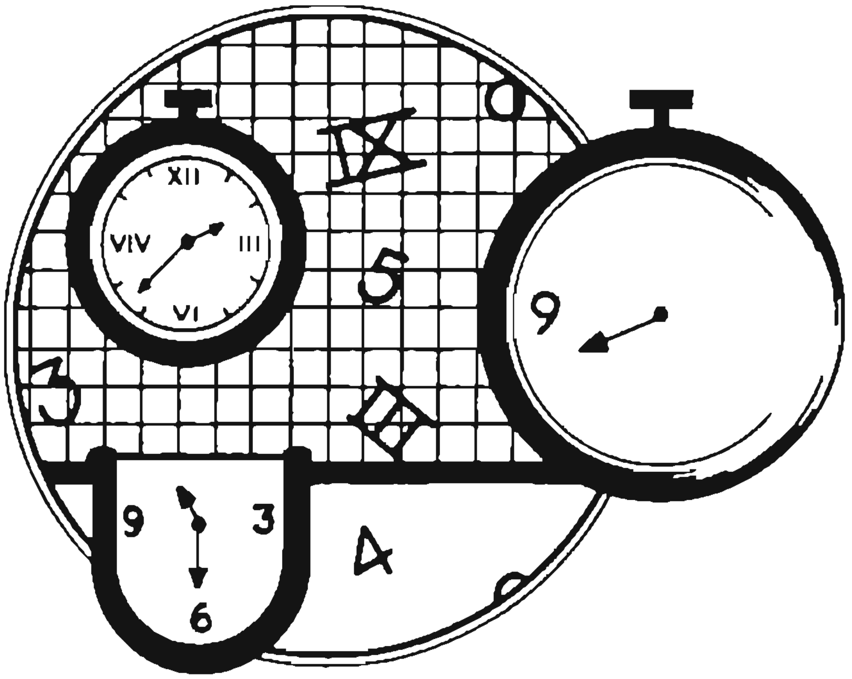
DISTRIBUTED SIMULATION

Edited by

Brian Unger, PhD
and
David Jefferson, PhD

Simulation Series
Volume 19
Number 3



THE SOCIETY FOR COMPUTER SIMULATION INTERNATIONAL

Distributed Simulation 1988
©1988 by The Society for Computer Simulation
ISBN 0-911801-29-4

Circuit simulation on a Hypercube

Erik P. DeBenedictis
Bryan D. Ackland

AT&T Bell Laboratories
Holmdel, New Jersey 07733

ABSTRACT

This paper discusses combined research on simulation and parallel processing. Hypercubes and an associated programming environment (called protocols) were constructed to support experiments in VLSI circuit simulation. Synchronous and asynchronous parallel simulation algorithms were developed and programmed in this environment. Empirical results for a hypercube containing 64 processing elements (PEs) are reported. Results show that for a machine with moderate communication costs, the synchronous algorithm is more efficient. Computer simulations of the algorithms suggest that the asynchronous version would perform better if low cost communication primitives could be developed. The synchronous algorithm is limited by processor load imbalance. An analysis of the load situation suggests that large processors (>1000 nodes) will indeed be able to efficiently simulate circuits containing hundreds of thousands of transistors.

1. INTRODUCTION. With the ever increasing complexity of VLSI circuits there is a continuing need to increase the power of the simulation tools used to verify their function and performance. One approach to improving the speed and capacity of these tools is to take advantage of the memory and compute power of a general purpose multiprocessor system. Such an approach allows us to exploit the intrinsic concurrency of a circuit by simulating different pieces of the circuit on separate processors.

Uniprocessor circuit simulators (programs that simulate analog circuit behavior) are limited to circuits in the range of a few hundred to a few thousand transistors. The simulation of very large circuits (> 100,000 transistors) will therefore require a multiprocessor system composed of hundreds to thousands of processing elements (PEs). Hypercubes, by virtue of their scalability are well suited to this task. The mapping of a uniprocessor algorithm onto a piece of multiprocessor hardware is not, however, a trivial task. First, it requires a partitioning of the problem that exposes the available parallelism. Secondly, it requires a software model of execution that can effectively represent the partitioned algorithm. Thirdly, it requires a set of programming primitives that can efficiently implement the required interprocessor communication and synchronization

In this paper we describe a set of experiments in which we map an MOS circuit simulator called Emu [Ackland 86] onto a 64 PE hypercube multiprocessor [DeBenedictis 85]. Section 2 gives a brief description of the simulation algorithm. Parallel partitioning of this algorithm is outlined in Section 3. Sections 4 and 5 describe the programming model and low level primitives used to implement the parallel algorithm. Emphasis here is on techniques that are general enough to be used in other applications and also extensible to multiprocessors having thousands of PEs. Some empirical results are reported in Section 6. Section 7 provides a simple analysis of some of these results together with some predictions as to how the system will perform with large numbers of PEs.

2. SIMULATION MODEL. EMU represents an MOS circuit as a set of capacitive nodes interconnected by voltage controlled current sources. The change in voltage at any node in the circuit can be determined by summing the currents entering a node and integrating their charging effect on the node capacitance. For a sufficiently small timestep, this change in voltage can be calculated using a single backward Euler integration, assuming that all other nodes in the circuit remain constant.

The accuracy and numerical stability of this scheme depends critically on the choice of timestep. Nodes that are tightly coupled require a smaller timestep to maintain the same level of simulation accuracy. For this reason the circuit is divided into *regions* of tightly coupled nodes. Two nodes are said to be tightly coupled if they are joined by a direct charge transfer path (e.g. the source-drain of an MOS transistor). Between regions, simulation occurs using a relatively coarse timestep specified by the user. Within regions, however, the timestep is automatically subdivided according to circuit activity. At any particular time, a few regions (the active ones) will be simulated with a fine timestep while the rest of the circuit is safely simulated using a coarser timestep.

Further efficiencies are obtained by checking to see if the inputs to a region change from one timestep to the next. If both the inputs and the internal state of the region has not changed, it is not necessary to resimulate the region since the result will be the same. In our experiments with large logic circuits (> 1000 transistors), only 15-25% of the regions need to be recalculated.

3. PARALLEL IMPLEMENTATION. We exploit a circuit's intrinsic concurrency by distributing the circuit description across the PEs. The division of the circuit into regions provides a natural starting point for generating the partitions. Since all connections between regions are unidirectional, the circuit can be represented as a directed graph as shown in Figure 1 where each node is a region and each edge represents a node voltage generated in the source region and used as an input by the destination region. The partitioning problem then becomes one of allocating regions to processors in such a way as to evenly distribute the computational load while minimizing interprocessor communication.

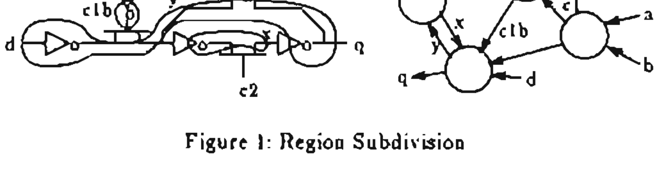


Figure 1: Region Subdivision

The concurrent simulation of a circuit on multiple processors requires two additional resources. The first is a communications mechanism for transmitting node voltages from one region to another. The second is a synchronization protocol to ensure that regions are receiving the correct data at the correct time. Two different models of parallel computation have been developed to provide these resources.

3.1. Synchronous Model. In the synchronous model, all regions are simulated in lock step. Each processor simulates each of its regions for exactly one timestep. It then writes the values of those node voltages that have changed to other processors that need that data. The processor then waits until all other processors have finished simulating that timestep It then reads its new input data and begins simulating the next timestep. Computational load balance between processors is limited by the dynamic load of each region (caused by timestep subdivision and conditional execution). Synchronization requires all processors to run at the speed of the slowest processor at each timestep.

3.2. Asynchronous Model. In the asynchronous version, source regions individually transmit their output values to destination regions as they are calculated. The source region can use that value to simulate a new timestep as soon as all of its inputs have arrived. The simulation of the circuit can then be viewed as the execution of a macro data flow graph [Ackland 86]. Synchronization is implicitly provided by the transmission of data. If buffering is provided in each of the communication paths, source nodes can proceed ahead of their destination nodes. This relieves the load imbalance problem by allowing a limited amount of temporal concurrency The disadvantage of this scheme is the increased communication load caused by the need to transmit unchanged node voltages in order to provide synchronization.

4. PROGRAMMING FOR A MULTIPROCESSOR.

Mapping the parallel algorithms of the previous section onto the hypercube architecture is accomplished using parallel programming scenarios, or *plans* [Soloway 86], that specify control and/or data operations requiring the interaction of two or more processors. A secondary goal is to develop plans that are general enough to be used in other applications. Once these plans have been specified, they are implemented using a number of lower level distributed objects These objects must ensure that there is no possible interaction between plans that might lead to errors or deadlock.

4.1. Master-Slave Plan. At the highest level, we need to provide for command control of the simulator by the user. For this purpose, the simulator can be viewed as a single *master* processor controlling a number of *slaves* The master provides the user interface as shown in Figure 2. Operation consists of a number of cycles in which the master issues a command which is then executed by the slaves The master may, for example, issue the command *simulate for 100 ns.* Each of the slaves will then begin the specified simulation for their portion of the circuit description. One by one the slaves finish. When all are done, the master regains control, prints a prompt for the user, and proceeds to the next cycle.

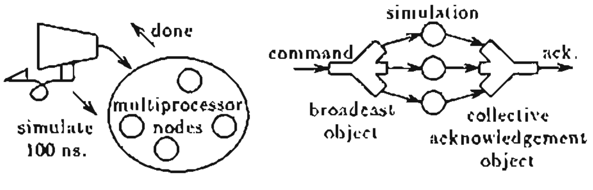


Figure 2: Master-Slave Plan

This plan is implemented using a distributed object with two functions -- *broadcast* and *acknowledge*. The broadcast protocol replicates a message and makes it available to the slaves as shown in the right half of Figure 2. The acknowledge protocol collects individual responses from each slave, finally sending one acknowledge signal to the master.

In addition, there is other data that the slaves must transmit to the master. For example, debugging information or plotting data during simulation. Our Master-Slave plan includes a facility for non-acknowledgement messages to the master, which, to ensure correct synchronization, have priority over acknowledgement messages.

4.2. Data Update Plan. Once each timestep, old simulation data values (node voltages) are updated with new data values. It is essential that the new values are calculated only in terms of the old values. This means that we must keep two copies of all node voltage data as shown on the left in Figure 3. In a uniprocessor, new values are calculated using the old values read from the *real* cells. These results are stored in cells called *scratch*. At the end of the timestep, data is copied from scratch to real.

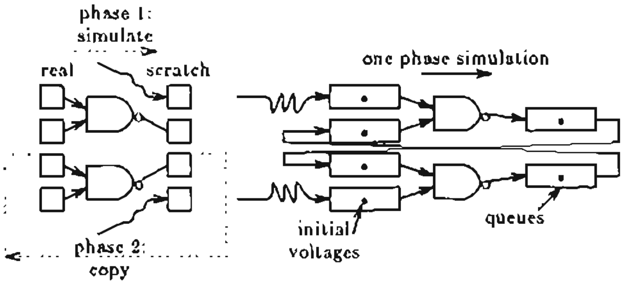


Figure 3: Simultaneous Update Plan

On the hypercube, the calculating and reading regions of this data may be on different processors. In addition (in the asynchronous version of the algorithm) these processors may be operating on different timesteps. Data update is accomplished using distributed queues as shown on the right in Figure 3. Old data is read from the output of the queue, new data is written to the input of the queue. Under time lockstep conditions, the queue will hold at most two values (old and new). By extending the size of the queue, however, the writing region may proceed a number of timesteps ahead of the reader.

4.3. Synchronization Plan. In the asynchronous algorithm, regions are synchronized by the arrival of data. In the synchronous version, however, there needs to be some form of explicit synchronization to inform processors that all input data has been appropriately updated. This is accomplished using a distributed synchronization object. Each processor interacts with this object via a single blocking call. The protocol that defines the operation of the object takes advantage of the hypercube network to determine and transmit synchronization in time logarithmic in the size of the multiprocessor. Basically, each processor synchronizes with each of its neighbors in dimension order. Once it has achieved synchronization with its neighbor on dimension 0, it waits for synchronization on dimension 1, and then 2, and so on until all dimensions have been synchronized. The object then returns control to the calling program and simulation of the next timestep can proceed.

5. HYPERCUBE PROGRAMMING MODEL.

Two types of code have been used to implement an application program on the hypercube. The first is *regular* program code which is characterized by a single control thread on each PE. The second is *protocol* code which is used to implement the distributed objects described in the previous section. Regular code interacts with the protocol code through a series of blocking or polling system calls in much the same way as a user program on a conventional machine interacts with the kernel operating system.

5.1. Protocol Specification. Protocols are represented using *state vectors* and *transition functions*. These specify how a distributed object will react to events generated both by the regular program and incoming messages. The following paragraphs illustrate the use of protocols by way of an example. A more complete description of this distributed object programming technique is given in [DeBenedictis 85] and [DeBenedictis 86].

Consider, for example, the implementation of a distributed queue. Here we concentrate on the flow-control aspects (i.e. when data and acknowledgement messages are sent or retained) and ignore data storage and initialization. Figure 4 illustrates the state transitions for a sender and receiver. Both sender and receiver exist *in* exactly one state at all times. Each starts off in the 0 or empty state. The state transition *arcs* represent three kinds of indivisible events: sending a message, receiving a message, and an interaction with the regular program. In this example the protocol has been drawn as a two-dimensional mesh. For the case of the sender, the vertical axis represents the number of elements in the queue while the horizontal axis represents the number of messages sent but not yet acknowledged. Note that these axes are not fundamental to the protocol - they just simplify the visual representation.

Figure 5 shows some possible input and output *transition functions* for the sender. The input function is executed whenever an acknowledgement message is received. No data operations are required in this case – just a change of state. The output function is executed whenever there is data in the queue and the network is ready to accept a message. The output function sends the message and then alters the state appropriately. Note that a single event may cause complex behavior. An acknowledge message may, for example, cause the input function to run. The input function changes the state of the protocol which causes the output function to run which, in turn causes the input function to run again, repeating the cycle.

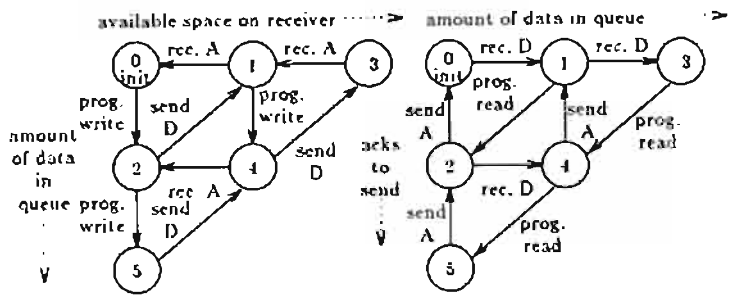


Figure 4: State Transition Dingram for a Queue

5.2. Run-Time System Support. The run-time system provides extensive support for protocols. In addition to supplying a number of standard protocols, it also allows the user to define custom protocols via the *declare_protocol* system call. Parameters to this call describe the states and their transitions and the addresses of the transition functions. Once a protocol is installed, the system decodes network events to determine which function (note there are different functions for different protocols) and which state vector (as there may be many instances of a given protocol running at once) to use.

```
input_function() {
  if (state==1) state=0;
  else if (state==3) state=1;
  else if (state==4) state=2; }

output_function() {
  if (state==2) { send(D); state=1; }
  else if (state==4) { send(D); state=3; }
  else if (state==5) { send(D); state=4; }}
```

Figure 5: Input and Output Functions

There is, in addition, a *dynamic* protocol facility where state vectors are created on demand. When a protocol is declared, one of the arguments is the amount of memory to set aside for state vectors. Subsequently, the first interaction with an instance of that protocol causes a zeroed state vector to be created from the set aside memory. In the simulation application, this allows each processing node to create protocols only for those queue objects with which it interacts (instead of all those associated with the application).

The protocols described here are independent of each other – which is essential to the plan-based programming approach. The plans, such as Master-Slave and Data Update, are in operation simultaneously, so it is imperative that one plan does not change the qualitative behavior of another plan. The system described for protocols is devoid of any mechanism which might cause an interaction (other than timing). As a result plans, which are based on protocols, are independent. This allows the plans to be composed arbitrarily without the necessity of considering their interactions. Conventional blocking message i/o would destroy this independence and is therefore disallowed. The ease-of-programming brought about by the plan approach has allowed us to code the versions of this simulator expeditiously.

6. EXPERIMENTAL RESULTS. Figure 6 shows results for both the synchronous and asynchronous versions of the simulator running on the hypercube processor. Results are plotted in terms of speedup relative to a single PE. The circuit being simulated is a controller for a fuzzy logic chip consisting of 8620 transistors grouped into 984 regions There are a total of 3700 node voltages that must be transferred between regions each timestep.

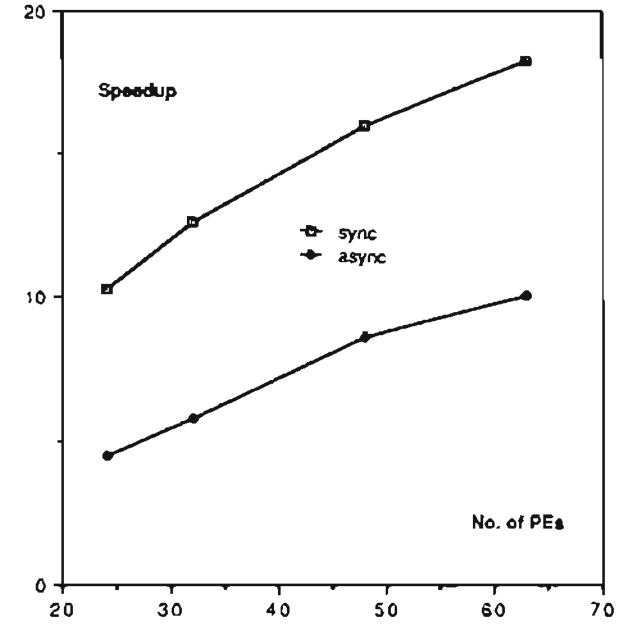


Figure 6: Empirical Results for Emu on a Hypercube

The obvious point to notice is the large performance difference between the two versions. With 63 PE's in operation, the synchronous version is running with a processor efficiency of 29% whereas the asynchronous version achieves only 16% efficiency. As mentioned previously, the synchronous version has the advantage that it requires less communication between regions. The asynchronous version, however, should achieve a better load balance. To understand the reasons for this large difference in performance, an execution time profiler was written for the hypercube. Results of this study are reported in [Lucco 87] and summarized in Figure 7. They show that the asynchronous version is spending over 40% of its time executing communications primitives. In the synchronous version, this overhead has been reduced to 9%. Large communication overhead

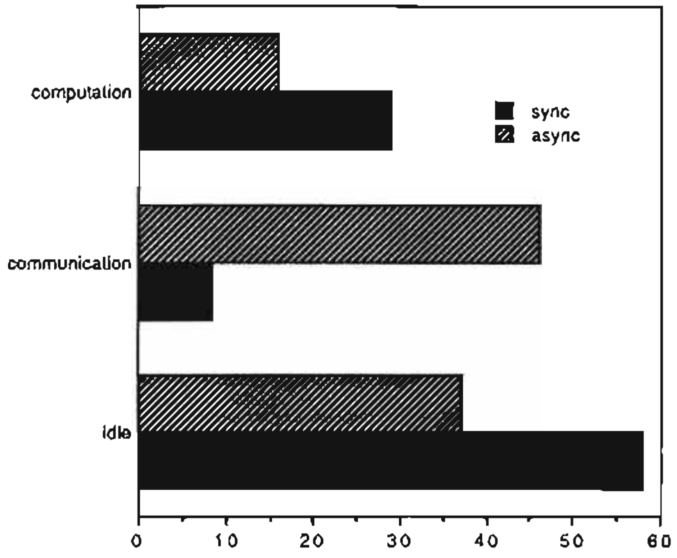


Figure 7: Contributions to Runtime

not only causes lost processing time but also introduces latency which can, in turn, lead to increased idle time (because the required input data has not yet arrived).

Another potential problem with the asynchronous model is the order in which regions are scheduled on a each processor. If this order is poorly chosen, data required by other processors may be unnecessarily delayed resulting in starvation effects. This matter was studied by simulating the macro data flow model used in the asynchronous version under conditions of zero communication cost [Nichols 87]. These studies confirmed that the performance of the system is indeed sensitive to the region scheduling algorithm and that an improvement of about 10% in efficiency can be gained by using a scheduler that always attempts to run the oldest (in simulation time) region first, as shown in Figure 8. Another interesting result from these studies was that in the absence of communication costs, the asynchronous version does indeed outperform the synchronous version by about 25-30% due to improved load balance.

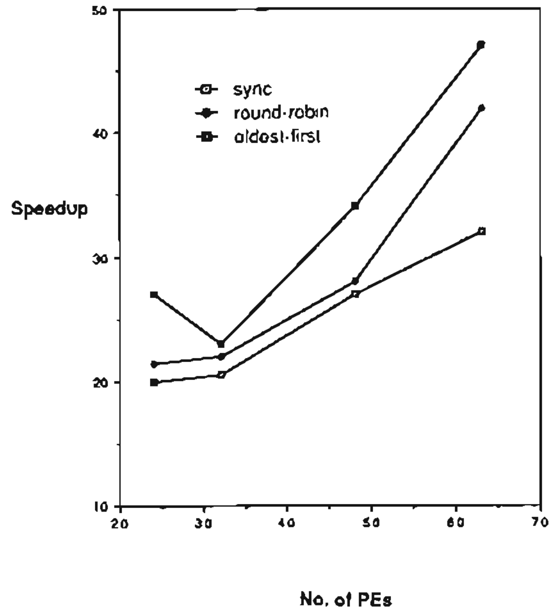


Figure 8: Analysis of Performance Ignoring Communication Costs

The profile of the synchronous version (Figure 7) shows that although processors spend only about 9% of their time doing communication, they lose about 58% as idle time due to poor load balance. More simulation studies were performed to determine how much this could be reduced using a dynamic allocation scheme to move regions between processors during the course of a simulation [Kravitz 87]. These results are shown in Figure 9. They compare dynamic allocation (ignoring re-allocation costs) to a random static allocation ignoring communication and synchronization overheads. They show a potential speedup of 40 with 63 processors - a processor efficiency of 63%. Efficiency here is limited by a residual load imbalance caused by the granularity of the regions. Note that in a real dynamic allocation scheme, this efficiency would be considerably less due to cost of moving regions from one processor to another.

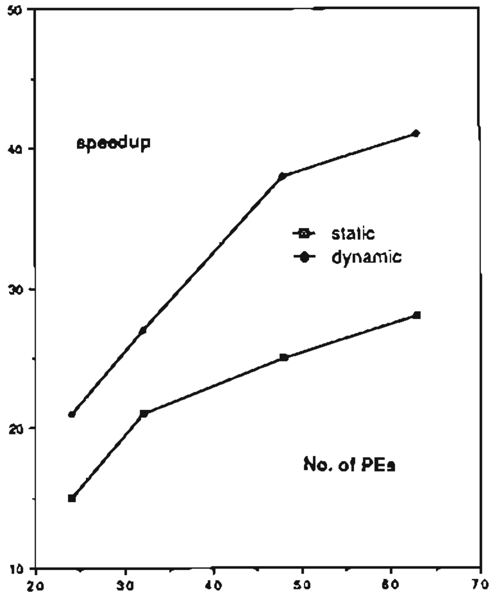


Figure 9: Comparison of Dynamic and Static Region Allocation

7. EXTRAPOLATIONS.

The empirical results reported in the previous section show that under conditions of moderate communication cost, the synchronous version appears to be the more efficient. They further show that the synchronous version is limited mainly by idle time resulting from unequal dynamic load. In this section, we perform a simple analysis of this load balance situation to try and predict how the simulator will perform for large circuits and with large numbers of PEs.

Assume we are simulating a circuit in which there are r regions on a system in which there are n PEs. Assume a uniform distribution of regions across the PEs. Further assume that communication costs in the system are zero and that performance is only limited by the variation in runtime for each region. The runtime distribution of a region for a single timestep is complex as shown in the left half of Figure 10. There is finite probability that the region will not have to be simulated at all; i.e., runtime will be zero. If it does need to be simulated, the runtime will

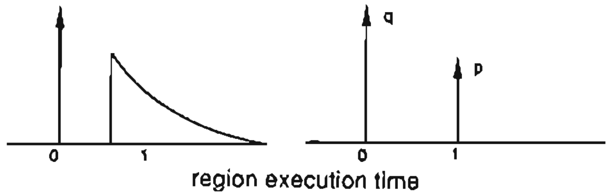


Figure 10: Region Runtime Distribution

depend on the amount of timestep subdivision. For the sake of our analysis we will assume the runtime distribution is as shown in the right half of Figure 10. That is, we assume that the runtime is 1 with probability p, and 0 with probability $q = 1-p$.

The distribution of total time X required to execute one timestep on one processor can be obtained by modeling the execution of the regions as r/n independent events. For a random placement with a large number of regions per processor, this is a reasonable approximation. This then gives X a binomial distribution, where $\mu = rp/n$ is the mean, and $\sigma = \sqrt{rpq}/n$ is the standard deviation.

The running time of the entire multiprocessor for one timestep can be modeled as the maximum time of n trials of X, denoted $X_{[n]}$. $\bar{X}_{[n]}$ is the average time over the entire multiprocessor per timestep. Load balance efficiency is then $e_l = \mu/\bar{X}_{[n]}$.

Figure 11 plots efficiency calculated using this model with $p = 0.2$ as a function of the total number of regions r and the number of PEs n. Note that to keep the efficiency the same when the number of processors is doubled, requires the number of regions to increase by a factor of about 2.5. The upper right corner of the graph is interesting: a 100K region (~800K transistor) chip simulating on a 1000 PE multiprocessor would incur a 40% load balance overhead (60% efficiency).

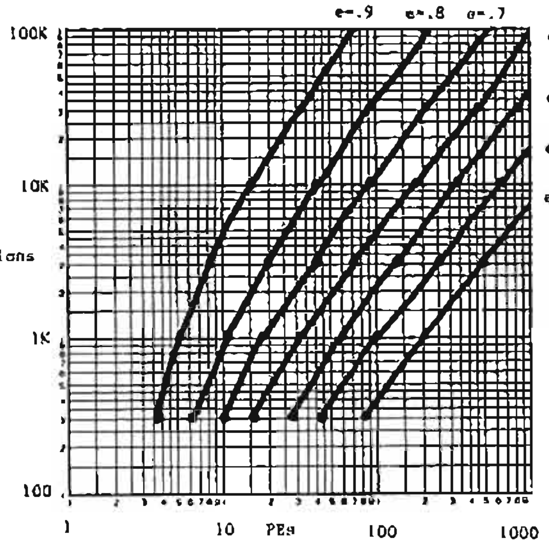


Figure 11: Efficiency Accounting for Load Balance

Since inter-PE communication occurs only when output data changes, it seems reasonable to assume that each PE's communication load is strongly correlated with its computation load. If the time to simulate a region is t_c and the time to communicate the results is t_m, then the communication efficiency will be $e_c = t_c/(t_m + t_c)$. Overall efficiency is then $e = e_l\, e_c$. For the fuzzy logic controller test results reported in the previous section, $r = 984$ and $n = 63$. Figure 11 predicts a load efficiency of $e_l = 0.38$. The communication efficiency (from Figure 7) is $e_c = 0.77$. This gives an overall efficiency of $e = 0.29$ which agrees closely with the empirical result.

8. CONCLUSIONS.

We have described a number of experiments in implementing a parallel version of an MOS circuit simulator on a hypercube multiprocessor. The hypercube architecture is well suited to this application because of its extensibility to large numbers of PEs. We have described two parallel simulation algorithms. We have also described a set of parallel programming plans (based on protocols) that have been used to implement these algorithms in a reliable deadlock free fashion.

Of the two algorithms, the synchronous shows the best performance on our hypercube processor. This is because of the relatively high cost of inter-PE communication. Commercial hypercube systems exhibit greater relative communication cost and are likely to also favor the synchronous algorithm. Simulation studies have shown, however, that under conditions of low communication cost, the asynchronous algorithm is more efficient. What is needed is a machine with hardware (or efficient firmware) support for high level communication primitives like queues. Further improvements in efficiency might be obtained with a dynamic partitioning strategy, although it is not clear what the costs of allocation would be.

Our empirical results showed that our 63 PE hypercube could be used to give speedups of 20 with moderate sized circuits. An analysis of the load balance of the synchronous algorithm predicts that these results are scalable to larger circuits simulating on larger numbers of PEs. Based on these predictions we hope soon to be able to demonstrate the simulation of very large circuits (>100,000 transistors) on a commercial 1000 PE hypercube.

9. References.

[Ackland 86] B. Ackland, S. Ahuja, E DeBenedictis, T. London, S. Lucco, and D. Romero, MOS Timing Simulation on a Message Based Multiprocessor. In *Proceedings of the IEEE International Conference on Computer Design*, October 1986

[DeBenedictis 85] E DeBenedictis, Multiprocessor Programming with Distributed Variables. In *Proceedings of the Conference on Hypercube Multiprocessors*, August 1985.

[DeBenedictis 86] E. DeBenedictis, Protocol-Based Multiprocessors. In *Proceedings of the Conference on Hypercube Multiprocessors*, August 1986.

[Kravitz 87] S. Kravitz and B. Ackland, Static vs. Dynamic Partitioning of Circuits for a MOS Timing Simulator on a Message-Based Multiprocessor. Elsewhere in these proceedings.

[Lucco 87] S. Lucco and K. Nichols, A Performance Analysis of Two Parallel Programming Methodologies in the Context of MOS Timing Simulation. In *Proceedings of Spring COMPCON 87*, February 1987.

[Nichols 87] K. Nichols and J. Edmark, Evaluating Multicomputer Systems with PARET. Submitted to IEEE Software.

[Soloway 86] E. Soloway, Learning to Program = Learning to Construct Mechanisms and Explanations. In *Communications of the ACM*, September 1986. Pages 850-858.