case of arbitrary trees due to larger degree of nodes or greater depth of the tree.

We make use of a spanning tree for the GVT computation algorithm¹ in order to design a load balancing scheme. As will be shown later, this spanning tree is actually used to carry the load information to the manager.

Let us discuss how the load transfer can help match the speed of different lps. Each logical processor L_i computes a simple estimate for its rate of progress of simulation as $R_i = (SC_2 - SC_1) / (\tau_2 - \tau_1)$, where SC_1 and SC_2 are respectively the simulation clock values of L_i , at two different points of real time, say τ_1 and τ_2 . The spanning tree used by the GVT computation algorithm defines a parent-child relationship between the lps. Whenever an lp sends its GVT update to its parent, it also sends its own R_i value and those of its children. So the node initiating the GVT computation receives the estimates of the rate of progress of simulation (R_i 's) from each lp and sorts them. With the help of the spanning tree, this list of R_i values are also broadcast to each of the processors along with the new GVT estimate. Every lp keeps track of the R_i values at its neighbors after receiving this list. Whenever a mismatch in the rate of progress is detected, the lp searches its neighbors in the lp-digraph of other physical processors with which it can share its load by means of remote procedure execution or by invoking process transfer. This mechanism is somewhat similar to the *buddy* strategy for load balancing in a distributed system [9].

We now describe how the entire process is handled by a given lp. Let $lvt_x(\tau)$ be the local virtual time of an lp, L_x , at time τ . From the earlier values of global virtual times received by L_x , it is possible to estimate the current GVT value. This estimate can be either parametric or non-parametric based on the system characteristics. For the time being we assume that we have a function $f(\cdot)$ to estimate the GVT value at time τ . In [10], Das and Sarkar have presented a non-parametric method for estimating GVT with a given confidence coefficient. We can define two boundary

points in the range of this GVT estimate and let us call them *low* and *high*. At any time if lvt_x is larger (resp. smaller) than high (resp. low), the corresponding lp L_x classifies itself to be in the state U (resp. O) denoting underloaded (resp. overloaded). Otherwise L_x is assumed to be in a *normal* (N) state.

Whenever an lp reaches the state U or O , it requests the states from its neighbors and creates two lists—one consisting of the lps with state O and the other with the lps having state U . The lps in these lists are sorted in the increasing order of their current R_i values. If the current lp is in state U (or O), it selects a neighboring lp from the list of O (or U) nodes and transfers a certain amount of load, say $\mathcal{L}$ to it.

Now the question is what should be the value of $\mathcal{L}$. If we assume that the load balancing takes place periodically only after every T seconds, then $\mathcal{L}$ should be such that the local clock values of the lps, between which the load transfer took place, become equal. Let us suppose that the load transfer took place between L_x and L_y having the local clock values as SC_x and SC_y , respectively. If we assume that their local clock values will become equal to SC after T seconds, then the time required for progressing L_x from SC_x to SC is $((SC - SC_x) / R_x)$ and the corresponding time required for L_y is $((SC - SC_y) / R_y)$. If $\mathcal{L}$ is the amount of computational load transferred, then

$$T = \frac{SC - SC_x}{R_x} - \mathcal{L} \text{ and } T = \frac{SC - SC_y}{R_y} + \mathcal{L}$$

which yields

$$\mathcal{L} = \frac{SC_x - SC_y - T(R_y - R_x)}{2}.$$

This load $\mathcal{L}$ is such that the load transfer will make simulation clock of both the lps (between which the load transfer is taking place) equal after time T , provided the characteristics of the simulation at these two lps have not undergone major

¹ Although we are using a spanning tree based algorithm for GVT computation, any other algorithm will work equally well.

changes. The following procedure formally presents the algorithm described above.

Procedure Load-Balance-Progress-Rate(x)

/ Procedure to be executed at node L_x for load balancing */*

begin

if (L_x is the GVT manager)

broadcast $\{R_i | 1 \leq i \leq n\}$ to all the lps. */* all-to-all communication */*

Choose γ , the confidence coefficient for GVT estimate.

Using the function $f(\cdot)$ and the R_i values received as samples, compute the confidence interval [low, high] for GVT with confidence coefficient γ .

if ($lvt_x > \text{high}$) state = U */* underloaded */*

else if ($lvt_x < \text{low}$) state = O */* overloaded */*

else state = N */* N indicates a normal state */*

endif

if (state $\neq N$) collect state of all the neighboring lps.

if (state = O)

Select at random a neighboring lp L_y with state U .

Transfer a fraction $\mathcal{L} = \frac{SC_x - SC_y - T(R_y - R_x)}{2}$ of load from L_x to L_y .

endif

if (state = U)

Select a neighboring lp L_y with state O .

Receive a fraction $\mathcal{L} = \frac{SC_y SC_x - T(R_x - R_y)}{2}$ of load from L_y .

endif

end

This algorithm is simple and relies on the assumption that physical load can be split and transferred from one processor to another. However, the situation may not be so simple in

practice. The method of estimating the GVT from R_i s can be either parametric or non-parametric [10]. The use of parametric methods will require knowledge about the distribution of GVT increments with respect to physical time. The confidence coefficient γ is subject to the user's choice. However, a value of 90% or 95% is likely to yield good results. If the topology of the physical processor network is regular, it is easier for a processor to compute its neighbors on the fly. Otherwise each processor need to maintain a list of its neighbors.

3.4. Load Balancing Based on Evolutionary Strategy

A *genetic* algorithm (GA) is a general technique for solving combinatorial optimization problems [16], by modeling the natural selection process where the fittest progeny always survives. A genetic algorithm consists of (i) a string representation of the *genes* of the *population* or the nodes in the search space of the optimization process, (ii) a set of genetic *operators* for creating new generation (iii) a probabilistic control function that controls the operation of the genetic operators. The basic scheme consists of the following steps. For a survey on genetic algorithms, refer to [38].

1. A set of the population nodes in the search space is randomly generated.
2. The fitness value of each of the nodes chosen in Step 1 is computed using the fitness function.
3. The genetic operators are applied to the existing nodes to create new solutions and only the fittest individuals in the combined population are retained.
4. Previous steps are repeated until the algorithm converges.

In the following we present a load balancing technique based on the concept of evolutionary strategy, a variation of the genetic algorithm, which can be used for both static and dynamic load balancing in optimistic PDES. The distribution of the Ips into various groups (referred as

clusters from now on) related to the physical processors corresponds to the strings in the genetic algorithm.

3.4.1. Static Load Balancing

In the static scheme, the computation requirement of various lps is first estimated. Let us assume that there are p processors $P, P_2, \dots, P_p$ and n lps $L_1, L_2, \dots, L_n$ where $n \approx k \cdot p$. Here k is the average number of lps per physical processor. Let L_x be the computation load of the lp L_x where $1 \leq x \leq n$. Then $W = ((\sum_{x=1}^n L_x)/p)$ is the average load per processor. At the beginning of our algorithm, we randomly divide n lps into p clusters, each cluster containing at most $\lceil k \rceil$ lps. Assume that C_i denotes the i -th cluster having a load of W_i , for $1 \leq i \leq p$.

We define two types of genetic operations – *transfer* and *exchange* – which operate on a pair of clusters. These operations will also be referred to as *mutations*. If C_i and C_j are two clusters satisfying $W_i > W_j$, then the transfer operation selects one lp at random from C_i and transfers it to the cluster C_j . Similarly, the exchange operation selects one lp from each of the two clusters and exchanges them.

Before starting the algorithm, the number of iterations to be performed is assumed to guarantee that the clustering process will converge. In each iteration, we select two clusters C_i and C_j and apply either a transfer or an exchange operation. The probability of occurrence of each of these operations are chosen to be equal. Let $W_{i,a}$ and $W_{j,a}$ be the load of the clusters C_i and C_j respectively after performing a mutation operation. If $|W - W_i| + |W - W_j| > |W - W_{i,a}| + |W - W_{j,a}|$, then we start the next iteration with the distribution obtained after applying the mutation operation. Otherwise we retain the distribution prior to applying the mutation and proceed to the next iteration. In the static load balancing scheme, the algorithm uses the estimates of the computation load and tries to distribute them among the physical processors as evenly as possible. In this case we do not emphasize the communication load, which

is automatically taken care of in a dynamic load balancing scheme described next.

3.4.2. Dynamic Load Balancing

In the dynamic load balancing scheme, our goal is to keep the local virtual clocks of different lps almost equal. This is achieved by distributing the lps among various processors so that the system time required by different processors to advance the simulation clocks of all the lps mapped onto it, are the same. As opposed to the static scheme, the load distribution is such that the lps advance their simulation clocks at the same rate. This is the reason for using R_i , the rate of progress of simulation with respect to the physical time, rather than the actual computation load. Since every lp L_i keeps an estimate of R_i , the system time needed to advance the local simulation clock by one time unit is given by $(1/R_i)$. To gather these R_i values, a special processor is chosen as the *manager*. Next we embed a binary tree in the processor network with the manager as the root, which broadcasts a message *send-load-info* to all other processors after every T time units. A processor on receiving the message *send-load-info*, passes it onto its children in the binary tree and sends the R_i values for each of its lp to the manager. Load balancing calculations, described in the following procedure, take place at the manager who in turn sends the information about the new distribution of the lps to the respective processors.

To balance the load on different processors, the maximum number of iterations (N_Max) to be performed and a *tolerance factor* (ϵ) are fixed first. The lps mapped onto a processor are treated as a cluster. For each cluster, we define a *fitness* measure which is based on the deviation of the load in the cluster from the ideal load. As the algorithm proceeds, pairs of clusters are chosen at random and a *mutation* (transfer or exchange) is applied to them. If the fitnesses of the clusters increase (or if the differences between their load and the average load decrease) after applying the mutation, the new configuration is chosen. Other-

wise the previous configuration is retained. If the algorithm does not achieve the desired result after performing N_MAX iterations, the tolerance factor is increased and the algorithm is applied again. This procedure is described as follows.

Procedure Dynamic-Load-Balance-Genetic

begin

Fix an $\varepsilon > 0$. /* ε is the tolerance factor*/

Fix N_MAX , the number of iterations per phase.

Form p clusters $p, C_1, C_2, \dots, C_p$ one for each processor.

Each cluster contains the lps mapped onto the corresponding processor.

Compute the value $f_i = \sum_{L_j \in C_i} \frac{1}{R_j} - \frac{1}{p} \sum_{j=1}^n \frac{1}{R_j}$ for each cluster.

/* The maximum value off is the fitness measure of the distribution in the genetic algorithm. */

$fit = \max_{i=1}^p \{f_i\}$

while($fit > \varepsilon$) **do**

$N = 0$

while ($N < N_MAX$) **do**

Select a pairs of clusters C_i and C_j with $f_i < 0$ and $f_j < 0$.

Apply one of the two mutations *transfer* and *exchange*, selected at random, to C_i and C_j .

Retain the previous distribution for the selected pair if the mutation does not reduce f_i .

$N = N + 1$

end

$fit = \max_{i=1}^p \{f_i\}$

if $fit < \varepsilon$ **Stop**

else $\varepsilon = 2 \cdot \varepsilon$

end

end

This algorithm assumes that an lp can be physically transferred across processors. But no efficient hardware/software platform yet exists in reality to implement such a thing. So, unless the

grain size of computation is considerably high, the algorithm may not perform well. The fitness function corresponding to a processor is really the time needed to advance all the lps mapped to a physical processor by one unit of simulation time. It represents the relative computational need for every cluster formed.

4. IMPLEMENTATION DETAILS

The goal in both of our proposed load balancing algorithms has been to keep the speed of simulation clock at each of the logical processes balanced. We have implemented and studied the performance of these two algorithms on a network of four heterogeneous computers (two DEC ALPHAs, a Sequent Symmetry multiprocessor and an IBM RS6000 computer) which are connected through Ethernet. Each of these machines works in a multiuser mode, and therefore the processing time is shared by all the users.

The message passing is implemented using the distributed networking software package PVM (Parallel Virtual Machine). Figure 3 shows how the PVM functions. In the cluster of machines, one node is designated to run the PVM console, the managing process. All other nodes of the cluster runs the PVM daemon (PVMD). The PVM console provides a means for interfacing with the PVM and PVMD takes care of the task of message passing. Each task spawned using PVM has a unique task id. A message is sent using the task id of the receiver by making a function call to PVM. The corresponding PVMD in turn processes the message and sends it to the PVMD on the receiver side, which passes the message to the receiver.

FIGURE 3 Process communication using PVM.

Several processes running on a single machine communicate between themselves by using a socket whereas PVMD running on two different processors communicate using the IP protocol. PVM also provides built-in routines to perform the following functions for the tasks spawned by it:

- Changing configuration of the cluster, *i.e.*, addition or deletion of workstations.
- Providing information about the cluster and the tasks running on it at any point of time.
- Handling message buffers.
- Computing group functions.
- Passing messages.

We have implemented an abstract model of an optimistic parallel discrete event simulation in our workstation cluster. The two load balancing algorithms implemented on this network of workstations include the algorithm (Section 3.3) for transferring the load between two processors and piggy backing on GVT, and the load balancing algorithm (Section 3.4) based on the evolutionary strategy. We have studied the effect of these algorithms in reducing the number of rollbacks. System restrictions on memory usage, the number of file descriptors and the number of tasks limit the number of lps that one PVMD can handle.

We create a master process which in turn creates all the lps. The master process decides how many lps should be created on each machine and also does the task of load balancing. The master process is always spawned on the faster DEC ALPHA machines. Since the proposed load balancing algorithms are simple and the master process runs on the faster machines, the time required for load balancing itself is very low and hence not reflected in the time needed for the simulation run.

Lps run independently on different machines. Initially, every lp creates certain events for itself. When an lp processes an event, it generates a new event. Each event is then classified as either internal or external based on the distribution of a random variable. *Internal* events are inserted into the event queue of the lp, whereas *external* events

are communicated to the receiving lp by means of a pvm message. The receiving lp is chosen at random from the set of lps except for the sender. Every event is attached with a computation time which is a hyper-exponential random variable with a certain mean, called the *grain size*. An lp while executing an event perform some basic computation for the duration of time equal to the computing time of the event. Two other parameters (generated as exponential random variables) are also attached to every event. One indicates what should be the increment in the local simulation clock after processing this event, while the other indicates the simulation time (*i.e.*, timestamp) when this event should have been processed.

Every lp has three queues. The *event queue* contains the set of events yet to be processed. There is an *old message queue* which contains the events generated since the last GVT computation. The *antimessage queue* contains the set of antimessages received by the lp. A message type is indicated by a flag which is generated at the time of message creation. When a rollback occurs, the set of events from the event queue is examined and the corresponding antimessages are generated and sent to other lps. Every lp receives the antimessages destined for itself in the antimessage queue. We follow a lazy cancelation approach here. Both event queue and the antimessage queue are maintained using a heap structure. When an event is to be processed, it has to be in the front of the queue. Before processing it, the antimessage queue is checked to see if the event is to be annihilated.

4.1. Limitations due to System Constraints

We have faced several constraints while performing experiments on the network of workstations, using PVM. Since we did not have any specific computation/application in mind, we implemented grain size by introducing a sleep time for that duration. But the system marks all the processes sleeping for a long time as suspended. So, we had to perform some basic computation several times

instead of 'sleep'. Furthermore, since our network is heterogeneous, the grain size had to be multiplied by appropriate constants (weights) according to the machine speed.

To measure processing as well as elapsed time has been a major problem. We intended to measure time with a precision of microseconds. Some of our machines do not have any function to measure the elapsed time in microseconds. So, we have used the function `gettimeofday()`. The resolution of the system clock is hardware dependent. The functions for measuring time is also different in different systems and they reside in different libraries which made it difficult to maintain the uniformity of the lps.

The number of file descriptors (FD) per user is limited to 32 in some of the machines we considered. This restriction is used since the system under consideration are shared by other users. Because of this constraint, it is almost impossible to create more than six lps in some machines. The number of messages the PVMD can handle also restricts the total number of lps. One additional problem arises when another such program is run using PVMD by another user. In such a case the number of messages goes beyond the acceptable limit and the system practically comes to a halt. This limits the total number of machines that could be included in our experiments.

Our load balancing algorithms are based on the rate of progress of simulation with respect to the physical time. Based on this rate, we recompute the grain sizes which need to be an integer because they determine the number of basic computations to be performed. When there are many users, lps get swapped out and quite often this makes the rate of progress of simulation to be zero. In such a case the physical time is much larger than the increment in simulation time, making the algorithm inapplicable. This restricts the number of lps created for testing these algorithms between 12 to 16 and their average grain sizes to less than 1000 milliseconds.

We have made some interesting observations with respect to the capability of PVM in handling

such applications. For example, the simulation stops when a certain value of GVT is achieved. But at this time even though the lps terminate, some messages remain in the system since their recipients are no longer available. These messages remain in the system as long as the PVMD's are run, thus decreasing the available memory considerably. As a consequence, several iterations of the algorithms become questionable. Since there is no built-in flow control, faster lps flood the PVMD's with messages making them stop. This happens since there is a FIFO policy for message passing and PVMD's can not send control messages before sending the simulation messages. Thus there is a restriction on the number of external messages that an lp can produce. Since memory is allocated when a message is generated, the total memory allocated to a process increases. The process swap space being limited, sometimes the operating systems become unable to swap processes and the entire system comes to a stand still.

5. EXPERIMENTAL RESULTS

In our implementation, we first vary the grain size from 1 to 10 milliseconds incrementing by one millisecond at every step. Then we vary the grain size from 10 to 100 milliseconds incrementing by 10 milliseconds at every step. The grain size increases by 100 milliseconds at every step in the range 100 to 1000 milliseconds. Note that the grain size specified here is not the computing time for every event at an lp, since such a case is not very realistic. On the contrary, the computing time for every event in an lp is a hyper-exponential random variable with mean equal to the grain size specified above. Since after a few invocations of the load balancing algorithms this grain size becomes almost equal, the lps periodically change their load by generating these hyper-exponential random variables. Figure 4 shows how the maximum and minimum grain size changes during a simulation run. The spikes observed in the difference

FIGURE 4 Load distribution over time during simulation run.

between the maximum load and minimum load is due to the lps changing their grain size periodically.

Physically transferring a process is difficult and no known software exists to handle this efficiently. In our case, all lps performed identical functions. So when an lp L_i is to be transferred from processor P to processor Q, we created another lp L_j in Q which is a copy of L_i and all new messages are then sent to L_j . The parameters are also transferred from L_i to L_j . The lp L_i runs as long as its message queue is not empty. Once the event queue of L_i becomes empty, before terminating itself L_i sends a signal to L_j to start execution. The computation related to an event can easily be simulated by making the process 'sleep' for that duration. However if this sleep time is long, the process gets marked as suspended by the operating system and does not reflect the true characteristics of the system. To avoid this problem, we have multiplied every such computation time by a factor and made the system perform some basic computation for those many iterations.

5.1. Results for Load Transfer Algorithm

Figures 5 and 6 show the empirical results for simulation time ≤ 2000 using the first algorithm

which is based on load transfer. We used 16 lps which were equally distributed over the four processors. The computation size at every lp followed a hyper-exponential distribution which is a constant added to an exponential distribution. The system was simulated both with and without load balancing schemes. In each of these cases we have counted the total number of messages exchanged in the system, the number of antimes-ages generated in the system, and the number of primary rollback messages. Primary rollback messages are those which initiate a primary rollback and may cause further rollbacks. Use of only 16 lps limits the scope of the algorithm to a great extent. Since load balancing is costly, in practice we have only chosen the fastest and the slowest lp for load transfer. That too is done only when the difference in their simulation clocks is very significant. Since the number of lps used is not very high, locating the fastest and the slowest lps is relatively inexpensive.

The graph in Figure 5 shows the speed up obtained due to load transfer mechanism. Initially, the speed up increases but a decreasing trend is observed beyond a certain value of the grain size. The average speed up is about 1.30 using four processors, showing a significant improvement due

FIGURE 5 Speed up obtained using the load transfer mechanism.

FIGURE 6 Ratio of the number of rollbacks to the total number of messages vs. grain size, using load transfer algorithm.

to load balancing. The graph in Figure 6 shows the ratio of rollback messages to the total number of messages for different grain sizes. Excluding extreme values, we observe an average decrease of about 2% in this ratio.

When the grain size is within the range 1 to 10 milliseconds, in eight out of ten cases load balancing has reduced the number of rollback messages. When the grain size is within the range 10 to 100 milliseconds, in seven out of ten cases

load balancing has reduced the number of rollback messages. Similarly in the range 100 to 1000 milliseconds for grain size, in seven out of ten cases load balancing has actually reduced the number of rollbacks. However, the performance of the algorithm is better when the grain size is larger than 10 milliseconds.

5.2. Results for Process Transfer Algorithm

While implementing our second load balancing algorithm, we observed that the introduction of exchange of two processes causes severe overhead. Hence we have restricted ourselves only to the transfer operation. Moreover, performing two or more transfers in the same iteration causes severe degradation in the performance, so we perform only one transfer in every invocation of the load balancing algorithm. This transfer takes place only between the processors containing the slowest lps and the processor containing the fastest lps. We transfer processes only when the difference between their fitness coefficient is significantly large (as high as the average fitness). The results shown

for the process transfer based load balancing algorithm is for the case when we start with 5 lps on both the Sequent Symmetry and IBM RS600 machines and one lp on each of the DEC ALPHAs.

Figures 7 and 8 show the effect of load balancing. We have simulated 12 lps on the four-available processors, and these results correspond to the situation when there are 6 lps in each of the two slower processors and 2 lps in each of the two faster processors. All of these lps are identical in characteristics. The system was simulated as long as the GVT value was smaller than 2000. We have counted the total number of messages, the number of antimessages and the number of primary rollback messages in the case when there was no load balancing and also when the load balancing algorithm was used.

The graph in Figure 7 shows the speed up obtained due to load balancing using the process transfer algorithm. Here also beyond a certain value of the grain size, we observe a decreasing trend. The average speed up is about 1.31 using four processors. But it is likely to be less when we

FIGURE 7 Speed up obtained using the process transfer algorithm.

FIGURE 8 Ratio of the number of rollbacks to the total number of messages vs. grain size, using process transfer algorithm.

start with a more uniform distribution of lps over the processors. The graph in Figure 8 shows the ratio of rollback messages to the total number of messages for different grain sizes. Excluding extreme values, we observe an average 3% decrease in this ratio.

When the grain size is within the range 1 to 10 milliseconds, in seven out of ten cases our simulation implementation performs worse with load balancing. In the range 10 to 100 milliseconds, in eight out of ten cases the implementation with load balancing performs well. In the range 100 to 1000 milliseconds again, in eight out of ten cases load balancing improves the performance. From this we can conclude that this load balancing algorithm is very effective for larger grain sizes.

6. CONCLUSIONS

We have presented two dynamic load balancing algorithms for reducing the total number of rollbacks in an optimistic PDES environment.

Each of these algorithms piggyback on the existing GVT computation procedure and hence the additional cost for executing these algorithms themselves is expected to be low. We have implemented both the algorithms on a network of workstations and studied their empirical performance. The load transfer based algorithm assumes a system in which lps can be broken into smaller ones when required. VLSI logic circuit simulation is a good example of such systems where an lp may be simulating a single gate. The other algorithm is based on process transfer. All the processes on a machine are treated as a cluster. Processes are moved from one cluster to another when necessary. We have used PVM to implement the message passing interface in these systems. Since there is no built-in flow control in PVM, we had to restrict certain characteristics, like the number of messages generated per second, the ratio of internal to external messages of the lps, *etc.* There is no efficient system for implementing process transfer in a distributed environment. Hence to study the effectiveness of our algorithm, the process transfer mechanism was mimicked.

Our results show that the dynamic load balancing is effective only when the computation times for the events are within a certain range and the number of external messages generated is restricted. Dynamic load balancing becomes ineffective if it is used too often since it may incur a large overhead.

Many load balancing algorithms assume the presence of effective means for migrating a process from one processor to another which is not the case in practice. In the case of a parallel simulation, we have clocks, system state queue, message queues and other data structures associated with a process. The situation becomes even more complicated due to the messages in transit. A message may reach its destination and find that the corresponding process has migrated by that time. A naive solution to this problem is to halt all the processes when the load balancing takes place. However, such a solution will be slow and inefficient. But the software implementing the message passing to keep the physical location of the process transparent will be of great use. Another interesting question is: do we need to move the entire old message queue? If rollbacks do not occur frequently, it may be enough to transfer just a few previous system states. We have assumed in our experiments that any lp can directly communicate with any other lp, *i.e.*, the underlying network topology is completely connected. But in a real system, there is always a communication pattern and an lp sends messages directly to only a few neighbors. Our future work will study the effect of our load balancing algorithms for this kind of systems. Additional experiments need to be conducted to study the scalability of our approaches for a large number of workstations. Our experiment in this paper is based on a theoretical model. An improved version of it would be to run a trace for a real application with and without loadbalancing. In our future research we plan to implement a real trace driven simulation application to study speed-up gain and effect of loadbalancing.

Acknowledgements

This work is supported by Texas Advanced Technology Program Grant under Award No. TATP-003594031 and Nortel Wireless Technology in Richardson, Texas.

References

- [1] Bagrodia, R. and Liao, W. T. (1992). "Transparent Optimization of Overheads in Optimistic Simulations," *Proceedings of the 1992 Winter Simulation Conf.*, pp. 637–645.
- [2] Boukerche, A. and Das, S. K. "Dynamic Load-Balancing Strategies for Conservative Parallel Simulation," *Proceedings of 11th Workshop on Parallel and Distributed Simulation (PADS'97)*, Austria, pp. 20–28, June 1997.
- [3] Boukerche, A. and Das, S. K. (1998). "Load Balancing Strategies for Parallel Simulations on a Multiprocessor Machine," to appear In: *The State-of-the-art in Performance Modelling and Simulation* (Eds., Walrand, J., Bagchi, K. and Zobrist, G.), Gordon and Breach Pub.
- [4] Boukerche, A. and Tropper, C. (1994). "A Static Partitioning and Mapping Algorithm for Conservative Parallel Simulation," *Proceedings of the 8th Workshop on Parallel and Distributed Simulation*, **24**(1), 164–172.
- [5] Bryant, R. E. *Simulation of Packet Communication Architecture Computer Systems*, Massachusetts Institute of Technology, Tech. Rep. MIT/LCS/TR-188, Cambridge, MA, Nov. 1977.
- [6] Butler, J. E. and Wallentine, V. E. (1994). "A Performance Study of the RPE Mechanism for PDES," *Proceedings of the Second International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, Durham, North Carolina, pp. 214–220.
- [7] Chandy, K. M., Holmes, V. and Misra, J. (1979). "Distributed Simulation of Networks," *Computer Networks*, North-Holland Publishing Co., **3**, 105–113.
- [8] Chandy, K. M. and Misra, J. (1981). "Asynchronous Distributed Simulation via a Sequence of Parallel Computations," *Communications of the ACM*, **24**(11), 198–206.
- [9] Chang, Y. C. and Shin, K. G. (1993). "Optimal Load Sharing in Distributed Real-Time Systems," *Journal of Parallel and Dist. Computing*, **19**, 38–50.
- [10] Das, S. K. and Sarkar, F. "A Hypercube Algorithm for GVT Computation and its Application in Optimistic Parallel Simulation," *Proceedings of the 28th Annual Simulation Symposium*, Phoenix, Arizona, Apr. 1995, pp. 51–60.
- [11] Das, S. K. and Sarkar, F. "Reducing Rollbacks Through Load Sharing in Parallel Discrete-Event Simulation," *Proceedings of the Conference on Parallel and Distributed Computing and Systems*, Washington, Oct. 1994, pp. 407–410.
- [12] Deo, N., Medidi, M. and Prasad, S. (1992). "Processor Allocation in Parallel Battlefield Simulation," *Proceedings of Winter Simulation Conference*, pp. 718–725.
- [13] Ferscha, A. (1995). "Parallel and Distributed Simulation of Discrete Event Systems," In: *Handbook of Parallel and Distributed Computing* (Ed., Zomaya, A.), McGraw-Hill.

- [14] Fujimoto, R. M. (1990). "Parallel Discrete Event Simulation," *Comm. of the ACM*, **33**(10), 31–53.
- [15] Glazer, D. W. and Tropper, C. (1993). "On Process Migration and Load Balancing in Time Warp," *IEEE Trans. on Parallel and Distributed Systems*, **4**(3), 318–327.
- [16] Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, MA.
- [17] Grošelj, B. and Tropper, C. (1988). "The Time-of-next-event Algorithm," *Proceedings of the SCS Multiconference on Distributed Simulation*, pp. 25–29.
- [18] Hagerer, A. and Lang, S. "An Approach for Balancing the Workload of the Munich Simulation Computer," *Proceedings of the SCS Multiconference on Advances in Parallel and Distributed Simulation*, **23**(1), Anaheim, CA, Jan. 1991, pp. 141–147.
- [19] Jefferson, D. (1985). "Virtual Time," *ACM Trans. on Programming Languages and Systems*, **7**(3), 405–425.
- [20] LeMaster, T. E., Dutta, A. K. and Ghosh, S. "Virtual Time—A New Paradigm for Synchronization in Distributed Systems," In: *Advances in Distributed and Parallel Processing*, Vol. I: System Paradigms and Methods (Ed., Harry W. Tyrer), Ablex Publishing, pp. 110–175.
- [21] Lin, Y. and Lazowska, E. D. (1991). "A Study of Time Warp Rollback Mechanisms," *ACM Trans. on Modeling and Computer Simulations*, **1**(1), 51–72.
- [22] Lin, Y. B., Preiss, B. R., Loucks, W. M. and Lazowska, E. D. (1993). "Selecting the Checkpoint Interval in Time Warp Simulation," *Proceedings of the 7th Workshop on Parallel and Distributed Simulation*, San Diego, CA, July pp. 3–9.
- [23] Lubachevsky, B., Shwartz, A. and Weiss, A. (1989). "Rollback Sometimes Works... If Filtered," *Proc. of Winter Simulation Conf.*, pp. 630–639.
- [24] Milojicic, D. S. (1994). "Load Distribution Implemented for Mach Microkernel," Vieweg.
- [25] Misra, J. (1986). "Distributed Discrete-Event Simulation," *ACM Computing Surveys*, **18**(1), 39–65.
- [26] Nandy, B. and Loucks, W. M. "On a Parallel Partitioning Techniques for Use with Conservative Parallel Simulation," *Proceeding of the 7th Workshop on Parallel and Distributed Simulation*, July 1993, San Diego, CA, pp. 43–51.
- [27] Nandy, B. and Loucks, W. M. "An Algorithm for Partitioning and Mapping Conservative Parallel Simulation onto Multicomputers," *Proceeding on the 1992 SCS Western Simulation Multiconference on Parallel and Distributed Simulation*, **24**(3), Jan. 1992, Newport Beach, CA, pp. 139–146.
- [28] Nicol, D. and Reynolds, P. F. (1985). *The Automated Partitioning of Simulations for Parallel Execution*, Tech. Report. TR-85-15, Univ of Virginia.
- [29] Nicol, D. and Fujimoto, R. M. "Parallel Simulation Today," to appear in: *Annals of Operations Research*.
- [30] Prasad, S. K. and Naqib, B. "Effectiveness of Global Event Queues in Rollback Reduction and Load Balancing," *Proc. of the 9th Workshop on Parallel and Distributed Simulation*, Lake Placid, NY, May 1995, pp. 187–190.
- [31] Preiss, B., McIntyre, I. and Loucks, W. (1992). "On the Trade-off Between Time and Space in Optimistic Parallel Discrete-Event Simulation," *Proceeding of the 6th Workshop on Parallel and Distributed Simulation*.
- [32] Prakash, A. and Subramanian, R. "Filter: An Algorithm for Reducing Cascaded Rollbacks in Optimistic Distributed Simulations," *Proc. of the 24th Annual Simulation Symp.*, New Orleans, LA, April 1991, pp. 123–132.
- [33] Reiher, P. and Jefferson, D. (1990). "Virtual Time Based Dynamic Load Management in the Time Warp Operating System," *Proceedings of the Workshop on Parallel and Distributed Simulation*, pp. 103–111.
- [34] Reiher, P., Bellenot, S. and Jefferson, D. "Temporal Decomposition of Simulations Under the Time Warp Operating System," *Proceedings of the SCS Multiconference on Advances in Parallel and Distributed Simulation*, **23**(1), Anaheim, CA, Jan. 1991, pp. 47–54.
- [35] Schlaghaft, R., Ruhwandl, M., Sporrer, C. and Bauer, H. "Dynamic Load Balancing of a Multi-Cluster Simulator on a Network of Workstations," *Proc. of the 9th Workshop on Parallel and Distributed Simulation*, Lake Placid, NY, May 1995, pp. 175–180.
- [36] Shirazi, B. A., Hurson, A. R. and Kavi, K. M. (Editors) (1995). *Scheduling and Load Balancing in Parallel and Distributed Systems*, *IEEE Computer Society*, NJ.
- [37] Sporrer, C. and Bauer, H. "Corolla Partitioning for Distributed Logic Simulation of VLSI-Circuits," *Proceeding of the 7th Workshop on Parallel and Distributed Simulation*, July 1993, San Diego, CA, pp. 85–92.
- [38] Srinivas, M. and Patnaik, L. M. (1994). "Genetic Algorithms: A Survey," *IEEE Computer*, **27**(6), 17–26.
- [39] Turner, S. J. and Xu, M. Q. "Performance Evaluation of the Bounded Time Warp Algorithm," *Proceeding on the 1992 SCS Western Simulation Multiconference on Parallel and Distributed Simulation*, **24**(3), Jan. 1992, Newport Beach, CA, pp. 117–126.

Authors' Biographies

Falguni Sarkar received B.S. and M.S. degrees in Statistics and M.Tech. degree in Computer Science in 1985, 1987 and 1989, all from the Indian Statistical Institute, Calcutta. He received Ph.D. in Computer Sciences from the University of North Texas, Denton, in 1996.

Dr. Sarkar has worked for Citibank from 1989–'92 in several countries, and at present he is a Manager in Nortel Technologies, Richardson, Texas. His research interests include parallel memory access, parallel simulation, interconnection networks, parallel data structures and algorithms, and cellular communication systems.

Sajal K. Das received the B.Tech. degree in 1983 from Calcutta University, the M.S. degree in 1984 from the Indian Institute of Science, Bangalore, and the Ph.D. degree in 1988 from the University of Central Florida, Orlando, all in Computer Science. Currently he is a Full Professor of Computer Science at the University of North Texas, Denton, and also the Director of the Center

for Research in Parallel and Distributed Computing. Dr. Das is a recipient of Honor Professor Awards from UNT in 1991 and 1997 for best teaching and scholarly research, and UNT's Developing Scholars Award in 1996 for outstanding research. He has been a Visiting Scientist at the Slovak Academy of Sciences in Bratislava and the Council of National Research in Pisa, Italy, and also a Visiting Professor at the Indian Statistical Institute, Calcutta.

His current research interests include wireless networks and mobile computing, parallel algo-

rithms and data structures, and parallel simulation. Dr. Das has published over 140 research papers in these areas. He serves on the Editorial Boards of the Journal of Parallel and Distributed Computing, Parallel Processing Letters, and the Journal of Parallel Algorithms and Applications. He has served on the Program Committees of numerous conferences and is the General CoChair of the International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'98). Dr. Das is also a member of the IEEE and the ACM.