Parallel Discrete Event Simulation

Benno Overeinder Bob Hertzberger Peter Sloot

Department of Computer Systems
University of Amsterdam
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands
e-mail: overeind@fwi.uva.nl

24 April 1991

Abstract

In simulating applications for execution on specific computing systems, the simulation performance figures must be known in a short period of time. One basic approach to the problem of reducing the required simulation time is the exploitation of parallelism. However, in parallelizing the simulation new problems arise. Due to the distributed generation of events causality errors can occur, as a result the sequence in which to process the events is essentially indeterminated.

In this paper we present a model to analyse the inherent parallelism of a simulation, together with a survey of existing strategies to perform the simulation in parallel. Some extensions to this model are discussed, resulting in reliable evaluation of the effectiveness of these strategies.

1 Introduction

In the Parallel Scientific Computing Working-group at the University of Amsterdam, we are interested in the execution performance of classes of applications on classes of computing systems. We distinguish the following levels that are involved in performance prediction: application, general abstract machine, simulation language, and discrete event simulator. Each level is supported by the level underneath. In this way the efficiency of a level is partially determined by the supporting level, thus imposing severe constraints to the simulator. Especially if the performance figures are iteratively used to optimize the application, the effectiveness of the simulator is of vital importance.

Large discrete event simulations are known to consume enormous amounts of time on sequential machines. One basic approach to reduce the required simulation time is the exploitation of parallelism. A major drawback however, is the inherent complexity of this type of simulation since the notion of global time does not easily map on a parallel computer. Sophisticated clock synchronization algorithms are required to ensure that cause-and-effect relationships are correct reproduced by the simulator.

The idea of parallel simulation—in literature also indicated by distributed simulation—was first proposed by K.M. Chandy and independently by R.E. Bryant. Papers by Chandy and Misra [Cha79], and Bryant [Bry77] contain basic ideas of parallel simulation, the problem of deadlock and schemes for deadlock resolution, detection and recovery [Cha81]. Alternative schemes proposed by D.R. Jefferson are based on the concepts of Virtual Time [Jef85].

This paper is structured in the following way. Section 2 gives an introduction to discrete event simulation. In section 3 a parallel view to the sequential simulation is proposed, and various methods for parallel simulation are described together with a discussion on their effectiveness. Finally, in section 4 an evaluation of these methods and some suggestions for further research are presented.

2 Concepts of Discrete Event Simulation

Modelling and simulation can be characterized as the complex of activities associated with constructing models of real world systems and simulating them on a computer.

Essential to every model is the time base on which events occur. Accordingly, models can be classified depending on their temporal behaviour [Zei76]. A model is a *continuous time* model when time flows smoothly and continuously. A model is a *discrete time* model if time flows in jumps of some specified time unit.

A second classification can be based on the range sets of a model's descriptive variables. The model is a *continuous state* model if the range of the descriptive variables can be represented by the real numbers. The model is a *discrete state* model if its variables only assume discrete values.

Continuous time models can be further divided into *differential equation* and *discrete event* classes. A differential equation model is a continuous time–continuous state model where changes in state occur smoothly and continuously in time. In a discrete event model, even though time flows continuously, state changes can occur only at countable points in time—i.e., time jumps from one event to the next, and these events can occur arbitrarily separated from each other.

2.1 Discrete Event Simulation

The concept of a system and a model of a system were already used in the definition of the classes of simulation. These concepts need to be specified in order to develop a framework for the design of a discrete event model of a system. The major concepts are:

System A collection of entities that interact together over time to accomplish one or more goals.

Model An abstract representation of the system under consideration, usually containing logical and/or mathematical relationships that describe the behaviour of the system.

System state A collection of variables that contain all the information necessary to describe the system at any time.

Entity Any object or component in the system that requires explicit representation in the model.

Attributes The properties of a given entity.

Event An instantaneous occurrence that may change the state of the system.

Activity A duration of time of specified length during which entities engage some operation.

Process A sequence of events ordered in time. These events must be logically connected, involving the same entity.

To illustrate these concepts, we consider a bank. In the dynamics of a bank, customers might be one of the entities, the balance in their accounts might be an attribute, and making deposits might be an activity. Possible state variables are the number of busy tellers, the number of customers waiting in line or being served, and the arrival time of the next customer. The arrival of a customer as well as the completion of service of a customer are possible events.

Every discrete event simulation contains a state variable called the *simulation clock* to model the flow of time. Simulated time is advanced from the time of the current event to the time of the next scheduled event; thus skipping periods of inactivity. Future events are stored in a calendar that contains the time and the type of all scheduled events, usually in chronological order. The nature of the routine depends on the world view used in the model. Let us therefore consider some different world views relevant to discrete event simulation.

2.2 World Views

All simulations contain an executive routine for the management of the calendar and clock, i.e., the sequencing of events and driving of the simulation. This executive routine fetches the next scheduled event, advances the simulation clock and transfers control to the appropriate routine. The operation routines depend on the world view, and may be events, activities, or processes.

A world view is the point of view from which the modeller sees the world or the system to be modelled. Most of the discrete event simulations use one of the three following perspectives [Hoo86]: *event scheduling*, *activity scanning*, or *process interaction*.

In *event scheduling* each type of event has a corresponding event routine. The executive routine processes a time ordered calendar of event notices to select an event for execution. Event notices consist of a time stamp and a reference to an event routine. Event execution can schedule new events by creating an event notice and place it at the appropriate position in the calendar. The clock is always updated to the time of the next event, the one at the top of the calendar.

In the *activity scanning* approach a simulation contains a list of activities, each of which is defined by two events: the start event and the completion event. Each activity contains test conditions and actions. The executive routine scans the activities for satisfied time and test conditions and executes the actions of the first selectable activity. When execution of an activity completes, the scan begins again.

The *process interaction* world view focuses on the flow of entities through a model. This strategy views systems as sets of concurrent, interacting processes. The behaviour of each class of entities during its lifetime is described by a process class. Process classes can have multiple entries and exits at which a process interacts with its environment. The executive routine uses a calendar to keep track of forthcoming tasks. However, apart from recording activation time and process identity, the executive routine must also remember the state in which the process was last suspended.

Evidently, large discrete event simulations, using one of these three world view strategies, put extreme computational demands on sequential computers. Intuitively, the process interaction world view seems to be attractive as a starting point in our effort to the parallelization of the simulation. The modeller perceives the simulation already as a set of concurrent objects interacting with each other by well-defined communication. Besides, parallel simulation is interesting because it represents a problem domain that often contains substantial amounts of inherent parallelism (e.g., see [Liv85]).

In the following section a parallel view to a sequential execution will be presented in order to analyse the inherent parallelism of the simulation. Next the problems involved in parallel execution and the methodologies to circumvent these problems are described.

3 From Sequential to Parallel Discrete Event Simulation

3.1 The Average Parallelism Measure

If we have made the decision to do the simulation in parallel, there are some fundamental questions to be answered. What is the parallelism inherent to the simulation? How much benefit do we expect from doing things in parallel? And, once the job is done, how well did we perform this?

One very interesting characterization of the simulation that can be used to answer these questions is the *average parallelism*. Average parallelism can be defined in two equivalent ways:

1. The ratio of the total service time required to process events, to the length of the critical path through the execution of the simulation.

2. The speedup figures, if a hypothetical machine contains an unbounded number of available processors and zero synchronization overhead.

As a consequence of the second definition, the average parallelism figure should be regarded as an upper bound to the speedup that can be achieved.

To reveal the average parallelism inherent to a simulation, we have implemented a tool to analyse a sequential simulation run and extract the average parallelism [Ove91]. A system model is defined to express the parallelism explicitly and consists of a software component and a hardware component. The software component is a graph representing the execution of a sequential simulation. The hardware component of our system model reflects our focus on the parallelism inherent to a simulation, and makes assumptions of ideal hardware.

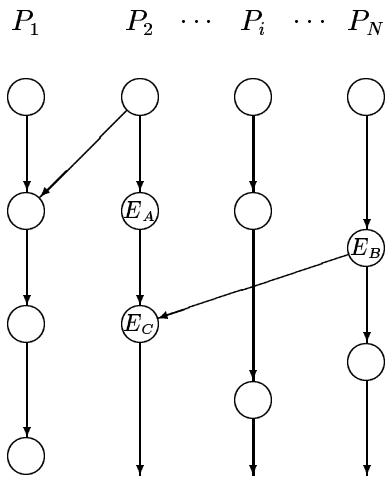

Figure 1: A program activity graph.

The execution of a simulation is represented by an acyclic directed graph (see Fig. 1). Each vertex of the graph corresponds to an event occurring in the simulation. Precedence constraints exist among the events, modelling the chronological order of events. These precedence constraints are modelled by the arcs of the graph: an arc from vertex E_A to vertex E_C means that event E_C cannot occur (or be executed) before event E_A is processed. Two types of arcs are distinguished: *intra-process arcs* and *inter-process arcs*. Intra-process arcs are precedence constraints between events that occur within the same process (e.g., arc between vertex E_A and E_C in Fig. 1). The intra-process arc denotes an independent unit of sequential work inside a process. We can consider inter-process arcs as precedence constraints between events that occur in different processes (e.g., arc between vertex E_B and E_C). These inter-process arcs represent synchronization requirements achieved by some communication primitive.

The hardware component of the system is modelled as an infinite number of identical processors, each of unit speed. The synchronization between processors has zero overhead and the entire computer is devoted to one single task.

A sequential run of the simulation generates an acyclic directed graph of events with their precedence constraints. When every process in the simulation is assigned to a different processor (i.e., one process to one exclusive processor), all *intra*-process dependent events occur at the same exclusive processor and all *inter*-process dependent events occur at different processors. As a consequence, the *intra*-process arc denotes an independent unit of sequential work on a processor, whereas the *inter*-process arc represents synchronization requirements between processors. Furthermore, the execution times of the independent units of work, measured during the sequential run, are assigned to the *intra*-process arcs and the zero synchronization costs to the *inter*-process arcs. In this way the graph is reduced to a representation of the

execution of the simulation on a hypothetical machine. The total amount of time required to process the events is equal to the sum of all the costs in the graph and the critical path through the execution of the simulation is now represented by the longest path in the graph.

Eager et al. [Eag89] use the average parallelism measure to express lower bounds on speedup and efficiency, and on the incremental benefit and cost of allocating additional processors. It is our opinion that average parallelism can be applied as a measure in the evaluation of effectiveness of various methods in parallel simulation. In other words, how much of the parallelism that is inherent to the simulation is actually exploited?

3.2 The Fundamental Problem in Parallel Discrete Event Simulation

We are especially interested in parallelization of asynchronous system simulation, where events are not synchronized by a global clock, but rather occur at irregular time intervals. In these simulations few events occur at any single point in simulated time and therefore parallelization techniques based on synchronous execution using a global simulation clock performs poorly. Concurrent execution of events at different points in simulated time is required, but this introduces interesting synchronization problems.

These problems become clear if one examines the operation of a sequential discrete event simulator. The sequential simulator typically uses three data structures: the state variables, an event list (the calendar), and a global simulation clock. For the execution routine (see section 2.2) it is crucial that the smallest time stamped event (E_{min}) from the event list is selected as the one to be processed next. If it would depart from this rule and select an other event with a larger time stamp (E_x), it would be possible for E_x to change the state variables used by E_{min}. This implies that one is simulating a system where the future could affect the past. We call errors of this kind *causality errors*.

Let us next consider the parallelization of a simulation based on the above paradigm. Most parallel discrete event simulation (PDES) strategies adhere to a process interaction world view that strictly forbids processes to have direct access to shared state variables. To this methodology some extensions have been made to support the parallel execution of the simulation [Cha79]. The system being modelled is viewed as being composed of some number of *physical processes* that interact at various points in simulated time. The simulation is constructed as a set of *logical processes* LP_0, LP_1,. . ., one per physical process. All interactions between physical processes are modelled by time stamped event messages sent between the corresponding logical processes. Each logical process contains a portion of the state corresponding to the physical process it models, as well as a local clock that denotes the progress of the process.

One can assure that no causality error occurs if one adheres to the local causality constraint:

> **Local Causality Constraint:** A discrete event simulation, consisting of logical processes that interact exclusively by exchanging time stamped messages, obeys the local causality constraint *if and only if* each logical process executes events in non decreasing time stamp order.

Consider two events. E_1 at logical process LP_1 with time stamp 10, and E_2 at LP_2 with time stamp 20 (see Fig. 2). If E_1 schedules a new event E_3 for LP_2 containing a time stamp less than 20, then E_3 could affect E_2, necessitating sequential execution of all three events. If one had no information what events could be scheduled by other events, one would be enforced to process the only save event, the one containing the smallest time stamp, resulting in a sequential execution.

During the simulation we must therefore decide whether E_1 can be executed concurrently with E_2. But how do we know whether or not E_1 affects E_2 without actually performing the simulation for E_1? It is this question the parallel discrete event simulation strategies must address.

In this paper we classify parallel discrete event simulation strategies by two categories: *conservative* and *optimistic*. Conservative approaches strictly avoid the possibility of any causality error ever occurring.

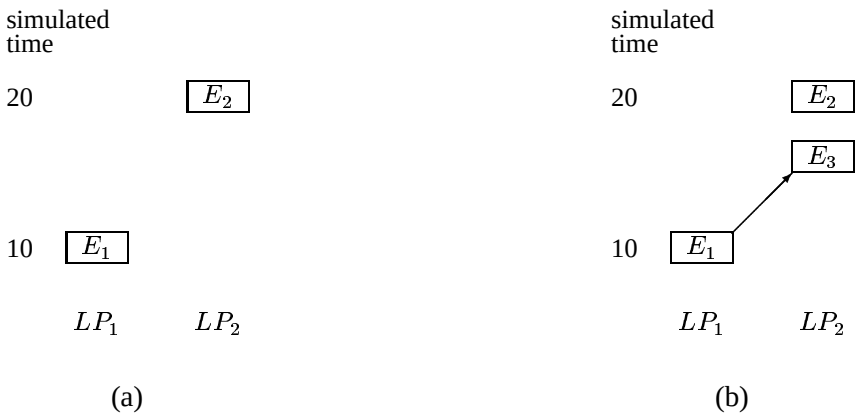

Figure 2: Causality error.

These approaches rely on some strategy to determine when it is safe to process an event. The optimistic approaches use a detection and recovery approach: whenever causality errors are detected a rollback mechanism is invoked to recover. We will describe some of the concepts behind conservative and optimistic simulation mechanisms.

3.3 Conservative Methods

The conservative approaches are the first distributed simulation mechanisms. The basic problem conservative mechanisms must address is to determine which event is save to process. If a process contains an event E_1 with time stamp T_1 and the process can determine that it is impossible to receive another event with time stamp smaller than T_1, then the process can safely process event E_1 without a future violation of the local causality constraint. Processes containing no safe events must block; this can lead to deadlock situations if no appropriate precautions are taken.

Independently, Chandy and Misra [Cha79], and Bryant [Bry77] developed the parallel discrete event simulation algorithms, where one statically specifies the links that indicate which process may communicate with which other processes. In order to determine when it is safe to process a message, it is required that messages from any process to any other process are transmitted in chronological order according their time stamps. Each link has a clock associated with it that is equal to either the time stamp of the message at the front of that link's queue or, if the queue is empty, the time of the last received message. The process repeatedly selects the link with the smallest clock and, if there is a message in that link's queue, updates its local clock to the link's clock and process the message. The order of event processing will be correct because all future messages received will have later time stamps than the local clock, since they will arrive in chronological order along each link. If the selected queue is empty, the process blocks. This is because the process may receive a message over this link with a time that is less than all the other input time stamps. Thus to insure correct chronology, the process is forced to wait for a message to update the clock on the link before the process can update its local clock. This protocol guarantees that each process will only process events in nondecreasing time stamp order, and thereby ensuring chronological integrity.

Deadlock occurs when there is a cycle of blocked processes and each process is blocked due to another process in the cycle. For example consider the network of Fig. 3. Each process is waiting on the incoming link containing the smallest clock value because the corresponding queue is empty. All three processes are blocked, even though there are event messages in other queues that are waiting to be processed.

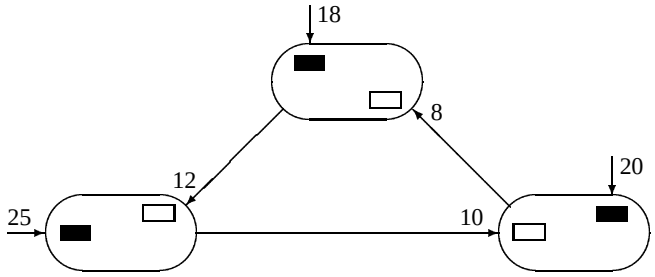

Figure 3: An example of deadlock. (The numbers indicate time stamps.)

Null messages are used to avoid deadlock. This scheme requires that there is a strictly positive lower bound on the *lookahead* for at least one process in each cycle. Lookahead is defined to be the amount of time that a process can look into the future. In other words, if the local clock of the process is any time T and the process can predict all messages it will send with time stamps less than $T + L$, where L is the lookahead. Thus, for a queueing network model, a strictly positive lower bound for the service time for some stations would be required. Intuitively, processes keep the clocks of their output links ahead of their local clocks by sending null messages. A null message with time stamp T_{null} from process LP_A to LP_B, tells LP_B that there will be no more messages from process LP_A with time stamp less than T_{null}. Whenever a process finishes processing an event, it sends a null message on each of its output ports indicating the lower bound on the time stamp of the next outgoing message. The receiver of the null message can then compute new bounds on its outgoing links, send this information to its neighbours, and so on.

Chandy and Misra [Cha81] also presented a two-phase scheme where the simulation proceeds until deadlocked, then the deadlock is detected and resolved. The mechanism is similar to that described above, except no null messages are created. Instead the computation is allowed to deadlock. The scheme involves a controller process to monitor for deadlock and control deadlock recovery. Deadlock detection mechanisms are described in [Gro89, Mis86]. The deadlock can be broken by the observation that the message with the smallest time stamp is always save to process; or, with use of a distributed computation, obtain a lower bound to enlarge the set of safe messages.

The mechanisms described above only attempt to detect and recover from global deadlocks. Prakash and Ramamoorthy [Pra88] suggested a hierarchical decentralized algorithm that takes advantage of the locality of these deadlocks. Another approach to detect and recover from local deadlocks can be found in [Mis86].

The performance of conservative mechanisms is critically determined by the degree to which processes can look ahead and predict future events; or more importantly, what will not happen in the simulated future. A process with lookahead L can guarantee that no events, other than the ones that it can predict, will be generated up to time *Clock* $+ L$. This may enable processes to safely process forthcoming messages that they have already received. Fujimoto describes lookahead quantitatively using a parameter called the lookahead ratio and presents empirical data to demonstrate the importance of exploiting lookahead to achieve good performance [Fuj89]. Other studies of the performance as a function of lookahead can be found in [Lin89, Lou90, Su89].

3.4 Optimistic Methods

In optimistic approaches a process's clock may run ahead of the clocks of its incoming links and if errors are made in the chronology a procedure to recover is invoked. In contrast to conservative approaches, optimistic strategies need not determine when it is safe to proceed. Advantages of this approach are that it has a potentially larger speedup than conservative approaches and that the topology of possible interactions between processes need not be known.

An optimistic approach to distributed simulation called Time Warp, based on the Virtual Time paradigm, was proposed by Jefferson and Sowizral [Jef82, Jef85]. Here virtual time is the same as the simulated time. The local clock, called the Local Virtual Time (LVT) of a process, is set to the minimum receive time of all unprocessed messages. Processes can execute events and proceed in local simulated time as long as they have any input at all. As a consequence, the local clock or LVT of a process may get ahead of its predecessors' LVTs, and it may receive an event message from a predecessor with time stamp smaller than its LVT, i.e., in the past of the process. If this happens the process *rolls back* in simulated time. The event causing the roll back is called a *straggler*. Recovery is accomplished by undoing the effects of all events that have been processed prematurely by the process receiving the straggler.

The premature execution of an event results in two things that have to be rolled back: the state of the logical process and the event messages to other processes. Rolling back the state is accomplished by periodically saving the process state and restoring an old state vector on roll back. Unsending a previously sent message is accomplished by sending a *anti-message* that annihilates the original when it reaches its destination. Messages that are sent while the process is propagating forward in simulated time are called *positive messages*. If a process receives an anti-message that corresponds to a positive message that is still in the input queue, then the two will annihilate each other and the process will proceed. If an anti-message arrives that correspond to a positive message that is already processed, then the process has made an error and must also roll back. It sets its current state to the last state vector saved with simulated time earlier than the time stamp of the message. A direct consequence of the roll back mechanism is that more anti-messages may be sent to other processes recursively.

The Global Virtual Time (GVT) is the minimum of the LVTs for all the processes and the time stamps of all messages sent but unprocessed. No event with time stamp smaller than GVT will ever be rolled back, so storage used by such event (i.e., saved states) can be discarded.

The procedure just described is referred to as Time Warp with aggressive cancellation. An alternative is lazy cancellation, where anti-messages are not sent immediately after roll back. Here, the process resumes executing forward in simulated time from its new LVT, and when it procedures a message it compares it with the messages in its output queue. If the same message is recreated, then there is no need to cancel the message. An anti-message created at simulated time T is only sent after the process's clock sweeps past time T without regenerating the same message. Thus, under lazy cancellation a roll back at the successor process may be avoided. On the other hand, if messages are not reproduced, then roll backs at the successor processes will be required under both mechanisms, and they will occur sooner with aggressive cancellation.

Depending on the application, lazy cancellation may either improve or degrade performance. States may be saved less frequently at the expense of greater overhead for roll back. As a consequence, lazy cancellation requires more memory than aggressive cancellation. Studies of the performance of optimistic approaches can be found in [Lin90, Mad90].

4 Conclusion and Discussion

Performance evaluation is critical for the design, implementation, and improvement of complex applications executing on parallel computers. Analytical approaches to performance evaluation are usually

inadequate because they are based on unrealistic assumptions and require many approximations. Therefore, simulation is a good alternative for obtaining accurate measures of performance. Currently, however, detailed simulations are extremely slow. Parallel simulation seems to be a promising approach for speeding up the simulations, although much more work needs to be done to increase the effectiveness of the existing methods.

Conservative methods offer good potential for certain classes of problems. A major drawback, however, is that they cannot fully exploit the parallelism available in the simulation application. If it is possible that event E_A might affect E_B either directly or indirectly, conservative approaches must execute E_A and E_B sequentially. If the simulation is such that E_A seldom affect E_B these events could have been processed concurrently most of the time. As a consequence, conservative algorithms heavily rely on lookahead to achieve good performance.

Optimistic methods offer the greatest potential as a general purpose simulation mechanism. A critical question faced by optimistic approaches is whether the system will spent most of its time on executing incorrect computations and rolling them back, at the expense of correct computations. An intuitive explanation why the behaviour tends to be stable is that incorrect computations can only be initiated by a premature execution of a correct event. This premature execution, and subsequent incorrect computations, are by definition in the simulated time future of the correct, straggler computation. Also, the further the incorrect computation spreads the further it moves into the simulated time future, thus lowering its priority for execution. Preference is always given to computations containing smaller time stamps. The incorrect computation will be slowed down, allowing the error detection and correction mechanism to correct before too much damage has been done.

A more serious problem with the optimistic mechanisms is the need to periodically save the state of each logical process. This limits the effectiveness of the optimistic mechanisms to applications where the amount of computation, required to process an event, is significantly larger than the cost of saving the state vector.

The type of application, or classes of applications, is important when determining an appropriate approach to distributed simulation. For dynamic topology systems and systems with irregular interactions, Time Warp methods are preferred over conservative methods, especially if state-saving overheads do not dominate. On the other hand, if the application has good lookahead properties, conservative algorithms can exploit the special structure within a fixed topology system. If the application has both poor lookahead and large state-saving overheads all existing parallel discrete event simulation approaches will have trouble obtaining good performance, even if the application has a considerable amount of parallelism.

A challenging, yet not fully exploited, problem is the use of hierarchical methods in parallel discrete event simulation (PDES). It is our contention that, if processes are forced to remember the values of all private variables, an object-oriented methodology can be employed. Here a class must encapsulate all relevant aspects of an entity: its attributes, actions, and life cycle. Communication between objects is allowed only through well-defined interfaces, described by the types of messages an object is willing to respond to. With the use of such object-oriented methodologies, the hierarchical decomposition of the problem under investigation can also be made available in the simulation. In conservative approaches there is some modest effort to use this hierarchical knowledge in the detection of local deadlock and recovery [Pra88]. In optimistic approaches, hierarchical knowledge could be used by the error detection and correction mechanism to quickly stop the spread of the erroneous computations. Furthermore, the proposed model in section 3.1 has to be extended for the evaluation of the various PDES strategies. Many performance evaluations of PDES strategies, found in the literature, compare the parallelism available in the application with the measured speedup of the application on a specific parallel computer. In consequence, there is interference with load balance and scheduling strategies that obscure the effectiveness of the PDES strategy. The extended model should eliminate this interference,

and measure the exploited parallelism by a PDES strategy. In this way, the exploited parallelism can be compared to the average parallelism to obtain the effectiveness of the strategy.

Acknowledgements

I would like to thank Sjaak Koot from our working-group for some valuable discussions.

References

[Bry77] Bryant, R.E., "Simulation of Packet Communications Architecture Computer Systems," MIT-LCS-TR-188, Massachusetts Institute of Technology, 1977.

[Cha79] Chandy, K.M., and J. Misra, "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs," *IEEE Transactions on Software Engineering*, vol. SE-5, no. 5, pp. 440–452, September 1979.

[Cha81] Chandy, K.M., and J. Misra, "Asynchronous Distributed Simulation via a Sequence of Parallel Computations," *Communications of the ACM*, vol. 24, no. 11, pp. 198–205, November 1981.

[Eag89] Eager, D.L., J. Zahorjan, and E.D. Lazowska, "Speedup Versus Efficiency in Parallel Systems," *IEEE Transactions on Computers*, vol. 38, no. 3, pp. 408–423, March 1989.

[Fuj89] Fujimoto, R.M., "Performance Measurements of Distributed Simulation Strategies," *Transactions of the Society for Computer Simulation*, vol. 6, no. 2, pp. 89–132, April 1989.

[Gro89] Groselj, B., and C. Tropper, "A Deadlock Resolution Scheme for Distributed Simulation," *Proceedings of the SCS Multiconference on Distributed Simulation*, pp. 108–112, March 1989.

[Hoo86] Hooper, J.W., "Strategy Related Characteristics of Discrete Event Languages and Models," *Simulation*, vol. 46, no. 4, pp. 153–159, April 1986.

[Jef82] Jefferson, D.R., and H. Sowizral, "Fast Concurrent Simulation using the Time Warp Mechanism, Part I: Local Control," Technical Report N-1906-AF, RAND Corporation, December 1982.

[Jef85] Jefferson, D.R., "Virtual Time," *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 3, pp. 404–425, July 1985.

[Lin89] Lin, Y-B., and E. Lazowska, "Exploiting Lookahead in Parallel Simulation," Technical Report 89-10-06, Department of Computer Science, University of Washington, Seattle (WA), 1989.

[Lin90] Lin, Y-B., and E. Lazowska, "Reducing the State Saving Overhead for Time Warp Parallel Simulation," Technical Report 90-02-03, Department of Computer Science, University of Washington, Seattle (WA), 1990.

[Liv85] Livny, M., "A Study of Parallelism in Distributed Simulation," *Proceedings of the SCS Multiconference on Distributed Simulation*, pp. 94–98, San Diego (CA), January 1985.

[Lou90] Loucks, W.M., and B.R. Preiss, "The Role of Knowledge in Distributed Simulation," *Proceedings of the SCS Multiconference on Distributed Simulation*, pp. 9–16, San Diego (CA), January 1990.

[Mad90] Madisetti, V., J. Walrand, and D. Messerschmitt, “Synchronization in Message-Passing Computers—Models, Algorithms, and Analysis,” *Proceedings of the SCS Multiconference on Distributed Simulation*, pp. 35–48, San Diego (CA), January 1990.

[Mis86] Misra, J., “Distributed Discrete Event Simulation,” *ACM Computing Surveys*, vol. 18, no. 1, pp. 39–65, March 1986.

[Ove91] Overeinder, B.J., and P.M.A. Sloot, “Parallelism in Architecture Simulation,” Technical Report, Department of Computer Systems, University of Amsterdam, Amsterdam, The Netherlands, under preparation.

[Pra88] Prakash, A., and C.V. Ramamoorthy, “Hierarchical Distributed Simulations,” *Proceedings of the 8th International Conference on Distributed Computing Systems*, pp. 341–347, San Jose (CA), June 1988.

[Su89] Su, W.K., and C.L. Seitz, “Variants of the Chandy-Misra-Bryant Distributed Discrete-Event Simulation Algorithm,” *Proceedings of the SCS Multiconference on Distributed Simulation*, pp. 38–43, March 1989.

[Zei76] Zeigler, B.P., *Theory of Modelling and Simulation*, John Wiley & Sons, New York, 1976.