Parallel Logic Simulation on General Purpose Machines

Larry Soule', Tom Blank
Center for Integrated Systems
Stanford University

Abstract

Three parallel algorithms for logic simulation have been developed and implemented on a general purpose shared-memory parallel machine. The first algorithm is a synchronous version of a traditional event-driven algorithm which achieves speed-ups of 6 to 9 with 15 processors. The second algorithm is a synchronous unit-delay compiled mode algorithm which achieves speed-ups of 10 to 13 with 15 processors. The third algorithm is totally asynchronous with no synchronization locks or barriers between processors and the problems of massive state storage and deadlock that are traditionally associated with asynchronous simulation have been eliminated. The processors work independently at their own speed on different elements and at different times. When simulating circuits with little or no feedback, the asynchronous simulation technique varies between 1 to 3 times faster than the conventional event-driven algorithm using 1 processor and depending on the circuit, achieves 10 to 20% better utilization using 15 processors.

1 Introduction

The time needed to simulate a digital system has been a major problem for a long time. People have tried to solve this problem by using hardware accelerators and by using general-purpose parallel machines. Hardware accelerators[1] are popular and although they do provide excellent performance, the relationship between utilization, price, and performance can make hardware accelerators less attractive. Our approach uses a general-purpose parallel machine which has the advantage that it not only speeds up logic simulation but it can also speed up all the other applications as well making it potentially more cost-effective. Unfortunately, few parallel algorithms for electrical simulation at the gate, RTL, and functional levels have been published. Further, these few algorithms usually can only use about 2-8 processors efficiently[2, 3, 4]. There are roughly five previous publications that are related to parallel software logic simulation and are described next.

Arnold[3] detailed a chaotic-time (asynchronous) switch level simulator. The circuit was statically partitioned so that each processor could work on its section at its own speed. In this scheme, each processor proceeds in an event-driven manner, periodically sending messages of the generated events and receiving messages of events from the other processors. If a processor simulates too far ahead in time and receives an event in its "past", it must rollback the state of the circuit to that time and continue from there. When a roll-back like this occurs, all the spurious events that were generated must be cancelled by "anti-events"[5]. Using this technique, a speed-up of 4 over the uniprocessor version was obtained with a 6 processor machine with the performance primarily limited by detecting and processing the "rollbacks". Further, since we must be able to back-up the state of the circuit to any time in the simulation, the "rollback" mechanism leads to a major state storage problem and intricate interprocessor communication.

At a much higher level of abstraction, MCC[6] developed a parallel architectural simulator. It was targeted for investigating future generations of parallel systems so the modeling level is abstract and the parallelism is somewhat "coarse-grained". Speedups obtained with a 12 processor Sequent B8000 were about 2.5 with 16 elements and about 8 with 256 elements.

The Encore Computer Company[2] also developed a parallel version of an event-driven functional simulator. A speed-up of 3 was observed when simulating a large design with 5 processors. Adding more processors did not increase the performance.

Chandy and Misra have presented an asynchronous algorithm for general distributed simulation and proved its correctness[7]. This algorithm views simulation as a sequence of parallel computations. It is similar to other asynchronous algorithms but there are no rollbacks - only events that are known to be valid are used. The simulation is run asynchronously until no more elements have events on all their inputs (i.e. deadlock). To break the deadlock, the "clock-values" of the elements are updated and the simulation is restarted. The algorithm was not implemented thus no performance figures were given.

In the "time-first" or T algorithm[8] the circuit elements are evaluated asynchronously on a uniprocessor. The events are not evaluated in simulation-time order as in the event-driven scheme, rather the events are processed asynchronously as they become ready.

The simulation algorithms that this paper presents are: a traditional event-driven simulator, and a compiled-mode simulator, and an asynchronous simulator. The parallel asynchronous algorithm presented here extends the ideas of the T algorithm for

CH2540-3/88/0000/0166$01.00 © 1988 IEEE

use on parallel machines, to work with models at different representation levels, and to automatically handle circuits with feed-back. Our algorithm is also very similar to the Chandy-Misra algorithm but the "clock-values" of the elements are updated incrementally so deadlock does not occur. Here "asynchronous" means that the processors never have to wait for any of the other processors - there are no synchronization locks or barriers.

2 The Synchronous Event-Driven Algorithm

The synchronous algorithm is a parallel implementation of the classic event-driven algorithm. In the uniprocessor version, the algorithm performs the following steps for each active time step:

1. Update all scheduled nodes
2. Evaluate all elements connected to the changed nodes
3. Schedule all output nodes that change.

Extending this algorithm to a parallel environment is seemingly straightforward: perform the node update phase in parallel, then perform the evaluate/schedule phase in parallel, making sure that *all* processors are done before continuing on to the next time-step. The initial implementation had only one centralized hash table for the node changes and one centralized queue for the activated elements. Unfortunately, the maximum speed-up obtained was about 2 with 8 processors. The problems with this approach were twofold: 1) The operating system would interrupt a process for about 0.1 to 0.25 seconds (comparable to the time needed to execute an *entire* simulation step) to do a working-set scan every 2 seconds causing all the other processors to go into an idle spin waiting for the process to finish, and 2) there was too much contention for the global queues. Modifying the operating system solved problem 1. Problem 2 is caused by the simulation problem itself; the "problem size" of each unit in the queue is very small. Consider the node update phase: to update a node, a processor must atomically remove a record from the global queue that specifys the node and the new value before it can do the actual update. However, it only takes a few instructions to update the node and activate its fan-out elements. Thus, the processor spends comparable times accessing the queue and performing useful work. To solve this problem, the queues were distributed with each processor having one queue for each of the other processors. This way, each processor can schedule the nodes to be updated on the other processors without any contention: when a node update is scheduled, the scheduling processor picks another processor, in a round-robin fashion, to schedule it on and adds it to its queue on that processor, thus splitting up the problem into *n* parts when *adding* to the list rather than when *removing* from the list. So when a processor goes to update nodes, the problem size is much bigger: instead of one node at a time, it can update all the nodes in its queues. To help with load-balancing, once a processor has finished *all* the tasks assigned to it, it looks at the queues on the other processors for more work. This introduces a little contention for the queues, but only at the very end of each phase. This load-balancing technique resulted in a 15-20% better utilization over static load-balancing. The queue for the element evaluation phase of the algorithm was distributed analogously.

2.1 Results of the Synchronous Algorithm

To evaluate the speed-up obtained with the synchronous algorithm, three circuits were used. The circuits were: a 16-bit multiplier with about 5000 elements at the gate level and about 100 elements at the RTL level, a pipelined micro-processor with about 3000 non-memory gates, and a 32x16 array of inverters as a control circuit. The speed-up versus the number of processors used is plotted in figure 1 (all times are for "pure" simulation time - the times for reading in the netlist and writing out the watched nodes are not included). The dip in performance when using more than eight processors is caused by increased cache accesses due to the organization of the Encore: there are 8 processor cards, with each card having two processors sharing one cache. However, the main reasons we don't get 100% utilization are: the number of available events at each time step is limited*, and the processors must synchronize at the end of every time step.

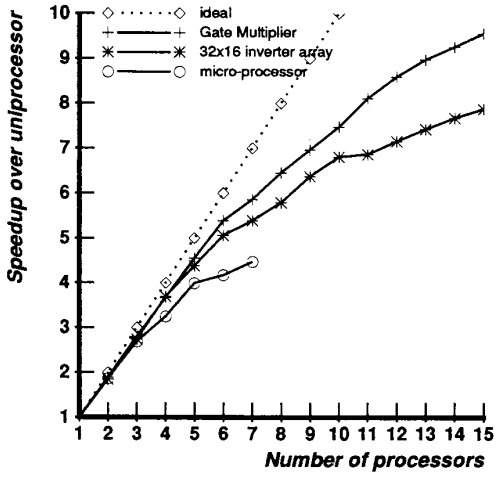

Figure 1: Event-driven Simulation Results

To help understand the effect on the speed-up of the number of available events at each time step, the inverter array was used. The number of events can be easily controlled by how often the inputs to the array are toggled. The speed-up results of simulations where the number of "inverter events" ranged from 512 to 64 are plotted in figure 2. Elements at the higher levels of abstraction will have execution times ranging from 1 to 100 inverter-events[4, 9]. This constant event distribution does not occur in most circuit simulations, but it is a good way to see the relationship between the problem size and the speed-ups obtained.

These plots indicate that to efficiently use more than 16 processors, there must be at least 1000 or so inverter-events in a significant percentage of the time-steps.

*The number of events available versus the simulation time is studied in[4]

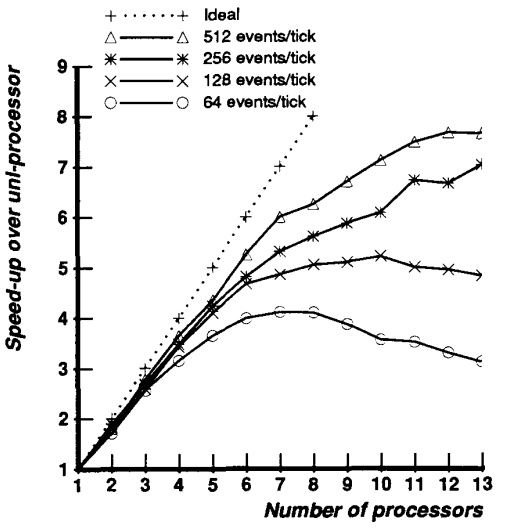

Figure 2: Event per Time-Step Results

3 The Compiled Mode Algorithm

The second algorithm is a parallel version of a unit-delay compiled mode simulator. In compiled mode, every element is executed every time step. To parallelize this, the elements are statically partitioned among the processors and each processor evaluates its assigned elements every time-step. The processors synchronize at the end of every time-step to ensure that all the new values for the nets have been evaluated. Compared to the event-driven algorithm, the "problem size" has been greatly increased. Also, since each processor evaluates the same elements every time step, the execution times are fairly predictable and load-balancing is easy. Unfortunately, this algorithm has the same problem that all other compiled-mode simulators have: if the circuit being simulated has a low element activity, a lot of unnecessary work is done. At the higher abstraction levels, this unnecessary work is not really a problem as most elements get executed every time step anyways. However, at the RTL level, the element activity becomes lower and at the gate level, the activity is typically 0.1%-0.5%[4] per time step and presents quite a problem. This varying element activity makes it hard to compare the general performance of this algorithm to the other algorithms.

3.1 Results of the Compiled Mode Algorithm

For the circuits that have a large number of similar elements (as most gate level representations have) the speed-ups obtained by the compiled mode algorithm, plotted in figure 3, are very good. For the functional level multiplier, however, there are only about 100 elements, and the elements have very different evaluation times (there are inverters, 8-bit adders, and 3-bit multipliers). The small number of elements cuts down on the work per step, and the dissimilar evaluation times makes load-balancing hard. Unfortunately, the class of circuits on which the compiled mode algorithm performs the best - gate level circuits - are the ones that usually have low element activities. Clearly we can efficiently use many processors, but if the element activity is low most of the speedup will be meaningless since the event-driven approach would be much faster overall. Whether the parallel compiled mode algorithm is to be used or not depends on the particular circuit being simulated.

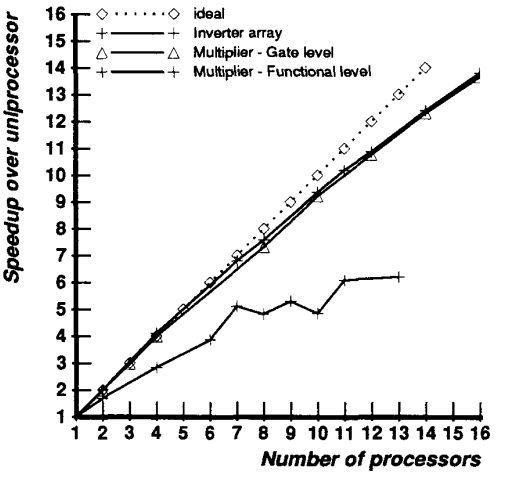

Figure 3: Compiled Mode Simulation Results

4 The Asynchronous Algorithm

The compiled mode algorithm achieves good speed-ups but at the cost of extra work which can be prohibitive when the circuit activity is low or detailed timing information is required. In contrast, the parallel event-driven algorithm, achieves good speed-ups when there are large numbers of events at every time-step, but sometimes even large circuits do not create enough events. Combining the very large problem-size of the compiled-mode algorithm with the efficiency of the event-driven algorithm motivated the development of the asynchronous algorithm. The bottlenecks of the synchronous event-driven algorithm are the synchronization at the end of every time-step and the lack of available events at each time-step. The synchronizing not only introduces overhead (i.e. the more processors, the more time needed to synchronize) but it also makes "load-balancing" -- splitting up the work among the processors equally -- more critical. Unfortunately, as shown by the compiled mode results, load-balancing is difficult in functional simulation since the models may range from simple 2-input logic gates to entire complex micro-processors and the execution times, even for multiple evaluations of the same model, are unpredictable since the time depends on the current inputs and state. Synchronization also makes the algorithm much more sensitive to practical considerations like page-faults, cache effects, and operating system interrupts. If one processor gets interrupted or stalled, *all* the other processors wait at the synchronization barrier until the interrupted process finishes (this effect was very noticeable until the modification to the operating system described in section 2 was made). Another

factor that limits the efficiency of the conventional algorithm when using a large number of processors is the lack of available events even in "large" circuits. It has been observed[4, 10] that even for circuits with 5000 gates, there can be less than 5 events available for evaluation about 50% of the time. (The specific numbers depend on the type of circuit and the clock granularity, but these numbers are typical for the circuits tested here)

The asynchronous algorithm presented here eliminates this synchronization by its semi-chaotic nature. The algorithm processes the circuit by elements rather than by time steps and thus decouples the processors resulting in a more efficient utilization of the available processors making it more economical to use large numbers of processors.

The algorithm is best understood by running through a simple example. The circuit for the example is shown in figure 4, where, *gen* represents a generator element, and *e1*, *e2*, and *e3* represent generic functional blocks. Since *gen* is a generator (i.e. it has no inputs - it just generates signals like the system clock, external interrupt, etc.) the value of node 1 at any particular instant can be determined by calling the code that models *gen* for that particular instant. By calling *gen* repeatedly, we can determine the value of node 1 for the entire simulation time. Since this updates the behavior of node 1, all the elements in its fan-out, *e1* in this case, are activated and placed on the "available for execution" list. The algorithm now takes the next element off the list, *e1*, and calls the code that models *e1* once for each valid event on node 1. This determines the behavior of node 2 for the entire simulation and the storage for the events on node 1 can be freed (Note, the storage can be freed only after all fan-out elements of a node have been processed). The updating of node 2 activates element *e2*. When *e2* is evaluated, it can not be evaluated for the entire simulation since one of its inputs, node 4, is only known to be X at time 0. This means that only one event, instead of the entire simulation time, can be evaluated now. This event is processed and the behavior of node 3 is updated, thus activating *e3*. Now node 4 can be evaluated, and *e2* is activated again. This *e2-e3* loop continues until nodes 3 and 4 have been evaluated for all time (this points out a problem of simulating circuits with large feed-back loops - which will be discussed in more detail later) Note that if *e2* is a simple logic gate, we may be able to take advantage of our knowledge of its behavior. For example, if *e2* is an AND gate and node 2 is 0 from time 0 until time 25, then we know that node 3 will be 0 from time t_d until time 25+t_d (where t_d is the output delay of the gate) and any events on node 4 between times 0 and 25 can be ignored.

This example informally illustrated the basic ideas of the asynchronous algorithm. This algorithm can be precisely specified as follows:

1. Initialization: Evaluate all generator and constant nodes for all time.
2. Each processor independently does:
    a. Atomically remove an element from the distributed activation list
    b. Get as much of the new output behavior from the inputs as possible
    c. Stimulate elements connected to updated outputs

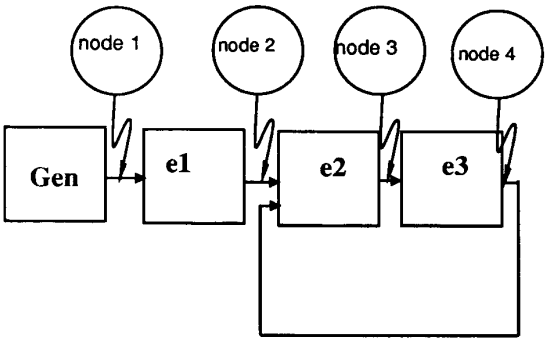

Figure 4: Example circuit to explain the Algorithm

To get the output behavior of an element, perform the following steps:

1. Get the minimum time of the first event of all the inputs. Call this the "current time"
2. Since several events may be queued on the input nodes, get the minimum time we know the behavior for *all* the input nodes. Call this *min_valid*
3. Set up the initial inputs, outputs and state corresponding to the current time
4. For each event on the input nodes that occurs at a time before *min_valid* do:
    a. Evaluate the element with the current inputs and state
    b. Add any new output values onto the behavior description of the output nodes
    c. Activate the elements in the fan-out of the updated output nodes (activate the elements only once).
    d. Get the next event on the input nodes
5. Update the valid times for the output nodes

Since the algorithm is asynchronous, adding and removing elements from the activation lists must be done asynchronously as well. To do this each processor owns *n* FIFO queues (including one for itself), where *n* is the number of processors, with each queue corresponding to one of the other processors. The processors only remove elements from queues they own, and add elements to queues that correspond to them. This way, each queue has only one processor that adds elements to it and only one processor that removes elements from it (one reader and one writer). Since no locks are used, the two processors corresponding to each queue must never modify the same location. Since elements are removed from the head and added to the tail, we just make sure that the head and tail never point to the same location to satisfy this constraint.

If we are simulating a reasonably large circuit, these element queues will fill up quickly giving all the processors some useful

work to do which in turn results in many elements executing concurrently. If we are simulating a smaller circuit or one with a long feed-back chain, the processors tend to pipeline the execution of events. For example one processor may be evaluating an element producing events and another processor can be evaluating one of the elements on the fan-out of that element. It is interesting to note that the algorithm adjusts to execute the events concurrently or "pipelined" as needed. This works automatically because when a new event is produced, the fan-out elements are stimulated if they are not stimulated already. Thus, if there are many elements available, several events can accumulate on the fan-out nodes before being processed. If there are only a few elements available, each event will be processed as it is produced.

Since the algorithm does not process the circuit in a time-synchronized manner, it must keep, for each node, all the events that have not been processed yet but are still needed. When a processor evaluates an element it uses the current behavior of the input nodes to determine the output behavior. Once all elements in the fan-out of a node have been evaluated up to a certain time, all input events prior to that time may be freed. Consistent with the nature of the algorithm, this "garbage collection" may also be done asynchronously.

Feed-back paths prevent complete processing of each node for all time. If you remember the example, the feed-back chain caused the simulation to proceed one event at a time. This type of circuit is the worst-case for the algorithm, but, in the uniprocessor case, the algorithm just reduces to the event-driven algorithm - one event at a time. However, the parallelism available may be reduced in some cases if the feed-back path contains a large portion of the circuit.

4.1 Asynchronous Algorithm Results

The results are promising. The speedup obtained for the functional and gate level representations of the multiplier, and the inverter array circuit using from 1 to 16 processors on an Encore Multimax is plotted in figure 4 normalized to the uniprocessor version. Again, the cache sharing effect is evident.

The inverter array achieves the best speed-ups since the processors always have an element with several events to process. When eight processors are used (ie no cache sharing between the processors), the utilization is 91%. The gate-level multiplier is a much larger circuit (5000 gates) and hence the simulation uses up much more memory. This causes the cache-sharing to affect this simulation the most. The functional level multiplier is very small (100 elements) so during the simulation of this circuit, the processors tend to evaluate the events in a pipelined fashion. This pipelined evaluation causes the number of events processed per element evaluated to drop thus causing more scheduling overhead and inter-processor dependency.

5 Comparison and Summary

A comparison between the relative speeds of the event-driven and asynchronous algorithms on the inverter array is plotted in figure 5. When 16 processors are used, the asynchronous

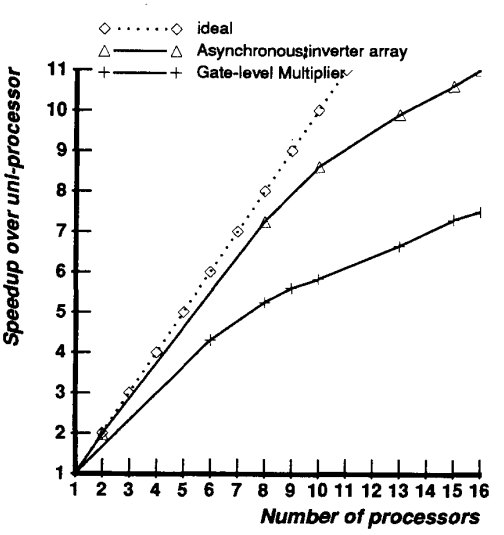

Figure 4: Speedups for the Asynchronous Algorithm

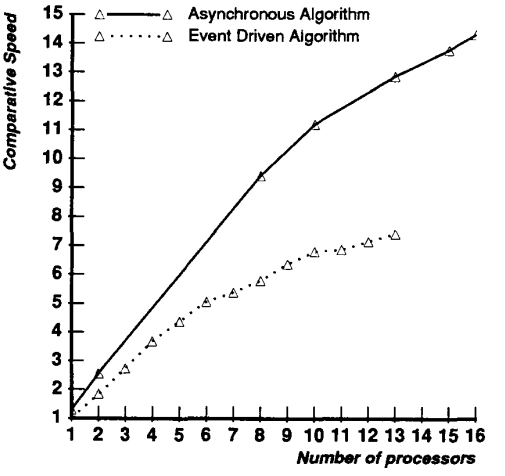

Figure 5: Comparative Speeds for the Inverter Array

algorithm has a utilization of 68% - 10-15% higher than the event-driven algorithm.

When the two algorithms are run on the functional-level multiplier, the asynchronous algorithm does far better. Since there are only 100 elements in the circuit, there are not nearly enough events at each time step to get good speed-ups from the event-driven algorithm. However, the asynchronous algorithm achieves reasonable speed-ups since it decouples the processors.

These results also show that general-purpose machines can be used efficiently for simulations at the gate through functional level.

In particular, the parallel asynchronous algorithm looks promising for a class of circuits than either the parallel event-driven algorithm or the parallel compiled-mode algorithm. The uniprocessor version of the asynchronous algorithm ranges between 1 to 3 times faster than the event-driven algorithm, and the parallel version couples this speed with better utilization. However, for circuits with long feed-back chains, it looks like the event-driven algorithm will be faster especially with a large number of processors.

We are currently converting more circuits to run under this environment to compare the three algorithms. Future work includes porting these algorithms to a hypercube architecture and to other general-purpose parallel machines like the Sequent. We are also investigating the effects of simulating circuits at different representation levels, and the effects of circuits with very large feedback chains and large busses on the algorithm's performance.

6 Acknowledgements

This material is based upon work supported under a National Science Foundation Graduate Fellowship and VLSI contract NDA 903-83-E-0335.

References

1. Tom Blank, "A Survey of Hardware Accelerators Used in Computer-Aided Design", *IEEE Transactions on Design and Test*, August, 1984, pp. 21-39.

2. Andrew Wilson, "Parallelization of an Event Driven Simulator on the Encore Multimax", Tech. report ETR 86-005, Encore Computer, 1986.

3. Jeffrey M Arnold, "Parallel Simulation of Digital LSI Circuits", Tech. report 86-295, M.I.T., January 1986.

4. Larry Soule, Tom Blank, "Statistics for Parallelism and Abstraction Level in Digital Simulation", *Proceedings of the 24th Design Automation Conference*, Stanford University, 1987, pp. 588-591.

5. David Jefferson, "Virtual Time", *ACM Transactions of Programming Languages*, Vol. 7, No. 3, July, 1985, pp. 404-425.

6. J Smith, K Smith, R Smith, "Faster Architectural Simulation Through Parallelism", *24th Design Automation Conference*, ACM/IEEE, June 1987, pp. 189-194.

7. Chandy, Misra, "Asynchronous Distributed Simulation via a Sequence of Parallel Computations", *Comm of the ACM*, Vol. 24, No. 11, April, 1981, pp. 198-206.

8. N Ishiura, H Yasuura, S Yajima, "Time First Evaluation Algorithm for High-Speed Logic Simulation", *ICCAD-84*, IEEE, Nov 1984, pp. 197-199.

9. E.W. Thompson, et.al., "The Incorporation of Functional Level Element Routines into an Existing Digital Simulation System", *17th Design Automation Conference*, IEEE/ACM, June 1980, pp. 394-401.

10. Wong, Franklin, "Statistics on Logic Simulation", *23rd Design Automation Conference*, ACM/IEEE, July 1986, pp. 13-19.