A New Scheduling Method for Parallel Discrete-Event Simulation

Edwin Naroska and Uwe Schwiegelshohn

University of Dortmund, Computer Engineering Institute, 44221 Dortmund, Germany
{edwin|uwe}@ds.e-technik.uni-dortmund.de

Abstract. In this paper we introduce a new conservative and non blocking algorithm for discrete event simulation on parallel computers with distributed memory. The new approach, called critical process first (CPF) algorithm, is especially well suited for simulating complex VLSI designs consisting of many logical processes. The algorithm avoids deadlocks by repeatedly sending lookahead information about critical processes to other computation nodes. Processes are called critical if they may directly influence processes of another computation node. To hide the communication latency of the parallel computer the CPF method gives priority to the execution of events which may affect critical processes. Simulation results show the superiority of the CPF algorithm over approaches without priority handling.

1 Introduction

Discrete-event simulation (DES) is an important part in the validation of large digital systems. With growing complexity, many uniprocessor simulation platforms reach their performance limits. The limiting factors are both computation power and memory constraints of the simulation computer.

To overcome these restrictions parallel discrete-event simulation (PDES) has been suggested. Typically, the numerous components of large designs are mostly connected with local links. This structure inherently supports the use of a parallel computer and PDES.

In our work we consider the simulation of VLSI models with a large number of logical processes on a parallel computer with *distributed* memory. Moreover, we assume that the sizes of the designs allow each simulation node to keep only a fraction of the whole design in its memory. While these machines have the advantage of better scalability over a shared memory multiprocessor, they have a relatively high communication latency compared to their computing power. Consequently, latency hiding becomes an important topic in PDES [9]. The huge memory demand also requires the distribution of the logical processes of the simulation task onto the various nodes of the computer in a step called partitioning.

As each node contains a set of logical processes the execution order of these processes is determined, that is the processes are scheduled. This step is subject to causality constraints. Although partitioning precedes scheduling both tasks

are not independent from each other. In this paper we concentrate on the scheduling problem. For the partitioning we only assume that the distribution of the logical processes is characterized by a small number of links between processes on different nodes.

Previous research in this area has resulted in two different classes of simulation approaches: conservative and optimistic algorithms [4]. In conservative methods all events of a process P are executed in the order given by their time-stamps [7]. It is further possible to distinguish between blocking and non-blocking conservative algorithms. For instance a blocking method has been introduced by Chandy and Misra [3]. Soule and Gupta [9] reported that these algorithms tend to spend a lot of computation time for deadlock detection, deadlock recovery and additional process computation when applied to simulation of digital circuits. Non-blocking algorithms prevent deadlocks but often require additional computation and communication overhead [9, 10].

Optimistic methods allow the processing of an event at a process without guaranteeing that no other event with a smaller time-stamp may arrive at the same process [5]. If the chronological order is not preserved, the simulation must be rolled back in time to a safe state. Therefore, an optimistic simulation algorithm requires a strategy for backing up the simulation. This results in additional memory requirements and may be unacceptable for large VLSI simulation tasks.

Our concept is conservative and non-blocking and suggests a tradeoff between the computation overhead and the memory required for node synchronization. As in [1, 2, 6, 8] we also base our approach on distances between processes. However, in order to adapt to parallel computers with message passing we avoid all barrier synchronization. Instead, we use a special kind of state messages for *node* synchronization similar to [8].

Most previous approaches [1, 2, 6, 7, 8] use a simple scheduling method where the execution order of ready events is arbitrary. Our method however is based on an algorithm which optimizes the order in which the events are executed by the nodes in respect to effective node synchronization. This effort supports hiding communication latency effects of the parallel computer.

The rest of this paper is organized as follows. First, a brief introduction to the problem is given in Section 2. In the next section we informally introduce our method. Then, node scheduling based on critical process lookahead is discussed and synchronizing algorithms with full and limited process lookahead are introduced in Section 4. For the proposed node scheduling schemes we give a simple and an improved algorithm called critical process first (CPF) in Section 5. Finally, in Section 6 some simulation results are presented.

2 Problem Description

A discrete-event simulation is represented by a weighted di-graph $G(P, W, l)$ with $W \subseteq P \times P$ and $l : W \rightarrow \mathbb{R}^+$. Intuitively, $p \in P$ and $w = (p_i, p_j) \in W$ denote a logical process and a link between two logical processes, respectively. During the simulation a message e, called an event, may be sent across a link (p_i, p_j).

Each event e consists of a time stamp $t(e)$ and a value. However, this value is of no relevance for the scheduling problem addressed in this paper.

Upon receiving an event e on link (p_i, p_j) the destination process $p(e) = p_j$ may change its internal state and may also send events on its output links. The virtual time of $t(p(e))$ is set to $t(e)$. The time stamp $t(e')$ of any outgoing event e' on link (p_j, p_k) is at least $t(p_j) + l((p_j, p_k))$. Thus, the weight $l(w)$ of a link $w = (p_i, p_j)$ denotes the lower bound of the event delay on this link.

Consequently, the earliest time a process p_i may influence the state of a process p_j is given by the $t(p_i) + d_{p_i p_j}$ where $d_{p_i p_j}$ is the shortest path in G from p_i to p_j. Note that $d_{p_i p_j} > 0$ as $l(w) > 0 \; \forall \; w \in W$.

As the simulation is causal $t(p_i)$ is not allowed to decrease. Hence, an appropriate scheduling method must guarantee that no event e_x may arrive at p_i with $t(e_x) \leq t(p_i)$.

From the above we can conclude that two events e_i and e_j with $p(e_i) = p_i$ and $p(e_j) = p_j$ can be evaluated in parallel when the following condition holds

$$t(e_j) < t(e_i) + d_{p_i p_j} \quad \wedge \quad t(e_i) < t(e_j) + d_{p_j p_i} . \tag{1}$$

Using these inequations either requires repeatedly solving an all pairs shortest path problem or saving the whole distance matrix in memory. The first approach is computational intensive [1] while the second one requires storing a table of size $|P|^2$.

In this paper we address simulation tasks with a large number of logical processes by proposing a tradeoff between memory and computation requirements.

3 The New Simulation Strategy

Our simulation algorithm is characterized as follows:

1. Processes are assigned to computer nodes. There they are handled by a single sequential node-simulator. This node-simulator determines the execution order of all local events.
2. Each node is required to keep only part of the whole distance matrix $D = [d_{p_i p_j}]$. In typical cases this reduces the memory complexity to far less than $O(|P|^2)$.
3. Different node-simulators are synchronized by state messages which include bounds on future event arrivals.

A dynamic imbalance of work distribution may cause an inefficient simulation as some nodes may have to wait for a (potential) event from another busy node. On the other side, the local events on a node are only subject to a partial order based on event and process time stamps. Usually, this leaves a multitude of permissible execution sequences in each node. This degree of freedom can be exploited to reduce the above mentioned inefficiencies.

Therefore, each node-simulator repeatedly estimates the earliest virtual time at which an internal event may affect processes located on other nodes. This

lookahead information is derived from the minimal process distances $d_{p_i p_j}$ and sent to the appropriate node-simulators. Then these node-simulator determine which of their internal events can be evaluated without violating the causality condition.

As all weights of the graph $\mathcal{T}$ are positive, our algorithm is free from deadlock.

4 Critical Process Lookahead

We first introduce some definitions with respect to a computing node n.

Definition 1 *The process set L_n of node n is the set of all processes allocated to this node.*
$C_n = \{p_i \mid p_i \in L_n \ \wedge \ \exists(p_i, p_j) \in W \ with \ p_j \notin L_n\}$.
$E_n = \{p_i \mid p_i \notin L_n \ \wedge \ \exists(p_i, p_j) \in W \ with \ p_j \in L_n\}$.
$\bar{d_n}(p_i) = \min\{d_{p_i p_x} \mid p_x \in C_n\} \ for \ p_i \in P$.

The critical process set C_n contains all source processes of outgoing external links from n, while all the source processes with external incoming links to n are in the process set E_n. Note, that all elements of E_n are not part of the process set L_n. Finally, the critical distance $\bar{d_n}(p_i)$ is the minimum distance from process p_i to any critical processes of n.

We also say that $\hat{t}(p_i)$ is a lower bound for the time-stamp of all unprocessed events at process p_i. Note that $\hat{t}(p_i)$ does not only take into account all existing events which are not yet processed but also potential events which have not yet been initiated.

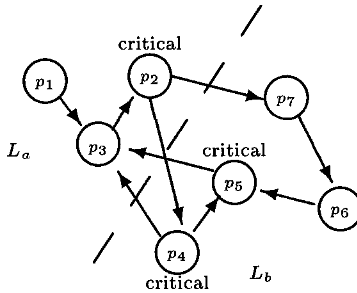

Fig. 1. A network with $C_a = \{p_2\}, C_b = \{p_4, p_5\}, E_a = \{p_4, p_5\}, E_b = \{p_2\}$

Fig. 1 shows an example network consisting of two nodes and seven processes. The critical and external process sets are given by $C_a = \{p_2\}$, $C_b = \{p_4, p_5\}$, $E_a = \{p_4, p_5\}$, and $E_b = \{p_2\}$.

It is the main objective of the scheduling algorithm to find a suitable lookahead $\hat{t}(p_i)$ for every critical process p_i. This lookahead information is then sent

to the other affected nodes, that is those nodes m with $p_i \in E_m$. Then it can be decided which events to process safely without violating the causality condition.

Definition 2 *Let e be an event with destination process $p(e) \in L_n$. The event e is called ready when*

$$t(e) < d_{p_x p(e)} + \hat{t}(p_x) \quad \forall \quad p_x \in E_n. \tag{2}$$

Otherwise, the event is called blocked.

A blocked event e cannot be processed by the associated node as an event on *another* node may result in an event with a smaller time-stamp than $t(e)$ at process $p(e)$. Note, that a ready event may not be processed immediately due to constraints between processes of L_n.

Apart from identifying the blocked events it is therefore also necessary to determine the processing order of ready events by respecting the local causality of the simulation. This causality is reflected in the definition of an event schedule:

Definition 3 *A tuple $S = (e_1, e_2, ...e_m)$ with $p(e_x) \in L_n \; \forall \; e_x \in S$ is called an event-schedule of node n if*

$$t(e_i) < t(e_j) + d_{p(e_j)p(e_i)} \quad \forall \quad 1 \leq i < j \leq m. \tag{3}$$

The events can now be legally executed in the order given by S until

1. the next event in S is blocked or
2. a new event has been internally generated.

Note, that internal events of n must be introduced into the event schedule immediately. This is not necessary for external events due to the blocking property. It is sufficient to introduce these external events into the event schedule whenever a blocked event in S is encountered or when the schedule is modified by using new lookahead information. After the introduction of new events into the event schedule S the processing restarts with the first event of the modified tuple S.

Due to the positivity of the distances an event $e_i \in S$ cannot create events which may affect any predecessor of e_i. Therefore, this scheduling scheme guarantees correct simulation with respect to causality.

In the case of encountering a blocked event however, evaluation must stop until new lookahead information is obtained from other nodes. As this may result in a loss of efficiency the scheduling scheme is extended.

Lemma 1. *Let $S = (e_1, e_2, ...e_y, e_{y+1}...e_m)$ be an event-schedule of node n. If event e_y is the blocked and e_{y+1} is ready then schedule $S_{new} = (e_1, e_2, ...e_{y+1}, e_y...e_m)$ is an event-schedule as well.*

Proof. As S is an event-schedule it meets Inequation 3. The only difference between S and S_{new} is the exchange of e_{y+1} and e_y. Therefore, it is sufficient to prove

$$t(e_{y+1}) < t(e_y) + d_{p(e_y)p(e_{y+1})}.$$

As event e_{y+1} is ready we have

$$t(e_{y+1}) < d_{p_x p(e_{y+1})} + \hat{t}(p_x) \quad \forall \quad p_x \in E_n.$$

However, there is a $p_\mu \in E_n$ with

$$t(e_y) \geq d_{p_\mu p(e_y)} + \hat{t}(p_\mu)$$

as event e_y is blocked. With shortest path property of $d_{p_i p_j}$ we then obtain

$$t(e_{y+1}) < \hat{t}(p_\mu) + d_{p_\mu p(e_{y+1})} \leq \hat{t}(p_\mu) + d_{p_\mu p(e_y)} + d_{p(e_y) p(e_{y+1})} \leq t(e_y) + d_{p(e_y) p(e_{y+1})}$$

Consequently *all* events with status ready can be executed in the order given by S while blocked events are skipped.

```
determine status of events;
determine a schedule S;
determine and dispatch lookahead information;
REPEAT
  insert new events into S if necessary;
  determine status of events;
  determine and dispatch lookahead information if necessary;
  e = first event of S with status ready;
  IF e ≠ ∅ THEN
    execute(e);
    remove e from S;
  END IF;
UNTIL (end of simulation);
```

Fig. 2. Synchronizing algorithm based on external process lookahead

For each new event the status must be determined. Moreover, the status of blocked events is checked when new lookahead information arrives as this may result in changing the status from blocked to ready.

The simulation algorithm based on the external process lookahead information is given in Fig. 2. At the beginning of a simulation an initial event-schedule must be created. Thereafter, a new event-schedule is determined only when new events or new lookahead information arrives. A new event may either appear as the result of the evaluation of another event or it may be received from another node.

Note, that there are no explicit synchronization points in the simulation algorithm. New events or lookahead information from other nodes can arrive at any time and will be processed whenever a new event-schedule S is determined or new lookahead information is evaluated.

4.1 Full Lookahead

When either an event is removed from the event schedule of node n or new lookahead information arrives from other nodes, the lookahead information for the critical processes of node n can be recomputed. This is done by evaluating the expression

$$\hat{t}(p_c) = \min\{\{\hat{t}(p_x) + d_{p_x p_c} \mid p_x \in E_n\},\ \{t(e) + d_{p(e) p_c} \mid e \in S\}\} \tag{4}$$

for every critical process $p_c \in C_n$. This way the time bound $\hat{t}(p_c)$ is determined as precisely as presently possible.

Unfortunately, computing the full lookahead information may lead to a significant computation overhead for large event schedules S. This overhead can be reduced by recalculating only parts of Expression 4.

A reduction of the computation overhead can also be achieved by saving the cause of the lookahead limitation, i.e. by preserving which lookahead information or which event restricts the lookahead time. Then Expression 4 must be recalculated only if the appropriate event has been deleted or the appropriate lookahead time has been changed.

To determine the lookahead a distance table of size $O(|C_n||L_n|)$ is required for each simulation node n.

4.2 Limited Lookahead

As already mentioned, Expression 4 may require a significant computation overhead if a growing number of events must be checked for their influence. This overhead can be reduced by relaxing the bound of the lookahead of a critical process $p_c \in C_n$ as follows:

$$\hat{t}(p_c) = \max\{t(p_c),\ \min\{\{\hat{t}(p_x) + \bar{d}_n(p_x) \mid p_x \in E_n\},\ \{t(e) + \bar{d}_n(p(e)) \mid e \in S\}\}. \tag{5}$$

This way the lookahead of every critical process p_c is set to the minimum timestamp of any unprocessed event at the critical process $p_x \in C_n$ or to the virtual time of p_c if it is greater.

The term $\min\{t(e) + \bar{d}_n(p(e)) \mid e \in S\}$ can be reused while computing $\hat{t}$ for all $p_c \in C_n$ as it is independent of all critical processes of node n. We will show later that this term is also used in the critical process first algorithm. Therefore, no additional computation is necessary. To hold the necessary part of the distance table only $O(|L_n|)$ memory is required.

However, this approach limits the lookahead of *all* critical processes to the lookahead of the most critical one.

5 Methods for Event-Scheduling

Next, we address the problem of finding or modifying an event-schedule S. As this task must be executed whenever a new event appears, it is important that the computational complexity is low.

In a simple approach events e_i can be arranged in ascending order of their time-stamps $t(e_i)$. This will obviously guarantee the validity of Condition 1.

To process the event-schedule only simple insert and delete operations are necessary.

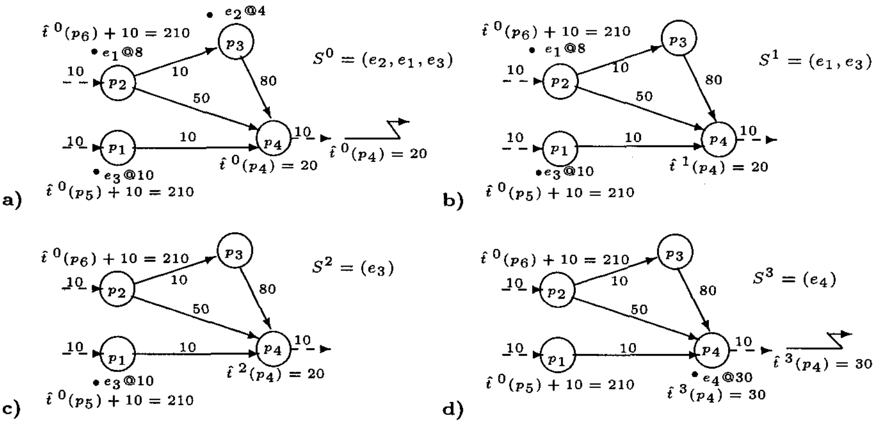

Fig. 3. Example simulation with a simple scheduling method

Figure 3 shows 4 steps of an example simulation using this simple scheduling method. The process links and their weights are also given in the figure.

In the figure events are written next to the corresponding processes. The notation $e_3@25$ denotes an event with number 3 and a time stamp 25. Filled and empty dots mark ready, respectively blocked, events.

For the sake of simplicity, only the processes located on node L_a are shown in the example. Connections to processes located on other nodes are indicated by dashed arcs.

In Figure 3a the lookahead information of process p_4 is determined to be $\hat{t}^0(p_4) = \min\{\hat{t}^0(p_6) + d_{p_6p_4},\ \hat{t}^0(p_5) + d_{p_5p_4},\ t(e_1) + d_{p_2p_4},\ t(e_2) + d_{p_3p_4},\ t(e_3) + d_{p_1p_4}\} = \min\{260, 220, 58, 84, 20\} = 20$. The new value $\hat{t}^0(p_4)$ is sent to the other node as indicated by the flash. The three events e_1 to e_3 arranged in ascending order of their time stamps form the schedule $S^0 = (e_2, e_1, e_3)$.

Evaluating the first event of S^0 results in a new lookahead value $\hat{t}^1(p_4) = \min\{\hat{t}^0(p_6) + d_{p_6p_4},\ \hat{t}^0(p_5) + d_{p_5p_4},\ t(e_1) + d_{p_2p_4},\ t(e_3) + d_{p_1p_4}\} = 20$ as shown in Figure 3b. As $\hat{t}^1(p_4) = \hat{t}^0(p_4)$ no update is sent. The new Schedule S^1 is given by $S^1 = (e_1, e_3)$.

Now e_1 is the next event to be evaluated. As no new event is initiated by e_1 the lookahead $\hat{t}^2(p_4)$ is defined by $\hat{t}^2(p_4) = 20$. Figure 3c presents the state of the simulated network after evaluating e_1.

Finally, the event e_3 is evaluated in Figure 3d. As a result a new event e_4 with $t(e_4) = 30$ arrives at process p_4 increasing the lookahead of p_4 to $\hat{t}^3(p_4) = \min\{\hat{t}^0(p_6) + d_{p_6p_4},\ \hat{t}^0(p_5) + d_{p_5p_4},\ t(e_4)\} = 30$. Afterwards, $\hat{t}^3(p_4)$ is sent to the other node.

Unfortunately, a new lookahead value for process p_4 is sent in step d at the earliest. Consequently, a node may become idle if it is dependent on this lookahead. However, by using a different scheduling method some of these situations can be avoided. In our example event e_3 determines the lookahead information of process p_4 in phase a to c. Unfortunately, it also has the biggest time-stamp of all events. Therefore, e_3 is put at the end of schedule resulting in a delayed processing of the event. It is easy to see that any order of the events in Figure 3a forms a valid schedule.

It is the goal of our event-scheduling method to evaluate the events and the lookahead information of critical processes as soon as possible. Hence, corresponding synchronization information can be sent sooner to other nodes. This approach is especially advantageous when a node must evaluate events which can neither directly nor indirectly initiate new events at critical processes ($\bar{d}_n \rightarrow \infty$). These events can be processed whenever the node is waiting for new lookahead information.

Definition 4 *In the critical process first (CPF) method events e_i located on node n are primarily ordered by $t(e_i) + \bar{d}_n(p(e_i))$ and secondarily by $t(e_i)$.*

Consequently, ready events are evaluated first if they could result in the earliest possible effect on a critical process of this node.

Lemma 2. *A tuple of events $S = (e_1, e_2 ... e_m)$ ordered by the critical process first method is an event-schedule.*

Proof. We must show that the CPF method guarantees the validity of Inequation 3. If $t(e_i) + \bar{d}_n(p(e_i)) = t(e_j) + \bar{d}_n(p(e_j))$ then $t(e_i) \leq t(e_j)$ and we are done. Otherwise, assume that the critical process p_c is destination of $\bar{d}_n(p(e_j))$. With the shortest path property and the definition of $\bar{d}_n$ we have

$$d_{p(e_i)p(e_j)} + \bar{d}_n(p(e_j)) \geq d_{p(e_i)p_c} \geq \bar{d}_n(p(e_i))$$

for any $1 \leq i, j \leq m$. Next we get

$$t(e_i) + \bar{d}_n(p(e_i)) < t(e_j) + \bar{d}_n(p(e_j)) \leq t(e_j) + d_{p(e_j)p(e_i)} + \bar{d}_n(p(e_i)).$$

Inequation 3 follows immediately by subtracting $\bar{d}_n(p(e_i))$ on both sides.

Figure 4 shows an example simulation using the CPF scheduling method. The initial state is the same as in Figure 3. To determine the execution order of the events we first have to calculate $t(e_i) + \bar{d}_a(p(e_i))$ for every ready event e_i. The appropriate values for e_1, e_2 and e_3 are $8+50 = 58$, $4+80 = 84$ and $10+10 = 20$. Sorting the events by these values forms the schedule $S^0 = (e_3, e_1, e_2)$ as shown in Figure 4a.

Due to this order, event e_3 is evaluated first in the next simulation step. As a result, the event e_4 arrives at process p_4 determining the lookahead value to be $\hat{t}^1(p_4) = \min\{\hat{t}^0(p_5)+d_{p_5p_4},\ \hat{t}^0(p_6)+d_{p_6p_4},\ t(e_1)+d_{p_2p_4},\ t(e_2)+d_{p_3p_4},\ t(e4)\} = \min\{220, 260, 58, 84, 30\} = 30$. This new lookahead is sent to the other node. Afterwards, the new event e_4 is inserted into the schedule S^0 leading to $S^1 = (e_4, e_1, e_2)$. Figure 4b presents the situation after evaluating e_3.

In the next step shown in Figure 4c the event e_4 is evaluated resulting in a new lookahead value $\hat{t}^2(p_4) = \min\{220, 260, 58, 84\} = 58$.

Finally, in Figure 4d evaluating event e_1 results in the new lookahead value $\hat{t}^3(p_4) = \min\{220, 260, 84\} = 84$.

Comparing Figure 4 with Figure 3 shows that the CPF scheduling method increases the lookahead faster than the simple scheduling scheme. In this example the simple scheme achieved a lookahead of 20 after 2 simulation steps whereas the CPF scheme increased the lookahead up to a value of 58 at the same time.

Additionally, only $O(|E_n||L_n|)$ memory on each simulation node is required to hold the necessary part of the distance table. Compared to algorithms which operate with the entire distance table this will lead to significantly less memory consumption if $|E_n| \ll |L_n|$.

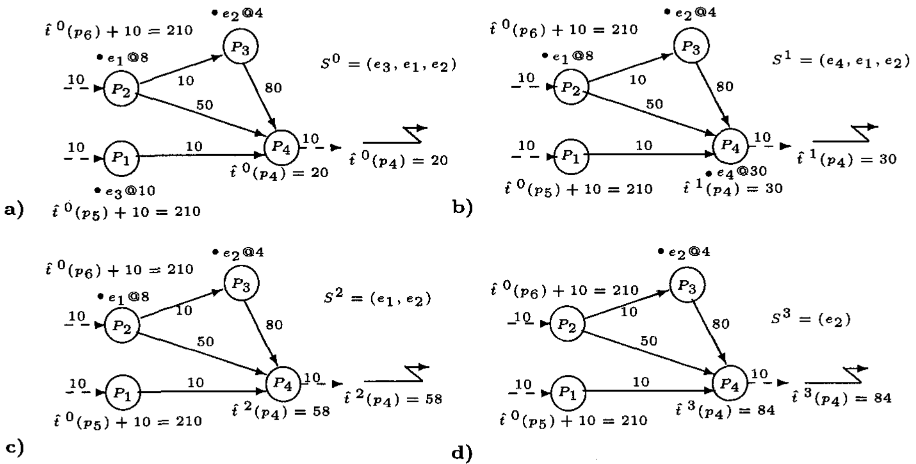

Fig. 4. Example simulation with CPF scheduling

6 Simulation Results

The performance of the various algorithms have been tested with several simulation runs. We used VHDL to model a parallel discrete-event simulator. This

model contains a simple process model and a simple delay scheme for the communication delay between virtual simulation nodes. The virtual execution times of the processes and the events generated by the processes are randomly chosen. However, the execution time of our scheduling algorithms has been neglected in our model.

Figure 5 shows the impact of the communication latency between the nodes on the efficiency of the simulation. The workload consist of 12000 logical processes equally distributed on 20 nodes. The figure outlines the superiority of the CPF algorithm over the simple approach. Additionally the CPF algorithm with limited lookahead performs almost identically compared to the CPF approach with full lookahead. Note that this is achieved with the least computation overhead of the analyzed algorithms.

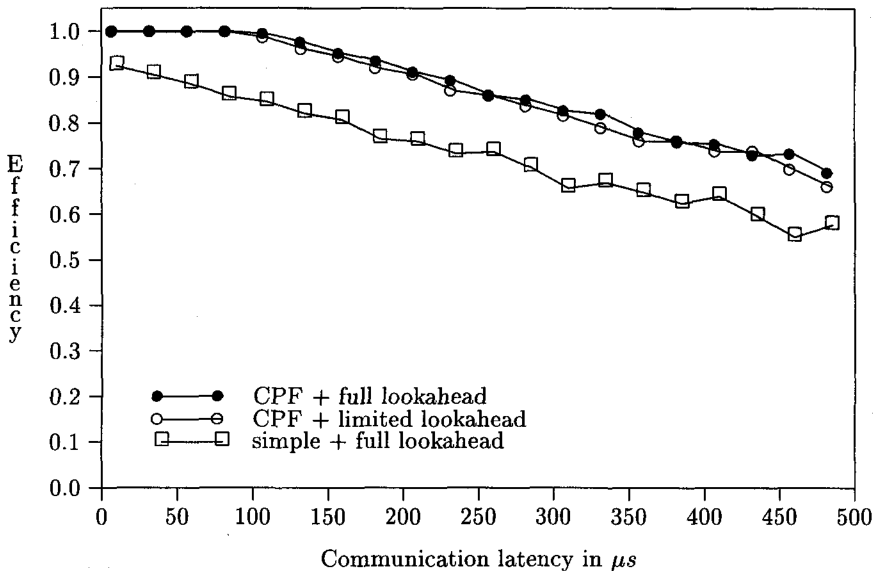

Fig. 5. Impact of the communication latency on the efficiency (symmetrical workload)

7 Conclusion

With the CPF algorithm we have introduced a conservative and non blocking approach to PDES for a parallel computer with distributed memory. Our analysis and our experimental simulations have shown that the algorithm is well suited for the simulation of systems with a large number of logical processes which are mostly locally connected. This kind of systems are typically found in large

VLSI designs. Our approach is especially advantageous if the large memory of a parallel computer is needed to meet the requirements of the simulation model.

The algorithm avoids deadlocks by determining lookahead information for critical processes which directly can produce events at a process of another node. Based on this lookahead every node independently decides which events can be processed safely without violating the causality of the simulation. To improve the synchronization mechanism between the nodes, evaluations of events which may lead to an event at a critical process are given priority. Due to this mechanism new lookahead information can be calculated and dispatched as soon as possible. Simulation results have shown the superiority of the CPF algorithms over the simple approach.

References

1. R. Ayani and H. Rajaei, Parallel Simulation Based on Conservative Time Windows: a Performance Study, Concurrency: Practice and Experience, vol.6(2), pp.119-142, April 1994
2. L.J. Barriga, A Enhanced Parallel Simulation Algorithm Based on Distances between Threaded Objects, Fourth Swedish Workshop on Computer System Architecture (DSA-92) Linköping, January 1992
3. K.M. Chandy and J. Misra, Asynchronous Distributed Simulation via a Sequence of Parallel Computations, Communications of the ACM, vol.24, no.11, pp.198-206, April 1981
4. R.M. Fujimoto, Parallel Discrete Event Simulation, Communication of the ACM, vol.33, no.10, pp.30-53, October 1990
5. D.R. Jefferson, Virtual Time, ACM Transaction on Programming Languages and Systems, vol.7, no.3, pp.404-425, July 1985
6. B.D. Lubachevsky, Efficient Distributed Event-Driven Simulations of Multiple-Loop Networks, Communication of the ACM, vol.32, no.1, pp.111-123, January 1989
7. J. Misra, Distributed Discrete-Event Simulation, Computing Surveys, vol.18, no.1, pp.39-65, March 1986
8. D. Nicol, Parallel Discrete-Event Simulation of FCFS Stochastic Queuing Networks, SIGPLAN Not., pp.124-137, September 1988
9. L. Soulé and A. Gupta, An Evaluation of the Chandy-Misra-Bryant Algorithm for Digital Logic Simulation, ACM Transaction on Modeling and Computer Simulation, vol.1, no.4, pp.308-347, October 1991
10. R.C. de Vries, Reducing Null Messages in Misra's Distributed Discrete Event Simulation Method, IEEE Transaction on Software Engineering, vol.16, no.1, pp.82-91, January 1990