

Multithreading VHDL Simulation

Hansen Dai and Bill Paulsen

Viewlogic Systems, Inc.
47211 Lakeview Blvd.
Fremont, CA 94538

Abstract

This paper describes SpeedWave/MT, a parallel VHDL simulator which uses multithreading to achieve simulation speedup on Symmetric Multi-Processing (SMP) machines. We take advantage of the inherent parallelism in VHDL, and also recent technology advances in both hardware and software for SMP machines to speedup VHDL simulation. Our approach parallelizes the execution of signal updates and VHDL processes in every simulation delta cycle. Multiple POSIX-style threads are created, up to the number of physical processors, to execute the computation tasks in parallel. Threads communicate with each other through shared memory and use spin lock for synchronization. Simulation speedup is expected for VHDL models with high activity at any simulation instant and high granularity of each computation task. Our simulation experiments show up to four times speedup on an 8-processor SMP machine with real world circuits.

1.0 Introduction

Fast simulation is essential to shortening design cycles since simulation is currently a key tool used in design verification. The computation power required to verify a design is usually exponential to the size of the design. Therefore, the demand of faster simulators is even stronger when designs get larger and more complex. Any technique of speeding up simulations is hence very important to improving productivity and time to market. Parallel simulation is one obvious technique which utilizes multiple physical processors to execute different parts of a system in parallel.

Parallel simulation algorithms [1] can be classified into three categories: lock time step, conservative [2], and optimistic [3] approaches. In the lock time step approach, the simulation time of different simulated parts is synchronized. Depending on the work load distribution among processors and the activity of the simulated system, some processors might starve and become idle. Also, due to long communication latency of loosely coupled multiprocessor systems, it is very difficult to obtain speedup. The latter two approaches try to relax the synchronous constraint and let different simulated parts advance simulation time asynchronously. The main purpose is to avoid processor starvation and keep processors busy doing useful work. The conservative approach allows a simulated part to advance its simulation time up to a point without causing any causality error. Deadlock could occur if different simulated parts block each other based on insufficient causality information. It turns out that the overhead of managing deadlock is very high. On the other hand, the optimistic approach allows causality errors and uses a rollback mechanism to recover from errors. A simulated part advances simulation until it receives an event in its past and then rolls simulation time back to the point for re-executing this event. In this approach, the handling of false messages is computationally intensive, and memory management for state saving and rollback is very expensive in current general purpose computers. Besides, all of these three approaches, in general, require the partitioning of the simulated system into subsystems. A partitioning task is not trivial and its outcome greatly affects simulation performance.

Our approach falls into the first category. We parallelize the computation in every VHDL simulation cycle. Computation tasks of a simulation cycle are arranged in a

global queue. Multiple threads, up to the number of physical processors, repeatedly retrieve tasks from the queue and execute tasks in parallel until the queue is empty. Then the simulator advances to the next simulation cycle. No circuit partitioning is required. Work load is balanced by letting threads compete with each other. However, the performance of our approach is limited by the concurrency of the circuit being simulated and the cost of contention. Our implementation has been emphasized reduction of the overhead of contention. Later, we will profile the type of circuit designs that can be accelerated by SpeedWave/MT.

In the remainder of this paper, we will describe SMP briefly. Then, section 3 presents our approach in detail. Experimental results of our approach are shown in section 4. We also discuss speedup factors and VHDL coding guidelines in section 5. Finally, conclusions and future directions are presented.

2.0 Symmetric Multi-Processing

Symmetric Multiprocessing (SMP) [4, 5] is a new parallel processing architecture consisting of technology advances in both hardware and software. It has demonstrated significant performance improvement in many applications with coarse grain parallelism such as client/server computing and multimedia. EDA simulation is one of the areas with great potential for achieving speedup.

2.1 Hardware Platform

With the advances of VLSI integration technology, it has become possible to integrate multiple processors into a desktop at very low cost. Sun Microsystems is currently the market leader selling both software and hardware. We understand that some workstation vendors plan to announce SMP products very soon. Some other vendors are also working on x86 multiprocessing. The benefit to users is good price performance.

In an SMP machine, multiple processors are tightly coupled through shared memory. All processors are identical and have equal access to shared memory. The hardware guarantees memory consistency of read/write operations among multiple processors. An atomic instruction such as test-and-set or load-and-set is supported by the hardware to implement software synchronization primitives.

2.2 Software Support

In addition, software support plays an important role in SMP technology. It provides not only the parallelized operating system, but also a programming interface for applications to access the power of multiple processors. With multithreading, an application can be decomposed into multiple tasks to take advantage of multiple processors. Multithreaded programs go through a threads interface (POSIX 1003.4a draft) to perform thread operations such as thread creation, scheduling, and synchronization.

A thread is a sequence of program execution steps. Within a UNIX process, it is an independent thread of control. Threads within a process also share most of the process address space and process resources such as file descriptors. Thus, context switching between threads is much cheaper than that between processes, and thread communication is easy. Multithreading techniques give an efficient way to utilize the parallelism of multiprocessor hardware which can provide multiple simultaneous execution points.

3.0 Parallel VHDL Simulation

3.1 VHDL Simulation

Figure 1 shows the execution flow of one VHDL simulation cycle. First the simulator advances to the earliest time with circuit activity. Then the simulation is divided into three sequential steps: transaction execution, signal evaluation, and process execution. Each of these three steps consists of many independent computation tasks. Transaction execution performs signal updates scheduled for the current simulation time. After this step, signal evaluation takes care of the updates of implicit signals such as resolved signals which require their sources to be updated first. Finally, process execution executes active VHDL processes which in turn schedule future events. Although it is not necessary to execute the three steps sequentially, this guarantees the equivalent functionality as the execution specified by the VHDL Language Reference Manual (LRM) [6].

According to the LRM, the order of processing explicit signal updates in one simulation cycle is not specified. Therefore, they can be executed concurrently. Similarly, active processes of one simulation cycle can be executed concurrently. As to resolved signals, a certain partial ordering of evaluation dependency is required for

correct simulation. However, independent signals can be evaluated concurrently. With the support of multiprocessors, parallel processing is possible by utilizing the inherent parallelism in VHDL.

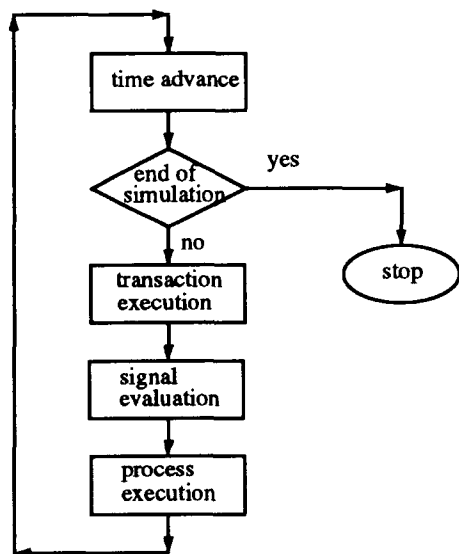


FIGURE 1. The execution of a simulation cycle.

3.2 Multithreaded Simulation Cycle

In SpeedWave/MT, a master thread controls the execution flow, prepares the task queue, and executes computation tasks. In addition, $N-1$ slave threads are created to execute computation tasks, where N is a user specified number up to the number of physical processors. Threads are one-to-one bound with physical processors.

Threads are synchronized by two concurrency control mechanisms:

1. *barrier*: is a mechanism to pause the execution of a thread at the barrier until the barrier is released. It is actually implemented by a busy wait. In this way, all threads are synchronized at one desired simulation point. For instance, a barrier is used to synchronize all threads at the end of a simulation cycle before advancing to the next simulation cycle.
2. *spin lock* [7]: provides a mutually exclusive access to a global resource or *critical section*. Once the lock is acquired by one thread, any other thread that needs to enter the critical section spins on the lock until the lock is released and then tries to acquire the lock again. The lock acquisition is implemented by an atomic machine instruction.

In the beginning of a simulation cycle, the master thread advances simulation time. Then all threads execute transactions until the transaction queue is empty. If any thread finishes execution earlier than other threads, it needs to wait at the barrier designated for the end of transaction execution. When all threads are synchronized, the master thread releases this barrier and advances simulation to signal evaluation step. All threads then begin to evaluate resolved signals in parallel and meet at the end of signal evaluation. Similarly, processes activated at this simulation cycle are executed in parallel by threads.

When threads execute computation tasks in parallel, a spin lock is used to serialize the access to a critical section. For example, the process queue is shared by all threads and therefore is a critical section. During process execution, each of the master and slave threads acquires the lock, gets one process, releases the lock, and executes the process. These operations are repeated until the process queue is empty. This is our approach to balance the load distribution among threads. Every thread is always busy before the process queue becomes empty. However, this approach creates contention overhead. If the computation amount of each process is too small and comparable to the overhead of spin lock, threads end up waiting for locks.

4.0 Simulation Results

We analyzed and benchmarked five real world circuits. The circuit analysis we conducted is intended to reflect circuit parallelism and granularity which are related to the speedup figure presented later. Analysis data were obtained by instrumenting our sequential simulator, SpeedWave.

The nature of these five circuit designs is described as follows:

- Circuit A is a structural design of an ATM model with fairly large primitive components.
- Circuit B is a behavioral design with many levels of hierarchy and lots of complex composite signals.
- Circuit C is a highly structural design with lots of primitive components coming from a third party ASIC library.
- Circuit D is a mixed behavioral and structural design with lots of gate-level components and a few large behavioral processes for finite state machines.

- Circuit E is a pure gate-level structural design with very fine grain primitive components.

In Table 1, the sizes of these circuits are characterized in terms of the numbers of components, processes, and signals.

TABLE 1. Sizes of circuits.

	components	processes	signals
Circuit A	206	250	31971
Circuit B	150	403	62564
Circuit C	7892	9111	26754
Circuit D	387	2232	197980
Circuit E	190	339	2960

Table 2 shows a snapshot of process execution patterns of these five circuits over 10 simulation cycles. These numbers representing the number of active processes in one simulation cycle are sampled from a random simulation time t for 10 consecutive simulation delta cycles. We can easily see the degree of parallelism of each circuit.

Table 3 presents the average number of active VHDL processes and the average number of scheduled transactions in one simulation cycle as well as average CPU time in terms of microseconds spent on executing each process. This table is intended to characterize parallelism and granularity. The average number of processes and transactions executed in one simulation cycle demonstrates parallelism, and the average CPU time spent on each process offers an idea about granularity. All the statistics were measured and averaged over a full simulation run.

TABLE 2. A snapshot of process execution patterns over 10 simulation delta cycles.

	time t	$t+\Delta$	$t+2\Delta$	$t+3\Delta$	$t+4\Delta$	$t+5\Delta$	$t+6\Delta$	$t+7\Delta$	$t+8\Delta$	$t+9\Delta$
Circuit A	1	100	50	50	1	100	1	100	50	100
Circuit B	1	1	123	15	1	1	123	10	1	1
Circuit C	2	2	883	20	56	15	2	43	2	2
Circuit D	1	14	2	1	123	1	1	16	3	1
Circuit E	1	28	26	26	70	29	1	2	24	28

TABLE 3. Statistics on parallelism and granularity.

	processes	transactions	CPU time (μ s/process)
Circuit A	56	2729	450
Circuit B	30	2600	720
Circuit C	130	157	1150
Circuit D	16	45	115
Circuit E	26	23	30

Simulations of these circuits were performed on a Sun Sparc Server 1000 with 8 processors. Figure 2 shows the speedups of simulation results. Speedup as high as four was obtained while simulating Circuit A with 8 processors. However, no speedup was achieved for Circuit D.

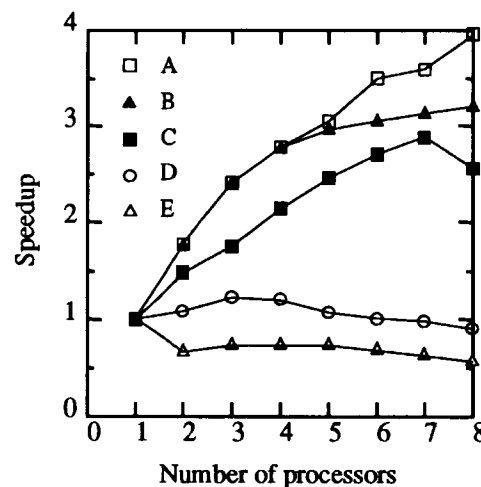


FIGURE 2. Speedups against number of processors.

5.0 Discussions

Among these circuits, Circuit A has the best amount of parallelism. Although its granularity is not the highest, Circuit A performs the best speedup scalability. Granularity seems not to affect performance as long as it exceeds a certain threshold which makes the elapsed time of contention overhead overlap with the execution of a process.

Circuit B has almost identical speedup scalability to that of Circuit A for number of processors less than five. When simulated with more than four processors, speedups flatten out due to insufficient parallelism.

Circuit C has the highest granularity. This is because it employs a very general third party ASIC library which includes timing checking, range checking, and delay calculations. The overhead of the ASIC library shows up in the granularity of process execution, considering the number of transactions scheduled by processes as listed in Table 3. In other words, each process schedules less than two signal updates, but takes more than 1000 microseconds CPU time. The parallelism distribution over simulation cycles is not as even as that of Circuit A. We did obtain fairly good speedup results, but it did not scale as well as Circuit A.

Circuit D shows very low circuit activity by comparing the number of scheduled signal updates (column 2 of Table 3) with total number of signals (column 3 of Table 1). Obviously, this circuit is modeled sequentially. The parallelism in VHDL is not utilized. Its speedup figure demonstrates that the parallelism and granularity of this circuit seems to be the minimal requirement for circuits to be sped up.

Circuit E is a small design with moderate parallelism but very low granularity. Also, one text I/O process which consumes almost 30% of total process execution time is always the only active process in a simulation cycle. No concurrency can be applied in the simulation cycle by our approach.

Based on our experimental study, we try to depict the factors that would affect simulation speedup and outline the coding styles that would help improve performance.

5.1 Speedup Factors

Speedup factors strongly depend on the parallel computing model employed by the simulator. Here, we try to avoid any factors that are too implementation specific.

In general, speedup is affected by the circuit activity at any simulation instant and the amount of computation in each computation task. Furthermore, it is known that file I/O can be sped up by multithreading.

5.1.1 Parallelism

In our approach, it is better to have as many scheduled transactions and active processes as possible at every simulation cycle. On the average, the five benchmark circuits show that less than 2% of circuit components is active at any moment. In other words, these circuits are modeled in a very sequential way. In general, larger circuits tend to have higher parallelism. Based on our experiences, there should be, on the average, at least 20 active VHDL processes in every simulation cycle in order to obtain speedup and to even out the *last process effect* that results when all threads but one are waiting at a barrier. In this sense, a synchronous design with at least 20 processes sensitive to clock is a good candidate.

5.1.2 Granularity

Due to synchronization overhead, the larger the granularity of each task, the better performance we can get. Especially, as more threads are involved in computation, the synchronization overhead and contention get higher. A rule of thumb for granularity is about 100 microseconds of CPU execution time per task in order to overcome the overhead. A gate level component with one AND/OR operation and one scalar signal assignment has very low granularity. In terms of physical meaning, any VHDL process with at least 20 scalar signal assignments or several equivalent composite signal assignments has sufficient granularity and is a good target for achieving speedup with SpeedWave/MT. In fact, most processes of Circuit A contain about 5 to 10 signal assignments including a few composite signal assignments or equivalent to totally 120 scalar signal assignments. However, this is just a guideline, not a strict rule. For instance, in a behavioral model like Circuit B, behavioral code other than signal assignments would certainly consume some CPU time.

5.1.3 I/O

In a single threaded simulator, I/O could block the execution of remaining tasks in the queue. It is obvious that multiple threads let process execution overlap with one or more I/O requests. Nevertheless, we haven't found a circuit design to demonstrate the benefit of multithreading.

5.2 Coding Styles for Improving Performance

VHDL is able to model the concurrent activity of circuits. Circuit designers should take advantage of this feature and improve parallelism in models, which will benefit the performance of parallel simulation.

One coding practice that could help reduce contention overhead is to collapse several related small processes into a big process. Usually, these processes are interconnected by bridge signals and activated sequentially. By grouping them together, bridge signals can be eliminated and granularity is improved.

We have noticed that some circuit designs with clock resolution in nanoseconds have simulation time increments in femtoseconds. The designers simply use this technique to model the dependency of circuit components. This causes the simulator to create lots of unnecessary simulation time points. In SpeedWave/MT, we maintain a single time wheel for scheduled time points. The time wheel is a global resource shared by multiple threads. Insertion of a new time point requires locking/unlocking the time wheel, which is very expensive. Simulation performance can be improved a lot if the circuit designs can utilize zero delay assignments which use an existing time point.

6.0 Conclusions

We presented a multithreaded VHDL simulator which parallelizes the computation of every simulation cycle. Simulation speedups of up to four times on an eight processor system were obtained. The results are very encouraging. We will continue developing new algorithms to reduce contention overhead and to expand the types of circuit designs that can be accelerated. Future work will also focus on relaxing the lock step constraint within one simulation cycle and between simulation cycles.

One important factor that makes this product possible is the SMP technology. It is expected to be inexpensive to upgrade a system with uniprocessor to multiprocessors. Multithreading will be the future computing model. While system clock rates are being pushed to the physical limit, the technology of integrating multiple processors into a system becomes more promising, so do the software applications riding on the technology wave.

References

- [1] R.M. Fujimoto. "Parallel discrete event simulation", *Communications of ACM*, October 1990.
- [2] K.M. Chandy and J. Misra. "Distributed simulation: A case study in design and verification of distributed programs", *IEEE Trans. on Software Engineering*, September 1979.
- [3] D.R. Jefferson. "Virtual time", *ACM Trans. on Programming Languages and Systems*, July 1985.
- [4] S. Kleiman, J. Voll, J. Eykholt, A. Shivalingiah, D. Williams, M. Smith. "Symmetric multiprocessing in Solaris 2.0", *COMPCON Conference Proceedings*, Spring 1992.
- [5] SunSoft. *SunOS 5.3 Guide to Multithreaded Programming*, November 1993.
- [6] IEEE. *IEEE Standard VHDL Language Reference Manual*. IEEE Std 1076-1987, March 1988.
- [7] *The SPARC Architecture Manual, Version 8*, Prentice-Hall, 1991.

Acknowledgments

The authors would like to thank Mr. Morgan Herrington of Sun Microsystems for his assistance in multithread programming techniques.