A Portable and Extendible Testbed for Distributed Logic Simulation *

Peter Luksch
Institut für Informatik; Technische Universität München; Germany
e-mail: luksch@informatik.tu-muenchen.de

Abstract

A flexible test environment is presented that allows for different methods of parallelizing discrete event simulation to be evaluated in a uniform environment. The testbed is portable and can be easily extended in order to analyze new parallelization methods. Run-time measurements performed on the iPSC distributed memory multiprocessors show efficiencies of up to 50 percent for both conservative and optimistic synchronization.

1 Introduction

In recent years, simulation has become an indispensable tool in VLSI design. As system complexity increases, simulation is used not only for validation purposes but also helps to design system hardware and software in parallel and to evaluate performance of different design alternatives. 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 run 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 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.

Parallel simulation efficiency depends on more than just the parallelization method employed. Properties of the simulation problem and parameters of the target architecture play an important role, too. For an unbiased comparison of different methods, run-time measurements have to be made under uniform conditions.

The goal of the testbed presented in this paper is to enable a detailed analysis of a great number of parallelization methods under uniform conditions.

According to the distribution of functions and data structures and according to the type of process synchronization the variety of methods for parallelizing discrete event simulation can be subdivided into a small number of classes representing fundamentally different approaches to parallelization. The classification scheme which has been used to make a representative selection of parallelizations for implementation in our testbed, is summarized in Fig. 1 (see [5] for more details).

2 A Testbed for Distributed Simulation

The main objective in the design of our testbed has been to provide a platform to implement and analyze a large number of distributed simulation strategies under uniform conditions. Besides the parallelization strategy the main factors that contribute to parallel simulation behavior are the target architecture and the field of application where discrete event simulation is used. Together with our primary topic of interest, parallelization strategies, these environment parameters span up a three-dimensional space as illustrated in Fig. 2. Our testbed has been designed to cover as large a portion of it as possible.

In order to eliminate the influence of the simulation application, some simulator had to be selected as the basis for the testbed. Given the importance of gate-level simulation in VLSI design and its ever increasing demands for computational power, we have chosen a gate-level logic simulator. It implements most of today's state-of-the-art techniques in the modeling of digital systems [3]. Thus, properties of the simulation application in our testbed are very close to those found in commercial CAE systems. Because of its computational complexity, computer-aided VLSI design is likely to become the first production-use application of distributed simulation.

Portability has been an important requirement in the design of our test environment. As implementation platforms we consider the whole class of scalable MIMD multiprocessors, especially distributed memory

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

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission. (c) 1994 ACM 0-89791-687-5/94/0009 3.50

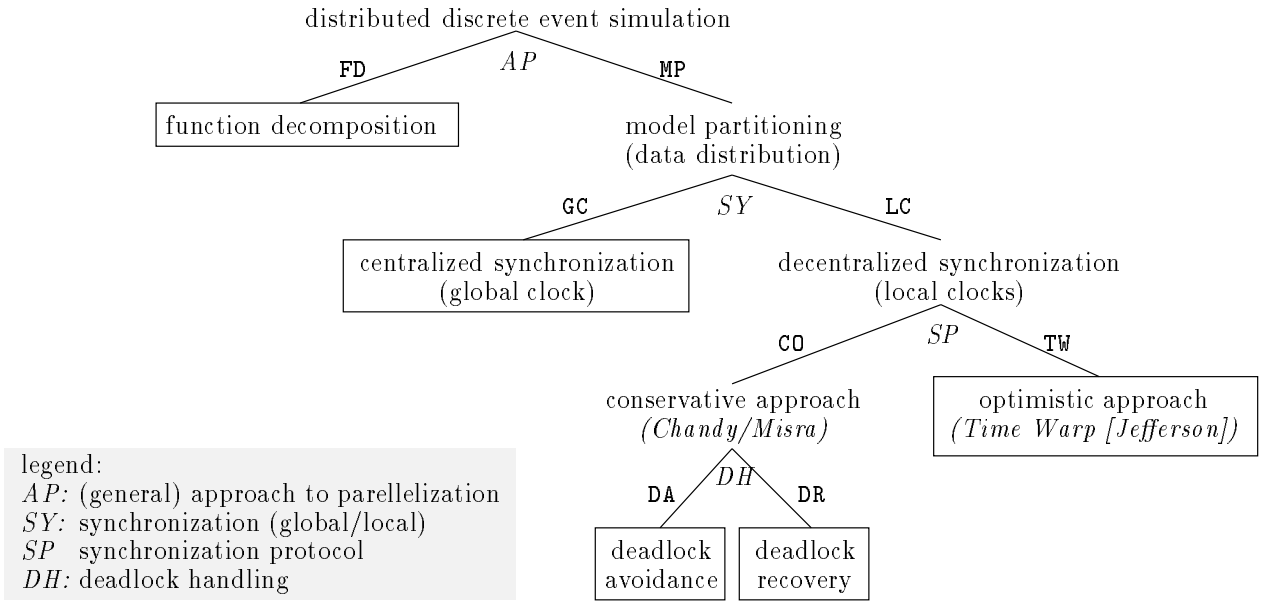

Figure 1: Classification of methods for distributed discrete event simulation

computers. The nodes of the multiprocessor are assumed to be virtually fully connected, without storing and forwarding messages on intermediate nodes. Parallelization is done explicitly on a medium to coarse grain level using the message passing model. In the implementation of parallelization strategies we have restricted ourselves to the basic communication primitives which can be found in nearly all of today's multiprocessor's programming models. Thus, a high degree of portability has been achieved despite the fact that until now there is no generally accepted standard for message passing programming.

Based on the classification scheme introduced in section 1, four parallelizations of the logic simulator have been implemented within the test environment. Each of them belongs to a different leaf in the tree of Fig. 1. The parallel simulators have been instrumented to collect detailed run-time statistics including parameters specific to each approach, such as the number of rollbacks in Time Warp.

On the one hand, implementation of a representative selection of parallelization strategies allows the main approaches to distributed simulation to be compared under uniform conditions. On the other hand, a library of functions is provided which forms the basis for a flexible test environment. It is summarized in Table 1. Using the library, a great variety of parallelizations can be analyzed with minimal implementation effort. In the following sections, the four parallelizations that have been implemented are presented.

Function Decomposition 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 [3]) are computed for current events; Event execution: compute new signal values, update signal list, determine fan-out elements; Element execution.

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 and min-cut partitioning.

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

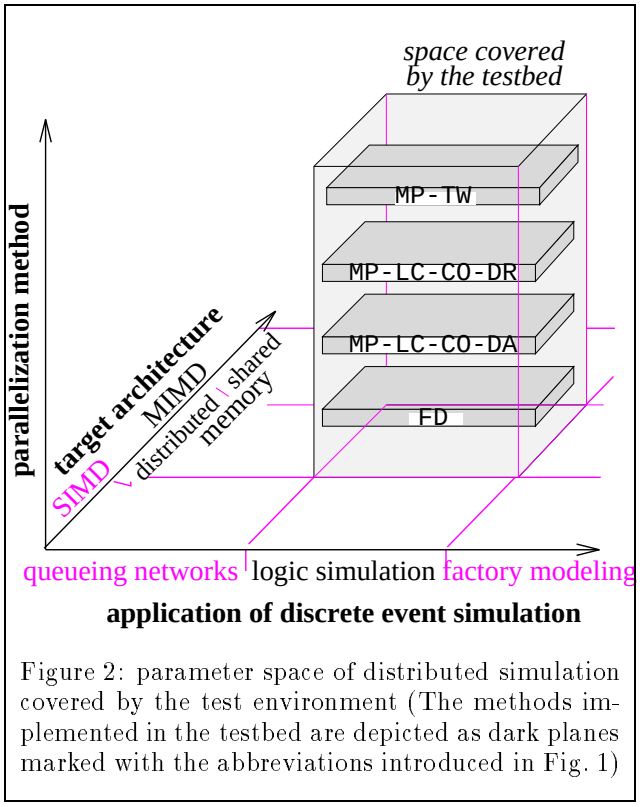

Figure 2: parameter space of distributed simulation covered by the test environment (The methods implemented in the testbed are depicted as dark planes marked with the abbreviations introduced in Fig. 1)

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 [8])
	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 1: library of functions provided by the test environment

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 in the conservative approach simulators remain in the suspended state unnecessarily 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.

Deadlock Avoidance with Time Requests Based on an algorithm proposed by Bain and Scott [1], a conservative protocol has been implemented that avoids deadlocks by means of time requests. A time request (T_i, S_i) is issued by a simulator S_i to all S_j with $l_i[j] < T_i$, i.e. all predecessors S_j that prevent S_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, S_i) asks the the receiving process S_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 channel time up to date. Otherwise, the request is queued and the requesting process is left waiting for the reply. If there are any predecessors S_k of S_j with $l_j[k] < T_i$, S_j sends a request (T_i, S_i) to these S_k. Note that the simulator S_i *originating* the request is entered as the second component, not the sender S_j. Note, that the request has S_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 S_l receives a request (T_i, S_i) from some process S_j while it has an identical request in its queue. Then an RYES ("reflected yes") reply is sent to S_j irrespective of the current local simulation time T_l. S_l has, however, to keep in mind that an RYES reply has been given to S_j. If later on an event e_1 with time stamp $t_1 < T_i$ is sent to S_j, the queued copy of request (T_i, S_i) which had been received, say, from S_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 S_i. This is the only situation where NO replies are generated.

A request by S_i is completed if all predecessors to which the request has been sent have sent their replies. If the request has been originated by S_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, S_i generates a new time request.

If a request $(T_k,\ S_k)$ has been completed which had been originated by another process, S_k, a reply is sent to the process S_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.

Deadlock Recovery with the Vector Method 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 [6]. 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 S_i has a (local) vector $\vec{L_i}$. If S_i sends a message to S_i, $L_i[j]$ is incremented by one; if S_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 S_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 S_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 circulation 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.

Time Warp In the 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 to 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 [7]. 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 [4] 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 [2] 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 complex elements.

[1] By re-simulation, we mean the renewed simulation of the rolled-back period of simulated time.

[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.

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, the load may be distributed unevenly if elements have different rates of activity or if activity distribution in the circuit changes over time.

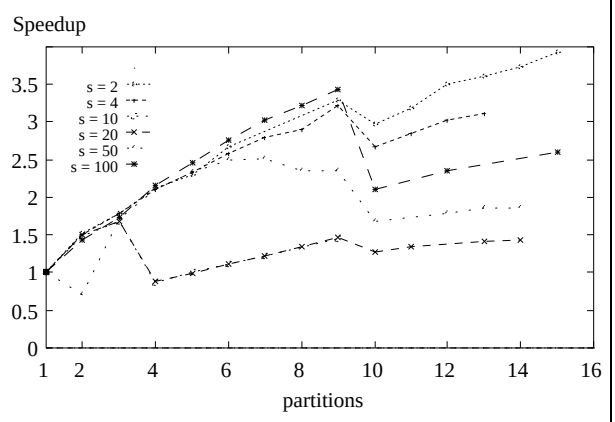

Figure 3: Time Warp: experimental results

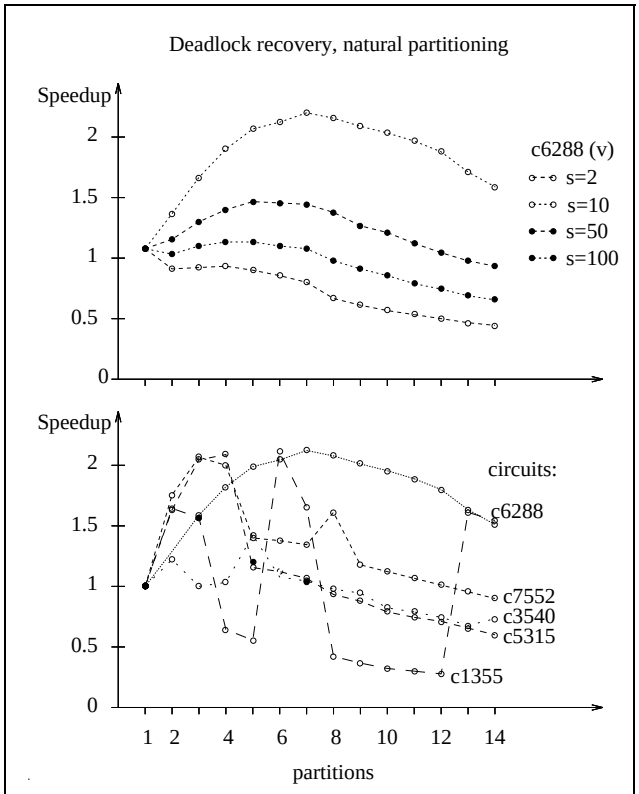

Figure 4: Deadlock recovery: experimental results

3 Experimental Results

The testbed has been implemented based on the machine-independent parallel programming library MMK which has been developed in our research group and currently is available for the iPSC/2, iPSC/860 and networks of Sun Sparc workstations. Run-time measurements have been performed on the iPSC distributed memory multiprocessors using the ISCAS-89 benchmark circuits as workloads.

Function decomposition has a theoretical speedup of 3–4. Parallelization overhead (without communication cost) has been measured to be less than 50%. Nevertheless, no speedup has been observed in our run-time measurements because of the implementation platform's high communication latency which is about 600 μs for MMK on the iPSC/860 and 2 ms on the iPSC/2. For function decomposition to be efficient communication latency must be low or circuits must be very large so that data exchanged between pipeline stages can be packed in long messages while keeping the pipeline busy.

In our measurements, performance of the parallelizations based on model partitioning showed to depend strongly on the circuit being simulated an the stimuli being applied to its primary inputs as depicted in Figures 3 and 4 for some examples. Maximum speedups are about half the number of simulators involved in the simulation. However, in many cases no clear relationship can be established between the number of simulators and the achieved speedup.

As the function of the ISCAS benchmarks is not known, random sequences of input vectors have been applied to the circuits at different frequencies. The parameter s in Figures 3 and 4 denotes the number of simulation time units between two successive input vectors.

The examples shown suggest that Time Warp outperforms conservative synchronization with deadlock recovery. However, our measurements do not clearly favor any of the three approaches that have been analyzed. Circuit topology and stimuli have impacted performance much more than the method of synchronization did for both of our static partitioning procedures. Run-time statistic revealed the reason for this rather unexpected behavior: Load has been distributed very unevenly among the simulators. This is illustrated in the lower diagram of Fig. 3 which shows the minimum an maximum reduction in the number of element evaluations per partition with respect to sequential simulation for a min-cut partitioned circuit with almost

equally sized partitions. Further analysis has shown that activity rates vary by several orders of magnitude from element to element. Also the "center of activity" within a circuit tends to move during simulation.

In Time Warp, uneven load distribution has resulted in an extreme divergence of LVT's. GVT approximation is sufficiently close. One simulator increases its LVT without rollbacks, another one proceeds at nearly the same rate but with frequent and short rollbacks. The other simulators periodically run far ahead of GVT and then rollback over long periods of simulated time. As a result of being far ahead of GVT, the latter processes use up all their memory for state saving if large circuits are simulated. In order to get such simulations finished, Time Warp's optimism had to be limited by suspending simulators which are running short of memory if they are more than a predefined amount of simulated time ahead of GVT.

4 Conclusion and Future Work

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".

From our measurements performed so far, the following conclusions can be drawn:

• Given its limited potential for speedup and its sensitivity to communication latency, the function decomposition approach can be applied sucessfully 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 weighting 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 proven to reduce the LVT divergence in Time Warp. However, further measurements will be necessary in order to evaluate its effects comprehensively.

Topics for future research using the testbed as a basis are:

• Implementing and analyzing optimizations of the existing parallelizations as well as new parallelization strategies.

• Porting the testbed to a widespread programming model, e.g. PVM or P4. Enlarging the set of hardware platforms where the testbed is available will allow us to evaluate different 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.

References

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

[2] 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, Jan. 1992.

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

[4] Y.-B. Lin and E. 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.

[5] P. Luksch. *Parallelisierung ereignisgetriebener Simulationsverfahren auf Mehrprozessorsystemen mit verteiltem Speicher*. Verlag Dr. Kovač, Hamburg, 1994.

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

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

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