Parallel Logic Simulation on Distributed Memory Multiprocessors Classification and Evaluation of different Approaches *

Peter Luksch
Institut für Informatik
Technische Universität München
D-80290 München
Germany
e-mail: luksch@informatik.tu-muenchen.de
Tel.: +49-89-2105-8164; Fax: +49-89-2105-8232

Abstract

A test environment is presented that allows for different methods of executing discrete event simulations in parallel to be evaluated in a uniform environment. A great variety of parallelizations have been proposed in the past. Up to now, however, an unbiased comparative evaluation of different approaches has been impossible because run-time measurements published in the literature have been obtained with different simulators on different multiprocessor systems and therefore cannot be compared.

Our approach to an unbiased comparison of different parallelization methods is as follows: The variety of existing methods is structured by classifying them according to the way how the simulation task is subdivided into processes and how these processes synchronize. Thus a small number of fundamentally different approaches to distributed simulation can be identified each of which comprises a whole class of parallel algorithms. As a basis for the testbed, one representative from each approach has been implemented. Thus a comparison of different approaches is possible while at the same time a library of functions is provided that allows further parallelizations to be implemented easily.

1 Introduction

Discrete event simulation (DES) is a widely used method to analyze complex systems or to validate them during the design phase. In VLSI design, simulation has become the only practical way to validate the logic and timing behavior of a system under construction. While *logic* validation is facilitated by the use of high level behavioral specifications which are transformed into gate level descriptions by synthesis tools, *timing* verification will require the system to be simulated at the gate level. Gate level simulation generally requires some orders of magnitude more computational power than simulation at a high level behavioral abstraction.

Sequential computers no longer can cope with the increasing demand for computational power that emerges from the desire for more comprehensive simulation of increasingly complex systems. Currently, parallel simulation on general purpose multiprocessors is the most promising and cost-effective way of providing the necessary compute power.

A great variety of methods for executing discrete event simulation in parallel have been proposed in the literature. For most of them, prototype implementations have been reported with

*This work has been partially funded by the DFG ("Deutsche Forschungsgemeinschaft", German science foundation) under contract No. SFB 342, TP A1.

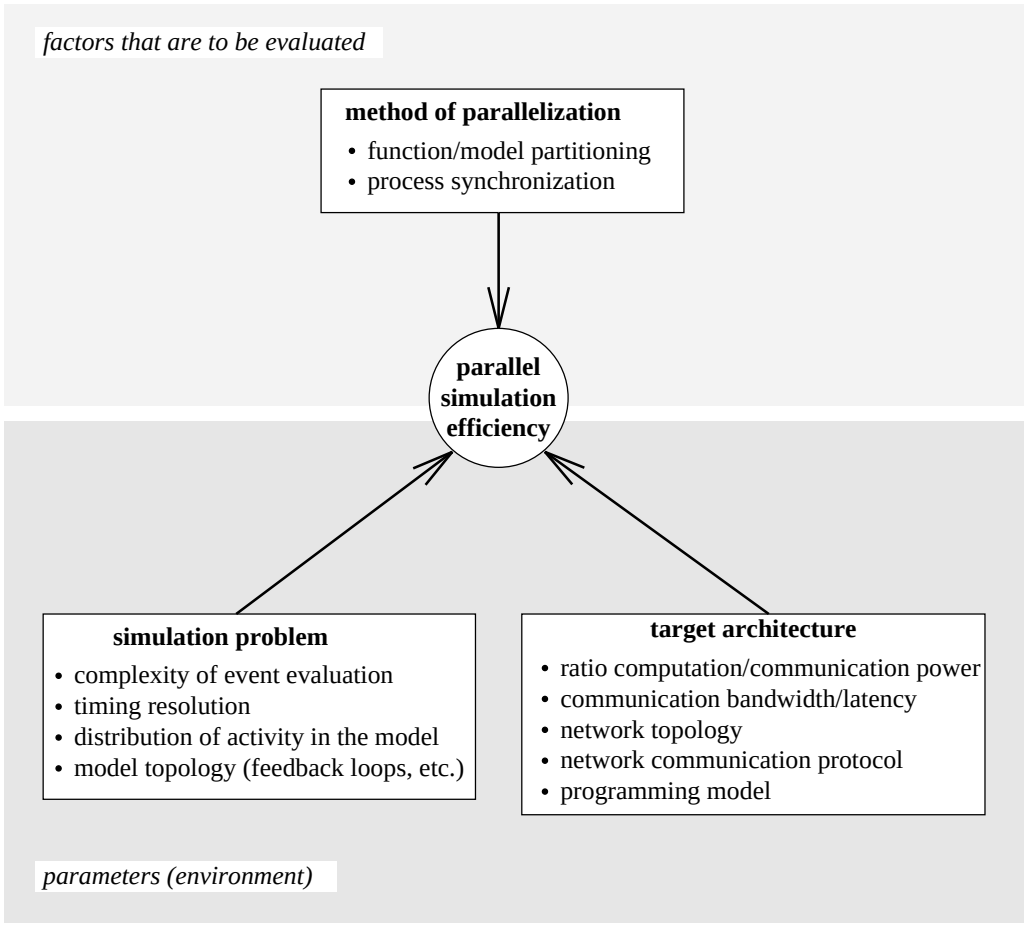


Figure 1: Factors that contribute to parallel simulation performance

different simulators on different target architectures. Run-time measurements ranging from no speedup at all to super-linear speedup do not clearly favor any specific approach. Table 2 in the appendix gives an overview of studies published in the literature. The main reason for the divergence of efficiency results in previous experiments is the fact that efficiency does not only depend on the method of parallelization but is also determined by the type of simulation problem considered and by parameters of the target architecture. Figure 1 summarizes the most important factors that contribute to a parallel simulator's performance.

2 Classifying DDES Methods

The discrete event simulation algorithm, which is displayed in fig. 2, can be parallelized in various ways. DDES algorithms are characterized by the following properties:

- the general *a*pproach to *p*arallelization, i.e. the distribution of functions and data structures (AP).
- the way how global *con*trol is organized (CN).
- the *sy*nchronization protocol used to coordinate processes (SY).

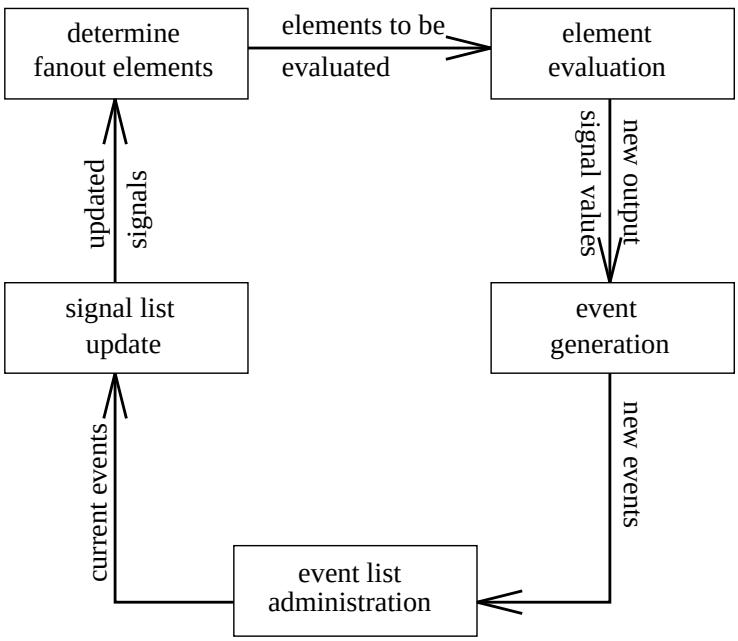


Figure 2: The discrete event simulation algorithm

Based on these criteria, DDES algorithms can be subdivided into a small number of fundamentally different approaches as explained in the subsequent sections.

2.1 General Approach to Parallelization

Generally, there a two ways of parallelizing a given algorithm: function decomposition and data partitioning.

In the function decomposition approach (AP = `FD`), the algorithm is subdivided into functions each of which is implemented a process. Applying this strategy to the DES algorithm means to implement the subtasks represented as boxes in fig. 2 as processes that process the stream of events in a pipeline. Global control is performed by the event list administrator which advances the simulation clock to the next event time as soon as all events for the current clock tick have been processed by the pipe.

In the data partitioning approach (AP = `MP`), the problem description, which generally is represented as a set of data, is distributed among several processes. Each process executes one instance of the sequential algorithm which, of course, has to be complemented by some functions for communicating data and synchronizing processes. If a digital logic simulator is parallelized that way, the circuit is divided into partitions each of which is worked on by one process. Such a process will also be called a simulator. This approach is referred to as model partitioning and is illustrated in fig. 3. Signal that connect elements of only one partition are referred to as *local* signals while signals connecting elements in different partitions are referred to as *cut* signals. A simulator that generates an event at a cut signal must communicate the event information to all simulators having fanout elements of that signal.

Both approaches can be combined. In a simulator parallelized by function decomposition, subtasks (e.g. element evaluation) may be replicated and assigned different partitions of the circuit. Such an approach is denoted by AP = `FD+MP`. In contrast, if a simulator has been parallelized by model partitioning, each simulator may be further parallelized by function decomposition. This approach is denoted by AP = `MP+FD` in our classification scheme. Both types of combination can

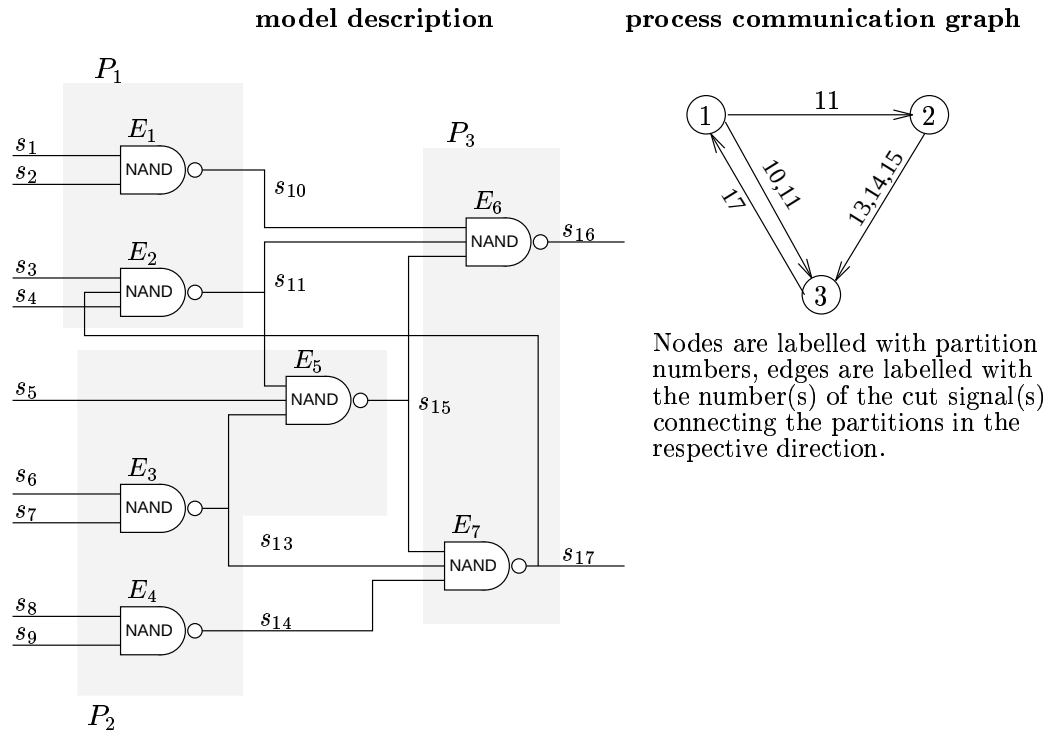


Figure 3: The model partitioning approach

be found in hardware accelerators [9]. Depending on the multiprocessor architecture considered as a target system for distributed simulation, these combination may also be promising on general purpose parallel computers.

2.2 Global Control

While global control is centralized at the event administration subtask in the AP = `FD` approach, AP = `MP` offers several options to coordinate processes. Control may either be centralized or distributed. With distributed control there is a wide range of possible synchronization protocols.

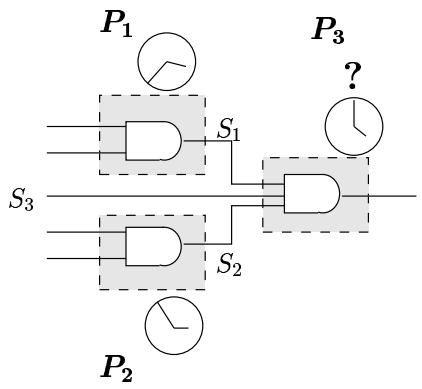
2.2.1 Centralized Control

In this approach, denoted by CN = `GC`, a *g*lobal (simulation) *c*lock is used to synchronize the simulators. At simulated time t, all current events can be processed in parallel. Before simulation time can be advanced to the next event time, a barrier synchronization has to be performed. All hardware accelerators with AP = `MP` implement centralized control.

2.2.2 Distributed Control

For a parallel simulation to be correct, it is sufficient that events depending on each other are processed in the correct sequence. Independent events with different time stamps may be processed concurrently.

This motivates the distributed control approach (CN = `LC`) which relaxes the strong synchronization restriction imposed by CN = `CG` and thus may exploit a higher degree of parallelism.



P_3 wants to simulate an event at signal S_3 with time stamp 4:00 and therefore needs signal values S_1, S_2 at 4:00. These signals, however are computed by P_1 and P_2, respectively.

P_3 does not know P_1' and P_2's simulation clocks.

If P_1, P_2 have not yet reached 4:00 in their simulations, they might send events time stamped before 4:00. As a consequence, P_3 cannot process its event safely.

Figure 4: The synchronization problem with distributed control

Decentralized control generally can be expected to outperform centralized control if time is modeled at a high degree of resolution (e.g. nano/pico seconds in logic simulation) which results in a relatively small number of events to be executed per simulation tic. Also, if barrier synchronization is expensive on the target system, decentralized control is to be preferred.

Control is distributed by having each simulator have its own *l*ocal *c*lock which it advances autonomously. Local simulation clocks give rise to a synchronization problem which is illustrated in fig. 4. Two approaches to its solution have been proposed: the *co*nservative one by Chandy,Misra and Bryant [12] ($SY =$ `CO`) and the optimistic one by Jefferson [20] ($SY =$ `TW`) which is also known as *T*ime *W*arp.

The Conservative Approach to Synchronization

The conservative approach assumes communication channels to deliver messages in FIFO order. All the local simulation clocks are required to increase monotonically. Event messages carry the simulation clock of their sender at the time of generating the event. Thus, if a process P_i has received an event message P_j, the generation time t_j from that message is a lower bound for P_j's current simulation clock.

Each simulator P_i remembers the generation time from the last event received from each of its predecessors P_j in a variable $l_i[j]$, called the *link time* for channel c_{ij}. $l_i[j]$ is P_i's lower bound on the local simulation time of P_j. P_i can only process its next event (at time T_i) if $l_i[j] \geq T_i$ for all its predecessors P_j. In the example of fig. 4, P_3 would suspend its simulation until an event message tells him, that $T_1 \geq T_3 \wedge T_2 \geq T_3$, i.e. the next event can be processed safely.

Deadlock may occur if some process P_i is waiting for another process P_j which, however, will not send any events to P_i because no signal connecting from P_j's partition into P_i's changes value. Another source of deadlock are feedback loops which may result in a cycle of processes each waiting for its predecessor to send an event message. Conservative protocols can be subdivided further according to how *d*eadlock is *h*andled.

*D*eadlock *a*voiding protocols ($DH =$ `DA`) make sure that no deadlock can arise. This may be done by using *n*ull *m*essages that are sent periodically in the absence of "real" events on a channel (`DA-NM`). Null messages, however, do not work for simulations where zero-delay feedback loops may occur such as in digital logic simulation. Another method to prevent deadlock are time requests which essentially are null messages that are sent on demand (`DA-TR`). A time request protocol that works for zero-delay feedback loops, too, has been published by Bain and Scott [3]. It will be

described in section 3.2.1.

*D*eadlock *r*ecovery protocols (DH = `DR`) do not take any precautions to prevent deadlock. Instead, they periodically check for deadlock. Upon deadlock detection, deadlock recovery is started. At least one process (the one with minimum next event time) can be resumed. There is a variety of algorithms to detect deadlock and to recover from it [30]. Of course, one may think of combining deadlock recovery with some precautions that are not guaranteed to avoid deadlock but considerably reduce its probability, as is the case for null messages in simulations were zero-delay feedback loops can occur.

The Optimistic Approach to Synchronization

In Time Warp, no simulator ever waits for receiving an event message form a predecessor. Instead, simulation proceeds regardless of the predecessor's simulation clocks making the optimistic assumption that P_i has received all events up to time T_i when processing its next event at T_i. This assumption turns out to have been wrong if later on a straggler arrives, i.e. an event message with a time stamp t_r in the past of the current local simulation time T_i. Then, P_i' simulation is rolled back to $T_i = t_r$, i.e. local simulation time decreases, which is why this approach commonly is referred to as *Time Warp*.

In order to undo the potentially wrong simulation, P_i has to cancel all events generated during $[t_r,\ T_i]$. Local events are cancelled by restoring the local state at time t_r. Events sent to other processes are canceled by sending anti-messages which will cause the corresponding event's effects to be undone by the receiving process. Thus, the possibility of rollbacks requires the simulator to keep track of previous states and of sent event messages. Sent messages generally are stored in a queue. Two different approaches exist for state saving: *c*heck*p*ointing (`CP`) and *i*ncremental *s*tate *s*aving (`ISS`).

With checkpointing, the simulator periodically saves its state as a whole. Incremental state saving, in contrast, saves state *changes* rather than snapshots of the simulation state as a whole. `ISS` reduces memory requirement at the price of extra computation for restoring a previous state. If state information is large and state changes can be described easily, as is the case in logic simulation, `ISS` clearly is to be preferred over `CP`.

Upon rollback, a sent event can either be cancelled immediately (*ag*ressive *c*ancellation, `AGC`) or it not cancelled before it is sure that the event will not be generated once again in the simulation of the rolled-back period of simulated time (*la*zy *c*ancellation, `LAC`).

Global progress is measured by the global virtual time (GVT) which is defined as the minimum of all local simulation times and the time stamps of all un-processed events in the system. No rollback can occur to a point before GVT. An approximation of GVT is computed periodically in parallel to the simulation in order to detect termination and to free memory used to store information about states before GVT (fossil collection). A great variety of algorithms for GVT approximation can be found in the literature (e.g. [36, 22, 27, 11, 30, 4]).

2.3 A Classification Scheme for DDES Algorithms

The set of options for parallelizing discrete event simulation sets up a classification scheme that assigns a given parallel simulator to one of the approaches illustrated as a classification tree in fig. 5. This tree is the basis for the test environment to be described in the next section.

3 A Portable Test Environment for Distributed Logic Simulation

The goal of the testbed described in this section has been to provide a portable environment that allows a great number of DDES algorithms to be analyzed in detail under uniform conditions. As mentioned in section 1, there are mainly three factors that contribute to the performance of a parallel simulator: the parallelization method which is our primary object of investigation,

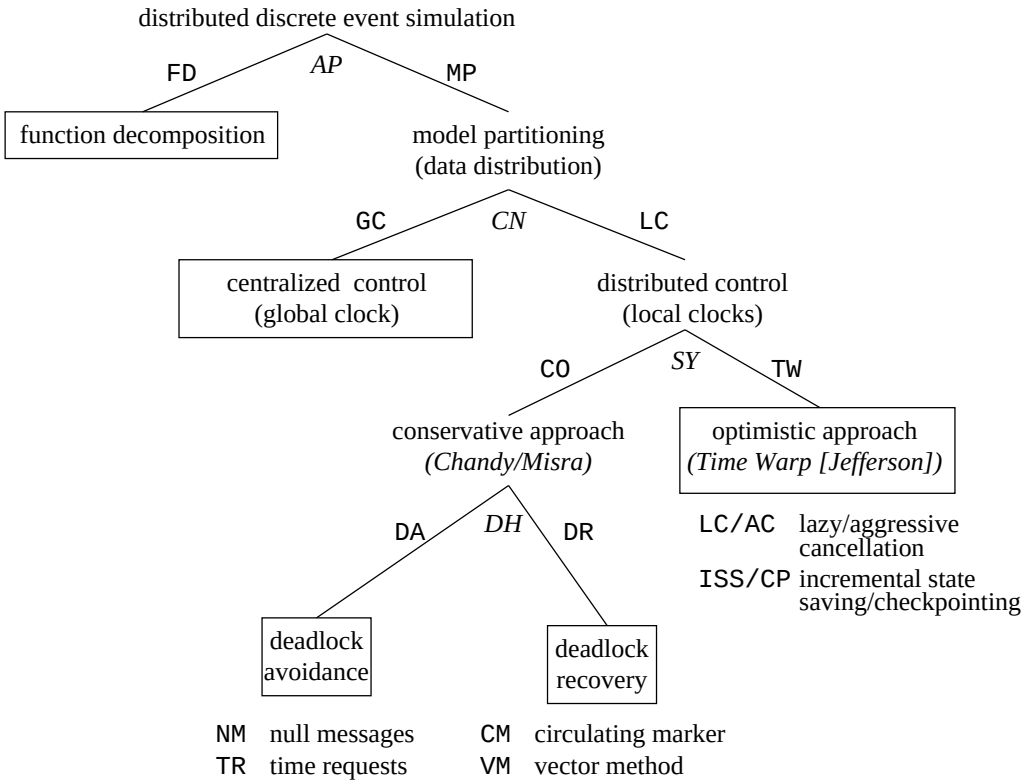


Figure 5: Classification of methods for distributed discrete event simulation

parameters of the target architecture and properties or the simulation application (see fig. 1). They span up a three-dimensional space, as depicted in fig. 6, which is to be covered by the test environment.

The testbed has been designed to cover the full range of parallelization methods as described in the section 2. For a flexible testbed it is necessary that the environment parameters be modifiable in a controlled manner over a wide range. However, a commitment to a certain application of DES is inevitable. Because of its practical importance we have selected digital logic simulation. The basis of the testbed is a (sequential) gate level logic simulator which has been developed at our university's EE department [25]. The simulator implements most of today's techniques in digital system modeling, e.g. various timing models, spike handling etc. A number of ISCAS benchmark circuits have been available as work-loads.

Flexibility with respect to the target system is achieved by implementing all parallelizations based on a machine-independent parallel programming library. Although message passing today is the generally accepted paradigm for programming distributed memory multiprocessors (DMMP), there is not yet a standardized programming model. Basic communication primitives such as

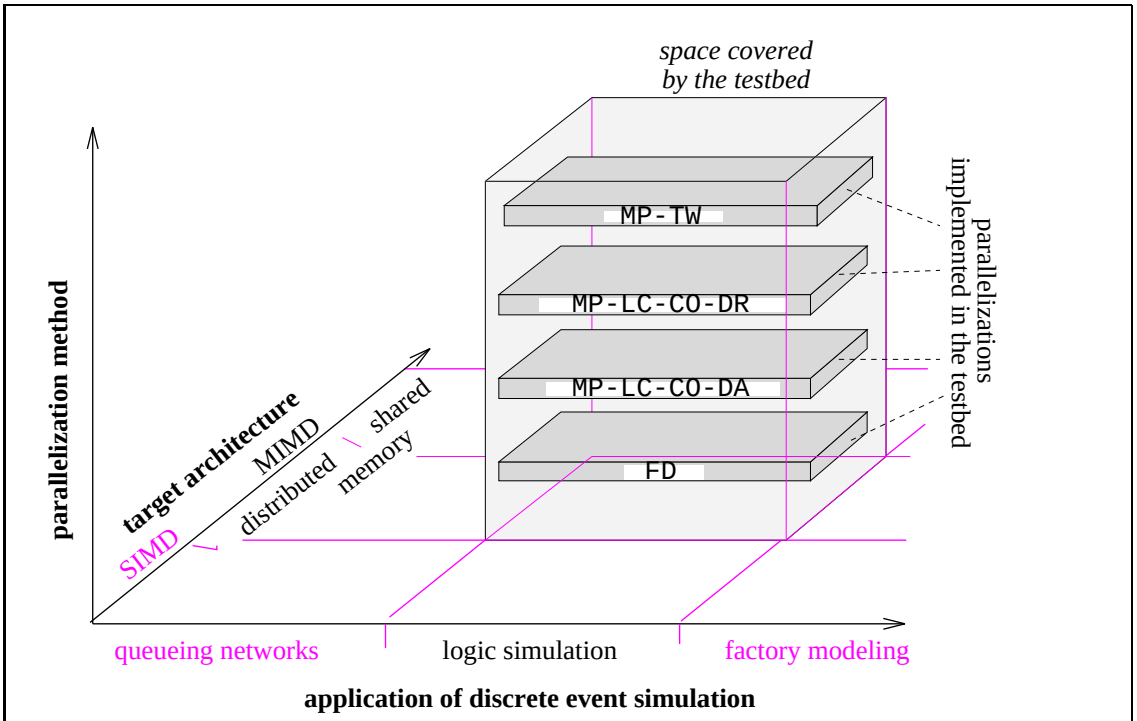


Figure 6: The parameter space of DDES covered by the testbed

(non)blocking send/receive, however, are available in virtually all of today's DMMP programming models. Our testbed is based on the parallel programming library MMK[7] which is available on the iPSC/2 and iPSC/860 DMMP's and, as an experimental version, on networks of Sun SPARC workstations. As only basic communication primitives are used in our parallel programs, the testbed can easily be ported to almost every other message passing programming model.

Based on the classification tree of fig. 5, four parallelizations, each belonging to another leaf of the tree, have been selected for implementation. Thus, the parallelizations implemented in our testbed are distributed evenly over the parallelization axis of fig. 6, enabling a comparative study of different approaches. In addition, a comprehensive library of functions is provided that supports easy implementation of further parallelization methods. It is summarized in tab. 2 in the appendix. The following parallelizations have been implemented [41, 42, 23, 1]:

- FD
- MP-CO-DA-TR
- MP-CO-DR-VM
- MP-TW-ISS with both LAC and AGC

They will be described in more detail in the following sections.

3.1 Function Decomposition (FD)

The sequential simulator is decomposed into six tasks that process the stream of events in a pipeline:

- Input: read stimuli for primary inputs.
- Output: write result.
- Event administration: event list management, advancing simulation time.

- Event generation: events are generated from new values at output signals, preliminary signal values (see [25]) are computed for current events.
- Event execution: compute new signal values, update signal list, determine fan-out elements.
- Element evaluation

The process graph is similar to that depicted in fig. 2.

3.2 Model Partitioning

In the model partitioning approach, a partitioning procedure is needed to assign elements to simulators. Our testbed currently implements two algorithms:

- natural partitioning: elements are assigned in the order in which they appear in the net-list.
- min-cut partitioning: A generalization of Fiduccia/Mattheyses' min-cut procedure for the case of non-bipartitioning has been implemented which has been proposed by Vijan [40]. In our implementation, elements and signals can be weighted individually to account for different activity rates and evaluation complexities.

Efficient communication is a key factor to parallel simulation performance. Today's distributed memory multiprocessors have quite high communication latencies – costs that have to be paid for each message irrespective of its length. This is why in our testbed an event buffering mechanism has been implemented which is controlled by two parameters, l_{min} and a_{max}. Instead of sending each event as one message, events are collected in a buffer which is sent as soon as its length reaches l_{min} events.

To prevent events from being withheld too long in the buffer, a second parameter is used. If sending an event is deferred for too long a time, synchronization overhead increases because simulators keep suspended unnecessary in the conservative approach while in the optimistic approach more speculative computation has to be undone by rollbacks. Therefore, if an event has been in the buffer for more than a_{max} units of simulated time, the buffer is sent regardless of its length.

The partitioning procedure and the event buffering mechanism described above have been used in all of the parallelizations based on model partitioning that have been implemented within the test environment. They will be described below.

3.2.1 Deadlock Avoidance with Time Requests (`MP-CO-DA-TR`)

Based on an algorithm proposed by Bain and Scott [3], a conservative protocol has been implemented that avoids deadlocks by means of time requests. A time request (T_i, P_i) is issued by a simulator P_i to all P_j with $l_i[j] < T_i$, i.e. all predecessors P_j that prevent P_i from advancing its simulation time to T_i which is the time stamp of the next event in its event list.

A time request (T_i, P_i) asks the the receiving process P_j whether its simulation time has already reached time T_i. If so, a YES reply is sent back. As an optimization, a YES reply carries the local simulation time of the replying process to keep the sender's link time up to date. Otherwise, the request is queued and the requesting process is left waiting for the reply. If there are any predecessors P_k of P_j with $l_j[k] < T_i$, P_j sends a request (T_i, P_i) to these P_k. Note that the simulator P_i *originating* the request is entered as the second component, not the sender P_j. Note, that the request has P_i as its second component – the simulator that has *generated* the request for T_i. Its identity is needed for cycle detection, as explained below.

A cycle is detected if a simulator P_l receives a request (T_i, P_i) from some process P_j while it has an identical request in its queue. Then an RYES ("reflected yes") reply is sent to P_j irrespective of the current local simulation time T_l. P_l has, however, to keep in mind that an RYES reply has been given to P_j. If later on an event e_1 with time stamp $t_1 < T_i$ is sent to P_j, the queued copy of request (T_i, P_i) which had been received, say, from P_m, must be answered by a NO reply, because

event e_1 might cause an event e_2 with time stamp $t_2 < T_i$ to be generated and sent to P_i. This is the only situation where NO replies are generated.

A request by P_i is completed if all predecessors to which the request has been sent have sent their replies. If the request has been originated by P_i, the replies decide whether simulation time can be advanced: If all replies are either YES or RYES, simulation will proceed. If any NO replies have been received, simulation must remain suspended. Having updated its channel times $l_i[j]$ according to the replies, P_i generates a new time request.

If a request $(T_k,\ P_k)$ has been completed which had been originated by another process, P_k, a reply is sent to the process P_j from which the request had been received: If there is at least one NO reply, a NO reply is sent. Otherwise, if all replies are YES, a YES reply is sent as soon as $T_i \geq T_k$. Otherwise, i.e. if there are no NO replies but at least one RYES reply, an RYES reply is sent. If $T_i \geq T_k$, the RYES can be converted to YES.

3.2.2 Deadlock Recovery with the Vector Method (`MP-CO-DR-VM`)

In the conservative approach, the alternative to deadlock avoidance is to allow for deadlock to occur, detect it an recover from it. Our implementation of deadlock recovery is based on Mattern's vector method [30]. Two variants of this deadlock detection algorithm have been implemented: a circulating control vector and a parallel version of the vector method. During deadlock detection the next event time is collected from each simulator. Deadlock is recovered from by computing the minimum of these times. All simulators with minimum next event times are restarted.

The circulating control vector The vector method detects deadlock by having each process count the number of messages that are sent to and received from other processes. Each simulator P_i has a (local) vector $\vec{L_i}$. If P_i sends a message to P_j, $L_i[j]$ is incremented by one; if P_i receives a message, $L_i[i]$ is decremented by one. A circulating control vector $\vec{C}$ collects this information on its way through the simulators.

A simulator P_i that has received the control vector keeps it until it has to suspend its simulation because $l_i[j] < T_i$ for some j. Then it updates $\vec{C}$ by adding its local vector to it which then is reset, i.e. $\vec{C} := \vec{C} + \vec{L_i};\ \vec{L_i} = \vec{0}$. The control vector is passed to a process P_j with $C[j] > 0$. If $\vec{C} = \vec{0}$ upon update, deadlock has been detected: all processes have suspended simulation and there is no event message in transit.

The parallel vector method In this variant of the vector method, the control vector $\vec{C}$ is kept by a designated control process, P_C, to which the simulators send their local vectors if they have to suspend their simulation. P_C updates $\vec{C}$ in the same way as with the circulating control vector. Also, a simulator resets its local vector after sending it to P_C. Again, if P_C finds $\vec{C} = 0$, parallel simulation is deadlocked.

3.2.3 Time Warp (`MP-TW-ISS-{AGC,LAC}`)

In our Time Warp parallel simulator, state information is saved incrementally instead of periodically saving the state as a whole (checkpointing). Upon execution events are not removed form the event list. Instead, the signal value prior to event execution is stored in the event data structure. If a rollback to time t_r occurs, a forward search is started in the event list beginning at time t_r. The value of a signal s is restored from the first event affecting s that is found in this search.

Incremental state saving is preferred checkpointing checkpointing in logic simulation because checkpointing would result in very inefficient memory usage since each event changes only a small part of the system state.

Both methods for undoing external events have been implemented: aggressive and lazy cancellation. With aggressive cancellation, an anti-message m^- is sent for each event message m^+ generated in the rolled back period immediately upon rollback. With lazy cancellation, an anti-message m^- is not sent before local simulation time (LVT) reaches the time stamp of m^+. Only

if m^+ is not generated once again in the re-simulation, m^- will be sent.[1] The idea behind lazy cancellation is that re-simulation will re-generate most of the events undone in the rollback.[2]

Global virtual time (GVT) is approximated using Samadi's GVT2 algorithm [36]. Despite being one of the earliest GVT algorithms, run-time measurements have shown a sufficiently close approximation of GVT. GVT2 outperformed a newer algorithm proposed by Lin/Lazowska [28] which does not require simulators to stop computation temporally but requires more messages to be sent. In our implementation of GVT2, however, the requirement of stopping simulation could be relaxed so that simulators may continue computation but must refrain from sending messages. Anyway, investigating newer GVT algorithms such as the one proposed in [4] will be an interesting application of the test environment.

Two extensions to the basic Time Warp mechanism have been implemented within our testbed:

- Being motivated by the same assumption as lazy cancellation *optimized re-simulation* aims at reducing the number of element evaluations during re-simulation, which is especially useful for circuits containing elements whose evaluation is computationally complex.
- *Dynamic re-partitioning* attempts to compensate uneven load distribution by moving elements from a heavily loaded processor to a lightly loaded one. Even if static partitioning has generated equally sized partitions, load may be distributed unevenly if elements have different rates of activity or if activity distribution in the circuit changes over time.

Optimized Re-Simulation
Assume, an element E has been evaluated during the rolled-back simulation at (simulated) time t resulting in an event e_1 to be generated. If during re-simulation, E is evaluated once again at time t, e_1 will be generated again if the state of E is the same as in the corresponding evaluation before rollback. The *state* of an element is defined as the vector of its input signals and its internal state variables. In the example of fig. 7, E's state is defined by the triple $(\mathtt{a},\mathtt{b},\mathtt{c})$[3].

The idea of our optimization is to re-use the event generated before rollback instead of evaluating the element once again if the above condition is met. More precisely, optimized re-simulation works as follows:

During "normal" simulation, the simulator keeps track of the causality relationship between events and element evaluations, i.e. it stores information of the form "event e_1 caused elements E_1, E_2 to be evaluated. Evaluation of E_1 generated e_3, evaluation of E_2 generated e_4." (e_3, e_4 are called *follow events* of e_1 caused by the evaluation of E_1 and E_2, respectively.) In addition the element state has to be remembered for each evaluation.

At rollback, local events are marked as "undone" instead of removing them from the list. If during re-simulation element E_1 is evaluated at time t, the simulator checks if there is information stored about follow events. If so, it compares E_1's state at the corresponding evaluation before rollback to its current state. If states are identical, follow event e_3 is re-scheduled by removing the "undone" mark. Only if there is no follow event information stored or states do not match must E be evaluated.

As long as state transitions during re-simulation are identical to transitions in the rolled-back simulation, "undone" events are "redone" instead of evaluating the element once again. In fig. 7 simulation is illustrated as a sequence of state transitions. As long as the lines for both simulations coincide, optimized re-simulation is in effect. The rolled-back simulation is represented by a dashed line, re-simulation by a solid line.

Dynamic Re-Partitioning There are three alternatives for implementing load balancing at simulation time:

1. There are several simulator processes on each node. The operating system determines the load on each node and migrates processes as necessary.

[1] By re-simulation we mean the renewed simulation of the rolled back period of simulated time, see also fig. 7.
[2] Strictly speaking, this assumption doubts Time Warp's efficiency. However, several studies have shown that lazy cancellation can be more efficient than aggressive cancellation.
[3] For simplicity, we assume that E does not have an internal state.

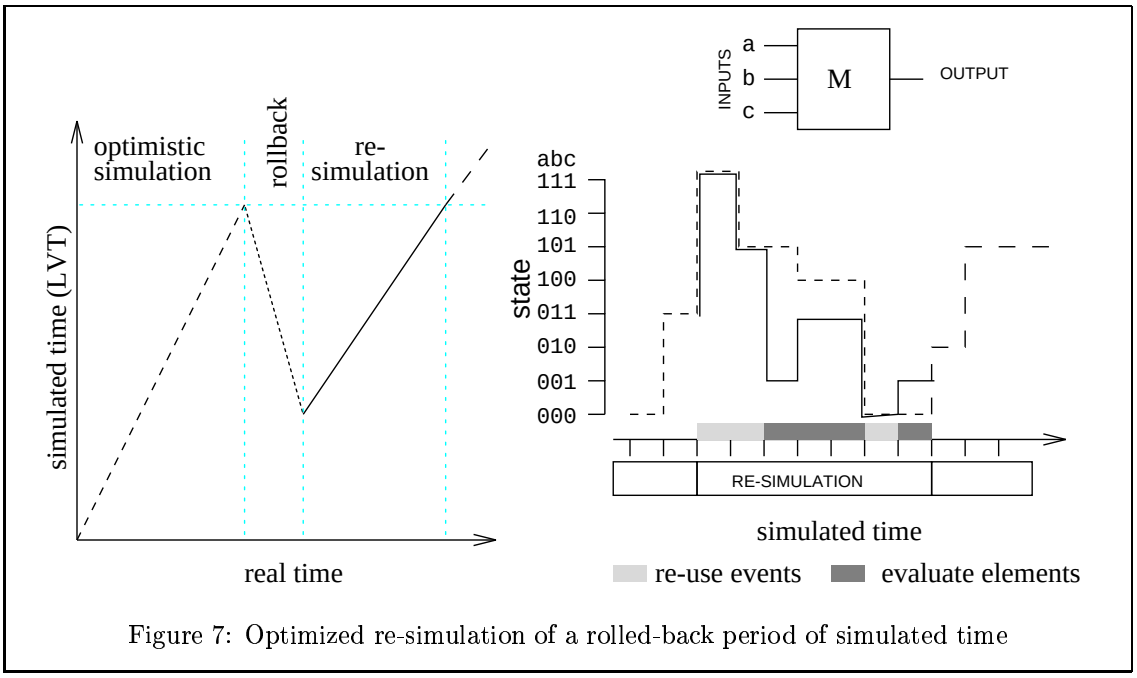


Figure 7: Optimized re-simulation of a rolled-back period of simulated time

However, currently no multiprocessor operating system supporting dynamic load balancing is available for production use. In addition, the Time Warp protocol makes it hard for an operating system to measure load since Time Warp simulators are ready to compute all the time. Moreover, optimal scheduling of several simulators on one node, i.e. lowest LVT first, cannot be implemented with any of today's operating systems.

2. Each simulator processes several partitions each of which has its own LVT. They are scheduled such that the partition with minimal LVT is simulated first. If load imbalance is detected, a heavily loaded process gives one or more of its partitions to a lightly loaded process.

   Load is measured as the minimum LVT of a simulator's partitions as reported in the snapshots taken for GVT computation. As LVT's may move forth and back quickly due to speculative computations followed by rollbacks, mean values taken over a number of snapshots provide a more realistic image of load distribution.

   Since partitions have their own LVT's, migrating them is relatively straightforward. However, as partitions may only be moved as a whole, their number must be much larger than the number of processors in order to be able to balance load exactly. Since the communication structure is fixed to a great extend by statically clustering elements into partitions, a good static partitioning policy is required.

3. To compensate for load imbalance, a set of elements is selected from the partition of a heavily loaded simulator and is moved into that of a lightly loaded one. Load is measured by observing LVT's like in 2). Compared to 2), element-wise re-partitioning allows very fine-grained redistribution of computational load. Communication relations between processes can be rearranged freely because there are no restrictions due to static partitioning.

   However, migrating elements is not as straightforward as migrating partitions which have their own LVT's. Usually, the source partition's LVT, T_{src}, is lower than the destination partition's LVT, T_{dest}. Therefore, the simulator processing the destination partition, P_{dest}, must perform a modified form of rollback to T_{src} in order to simulate un-processed events for the "new" signals. (This rollback does not require local events to be undone.) Un-processed events for signals that migrate into the destination partition have to be sent from P_{src} to

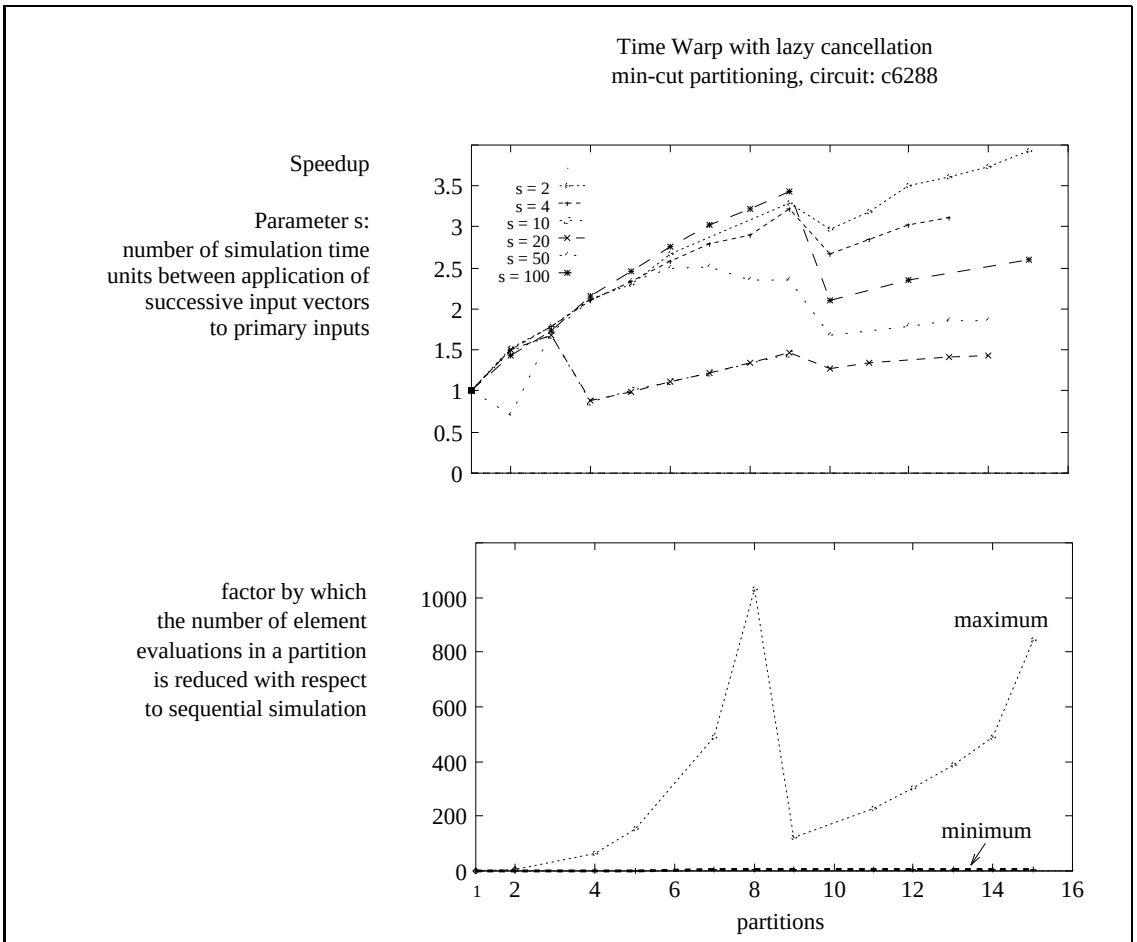


Figure 8: Time Warp: experimental results

P_{dest}. Also, rollback at P_{dest} may require the signal history for $[GVT, T_{src}]$ to be transferred to P_{dest}.

In the current version of the testbed, method 3 has been implemented. Comparison of methods 2 and 3 will be an interesting point for applications of and extensions to our testbed.

4 Experimental Results

The testbed has been implemented based on the parallel programming library MMK [7]. Run-time measurements were performed on the iPSC/2 and iPSC/860 DMMP's. Tab. 1 summarizes the communication performance of our implementation platform.

Function decomposition did not achieve significant speedup on these systems. The maximum possible speedup is between three and four. Parallelization overhead without communication costs has been measured to be nearly 50%. High communication latency makes sending short messages extremely costly. However, packing many items into one message reduces pipeline efficiency. For most of the benchmark circuits, simulation runs completely sequentially if items are combined to form messages that are long enough to achieve an acceptable effective bandwidth. (To achieve more than 80% of the maximum bandwidth, messages must be longer than about 6 kByte.)

Run-time measurements with the three parallel simulators that are based on model partitioning have shown a quite surprising result: Variations in speedups achieved for simulations of different

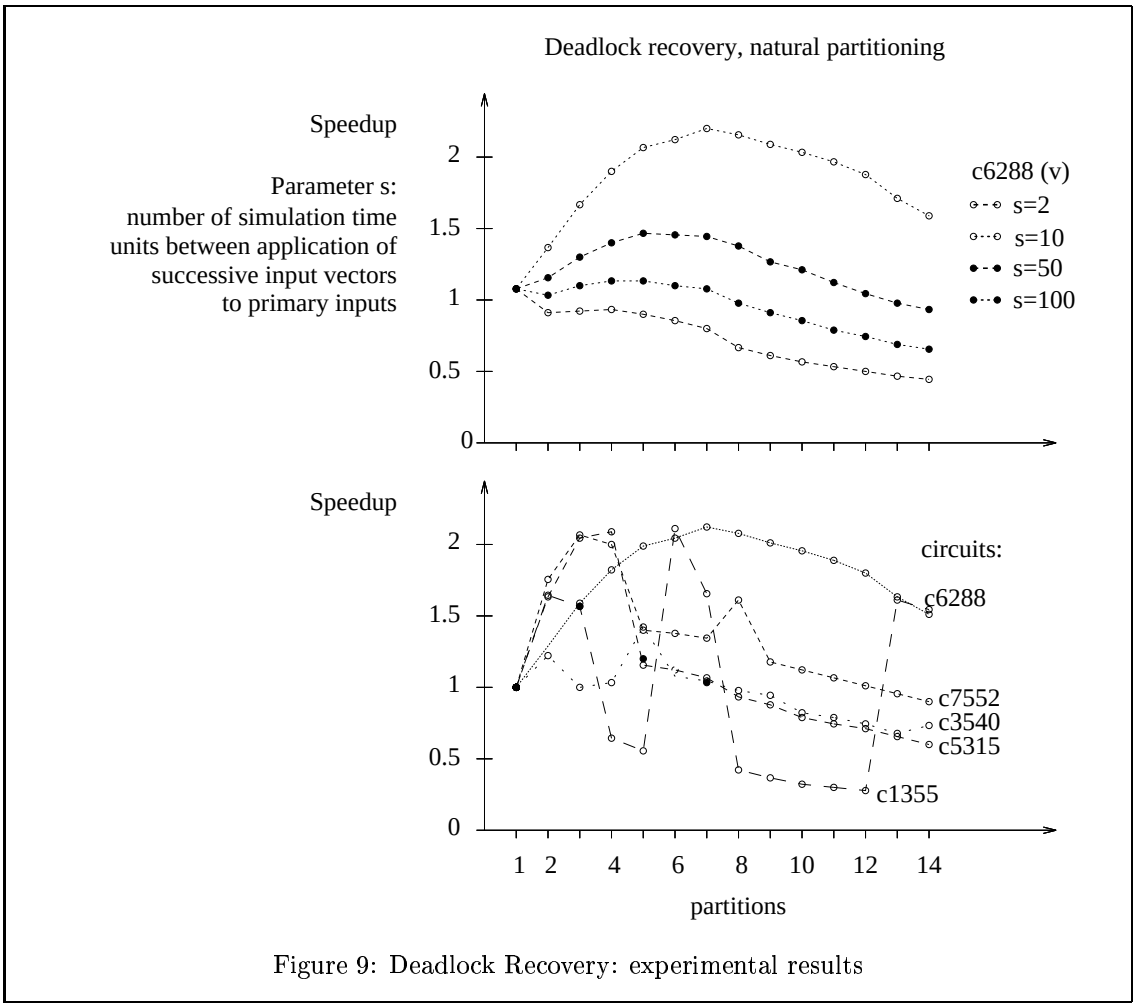


Figure 9: Deadlock Recovery: experimental results

	iPSC/2(MMK)	iPSC/860(MMK)
latency (time needed to transfer a 0-byte message)	1.95 ms	0.6 ms
maximum bandwidth (achieved for maximum length message [32kB])	1.35 MByte/sec	2.79 MByte/sec

Table 1: Communication performance of the implementation platform

circuits using the same parallelization method have been stronger than variations between different parallelization strategies. In addition, speedup often does not depend linearly on the number of nodes. Fig. 8 and 9 display some sample results for `MP-TW` and `MP-CO-DH`, respectively. Although these examples might suggest Time Warp to be superior to the conservative approach, none of the three methods that have been implemented can be clearly favored if all our run-times measurements are taken into account.

The reason for this somewhat strange result is that load has been distributed unevenly upon the processors. Although both partitioning procedures tend to produce partitions of almost equal size, the total number of element/event evaluations differs by more than an order of magnitude from node to node for most of the benchmark circuits. Fig. 10 and 11 show how activity varies in both space and time.

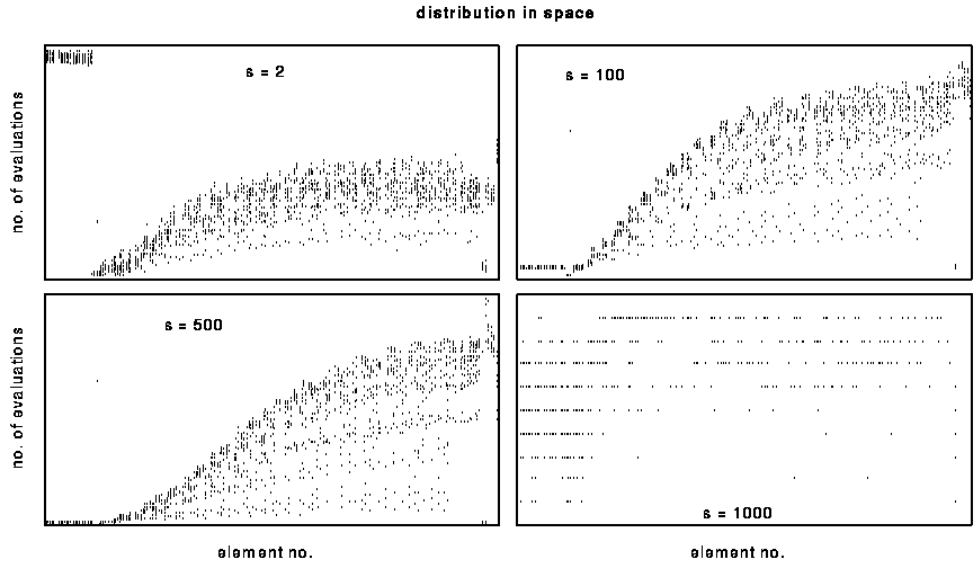


Figure 10: Distribution of activity in space

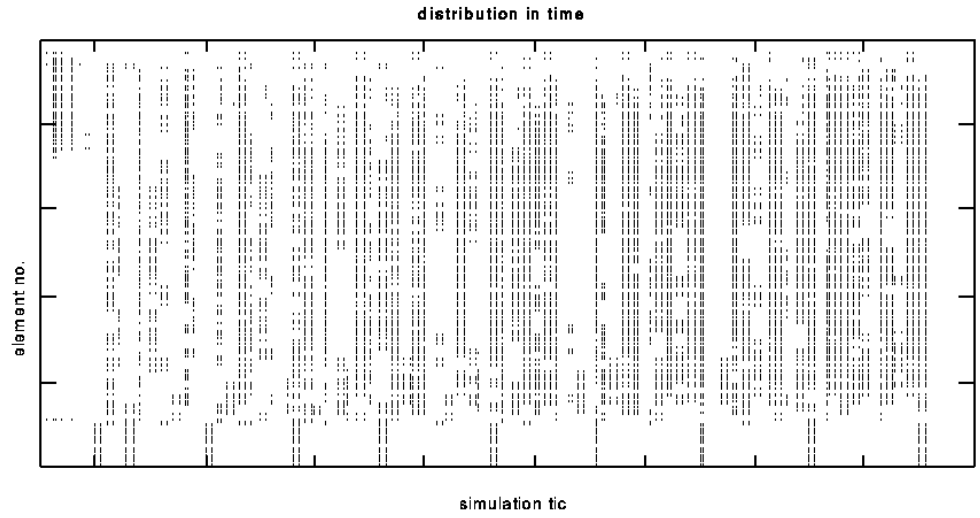


Figure 11: Distribution of activity in time

Fig. 10 displays the total number of evaluations of each element for different values of s. Parameter s is defined in fig. 8. A dot (x, y) in fig. 10 means: element x has been evaluated y times. For simplicity, no scale labels are given at the axes. As all axes begin at 0 and are scaled linearly, the diagram gives an impression of the relative variations even without scale labels. Fig. 11 shows for each simulation tic (x-axis) which elements have been evaluated.

Fig. 12 shows how Time Warp's behavior is affected by an uneven distribution of load. The diagram has been obtained from an observation of LVT's and GVT using the TOPSYS distributed monitoring system [8] which provides the best approximation to a global time base that is available

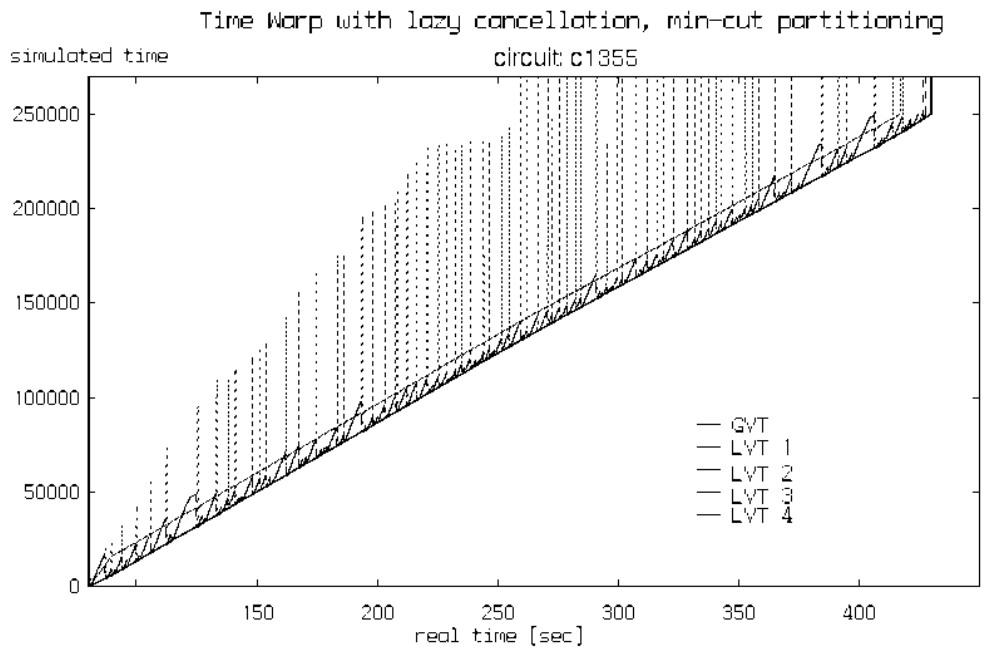


Figure 12: LVT's and GVT observed using the TOPSYS distributed monitoring system

on the IPSC DMMP's. LVT's diverge extremely due to load imbalance. While one simulator advances slowly but without rollback, others advance rapidly but roll back frequently. The first process is overloaded while the latter are only lightly loaded which results in much speculative computation that has to be rolled back later on. As a consequence of LVT's running far ahead of GVT, these processes run out of memory for large simulations because the amount of state information increases with the difference LVT–GVT. In order to be able to run large simulations, Time Warp's optimism has been limited as follows: as soon as a simulator starts to run short of memory it will be blocked whenever it tries to advance its LVT by more than some predefined amount of units ahead of GVT.

5 Conclusions and Future Work

Based on a classification of DDES methods, a test environment has been designed which allows easy implementation of a great number of parallelization strategies by providing a comprehensive library of functions and enables an unbiased evaluation of different parallelization strategies. Four parallelization have been implemented and analyzed. However, the number of run-time measurements has been limited by the instability of both the iPSC multiprocessors and the programming environment.

Since some of the results obtained have been quite unexpected, further run-time measurements should be carried out in the future including larger circuits and circuits of known function for which input stimuli can be provided that "make sense". As the function of the ISCAS benchmark circuits is not known in detail, randomly generated input sequences have been applied to their primary inputs. Although parameters for generating these sequences have been varied over a wide range, random input may have resulted in un-typical operating conditions for the circuits. Therefore, checking our results against results obtained from simulations with "real" stimuli is desirable. For this purpose, a program has been written that makes it possible to convert a number of circuits designed on a (commercial) CAE system at our laboratory, so that they can be used as work-loads

for our parallel simulator.

Given its limited potential for speedup and its sensitivity to communication latency, the function decomposition approach can be applied successfully only in combination with the model partitioning approach. In future multiprocessors where each node has several CPU's sharing a common memory, a simulator running on one node may be parallelized using function decomposition while simulation is distributed among the nodes using the model partitioning approach.

Different activity rates must be accounted for in the static partitioning procedure. Most heuristic algorithms can be modified to have individual weight factors for elements and signals. Since in the design phase of a circuit typically a number of nearly identical simulations is run in a sequence (e.g. for debugging the design), these weight factors can be easily obtained from statistics collected in a previous run at no extra cost.

Dynamic re-partitioning has proved to reduce the LVT divergence Time Warp. However, further measurements will be necessary in order to evaluate its effects comprehensively.

Besides comprehensive analysis of the parallelizations currently implemented, further parallelization strategies are to be implemented and analyzed in the future. To enlarge the set of hardware platforms where the testbed is available, it will have to be ported to one of the evolving standards for message passing programming such as PVM, P4 or MPI. Thus it will be possible to evaluate different types of multiprocessors with respect to their appropriateness for distributed discrete event simulation.

Considering other application areas of discrete event simulation will show to what extent results obtained from logic simulation can be generalized to other types of simulation problems. Parallelization of a commercial simulator designed for modeling production processes in factories just has begun.

A Appendix

A.1 Summary of DDES studies published in the literature

Ref.	paralleliz. method[(a)]	type of simul.[(b)]	hardware[(c)]	work-loads[(d)]	results[(e)]
[15]	CO-DA-NM CO-DR	QN	BBN Butterfly (SM)	up to 64 lp (synth. wl)	SU: DA: 2–11.5 (16P.); DR: 0.3–3(4P.), 1.2–5.8(8P.)
[14]	TW CO-DA CO-DR	QN	BBN Butterfly (SM)	16–64 lp	SU: TW: 6–9(16P.); CO-DA: 1.5–3(16P.); CO-DR: < 1(1–16P.);
[16]	TW	QN	SIM	1024 processes	Eff.: 0.97(16P.), 0.91(128P.), 0.89(256P.)
[17]	CO-DA-NM CO-DR-ZM	SL	iPSC (DM)	small	not asserted
[29]	TW	QN	SIM	170 processes	SU: 20–28 (43P.)
[33]	CO-DA appoint-ment pro-tocol	QN	Flex/32 (SM)	homogeneous networks	SU: 10–11 (16P.)
[34]	CO-DA	QN	Sequent Balance 2100 (SM)	5–8 log. proc.	SU: DA: $\approx$1(5P.), DR: 2–5(20P.)
[6]	GC	LG	Sequent Balance 8000 (SM)	circuits: 2608 u. 3827 elements	Eff.: 0.5–0.6

Ref.	paralleliz. method[a]	type of simul.[b]	hardware[c]	work-loads[d]	results[e]
[43] [44]	TW	BF	BBN Butterfly (SM); Caltech MarkIII Hypercube	2×18 Corps	SU: 28.6 (60P. MarkIII), 38.5 (128P. MarkIII), 36.8 (100P. BBN)
[10]	TW	LM	BBN GP1000 (SM)	up to 31680 gates, 11820 transistors	SU: 5–10(32P., transistor level), 5–20(32P., gate level)
[18]	CO-DA	LG VHDL	iPSC/2 (DM)	4–33 lp	not asserted
[21]	TW	Pool	Caltech MarkIII Hypercube (DM)	128 balls	SU: 10.5(24P.)
[31]	TW	BF	BBN Butterfly (SM)	51+46 divisions	SU: 4.5–5(10–20P.) (ZF = EB), 7(10P.), 10(20P.) (ZF = EI)
[32]	CO-DA-NM	QN	BBN Butterfly (SM)	3 topologies	SU: 32–48 (16P.) (!)
[5] [38]	TW	LG	network of workstations	ISCAS Benchmark circuits	SU: 2–5(5P.), 4–8(20P.)
[19]	TW	LM	SIM	4-,8-bit multiplier, adder	SU: TW: 1.97–3.90(4P.), 7.16(16P.), 17.46(32P.); CO-DA: 1.96–2.83(4P.), 6.22(16P.), 16.16(32P.)
[24]	GC DS-CO-DA DS-TW-AC DS-TW-LC DS-TW-DC	LB	ALLIANT fx80) (SM)	16×16 Omega-, 4×4 torus networks	SU: GC: 1, 4 (8P.); CO-DA(dp): 2.5–9(8P.); CO-DA(sp): 0.4–3(8P.) TW-AC: 3.5–6(8P.) TW-LC: 2.3–6(8P.)
[37]	GC DS-CO-DA TW	LG	Transputer (DM)	industrial circuits up to 14000 gates	SU: GC: < 1, CO-DA: 2.24(4P.), 3.42(16P.), TW $\approx$ 20% slower than conservative synchronization
[39]	TW	CN	Transputer (DM)	simple LAN model	not asserted
[45]	CO-DA-NM	LG	Transputer (DM)	circuits of 112–1664 gates	SU: 6.0–7.9(4P.)(!), 1.8(4P.), 3.0, 3.4(4P.)
[26]	CO-?	LG	Transputer (DM)	72 elements	strong impact of partitioning
[2]	TW	LG VHDL	network of workstations	14–38 log. proc.	analysis of different optimization techniques and their effects
[35]	CO-DA-NM	TK	Convex C3400 (SM)	27 switches	SU: 1.4–1.8(2P.)

abbreviations:

[a] see fig. 5; for all parallelizations described here: AP = MP (omitted in the table); for all parallelizations except [6] CN = LC.
sp: static partitioning; dp: dynamic partitioning

[b] QN : queueing network simulation; LG : gate level logic simulation; LB : behavioral level logic simulation; LM : multi-level logic simulation; BF : battlefield simulation; CN : computer network simulation; SL : simulation library; TK : simulation of telecommunication networks

[c] SM : shared memory multiprocessors; DM : distributed memory multiprocessors
SIM: simulation of a multiprocessor

[d] wl: work-load; lp: logical process(es)

[e] SU : speedup; Eff: efficiency
DA 2–11.5 (16P.): deadlock avoidance: speedups between 2 and 11.5 on 16 processors

approach		function
function decomposition		decomposition into six processes
		element evaluation: one- and two-phase approach
		communication mechanism: number of items per message adjustable as a parameter
		instrumentation for run-time statistics
model partitioning		interface to circuit partitioning
		static partitioning: natural partitioning and min-cut (generalization of Fiduccia/Mattheyses' algorithm by Vijan [13, 40])
	conservative approach	modified control structure for conservative synchronization
		deadlock avoidance by time requests
		deadlock recovery with the vector method: circulating control vector, parallel vector method
		two options for the definition of external events
		instrumentation for run-time statistics
	Time Warp	rollback mechanism
		optimized incremental state saving
		aggressive and lazy cancellation
		optimized re-simulation after rollback
		dynamic re-partitioning
		instrumentation for run-time statistics

Table 2: library of functions provided by the test environment

References

[1] Markus Abt. Parallelisierung eines ereignisgetriebenen Logiksimulators mittels eines konservativen, verklemmungsbehandelnden Synchronisationsprotokolls. Diplomarbeit, Technische Universität München, Institut für Informatik, München, August 1993.

[2] Sandeep Aji, Avinash C. Palaniswany, and Philip A. Wilsey. Interactions of Optimizations to a Time Warp Synchronized Digital System Simulator. In *Modelling and Simulation ESM 93, Proceedings of the 1993 European Simulation Multiconference*, pages 593–597, Lyon, June 1993. SCS.

[3] W.L. Bain and D.S. Scott. An algorithm for time synchronisation in distributed discrete event simulation. In *Distributed Simulation*, 1988.

[4] H. Bauer and C. Sporrer. Distributed Logic Simulation and an Approach to Asynchronous GVT-Calculation. In *Proceedings of the 1992 SCS Western Simulation Multiconference on Parallel and Distributed Simulation (PADS92)*, pages 205–209, Newport Beach, California, January 1992.

[5] Herbert Bauer, Christian Sporrer, and Thomas H. Krodel. On distributed Logic Simulation Using Time Warp. In *IFIP TC 10/WG 10.5 International Conference on Very Large Scale Integration, VLSI 91*, pages 4.1.1.1–4.a.1.10, Edinburgh, Scotland, August 1991.

[6] G. Beihl. A Shared-Memory Multiprocessor Logic Simulator. In *Eighth Annual International Phoenix Conference on Computers and Communications*, pages 26–28, Wyndham Paradise Valley Resort, Scottsdale, Arizona, March 1989.

[7] T. Bemmerl, A. Bode, T. Ludwig, and S. Tritscher. MMK - Multiprocessor Multitasking Kernel (User's Guide and User's Reference Manual). SFB-Bericht 342/26/90 A, Technische Universität München, Institut für Informatik, August 1990.

[8] T. Bemmerl, R. Lindhof, and T. Treml. The Distributed Monitor System of TOPSYS. In H. Burkhart, editor, *Proceedings of CONPAR90 VAPP IV*, volume 457 of *Lecture Notes in Computer Science*, pages 756–765, Zürich, Schweiz, 1990. Springer-Verlag.

[9] Tom Blank. *A Survey of Hardware Accelerators Used in Computer-Aided Design*, pages 90–108. IEEE Computer Society Press/North Holland, 1984.

[10] Jack Vedder Jr. Briner. *Parallel Mixed-Level Simulation of Digital Circuits Using Virtual Time*. PhD Dissertation, Duke University, Durham, NC 27706, August 1990.

[11] K. Mani Chandy and Leslie Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[12] K.M. Chandy and J. Misra. Asynchronous Distributed Simulation via a Sequence of Parallel Computations. *Communications of the ACM*, 24(11), April 1981.

[13] C.M. Fiduccia and R.M. Mattheyses. A Linear-Time Heuristic for Improving Network Partitions. In *19th Design Automation Conference*, pages 175–181, 1982.

[14] Richard M. Fujimoto. Performance Measurements of distributed simulation strategies. In *Distributed Simulation 1988*, pages 14–20, 1988.

[15] R.M. Fujimoto. Lookahead in Parallel Discrete Event Simulation. In *International Conference on Parallel Processing*, 1988.

[16] Jr. Gilmer, John. B. An assessment of "'Time warp"' parallel discrete event simulation algorithm performance. In *Distributed Simulation*, pages 45–49, 1988.

[17] T.C. Hartum and B.J. Donlan. HYPERSIM: Distributed Discrete-Event Simulation on an iPSC. In *The Third Conference on Hypercube Concurrent Computers and Applications*, volume I, pages 745–747, 1988.

[18] T.C. Hartum, A. Lee, and J. Sartor. Parallel Simulation Speedup on the iPSC/2. Intel Supercomputer Users' Group Proceedings, Aug. 17-29, 1990.

[19] Friedrich Hoppe. *Ein Verfahren zur Synchronisation und Lastverteilung f"ur die parallele Mehrebenenlogiksimulation.* Dissertation, Technische Universität Berlin, February 1991.

[20] D. Jefferson. Virtual Time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[21] D. Jefferson, B. Beckman, F. Wieland, J. Blume, M. Di Loreto, P. Hontalas, P. Reiher, K. Sturdevant, J. Tupman, J. Wedel, and H. Younger. The Status of the Time Warp Operating System. In *The Third Conference on Hypercube Concurrent Computers and Applications*, volume I, Architecture, Software, Computer Systems and General Issues, pages 738–744, 1988.

[22] D. Jefferson and H. Sowizral. Fast Concurrent Simulation Using the Time Warp Mechanism, Part I: Local Control. Rand Note N-1906-AF, Rand Corporation, Santa Monica, CA., 1982.

[23] Marc Köhler. Verklemmungsfreie Parallelisierung eines ereignisgetriebenen Logiksimulators mittels eines konservativen Synchronisationsverfahrens. Diplomarbeit, Technische Universität München, Institut für Informatik, München, August 1992.

[24] Pavlos Konas and Pen-Chung Yew. Parallel Discrete Event Simulation on Shared-Memory Multiprocessors. In Alan H. Rutan, editor, *Annual Simulation Symposium*, volume 24 of *Annual Simulation Symposium*, pages 134–148, New Orleans, Louisiana, April 1991. ACM, IEEE Computer Society SCS, IMACS, IEEE Computer Society Press.

[25] T.H. Krodel and K. Antreich. An Accurate Model for Ambiguity Delay Simulation. In *27th ACM/IEEE Design Automation Conference*, pages 122–127, 1990.

[26] Phillip Lanchès and Utz G. Baitinger. A Parallel Evaluation Environment for Distributed Logic Simulation. In *Modelling and Simulation ESM 92, Proceedings of the 1992 European Simulation Multiconference*, pages 465–469, San Diego, CA, June 1992. SCS International.

[27] Y.-B. Lin and E.D. Lazowska. Comparing Synchronization Protocols for Parallel Logic-Level Simulation. In *Proceedings of the 1990 International Conference on Parallel Processing*, volume III, pages 223–227, 1990.

[28] Y.-B. Lin and E.D. Lazowska. Determining the Global Virtual Time in a Distributed Simulation. In *Proceedings of the 1990 International Conference on Parallel Processing*, volume III, pages 201–209, 1990.

[29] Greg Lomonow, John Cleary, Brian Unger, and Darrin West. A performance study of Time Warp. In *Distributed Simulation*, pages 50–55, 1988.

[30] Friedemann Mattern. *Verteilte Basisalgorithmen*, volume 226 of *Informatik-Fachberichte*. Springer-Verlag, Berlin, 1989.

[31] Katherine L. Morse. Parallel Distributed Simulation in ModSim. In *International Conference on Parallel Processing*, volume III, pages III–210–III–217, 1990.

[32] J. Nathaniel, D. Mannix, and T. Shaw, W. Hartum. Distributed Discrete-Event Simulation Using Null Message Algorithms on Hypercube Architectures. *Journal of Parallel and Distributed Computing*, 8:349–357, 1990.

[33] D.M. Nicol. Parallel Discrete Event Simulation of FCFS Stochastic Queueing Networks. *SIGPLAN Notices*, 23(9), September 1988.

[34] Daniel A. Reed and Allen D. Malony. Parallel discrete event simulation: The Chandy-Misra Approach. In *Distributed Simulation 1988*, pages 8–13, 1988.

[35] Matti Salmi, Jarmo Harju, and Jari Porras. A Chandy-Misra Parallel Simulation Environment for the Simulation of GSM Mobile Communication Network. In *Modelling and Simulation ESM 93, Proceedings of the 1993 European Simulation Multiconference*, pages 575–579, Lyon, June 1993. SCS.

[36] B. Samadi. *Distributed Simulation, Algorithms and Performance Analysis.* Technical Report, University of California, Los Angeles, (UCLA), 1985.

[37] Jürgen Sang and Manuela Sang. Untersuchung von Algorithmen zur verteilten ereignisgesteuerten Simulation. In Djamshid Tavangarian, editor, *Simulationstechnik, 7. Symposium*, volume 4 of *Fortschritte in der Simulationstechnik*, pages 248–252, Hagen, September 1991. ASIM, Vieweg.

[38] Christian Sporrer and Herbert Bauer. Corolla Partitioning for Distributed Logic Simulation of VLSI-Circuits. In *7th Workshop on Parallel and Distributed Simulation (PADS)*, San Diego, USA, May 1993.

[39] Fannie Tallieu and Frank Verboven. Using Time Warp for Computer Network Simulations on Transputers. In Alan H. Rutan, editor, *Annual Simulation Symposium*, volume 24 of *Annual Simulation Symposium*, pages 112–117, New Orleans, Louisiana, April 1991. ACM, IEEE Computer Society SCS, IMACS, IEEE Computer Society Press.

[40] Gopalakrishnan Vijayan. Min-Cost Partitioning on a Tree Structure and Applications. In *26th ACM/IEEE Design Automation Conference*, pages 771–774, 1989.

[41] Ricarda Weber. Parallelisierung eines ereignisgetriebenen Logiksimulators durch Aufteilung des Algorithmus in parallele Prozesse. Diplomarbeit, Technische Universität München, Institut für Informatik, München, May 1992.

[42] Holger Weitlich. Parallele Logiksimulation nach der Time-Warp-Methode auf einem Multiprozessorsystem mit verteiltem Speicher. Diplomarbeit, Technische Universität München, Institut für Informatik, München, August 1992.

[43] F. Wieland, L Hawley, A. Feinberg, M. di Loreto, L. Blume, J. Ruffels, P. Reiher, B. Beckman, P. Hontalas, and S. Bellenot. The Performance of a Distributed Combat Simulation with the Time Warp Operating System. *Concurrency: Practice and Experience*, 1(1):35–50, September 1989.

[44] F. Wieland and D. Jefferson. Case Studies in Serial and Parallel Simulation. In *International Conference on Parallel processing*, volume III, pages 255–258, 1989.

[45] Kenneth R. Wood. Distributing gate-level digital simulation over arrays of transputers. *Concurrency: Practice and Experience*, 3(4):367–379, August 1991.