Lookahead Accumulation in Conservative Parallel Discrete Event Simulation.

Jan Lemeire, Wouter Brissinck, Erik Dirkx

Parallel Systems lab,
Vrije Universiteit Brussel (VUB)
Brussels, Belgium
{jlemeire, wouter, erik}@info.vub.ac.be

Abstract. Lookahead is a key issue in distributed discrete event simulation. It becomes explicit in conservative simulation algorithms, where the two major approaches are the asynchronous null-message (CMB) algorithms and the synchronous window algorithms (CTW). In this paper we demonstrate how a hybrid algorithm can maximize the lookahead capabilities of a model by lookahead accumulation. Furthermore, per processor aggregation of the logical processes allows for tuning of the granularity. A qualitative performance analysis shows that in case of no hop-models our algorithm outperforms the traditional conservative algorithms. This is due to reduced synchronisation overhead caused by longer independent computation cycles, generated by the lookahead accumulation across the shortest lookahead path.

1 Introduction

This paper deals with parallel discrete event simulation (PDES) [Ferscha95, Fujimoto90] of logical process based models. There are 2 main approaches in conservative parallel simulation algorithms: the asynchronous approach, called CMB (after Chandy, Misra and Bryant), using null messages for synchronisation [Misra86, Lin95 & Ferscha95], and the synchronous window approach, CTW, (Conservative Time Windows) [Lubachevsky89, ayani92], which uses a window ceiling for synchronisation.
The algorithm that we developed is based on the deadlock avoidance CMB algorithm, and incorporates the concepts of the CTW approach. Our algorithm tries to maximize the performance by optimally tuning two attributes of the model: granularity and lookahead.
Granularity or grain size is defined as 'amount of computations between communication points' [Choi95]. Our algorithm tries to get better performance by maximizing granularity and thus attaining less communication overhead. This is done by per processor aggregation of all its dedicated logical processes forming a multiprocess, which can be simulated sequentially on each processor [Brissinck95, Praehofer94].

Next to granularity, our algorithm exploits maximally the performance gain coming with the lookahead capacities of the model. Better lookahead leads to less synchronisation overhead and better load [Preiss90, Peterson93, Fujimoto88]. Our algorithm tries to accumulate lookahead while calculating the global lookahead of the multiprocess.

The next section explains the algorithm, section 3 discusses the various aspects of the algorithm and compares it with the traditional conservative algorithms (CMB & CTW). Section 4 analyses the performance on a qualitative basis and compares it with CMB & CTW performance. Section 5 finally shows the impact on 2 example models.

2 The Algorithm

At first, the model is partitioned among the available processors and all logical processes on the same processor are aggregated to form a multiprocess. Parallel simulation happens in cycles of independent simulation alternated with communication of the events that travel through the channels connecting the multiprocesses. The independent simulation phase on each processor is based on the chronological processing of all events that are ordered in an *event queue*.
Since we use the conservative approach, simulation is only continued when all events are known until that time. The synchronization algorithm will calculate this safe time. Our algorithm therefore needs to synchronize the multiprocesses and the simulation inside each multiprocess.
Synchronization of the multiprocesses is based on the CTW algorithms. After a phase of independent simulation, a multiprocess will send outgoing events to the other multiprocesses. It then waits for receiving incoming events at the incoming channels from neighbor multiprocesses. All events come together with a time window. The window assures that all events during that time period are known, so that simulation can advance.
Synchronization inside each multiprocess is based on a deadlock avoidance CMB approach that uses null messages to indicate safe simulation. Null messages or null events are defined as 'a promise not to send any other message with smaller timestamp in the future' [Ferscha95]. After these null messages, *conditional events* are possible, because it is not sure that *all* events are known for that time. In our algorithm, a logical process will simulate *until* a first null message appears at *one* input, whereas in CMB algorithms a process has to *wait* for null messages at *all* inputs before it can simulate. This is possible because inside a multiprocess normal and null events are processed in chronological order. When a null message enters a process, this process is *killed*, stopping the simulation of that process. Future events are conditional and may not be processed during the present cycle. They are scheduled in the conditional queue to be simulated in the next cycle. Next, when the process is killed, null messages are scheduled for all output channels at the local virtual time plus the processes' lookahead. They will kill the succeeding processes (Figure 2). The first null message arriving at an outgoing channel of the multiprocess

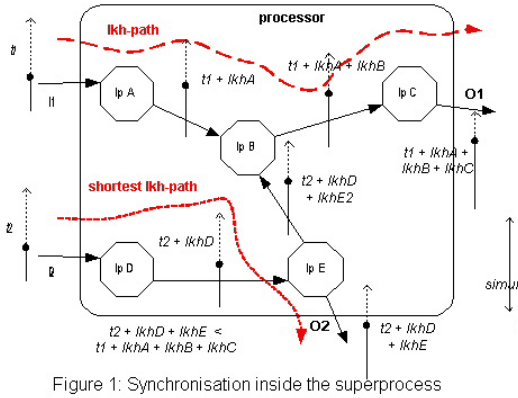

Figure 1: Synchronisation inside the superprocess

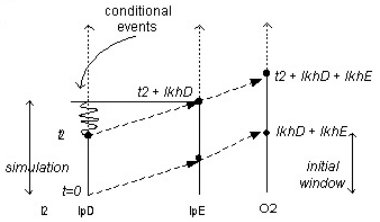

Figure 2: Window propagation

determines the window ceiling of the window that is sent (Figure 1). At start of each simulation cycle, null messages are scheduled for all global inputs of the multiprocess at the time of the incoming window ceiling.
In the initialization phase of the simulation, the first windows are generated. A cycle of independent simulation is performed with an empty input windows (ceiling time zero). The edge processes will be killed at time 0, generating lookahead-incremented null messages for the succeeding processes. In this way, the first global lookahead together with the first output events are generated at the processor outputs for the initial synchronisation of the multiprocesses.

Asynchronous algorithm

Our algorithm can be seen as a window algorithm: each processor receives a safe window to simulate. More precisely, each channel has a safe simulation window during each cycle. Also inside the processor, each channel receives a window, where the ceiling is defined by a null message. However, it is an *asynchronous* algorithm. There is no global (barrier) synchronization as with CTW algorithms. Each multiprocess decides independently when and how much it can simulate, like in CMB algorithms.

3 Discussion of the cycle time

The shortest lookahead-path

Simulation takes place in cycles of communication and independent simulation. A simulation cycle on a processor lasts until the first output is killed by a null event. This null event is generated by a previously killed process, which on his turn is killed by another null event, etc. This chain of null events starts at a certain input and

propagates through the model, forming what we call a *lookahead path*, and ending at an multiprocess output (Figure 1).
Each global output will be killed by a lookahead-path. The *shortest lookahead-path* kills the first output and determines thus the cycle size.

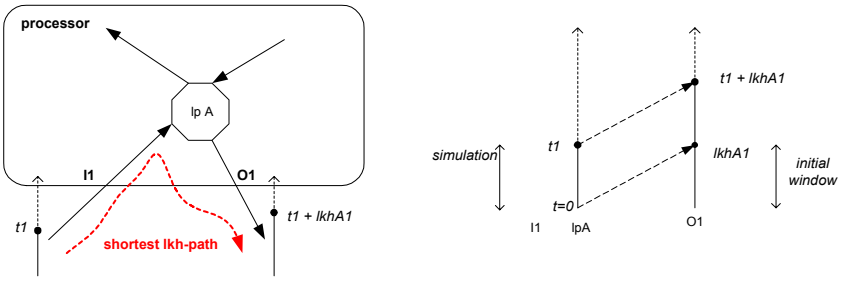

Figure 3: Model with a *hop*

If the shortest lookahead-path passes through multiple processes we can speak about *lookahead accumulation*, the global lookahead of the multiprocess is formed by the sum of the lookahead of all processes in the path. If on the contrary the shortest lookahead-path comes in and leaves the multiprocess out immediately, we talk about a *hop* (Figure 3). For those models, there is no lookahead accumulation and the global lookahead simply equals the lookahead of the edge process.

The shortest lookahead-path is the largest possible safe simulation cycle. By construction, any larger cycle can cause conditional events.

The cycle in CMB algorithms

In a similar way, a cycle can also be defined for CMB algorithms. The cycle is defined as the frequency of event communication and null event generation. In the example of figure 4 [after Lin95], the process can simulate in cycles of the sum of the lookaheads. We see that it is also determined by the shortest lookahead path from a process output back to an input. Each process got its own shortest lookahead path, as opposed to our algorithm where it is calculated per multiprocess.

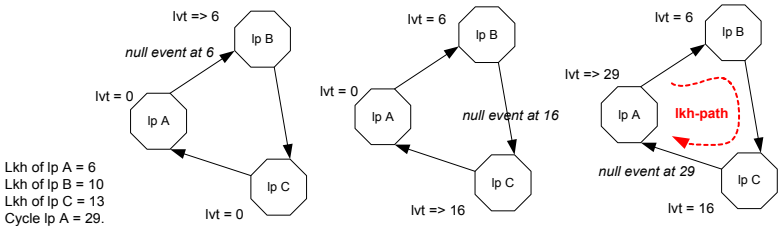

Figure 4: Cycle in CMB

The cycle in CTW algorithms

For conservative time window algorithms, the window size is calculated with the minimal lookahead of the edge processes (connected with other multiprocesses). The main CTW algorithms define a *distance* [Ayani92], an *event horizon* [Steinman94] or a *mimimum propagation delay* (static lookahead) with *opaque periods* (dynamic lookahead) [bounded lag algorithm, lubachevsky89]. All these concepts reflect the lookahead of a process. For these algorithms, no lookahead accumulation takes place and thus is the cycle time the same as in our algorithm for hop-models.

4 Qualitative Performance Analysis

This section discusses the performance of the algorithm and compares it qualitatively with the two traditional approaches. As a first order approximation, we assume the sequential simulation time *SeqSimT* to be proportional to the number of simulated events *#evSim*:

$$SeqSimT = C_1.\#evSim$$

Parallel simulation on p processors is then the simulation of p times less events plus the overhead induced by the parallel nature of the simulation:

$$ParSimT = C_1.\#evSim/p + \sum_i^{\#O} overheadT_i$$

with *ParSimT* the parallel simulation time and $overheadT_i$ the time of overhead i, ranging from 1 to *#O*, the number of overheads. Performance is measured by the speedup, which is the ratio of sequential simulation time versus the parallel simulation time. The impact of the overhead on the speedup is then the ratio of the overhead time with the ideal parallel simulation time:

$$Speedup = \frac{SeqSimT}{ParSimT} = \frac{C_1.\#evSim}{C_1.\#evSim/p + \sum_i^{\#O} overheadT_i} = \frac{p}{1 + \sum_i^{\#O} \frac{overheadT_i}{C_1.\#evSim/p}}$$

The ratio $overheadT_i/ParSimT$ is defined as the *overhead ratio* of overhead i.
Our parallel simulation algorithm generates 3 main types of overhead: communication, synchronization and idle time. These result in 5 overhead ratios Ovh_i and 5 performance factors reflecting the impact of simulation statistics on the different overheads [Lemeire 2001], as shown in Table 1:

Table 1: Overhead classification of the conservative simulation algorithm

Overheads		*Overhead Ratios*	*Performance factors*
Communication	Ovh_1	per event overhead	#evComm / #evSim
	Ovh_2	constant overhead	#evSim / cycle
Synchronisation	Ovh_3	synchronization	#evNull / #evSim
	Ovh_4	conditional queue	#evCond/#evSim
Idle time	Ovh_5	load imbalance	Differences in #evSim per processor

The communication overhead is the time not overlapping with computation for communicating the events. This can be split in the variable overhead (Ovh_1), proportional to the data size, and the constant communication overhead (Ovh_2), induced by setting up the communication link. The communication overhead ratio Ovh_1 is proportional to the number of communicated events between the processors (*#evComm*) versus the number of simulated events. This results in the first performance factor, namely *#evComm/#evSim*. The constant overhead ratio Ovh_2 leads to *#evSim/Cycle*, the number of simulated events per cycle. This ratio is also called **granularity** or grain size [also event simultaneity in Peterson93].

The synchronization overhead is the processing in each cycle of the synchronisation information. For CMB-algorithms and our algorithm this is the null event processing, whereas for CTW-algorithms it is the window size calculation. The processing time for this depends in the first place on the number of null events #evNull. This results for Ovh_3 in a performance factor *#evNull/#evSim*. Our algorithm induces an extra synchronization overhead (Ovh_4) due to the conditional events *#evCond* that are queued to be processed in the next cycle. This leads to a constant overhead and one proportional to *#evCond/#evSim*.

Unequal simulation phases on the different processors lead to idling, when processors have to wait for incoming events. This is mainly caused by load imbalances, here unequal number of events to be simulated. This overhead ratio (Ovh_5) is proportional to the relative deviation of the number of events simulated on each processor.

The Lookahead Accumulation Benefit

The synchronization algorithm influences all but the per event communication overhead OT_1, which is only determined by the model partitioning. The other overheads depend on the cycle time [Peterson93, Choi95], which is determined by the lookahead properties of the model. In case of no-hop models, our algorithm gets larger cycles and will attain a better performance. There will be less constant communication overhead (OT_2), less synchronisation overhead (OT_3), discussed in the next section, and better elimination of temporal load imbalances (OT_5).

The Synchronization Overhead
The per cycle synchronization calculation depends strongly on the algorithm. For CMB algorithms, it is the processing of one null event per channel, whereas for CTW, it is proportional to the number of edge-processes. The synchronization information is thus the lowest for the CTW, and the highest for CMB approaches. In our algorithm it is one null event per interconnection plus the depth of the null event propagation. In case of a hop model, our algorithm looses the lookahead accumulation advantage, the synchronisation overhead will be similar as with CTW (only the lookahead of the edge-processes is taken into account) and so the performance will be equal.
It is proportional to the cycle frequency. For CTW algorithms it is also proportional to the number of interconnections. Whereas for CMB, the number of null events per cycle equals the number of channels. Our algorithm performs in between both: the number of null events per cycle is proportional to the number of interconnections plus the depth of the lookahead propagation
Note that a lot of algorithms optimize the synchronization overhead, like diverse null event reduction techniques in CMB algorithms [Ferscha95, etc] and for example, the bounded lag in Lubachevsky's CTW algorithm [Lubachevsky89].

The overhead OT_4 is specific for our algorithm. The cost for the extra lookahead of our algorithm is the conditional queue. In case of a hop, simulation will stop by the first killed process, no other processes were killed so far and thus, there are no conditional events and no conditional queue overhead. But in case of lookahead accumulation, conditional events of the killed processes must be stored in the conditional queue to be simulated in the next cycle. These extra operations cause the extra overhead: the check whether the process is killed and the queuing. These events come in chronological order out of the event queue and therefore sorting of the conditional queue is not necessary. This results in one extra operation for each event and one for each conditional event. In Figure 2 it can be seen that the number of conditional events could reach half of the number of processed events, as for lp D. But in most cases, it will be much less, because the last lookahead of the lookahead-path causes no conditional events. Moreover, deep processes (far from the edge) will not be killed soon. In total, the extra overhead is thus between 1 and maximally 1.5 extra operations (check and append) per simulated event, which will be much smaller compared to the time to simulate one event C_1. We can conclude that the extra overhead induced by our algorithm is small, what is confirmed by the experimental results.

5 Examples

Two models will demonstrate our claims. One gives good results by exploiting the lookahead accumulation, while the other fails due to low lookahead. Both are simulated on a cluster of 4 Pentium II processors of 333MHz connected by a 100Mb/s non-blocking switch.

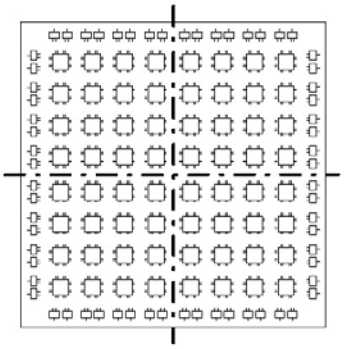

Figure 5: FPGA model

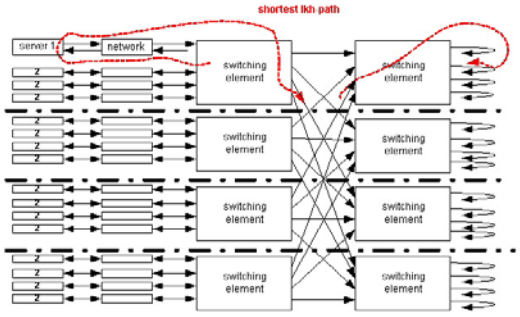

Figure 6: ATM switch model

FPGA

Field Programmable Gate Arrays (FPGAs) are prefabricated devices used to implement digital logic. They feature a matrix structure of logic cells interconnected by routing channels, and a periphery of I/O cells. FPGAs can be programmed by a stream of configuration bits to form a logic circuit. The simulation model consists of 2387 processes and 10978 channels [Bousis00]. Geometrical partitioning (the dashed lines in Figure 5) gives best load balancing and least communication. However, the model is heavily interconnected and contains many hops (namely 453). The shortest lookahead path is only 8 ns, resulting in only 70 events simulated per cycle of 8ns. The performance results are shown in Table 2.

ATM switch

The high capacity ATM switch model [Geudens00] demonstrates the benefits of our algorithm (Figure 6). The model consists of a detailed 4 by 4 switch with 16 entries. Each input receives IP-traffic by a simulated network.

Table 2: Performance results for parallel simulation with 4 processors

		FPGA	ATM switch
	Global Performance		
	Speedup	**0.74**	**3.5**
	#evSim per realtime second	6592 events/s	44000 events/s
	Cycle time	8ns	50000ns
	Communication overhead	17.6%	0.6%
OT_1	#evCom / #evSim	18%	5.7%
OT_2	#evSim / Cycle	70	10100
	Synchronisation overhead		
OT_3	#evNull / #evSim	470%	0.45%
OT_4	#evCond / #evSim	0	1.6%
OT_5	**Idle time**	9.4%	11%

Here again, a geometrical partitioning (horizontal) is the only plausible one (dashed lines in Figure 6). The model can accumulate the lookahead along a path that leaves the switch, passes the network, enters the server and returns back to the switch. This results in long cycles giving an optimal speedup as shown in Table 2.

6 Conclusion

In this paper, we demonstrated the benefit of accumulating lookahead with a hybrid conservative parallel simulation algorithm, based on per processor aggregation of its processes. It uses the traditional CMB approach for synchronisation between the processors. The processors' global lookahead is determined by lookahead accumulation across the shortest lookahead path.
A qualitative performance analysis proved that our algorithm gets a performance benefit over the traditional conservative algorithms CMB (asynchronous null-message algorithms) and CTW (synchronous window algorithms) in case of partitioned models without 'hops'.
The performance analysis, applied to 2 different models, identified the different types of overhead in conservative simulation, allowing model sensitivity analysis of parallel simulation.

7 References

Ayani R., Rajaei H. Parallel simulation using conservative time windows. In 1992 Winter Simulation Conferences Proceedings, pp 709-717, 1992.

Bousis L. Study and Implementation of a Scalable Simulator for Complex Digital Systems. Thesis, Free University of Brussels, 2000.

Brissinck W., Steenhaut K., Dirkx E. A Combined Sequential/Distributed Algorithm for Discrete Simulation. Proceedings of IASTED, Modelling and Simulation, Pennsylvania, 1995.

Choi E., Chung M. J. An important factor for optimistic protocol on distributed systems: granularity. In 1995 Winter Simulation Conferences Proceedings, pp 642-649, 1995.

Ferscha A. Parallel and Distributed Simulation of Discrete Event Systems. Handbook of Parallel and Distributed Computing, McGraw-Hill, 1995.

Fujimoto R.M. Parallel Discrete Event Simulation. Communications of the ACM, 33, pp 29-53, October 1990.

Fujimoto R.M. Performance Measurements of Distributed Simulation Strategies. Proc. 1988 SCS Multiconference on Distributed Simulation Strategies, pp 14-20, February 1988.

Geudens S. Quantitative Study of a Highly Formant Network Switch with Distributed Simulation. Thesis, Free University of Brussels, 2000.

Lemeire, J. and Dirkx, E.: Performance Factors in Parallel Discrete Event Simulation. In: Proc. of the 15th European Simulation Multiconference (ESM), Prague, 2001.

Lin Y., Fishwick P.A. Asynchronous Parallel Discrete Event Simulation. 1995.

Lubachevsky B.D. Efficient distributed event-driven simulations of multiple-loop networks. Communications of the ACM, 32, 111-123. 1989.

Misra J. Distributed Discrete-Event Simulation. ACM Computing Surveys, Vol. 18, No. 1, March 1986.

Peterson G.D., Chamberlain R.D. Exploiting lookahead in synchronous parallel simulation. In 1993 Winter Simulation Conferences Proceedings, pp 706-712, 1993.

Praehofer H. and Resinger G. Distributed Simulation of DEVS-Based Multiformalism Models. IEEE, 1994.

Preiss B.R., Loucks W.M. The impact of Lookahead on the Performance of Conservative Distributed Simulation. 1990.

Steinman J.S. Discrete-event simulation and the event horizon. Proceedings of the 8th Workshop on Parallel and Distributed Simulation(PADS), 1994.