Parallel Logic Simulation of VLSI Systems

Roger D. Chamberlain
Computer and Communications Research Center
Department of Electrical Engineering
Washington University, St. Louis, Missouri

Abstract – Design verification via simulation is an important component in the development of digital systems. However, with continuing increases in the capabilities of VLSI systems, the simulation task has become a significant bottleneck in the design process. As a result, researchers are attempting to exploit parallel processing techniques to improve the performance of VLSI logic simulation. This tutorial describes the current state-of-the-art in parallel logic simulation, including parallel simulation techniques, factors that impact simulation performance, performance results to date, and the directions currently being pursued by the research community.

I. Introduction

The benefits of faster logic simulators are self evident to just about anyone in the electronic design automation field. Due to increased complexity in the VLSI system design process, logic simulation has taken on an essential role in the verification of designs prior to fabrication, yet the time required to complete simulations has grown. Larger designs require longer simulation runs for two primary reasons: a greater number of test vectors are needed to verify the correctness of larger systems and each test vector requires more computation to simulate the effects of the vector. The result is that simulation has become a significant bottleneck in the development of VLSI systems.

To address this important issue, the research community has expended considerable effort investigating the use of parallel processing to accelerate logic simulation. This work was recently surveyed by Bailey et al. [4]. In addition, a great deal of effort has been expended on parallel techniques for general discrete-event simulation (e.g., the Workshop on Parallel and Distributed Simulation is in its ninth year [1, 2, 3, 5, 19, 22, 24, 30, 31]). An excellent survey of this work is described by Fujimoto [13].

This tutorial provides an introduction to the problem of parallel logic simulation, including the logic simulation model, parallel simulation techniques, factors that impact simulation performance, performance results, and the directions currently being investigated by the research community.

[1] The author's research is supported by the National Science Foundation under grant MIP-9309658. He can be contacted at Campus Box 1115, One Brookings Dr., St. Louis, MO, 63130, or roger@wuccrc.wustl.edu.

II. Parallel Simulation Model

During design verification, VLSI systems are frequently simulated across a wide variety of abstraction levels, from continuous models at the circuit level to block-structured models at the behavioral level. Here, the term *logic simulation* is used to refer to any discrete-event simulation of a VLSI system, where the system components can vary from the transistor level (modeled as ideal switches), through the gate level (e.g., NANDs, flip-flops), to the behavioral level (e.g., multipliers, functional units).

In discrete-event simulation, system state variables are modeled as discrete-valued quantities that change value at discrete points in time. In logic simulation, the state variables typically represent signal levels on wires that interconnect circuit elements. In the simplest two-valued logic simulations, state variables are constrained to two quantities representing Boolean values (i.e., 0 or 1). Most modern logic simulators use multi-valued variables to represent additional information. For example, many switch-level simulators add an X state to represent unknown or floating signals, and gate-level simulators add states to represent drive strength and high impedance conditions. The IEEE standard logic system for VHDL simulation (STD_LOGIC_1164) uses a 9-valued logic [6].

There are a number of ways in which parallelism can be exploited to improve simulator performance. *Algorithm parallelism* uses pipelining techniques to accelerate the simulation loop by executing individual program steps on different processors (e.g., event queue management, functional evaluation). A limited amount of parallelism is available using this technique, since there are a limited number of steps in the simulation loop. *Data parallelism* uses different processors to simulate the circuit for distinct input vectors. This technique is quite effective for fault simulation, where a large number of independent input vectors need to be simulated. It is less effective, however, during design verification, where the goal is to minimize the completion time of an individual input vector. *Model parallelism* uses different processors to perform the functional evaluations for distinct logic elements. This tutorial concentrates on techniques for exploiting model parallelism.

To facilitate parallel execution of the simulation, the system components (at whatever level of abstraction) are considered to be atomic elements that are each encapsulated into a *logical process* (LP). Many implementations combine more than one component into a single LP, but this does not impact the basic simulation model described below. It can, however, impact the performance of the simulator.

The LPs are responsible for managing local state information for their component(s), processing simulation events, and maintaining a local simulated time reference. The components, or LPs, interact via communications *channels*, which model the circuit connectivity of the VLSI system. A change in the output of an LP (e.g., a 0 to 1 transition at a gate output) is communicated to the fanout LPs by delivering a time stamped message to each fanout LP.

For parallel execution, the LPs are partitioned and assigned to the available processors, and some time synchronization algorithm (a number are described below) is used to ensure correct coordination of the simulated times within each LP.

Although a functionally correct parallel logic simulation is relatively straightforward to design and implement, the primary purpose of parallelism is improved performance. Bailey et al. [4] identified five primary factors that influence the performance of parallel logic simulation: timing granularity (the resolution of simulated time), circuit structure (topology, component fanouts, etc.), target architecture, partitioning and mapping (assignment of LPs to processors), and time synchronization algorithm (used to coordinate simulated time between LPs).

Since the first two are typically determined by the simulation modeler (e.g., a timing granularity of 1 ns is of limited usefulness for very high frequency designs), the simulator design must react to variations in these factors, but it cannot control them. Clearly, course timing granularity provides for greater event simultaneity, simplifying the task of extracting parallelism. The relationship between circuit structure and simulator performance is not at all understood well; however, there definitely is a connection between the two. With all other factors equal, parallel simulator performance can vary dramatically from one circuit to the next.

The architecture of the execution machine clearly impacts the performance of any parallel algorithm. Due to the fine grain nature of logic simulation, communications capability in the parallel system is often the discriminating property between candidate execution platforms. Although implementations exist on both SIMD and MIMD machines, MIMD is clearly superior when many different component models are present in a circuit (e.g., hierarchical systems). A number of implementations exist on networks of workstations, often taxing the communications performance of the interconnecting local-area network. Emerging high-performance networks (e.g., ATM based) are seen an opportunity to help alleviate this communications performance bottleneck for the workstation network execution platform.

III. Circuit Partitioning

When assigning LPs to processors for execution, two competing requirements need to be balanced, a uniform computational load across the processors and a minimum of communications volume between processors. Since finding an optimal partitioning is computationally complex (NP-hard), the emphasis has been on developing efficient heuristics with near optimal results. Many of these heuristics are based on direct graph-partitioning algorithms or iterative adjustment algorithms that attempt to minimize a given cost function. They are often derived from algorithms originally developed for physical partitioning (e.g., min-cut algorithms and/or simulated annealing).

One of the earliest partitioning algorithms described specifically for logic simulation is the strings algorithm of Levendel et al. [17]. Starting at a primary input component, the component output is followed to a fanout component, the fanout component's output is followed to one of its fanout components, etc. until a primary output is reached. The "string" of components formed above is assigned to a processor, and the process repeats, forming another string. Analogous to the depth first search implicit in string partitioning, fanin and fanout cones (proposed by Smith et al. [25]) spread out from an initial gate in a breadth first manner.

Many logic partitioning algorithms borrow ideas from physical partitioning algorithms originally developed to address the placement problem. For example, Fiduccia and Mattheyses' [12] min-cut algorithm and other graph-based bisection algorithms [16] have been used extensively for logic partitioning with good results. In addition, simulated annealing has been used; however, its results are mixed. Simulated annealing has suffered from two problems: (1) the execution time required of the partitioning algorithm is prohibitively long, and (2) it is difficult to develop appropriate cost functions to guide the annealing process.

One of the major difficulties implicit in any partitioning algorithm for logic simulation is the fact that the computational workload associated with each LP is a function of its evaluation frequency. If the inputs to a gate are stable, event-driven simulation algorithms do not evaluate the gate. Since the input signals to the individual gates are a function of the test vectors used for a particular simulation execution, the evaluation frequency of each gate and, therefore, its computational workload requirements are unknown prior to execution. To address this issue, the idea of *pre-simulation* has been proposed [9]. Essentially, the simulation is run for a period of time and the evaluation frequency of each gate is measured. This measured evaluation frequency is then assumed to persist for the remainder of the simulation execution. Although it is unclear how well this technique will work in the general case, it has proven successful when using random test vectors.

Another important issue typically addressed during the partitioning process is the granularity of the LPs (i.e., how many atomic components are contained within each LP). Only one gate per LP can result in high overhead processing incoming messages, while only one LP per processor can result in unnecessarily blocked computation or high rollback overheads (see below). As a result, the optimum granularity is somewhere between these two extremes.

IV. Time Synchronization

One topic that has clearly dominated parallel simulation research is the algorithm used to coordinate simulated time across the LPs. Although not an ideal classification, time synchronization algorithms are often put into one of four categories: oblivious, synchronous, conservative asynchronous, and optimistic asynchronous. Each of these categories is explained below in its basic form; however, there are many variations of each of these algorithms [13].

The oblivious algorithm is not event driven at all. At every point in simulated time, every LP is evaluated, whether or not its inputs have changed. This completely eliminates the need for an event queue, and if the evaluations of LPs are properly scheduled, correctness can be guaranteed (components are evaluated after their inputs values are known). The appropriateness of this style of algorithm is highly dependent upon the activity (frequency of state changes) within a circuit. At low activity levels, redundant evaluations are an enormous overhead. At higher activity levels, the elimination of the event queue (and its associated overhead) can lead to a performance advantage.

The simplest event-driven algorithm is the synchronous technique. Here, the simulated time at all of the LPs is constrained to be the same. The LPs process their events at the present simulated time and then coordinate (typically via a barrier synchronization) to determine the next point in simulated time that has events to be processed. This technique is also referred to as a global-clock algorithm, since there is one globally consistent value of simulated time.

The two asynchronous algorithms (conservative and optimistic) allow simulated time to vary from one LP to the next. They differ in the rules used to process incoming messages and advance simulated time at individual LPs. Conservative algorithms process messages in strictly non-decreasing order, preserving causality constraints at all times [11, 20]. This *safety* condition is enforced by advancing local simulated time to the smallest time stamp received from any neighboring LP. This rule (called the input waiting rule) can lead to blocking and even deadlock; therefore, techniques are needed to prevent (or detect and resolve) deadlock.

Deadlock prevention is usually accomplished via *null* messages, messages with a time stamp but no other content. Essentially, a null message is a way for an LP to notify its downstream neighbors that their inputs are stable up to the time of the time stamp. Deadlock detection is often accomplished via circulating marker algorithms that invoke a deadlock resolution algorithm when a marker completes an entire cycle without detecting simulation activity.

The original optimistic algorithm is the Time Warp algorithm of Jefferson [15]. In the optimistic approach, simulation messages are processed immediately upon receipt at an LP. If a straggler message is received with a time stamp earlier than the local simulated time, then the LP executes a *rollback*. The rollback restores the state of the LP to an earlier state so that the straggler message can be processed without violating causality. Thus each LP must save state so that it can rollback. Since state saving can be a time consuming operation, frequently only the change in state is saved, not a complete copy of the state. This technique is referred to as incremental state saving. As part of a rollback, if outgoing messages have been delivered to downstream LPs, they are sent anti-messages to cancel the original message. The receipt of an anti-message at an LP will also trigger rollback, since the effects of the original message must now be canceled.

If the simulation runs for a long time, and memory is finite, then saved state must be reclaimed. Therefore, optimistic algorithms periodically compute a bound (called global virtual time, or GVT) such that all but one state that has time stamp less than GVT can be discarded. GVT is simply the minimum of the local simulated times at each LP and the time stamps of messages currently in transit.

Gafni's *lazy cancellation* strategy reduces the impact of roll back on the performance of simulation [14]. Instead of *aggressively* cancelling previously sent messages whenever roll back occurs, the lazy cancellation algorithm waits to cancel the message until it is known that the wrong message had been sent. Thus, if the right event had been calculated for the wrong reasons, the receiving processor is not inhibited because of excessive causality constraints.

V. Simulator Performance

The biggest difficulty in comparing the performance of different parallel logic simulators is the fact that there are no acceptable benchmarks to standardize the workload across distinct implementations. Although both the ISCAS-85 combinational benchmarks [8] and the ISCAS-89 sequential benchmarks [7] have been pressed into service, they were not originally designed to be simulation benchmarks. As a result, they

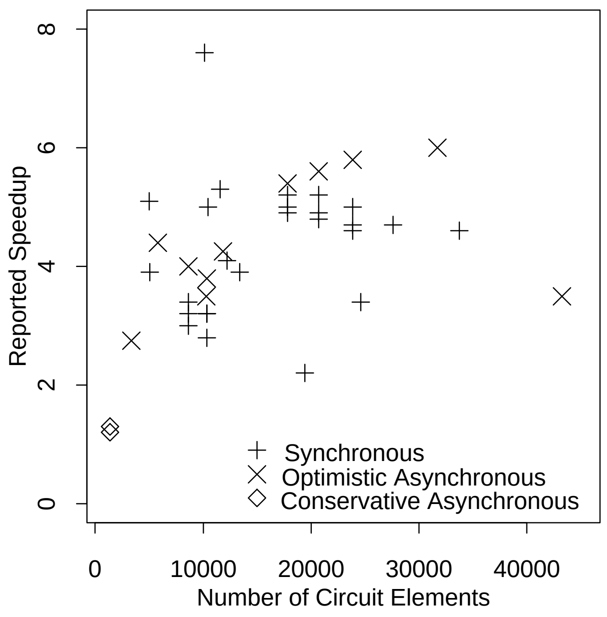

Figure 1: Representative performance results [4].

do not include test vectors (they are typically simulated using random vectors), they are all at the gate level of abstraction, and they are insufficient in size to satisfactorily evaluate performance on large circuits.

In spite of the difficulties in comparing results, a number of implementations exist. One of the first successful implementations was the optimistic asynchronous simulator of Briner et al. [10]. He reported speedups of up to 23 on 32 processors of a BBN GP1000 system. Bailey et al. [4] combine reported speedup results (for 8 processors) from a number of implementations using synchronous, conservative asynchronous, and optimistic asynchronous algorithms. These results are presented in Figure 1.

Note that there are a large number of differences between these implementations, including different abstraction levels, timing models, example circuits, execution platforms, and implementors (e.g., Briner et al. [10], Mueller-Thuns et al. [21], Soule and Gupta [26] Sporrer and Bauer [27], and Su and Seitz [29]). This limits the ability to draw firm conclusions; however, a number of trends are evident. First, none of the conservative asynchronous implementations reported good performance, while a number of synchronous and optimistic asynchronous implementations performed well. The timing granularity of the optimistic results varies from fine grain to coarse grain, but all of the synchronous implementations use coarse grain timing.

One problem that is of concern with the optimistic asynchronous algorithms is inconsistency in performance. Seemingly small variations in circumstances can trigger dramatic swings in performance results. This problem has also been observed in simulations of other application domains [18]. The synchronous algorithm does not seem to be prone to this type of behavior. In addition, incremental state saving is crucial to achieving good performance with optimistic algorithms.

Synchronous algorithms have their own problems, however. They have difficulty scaling to large numbers of processors since the time required to perform the barrier synchronization grows with processor population. Also, they are prone to load imbalance. An even distribution of LPs across the processors is insufficient to balance the computational workload if the evaluation frequency of individual LPs varies.

VI. Future Directions

The performance results to date seem to indicate that for coarse timing granularity a synchronous algorithm is sufficient and for fine timing granularity an optimistic asynchronous algorithm is needed. This is an oversimplification of the situation, however, since there are circumstances that can significantly impact the performance results that go well beyond timing granularity. As a result, there are no known implementations that consistently perform well independent of the circuit simulated and the test vectors applied.

This points to an important body of ongoing work. In partitioning, pre-simulation has been proposed to estimate the computational workload of components for load balancing purposes. Appropriate cost functions are being investigated for both simulated annealing and other iterative improvement partitioning algorithms. Also, dynamic load balancing is being considered to react to variations in computational workload.

In the area of time synchronization algorithms, the synchronous algorithm is being expanded to include many of the features found in asynchronous algorithms, with an attempt to avoid the performance instabilities found in the asynchronous algorithms. Positive results have been presented for other application domains (military simulation and queueing network simulation) by Steinman [28] and Noble et al. [23]. Optimistic asynchronous algorithms are being extensively studied in an attempt to understand how they can be effectively controlled to deliver consistent performance.

Hybrid algorithms are also under investigation. Ideas include hierarchical synchronization, using either a synchronous or conservative asynchronous algorithm within a cluster of processors and using an optimistic asynchronous algorithm across clusters. This appears especially attractive for naturally hierarchical execution platforms (e.g., networks of workstations where the individual workstations are bus-based multiprocessors).

Finally, there is a strong need for a benchmark set that addresses the needs of the logic simulation research community. This set should have large circuits, at varying levels of abstraction, with varying timing granularity, and test vectors typical of those used during the design verification process.

Acknowledgment

The author would like to thank Mary Bailey for her insightful comments on an early draft of this manuscript.

References

[1] M. Abrams and P.F. Reynolds, Jr., eds. *Proc. of the 6th Workshop on Parallel and Distributed Simulation*, SCS, 1992.

[2] D.K. Arvind, R. Bagrodia, and Y.-B. Lin, eds. *Proc. of the 8th Workshop on Parallel and Distributed Simulation*, SCS, 1994.

[3] R. Bagrodia and D. Jefferson, eds. *Proc. of the 7th Workshop on Parallel and Distributed Simulation*, SCS, 1993.

[4] M.L. Bailey, J.V. Briner, Jr., and R.D. Chamberlain. Parallel logic simulation of VLSI systems. *ACM Computing Surveys*, 26(3):255–294, September 1994.

[5] M.L. Bailey and Y.-B. Lin, eds. *Proc. of the 9th Workshop on Parallel and Distributed Simulation*, IEEE, 1995.

[6] W.D. Billowitch. IEEE 1164: Helping designers share VHDL models. *IEEE Spectrum*, 30(6):37, June 1993.

[7] F. Brglez, D. Bryan, and K. Kozminski. Combinational profiles of sequential benchmark circuits. In *Proc. of the Int'l Symposium on Circuits and Systems*, IEEE, 1989.

[8] F. Brglez and H. Fujiwara. A neutral netlist of 10 combinational benchmark circuits and target translator in Fortran. In *Proc. of the Int'l Symp. on Circuits and Systems*, IEEE, 1985.

[9] J.V. Briner, Jr. Parallel Mixed-Level Simulation of Digital Circuits Using Virtual Time. Ph.D. thesis, Duke University, Durham, N.C., 1990.

[10] J.V. Briner, Jr., J.L. Ellis, and G. Kedem. Breaking the barrier of parallel simulation of digital systems. In *Proc. of the 28th Design Automation Conf.*, pages 223–226, ACM, 1991.

[11] K. M. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Communications of the ACM*, 24(11):198–206, 1981.

[12] C.M. Fiduccia and R.M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proc of the 19th Design Automation Conf.*, pages 175–181, ACM, 1982.

[13] R.M. Fujimoto. Parallel discrete-event simulation. *Communications of the ACM*, 33(10):30–53, October 1990.

[14] A. Gafni. Rollback mechanisms for optimistic distributed simulation. In *Proc. of the SCS Multiconf. on Distributed Simulation*, pages 61–67, SCS, 1988.

[15] D. Jefferson. Virtual time. *ACM Trans. Programming Languages and Systems*, 7(3):404–425, July 1985.

[16] B.W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49(2):291–307, 1970.

[17] Y.H. Levendel, P.R. Menon, and S.H. Patel. Special-purpose computer for logic simulation using distributed processing. *Bell System Technical Journal*, 61(10):2873–2909, 1982.

[18] Y.-B. Lin and E.D. Lazowska. Processor scheduling for Time Warp parallel simulation. In *Proc. of the SCS Multiconf. on Advances in Parallel and Distributed Simulation*, pages 11–14, SCS, 1991.

[19] V. Madisetti, D. Nicol, and R. Fujimoto, eds. *Proc. of the SCS Multiconf. on Advances in Parallel and Distributed Simulation*, SCS, 1991.

[20] J. Misra. Distributed discrete-event simulation. *ACM Computing Surveys*, 18(1):39–65, March 1986.

[21] R.B. Mueller-Thuns, D.G. Saab, R.F. Damiano, and J.A. Abraham. VLSI logic and fault simulation on general-purpose parallel computers. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 12(3):446–460, March 1993.

[22] D. Nicol, ed. *Proc. of the SCS Multiconf. on Distributed Simulation*, SCS, 1990.

[23] B.L. Noble, G.D. Peterson, and R.D. Chamberlain. Performance of synchronous parallel discrete-event simulation. In *Proc. of 28th Hawaii Int'l Conf. on System Sciences*, Vol. II, pages 185–186, IEEE, 1995.

[24] P. Reynolds, Jr., ed. *Proc. of the Conf. on Distributed Simulation*, SCS, 1985.

[25] E.J. Smith, B. Underwood, and M.R. Mercer. An analysis of several approaches to circuit partitioning for parallel logic simulation. In *Proc. of the Int'l Conf. on Computer Design*, pages 664–667, IEEE, 1987.

[26] L. Soule and A. Gupta. An evaluation of the Chandy-Misra-Bryant algorithm for digital logic simulation. In *Proc. of the 6th Workshop on Parallel and Distributed Simulation*, pages 129–138, SCS, 1992.

[27] C. Sporrer and H. Bauer. Corolla partitioning for distributed logic simulation of VLSI circuits. In *Proc. of the 7th Workshop on Parallel and Distributed Simulation*, pages 85–92, SCS, 1993.

[28] J. Steinman. SPEEDES: A multiple synchronization environment for parallel discrete-event simulation. *Int'l Journal in Computer Simulation*, 2:251–286, 1992.

[29] W.-K. Su and C.L. Seitz. Variants of the Chandy-Misra-Bryant distributed discrete-event simulation algorithm. In *Proc. of the SCS Multiconf. on Distributed Simulation*, pages 38–43, SCS, 1989.

[30] B. Unger and R. Fujimoto, eds. *Proc. of the SCS Multiconf. on Distributed Simulation*, SCS, 1989.

[31] B. Unger and D. Jefferson, eds. *Proc. of the SCS Multiconf. on Distributed Simulation*, SCS, 1988.