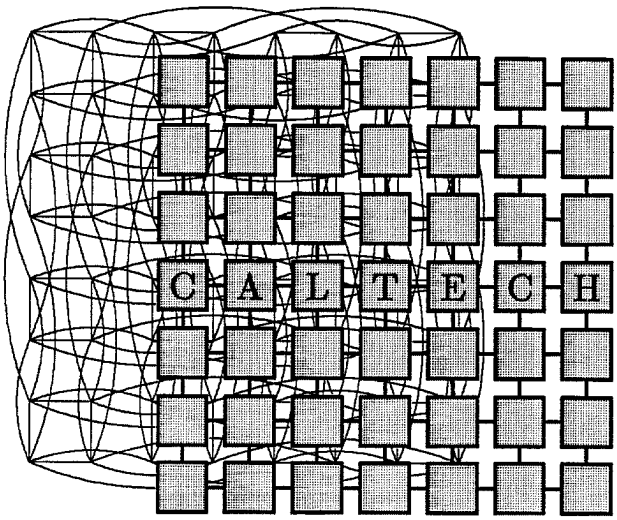

Department of Computer Science
California Institute of Technology
Pasadena, CA 91125

Variants of the Chandy-Misra-Bryant Distributed Discrete-Event Simulation Algorithm

by

Wen-King Su and Charles L. Seitz

Caltech Computer Science Technical Report

Caltech-CS-TR-88-22

19 December 1988

This paper is to be published in the Proceedings of the 1989 Distributed Simulation Conference, which is part of the 1989 SCS Eastern Multiconference

The research described in this report was sponsored by the Defense Advanced Research Projects Agency, DARPA Order number 6202, and monitored by the Office of Naval Research under contract number N00014-87-K-0745.

Variants of the Chandy-Misra-Bryant Distributed Discrete-Event Simulation Algorithm

Wen-King Su and Charles L. Seitz
Department of Computer Science
California Institute of Technology

Caltech-CS-TR-88-22

1. Introduction

We have been using variants of the Chandy-Misra-Bryant (CMB) distributed discrete-event simulation algorithm [1,2,3] since 1986 for a variety of simulation tasks [4]. The simulation programs run on multicomputers [5] (message-passing concurrent computers), such as the Cosmic Cube, Intel iPSC, and Ametek Series 2010. The excellent performance of these simulators led us to investigate a family of variants of the basic CMB algorithm, including lazy message-sending, demand-driven operation with backward demand messages, and adaptive adjustment of the parameters that control the laziness.

These studies were also motivated by our interest in scheduling strategies for reactive (message-driven) multiprocess programs [5,6,7], which are semantically similar to discrete-event (event-driven) simulators. The simulator itself is implemented in the reactive programming environment that we have developed for multicomputers: the Cosmic Environment and the Reactive Kernel [8].

We performed the studies reported here using logic networks. Logic simulation is expected to stress a distributed simulator, and is itself of practical interest. It is easy to construct examples of logic networks with a diversity of behaviors and structural difficulties, such as large fan-in and fan-out. Low-level logic elements such as logic gates exhibit responses in which an input event may or may not influence the outputs, depending on the internal state of the element and on the states of other inputs; yet, they require very little computation to simulate their behavior. Thus, the performance results shown later in this paper involve practically no computation other than the distributed simulation itself.

This paper is a brief and preliminary report of the simulation algorithms and performance results. A more definitive report will be found in the first author's forthcoming PhD thesis.

The research described in this paper was sponsored in part by the Defense Advanced Research Projects Agency, DARPA Order number 6202, and monitored by the Office of Naval Research under contract number N00014-87-K-0745; and in part by grants from Intel Scientific Computers and Ametek Computer Research Division.

2. The CMB Simulation Framework

As usual, the system to be simulated is modeled as a set of communicating elements. A CMB simulator can be implemented by coding the behavior of elements in processes that communicate by messages. A message conveys both a time interval and any events within this interval. A process reacts to the receipt of an input message by updating its internal state, and, if outputs can be advanced in time, by sending messages to connected processes. These messages may include *null messages* that convey no events (changes in the state information), but serve only to advance the simulation time.

It is easy to show that such a simulator is correct [3], in the sense that it computes a possible behavior of the system being simulated. A sufficient condition for freedom from deadlock in this eager message-sending mode is that there is a positive delay in every circuit in the graph of element vertices and communication arcs. Intuitively, it is the delay of the elements being simulated that permits the element simulators to compute the outputs over an interval that is later than the time of the inputs, so that time advances. Simulation time is determined locally, and may get as far out of step at different elements as their causal relationships permit.

This conservative (also known as pessimistic) type of simulator is a concurrent program that exploits the concurrency inherent in the system being simulated. In practice, just as with other concurrent programs, if the number of concurrently runnable processes substantially exceeds the number of processors, one can achieve high utilization of concurrent resources. The speculative (also known as optimistic) type of simulator attempts to exploit additional concurrency by computing beyond the interval during which inputs are defined, at the risk of having to roll back if the speculations prove incorrect. Such approaches are attractive for simulating systems whose inherent concurrency is insufficient to keep concurrent resources busy, and in which speculations can be made with high confidence. Our studies have concentrated on conservative variants of the CMB algorithm.

The design of distributed simulation programs is also influenced by a characteristic of the element simulators. In practice, an element simulator may or may not take as long to process a null message as an event-containing message. For the simulation of some systems, the processing of an event-containing message might involve a lengthy simulation of a physical process, whereas the processing of a null message might be very fast. Such simulations do not seriously stress the distributed-simulation aspect of the computation. However, for the simulation of systems of extremely simple elements, such as logic gates, the time required to compute the output of the gate is so small that it is comparable to the time required to process a null message.

Due to our interest in understanding the limits of event-driven distributed simulation, and the implications for scheduling strategies for message-driven multiprocess programs, our studies have concentrated on the case in which the time required to process null messages is comparable to the time required to process event-containing messages. It is straightforward to extrapolate the performance results for this difficult case to situations in which null-message processing is relatively fast.

The principal trouble with naive implementations of conservative CMB distributed simulation programs in any situation in which processing null messages is as costly as processing event-containing messages is that the volume of null messages may greatly exceed the number of event-containing messages. This difficultly is most evident when simulating systems with many short-delay circuits that have relatively low levels of activity.

In distributing the simulation, we seek to reduce the time required to complete the computation; however, we have an immediate problem if the element simulators must perform many more message-processing operations in the distributed simulation than they would perform event-processing operations in a sequential simulation. The centralized regulation of the advance of time achieved through the ordered event list maintained by sequential simulation programs allows these simulators to invoke element routines only once for each input event. The null messages inflate not only the volume of messages the system must handle, but also the computational load. Thus, if we are going to compete with the best sequential simulators, we must reduce the volume of null messages.

3. Indefinite Lazy Message Sending

To reduce the volume of messages, we use various strategies to defer sending outputs in the hope that the information can be packed into fewer messages. For example, one of the most obvious schemes is to defer sending null messages, so that a series of null messages and an event-containing message can be combined to form a single message that spans a longer interval. Since output events are often triggered only by input events, deferring the delivery of preceeding null messages is less likely to hamper the progress of the destination element than deferring the delivery of event-containing messages.

The first problem that must be addressed in employing such strategies is deadlock. When element simulators defer sending output messages, they may cyclically deny themselves input messages, leading to deadlock. All of our simulators have employed a technique of *indefinite lazy message sending* to permit arbitrary strategies for deferring message sending while still avoiding deadlock. The following is an idealized inner loop of the simulator, shown in the C programming language:

```
while(1)
  if (p = xrecv())
     simulate_and_optionally_send_messages(p);
  else
     take_other_action();
```

The function `xrecv` returns a pointer, `p`, that points to a message for the simulation process if a message has been received. The simulator then dispatches to the appropriate element simulator, and may either send or queue the outputs that the element simulator produces. If there is no message in the node's receive queue, the pointer returned is a NULL (0) pointer. In this case, the simulator takes other

action to break any possible deadlock. For a source-driven simulator, it selects a queued output to send as a message. For a demand-driven simulator, it selects a blocked element, and sends a *demand* message to its predecessor to request that queued outputs be sent. A deadlock in deferring messages cannot occur without "starving" a node of messages. When this situation is detected by `xrecv` returning a NULL pointer, the resulting action breaks the potential deadlock.

Within this indefinite lazy message-sending framework, we can experiment with *any* scheme for deferring and combining messages without concern for deadlock. A message is free to carry any number of events, and an element is free to defer message sending on any basis.

4. Variant Algorithms

We have experimented with many CMB variants; in the interests of comprehension, we will describe the operation and report the performance of six that are representative of the range of possibilities that we have studied:

A *Eager message sending:* This basic form of CMB serves as a baseline for comparison against the variants.

B *Eager events, lazy null messages:* Null outputs are queued. Event outputs, combined with any queued null outputs, are sent immediately. When `xrecv` returns a NULL pointer, the null output that extends to the earliest time is sent as a null message.

C *Indefinite-lazy, single-event:* All output from element simulators is queued. The output queues may contain multiple events. Messages are sent only when `xrecv` returns a NULL pointer. The output queue that extends to the earliest time is selected to generate a message up to the first event, if any, or a null message to the end of the interval.

D *Indefinite-lazy, multiple-event:* This scheme is a slight variation on *C*, motivated by characteristics of multicomputer message systems that make it economical to pack multiple events into fewer messages. All output from element simulators is queued. The output queues may contain multiple events. When `xrecv` returns a NULL pointer, the output queue that extends to the earliest time is selected to generate a message up to the *last* queued event, if any, or a null message to the end of the interval. However, to allow a direct comparison with sequential simulators, events are processed singly.

E *Demand-driven:* Although we usually think of simulation as source driven from inputs, one can equally well organize the simulation as demand driven from outputs. In the pure demand-driven form, all output from element simulators is queued. When `xsend` returns a NULL pointer, the input that lags furthest behind selects the destination for a demand message. Upon receipt of a demand message, if the output queue is not empty, the simulator sends all the information in the output queue; if the output queue is empty, the simulator generates another demand message to the source of lagging input to this element.

F Demand-driven, adaptive: Demand messages single out critical paths in a simulation. In an adaptive form of demand-driven simulation, a threshold is associated with each communication path. Outputs of element simulators are queued only up to the threshold; when the threshold is exceeded, the contents of the queue are sent as a message. Demand messages operate as in E, but also cause the threshold to be decreased. In the cases shown below, the threshold is halved. The simulator is accordingly able to adapt itself to the characteristics of the system being simulated.

Although these variants are described here in terms of message passing, the same variants also appear as different scheduling strategies in shared-memory implementations.

5. Experimental Method

In common with other highly evolved message-passing programs, the simulator is implemented with one simulation process per multicomputer node (or, in the Cosmic Environment, with one simulation process per host computer or per processor in a multiprocessor).

Basis of comparison: Although execution time is one of the most natural bases of comparison between any two programs that perform the same function, and is used below to illustrate the performance of our distributed simulators on different commercial multicomputers, execution time on these concurrent computers depends both on the algorithm and on the characteristics of the particular computer. When we wish to isolate the characteristics of the algorithm from those of the computer, the instrumented simulator operates as a simulator within a simulator. Execution time is then measured in a unit called a *sweep* [5, 6], which corresponds here to a fixed time required to call an element once. The time required for other operations, such as sending a message, can be set to a particular number of sweeps. Normally, a message sent by one node in one sweep is available in the destination node at the next sweep. However, to test the sensitivity of the algorithms to message latency, we can also set the latency to larger values.

Instrumentation: The simulator is a reactive program written in C, and is instrumented to function in two operational modes. In the *sweep mode*, a multicomputer-emulation program runs a simulation of a multicomputer; this in turn runs the reactive simulators. Time is measured in sweep units; on each sweep, each node is allowed to make one element call. In the *real mode*, the simulator runs directly on the multicomputer. There is one copy of the simulator process in each node, and each simulator process runs a subset of the elements as embedded reactive processes. Each node runs at its own pace, and execution time is measured with UNIX's real-time clock.

6. Experimental Results

Performance measurements have been made on a variety of logic networks, including those that are representative of networks found in computers and VLSI chips, and

those that are designed specifically to test or to stress the simulator. Six different network types, each in several sizes up to 4000 logic gates, have been the principal vehicles for these experiments. A larger variation in performance is observed among networks with different characteristics than between algorithm variants.

Multiplier example: The parallel multiplier is a good example of an ordinary logic network. The 14×14 multiplier used in several experiments employs 1376 logic gates to generate the 28-bit product of two 14-bit binary inputs. The multiplier network contains only limited concurrency, and does not contain tight circuits that give the simulator artificial performance boosts or troubles, depending on element distribution. It also contains moderately high fan-out in the multiplier and multiplicand lines; this puts pressure on the message system. In all fairness, the distributed simulation of this multiplier network is not expected to do too badly nor too well.

For the simulation, the most-significant bit of the product is connected back to the multiplier input via an inverting delay. The delay is such that the multiplier reaches a stable state before the multiplier input changes. The multiplicand input is set to a value that causes the circuit to oscillate. A trace of the product outputs shows that the simulator and the circuit are running correctly.

Measurements in the sweep mode: The plot in Figure 1 portrays in a log-log format the sweep count in the sweep mode versus the number of nodes, N, for the simulation of the 14×14 multiplier network under all six CMB variants. It is not useful to continue the plot beyond 2^{11} nodes, since at this point there are as many nodes as simulated gates. The placement of elements in nodes for these trials is balanced but random.

Each horizontal division represents a factor of two in resources; each vertical division represents a factor of two in sweep count or time. We have found this format (*cf* [5]) for portraying the performance of concurrent programs to be more useful than "speedup" graphs, for two reasons. First, we can observe the factor by which the execution time is reduced as resources are increased over very wide ranges. Second, since the ordinate is a physical measure, time or sweep count, we can compare different algorithms directly. For example, in addition to the plots of the sweep counts of the CMB variants, the heavy horizontal line represents the number of sweeps a sequential simulator requires for this same simulation.

The first remarkable characteristic of these performance measurements is that they are so similar across this class of variant algorithms. Algorithms A, E, and F produce more messages than B, C, and D, but in this mode in which messages are free but element invocations are expensive, there is little difference between the variants. The performance under sweep-mode execution exposes the intrinsic characteristics of the algorithm, and is not related to such multicomputer characteristics as the relationship between node computing time and message latency.

The gross characteristics of these curves are similar to those of other concurrent programs [5], and are quite understandable and predictable.

We observe at $\log_2 N{=}0$ (1 node) that all of the CMB variants are somewhat inefficient in comparison with the sequential event-driven simulator. For this

multiplier example, the null messages inflate the number of element invocations by a factor of 2–5 times; this is consistent with the 1–2.5-octave increase in sweep count over that of the sequential simulator. The null messages also inflate the concurrency over that which is intrinsic to the system being simulated. We shall refer to this inflation in the number of element invocations as the *overhead* of distributing the simulation. If the time required to process a null message were smaller than the time required to process an event-containing message, the overhead would be reduced proportionately.

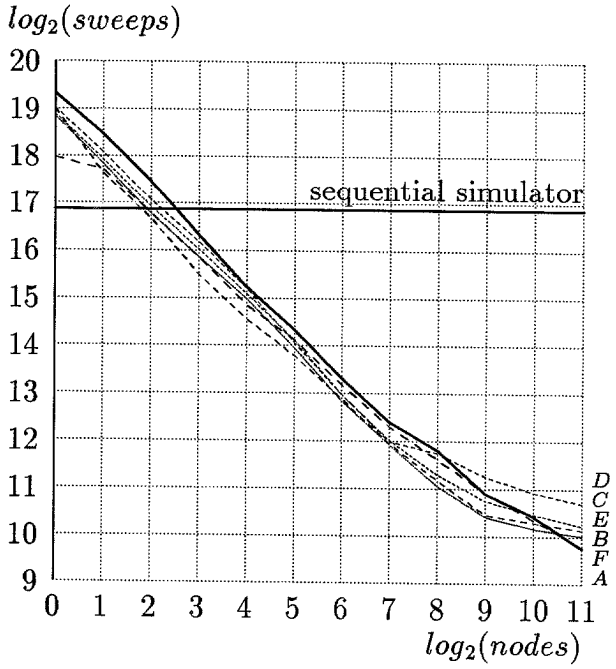

Fig 1: A 1376-gate multiplier, sweep mode

The performance is then divided roughly into two regimes, the first regime being one of near-linear speedup in N for the first 7–8 octaves, and the second regime being one of diminishing returns in N as the computing time approaches an asymptotic mimimum value. In the linear speedup regime, these simulators nearly halve the sweep count with each doubling of resources until limiting effects are reached. Load balance is assured by the weak law of large numbers when there are many elements per node. While each node has a sufficiently large pool of work, node utilization remains high. The simulators approach asymptotic minimal time as they exhaust the available concurrency in the system being simulated. The gradual "knee" of the curve originates from progressively less-effective statistical load balancing as the number of elements per node diminishes with larger N.

Additional statistics have been collected to measure other effects. For example, in the linear-speedup regime, when there are many logic elements per node, the simulators are quite insensitive to message latency. When there are few elements per node, the performance begins to deterioriate as message latency is increased. These

effects will be evident in the measurements performed on real multicomputers.

Measurements on real multicomputers: The results of simulating the same 1376-gate multiplier network on a 16-node iPSC/2 is shown in Figure 2, and on a 128-node iPSC/1 for variants B, C, and D is shown in Figure 3. The iPSC/2 is $\approx$6 times faster per node than the iPSC/1, so the time scales do not correspond. This simulation will not run on an iPSC/1 for $N < 4$ because the data and message queues for an increased number of logic elements per node will not fit in the node memory. Due to the same limitations of the iPSC/1 message system, neither the demand-driven nor the eager-message-sending simulation variants will run in most machine sizes. This choice of performance data is dictated by the desire to show performance results over the largest range of N possible with the machines that are currently operated by our research group. Results essentially identical to those shown in Figure 2 are also obtained on a 16-node Ametek Series 2010.

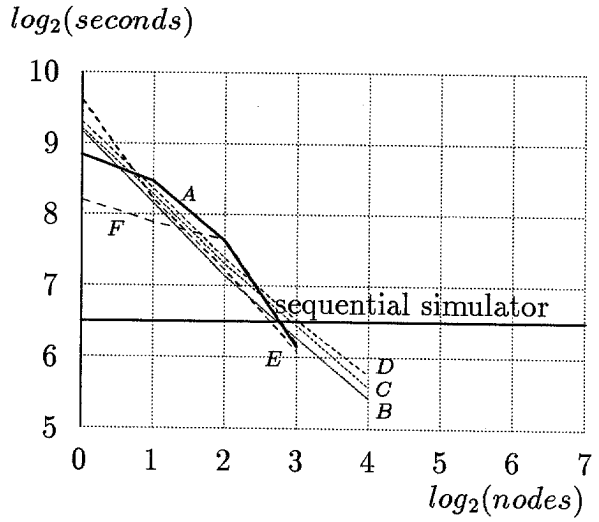

Fig 2: A 1376-gate multiplier for 40μs on an iPSC/2

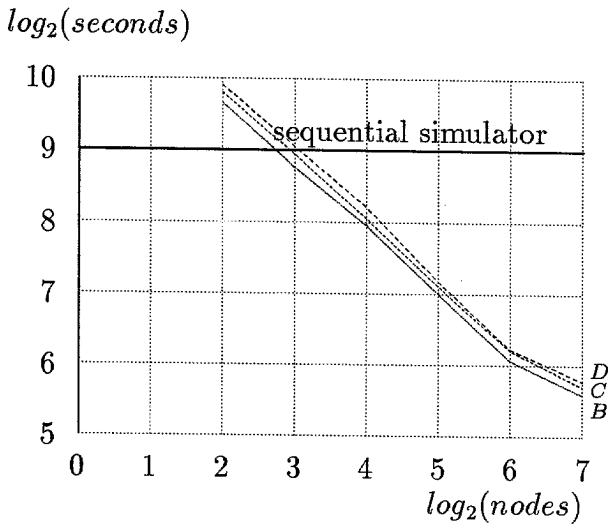

Fig 3: A 1376-gate multiplier for 40μs on an iPSC/1

The simulation of this network for $2^0 \leq N \leq 2^7$ is in the relatively uninteresting (but useful) linear-speedup regime, with some limiting effects starting to be seen in Figure 3 at $N=2^7$. The number of gates being simulated per node is sufficiently high to keep the node utilization high and the sensitivity to message latency low.

In order to exhibit the performance results in the more interesting (but less useful) diminishing-returns regime, we have scaled the network down to a 4-bit multiplier with 116 logic gates. The peformance on an Intel iPSC/2 up to 16 nodes is shown in Figure 4, and on an Intel iPSC/1 up to 128 nodes is shown in Figure 5. This network is small enough to exhibit interesting limiting effects as the simulation is increasingly distributed. The sublinear speedup is due to message latency in inter-node communications, increased null messages as the simulation is increasingly distributed, and load imbalance. The asymptotic time is limited by the message latency rather than by the available concurrency. In particular, Figure 5 shows that the asymptotic execution time of algorithm A, which is not very economical in its use of messages, is more than a factor of two worse than the asymptotic execution time of variants B, C, and D.

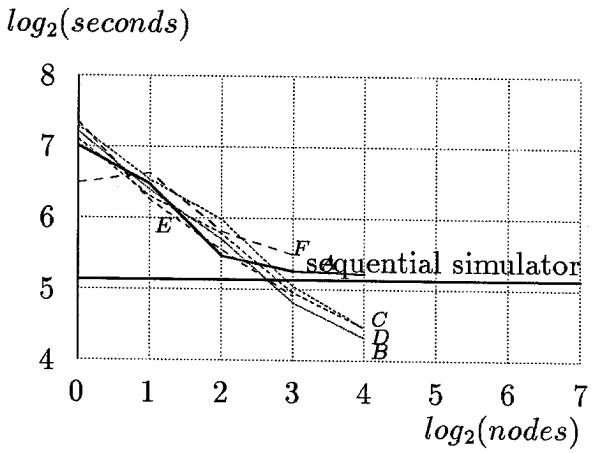

Fig 4: A 116-gate multiplier for 100μs on an iPSC/2

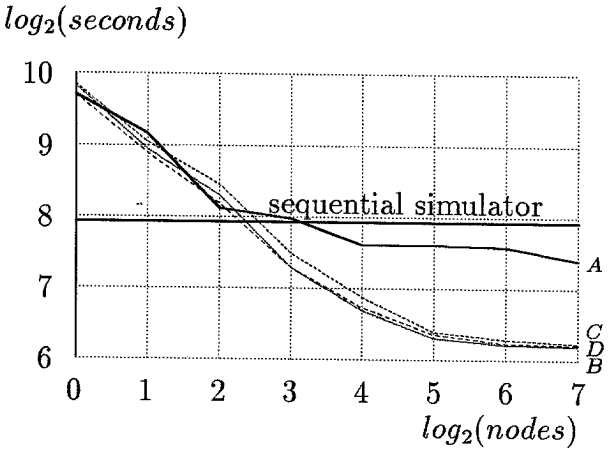

Fig 5: A 116-gate multiplier for 100μs on an iPSC/1

7. Hybrid CMB Variants

Although the CMB variants exhibit good speedup over wide ranges of N, speedup measures only the performance of the algorithm relative to less-distributed instances of itself. In comparison with the sequential simulator, the distributed simulators must pay the overhead of processing null messages. If the elements used in a simulation are such that the time required to process null messages is considerably less than the time to process event-containing messages, these conservative CMB variants will provide excellent performance and efficiency.

However, if the time required to process null messages is comparable to the time required to process event-containing messages, as it is for logic simulation, this overhead makes the CMB algorithm and its variants problematic for simulations on parallel computers in which N is small. What might be done to extend the CMB approach into this difficult small-N range?

A component of the overhead that cannot be eliminated within the CMB framework, in which elements are independent processes, is the null messages used to force progress in cycles of idling elements. However, we can take advantage of multiple elements sharing the same node by lumping members of low-latency, low-activity cycles, such as the gates that form a latch, into macro elements, and applying sequential simulation to them internally. The null-message-processing overhead for such cycles is eliminated at the cost of reduced concurrency for their members.

In this type of hybrid CMB variant simulator, all elements in each node are combined into one macro element, which is simulated internally with a conventional, ordered-event-list, sequential simulator. These sequential simulators are tied together externally with one of the CMB variant simulators. Since there is only one macro element per node, the hybrid variants are identical at N=1 to a sequential simulator. As N increases, however, more cycles are partitioned over multiple nodes, and each hybrid variant eventually converges with its corresponding CMB variant.

Measurements in sweep mode: Figure 6 shows the performance results for the CMB variants simulating a ring of 28 self-timed FIFO units. Each FIFO unit contains one FIFO-control cell and eight register cells, implemented with a total of 1067 logic gates. The FIFO ring is 50% full, holding 14 alternating 1- and 0-bytes. The overhead at N=1 is caused by the idling of the cross-coupled NAND latches in the registers and the FIFO controls. The CMB variants show a good speedup with increased N. Except for the initial overhead, the performance of all of the CMB variants is excellent.

Figure 7 shows the simulation results for the same circuit using the hybrid CMB variants with an element-distribution method that tends to place elements of each cycle in the same node.

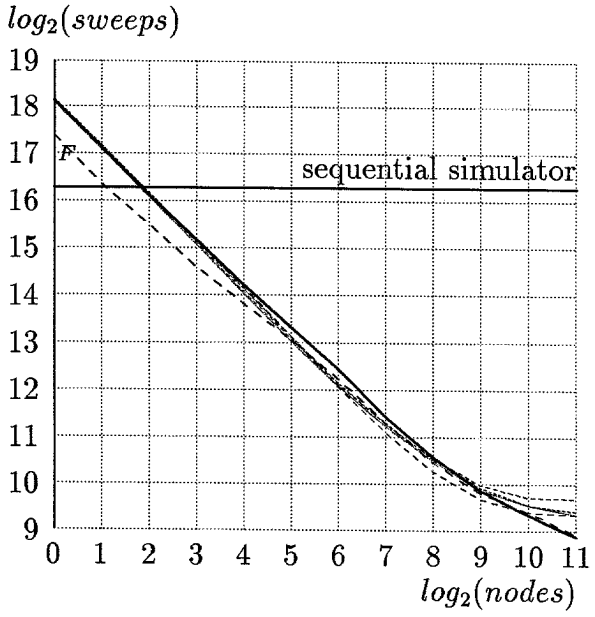

Fig 6: FIFO ring, non-hybrid simulator, emulation mode

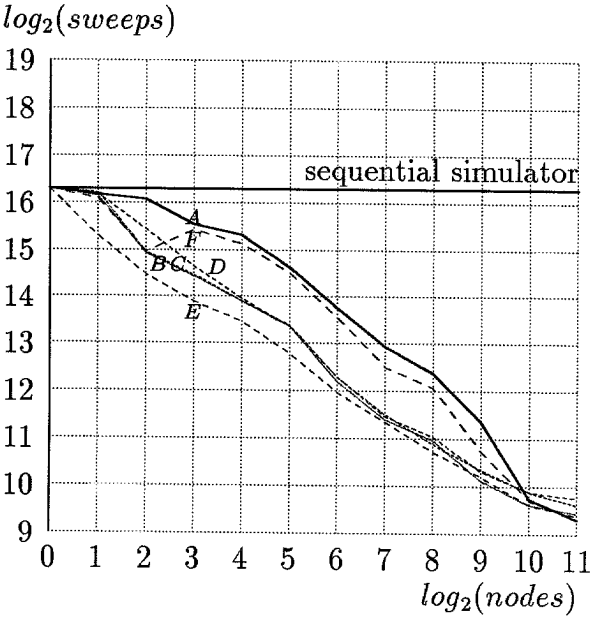

Fig 7: FIFO ring, hybrid simulator, emulation mode

Although the hybrid simulator exhibits a generally decreasing sweep count with increasing N, and extremely good small-N performance for the demand-driven variant E, less desirable behaviors have been observed for the hybrid variants. In particular, if the elements are not properly distributed, or cannot be properly distributed, the simulation time may increase starting at N=2 before starting to decrease. This effect is the result of cycles being broken and scattered over multiple nodes, so that it is the CMB rather than the sequential algorithm that dominates the execution time. Figure

8 illustrates the performance of the simulator for the same circuit used in Figures 6 and 7, but with random placement of the elements across the nodes.

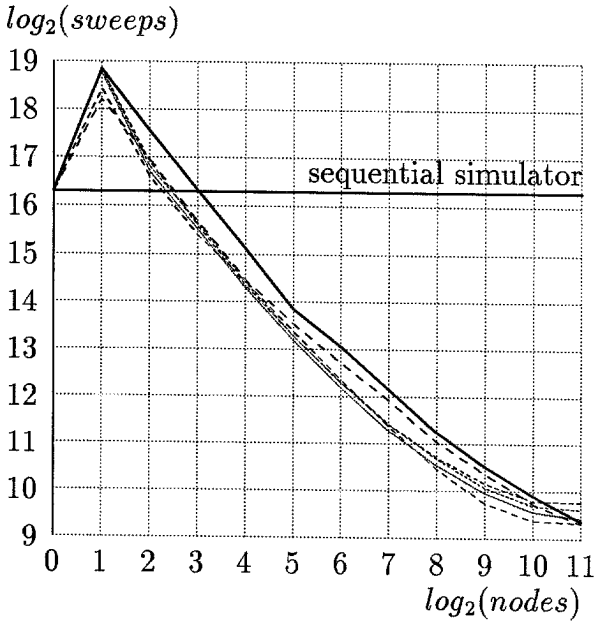

Fig 8: FIFO ring, hybrid simulator, randomized

Some programming short-cuts were used to produce these sweep-mode performance measures for the hybrid variants without implementing a regular sequential simulator; thus, we are not able to include corresponding performance graphs for real multicomputers. However, the instrumentation of the hybrid sweep-mode simulations, together with the performance parameters of second-generation multicomputers such as the Intel iPSC/2 and Ametek Series 2010, indicate that the performance on real multicomputers will be essentially similar to that in the sweep-mode. We are currently implementing distributed simulation programs and instrumentation to run the hybrid CMB variants on real multicomputers.

8. Conclusions

We selected logic simulation for these experiments because we wished to examine the limits of the applicability of the conservative CMB algorithm and its variants. Simulating the behavior of relatively simple elements that have a high degree of connectivity was expected to be a difficult case for distributed simulation. Indeed, the performance results presented here have been much more revealing of the capabilities and limitations of the distributed discrete-event simulation algorithms than earlier simulations that we performed of systems such as multicomputer message networks.

The reader should accordingly be cautious about drawing negative conclusions about the CMB framework from our comparisons of the performance of the CMB variants with the ordered-event-list sequential simulator. For objects of distributed simulation that are less demanding than logic simulation, such as systems in which

processing null messages is much faster than processing event-containing messages, the overhead is proportionately scaled down, and the following general conclusions remain valid:

1. Selected CMB varients exhibit excellent speedup over a wide range of N, limited eventually only by the concurrency of the system being simulated.
2. The CMB variants presented here, all based on the indefinite-lazy-message-sending framework, provide a useful improvement over the basic eager-message-sending CMB algorithm.
3. The hybrid CMB variants offer promise of efficient distributed simulation on small-N concurrent computers.

In some respects, the CMB and sequential algorithms make poor comparison subjects because these two algorithms represent relatively orthogonal optimizations in the basic task of simulation. While the execution time of the sequential simulator is sensitive only to the activity level of the circuit, the execution time for the fully distributed CMB algorithm is sensitive only to the structure of the circuit. In the FIFO-ring example, we can use more data bytes, fewer data bytes, or a different set of data bytes, and shift the sequential simulator's execution time proportionately without significantly changing the CMB variants' curves. Similarly, we can shift the CMB variants' curves without affecting the execution time of the sequential algorithm by varying the delay of the gates in the latches.

The hybrid CMB variants attempt to combine the best aspects of the sequential and CMB algorithms by allowing the sequential simulator to dominate when N is small, and the CMB variants to dominate when N is large. This approach may or may not produce a favorable result, depending on whether the elements can be properly distributed. More research needs to be done in the area of element distribution and its effect on the hybrid variants.

9. Acknowledgment

We very much appreciate the constructive suggestions, ideas, and encouragement that we have received from K. Mani Chandy.

10. References

[1] K. Mani Chandy and Jayadev Misra, "Asynchronous Distributed Simulation Via a Sequence of Parallel Computations," *CACM* 24(4), pp 198–205, April 1981.

[2] Randal E. Bryant, "Simulation of Packet Communication Architecture Computer Systems," MIT-LCS-TR-188, Massachusetts Institute of Technology, 1977.

[3] Jayadev Misra, "Distributed Discrete-Event Simulation," *Computing Surveys* 18(1), pp 39–65, March 1986.

[4] "Submicron Systems Architecture," Semiannual reports to DARPA, Caltech Computer Science Technical Reports [5220:TR:86] and [5235:TR:86], 1986.

[5] William C. Athas and Charles L. Seitz, "Multicomputers: Message-Passing Concurrent Computers," *IEEE Computer* 21(8), pp 9–24, August 1988.

[6] William C. Athas, "Fine Grain Concurrent Computation," Caltech Computer Science Technical Report (PhD thesis) [5242:TR:87], May 1987.

[7] William J. Dally, *A VLSI Architecture for Concurrent Data Structures,* Kluwer Academic Publishers, 1987.

[8] Charles L. Seitz, Jakov Seizovic, and Wen-King Su, "The C Programmer's Abbreviated Guide to Multicomputer Programming," Caltech-CS-TR-88-1, January 1988.