

Parallel Simulation in Metropolis

Guang Yang

guyang@eecs.berkeley.edu

Abstract. Electronic system design has been becoming more and more complex. It is very common to have dozens of IP modules in a single system. To quickly simulate the system is becoming a challenge. On the other hand, computing facilities, on which people run actual simulation, are becoming more and more powerful by having more powerful single processor and by building parallel machines. However, there is not too much attempts in exploring parallelism in electronic design automation community esp. for system level design. In this paper, I present a parallel simulator targeting Symmetric Multi-Processor (SMP) machines for Metropolis design environment. Since the careful implementation of parallelism, the portability and scalability are maximized.

1 Introduction

In today's electronic design automation (EDA) field, there are two main approaches to do design validation, formal verification and simulation. Although formal verification has been improved for almost 20 years, it is still quite limited in the applicable problem sizes. On the other hand, simulation has been dominating design validation and will be so in the coming decade.

From the design aspect, to deal with constantly increasing complexity, electronic system designers are turning to more rigorous design methods that favor the adoption of higher levels of abstraction in system specification, correct-by-construction deployment, and re-usability. A paradigm called *platform based design* [9] has been proposed to offer a way of coping with the difficulties of designs. In this paradigm, designers evaluate several configurations of platforms before selecting one that meets design goals. This process is called *design space exploration*. It requires building a series of models, one for each combination of configurations to be evaluated usually by simulation.

Therefore, as the system/platform complexities become higher and higher, the performance of simulators is becoming a more and more crucial issue. One way to improve simulation performance is to optimize the simulation algorithm by statically analyzing the system. This is sometimes not so helpful, e.g. for those computation intensive designs like MPEG players. The other way is to explore the parallelism in the design and simulate on a parallel machine. This is especially applicable to today's system level designs, because in almost all design methodologies, the entire system is composed by smaller hardware/software blocks which run in parallel.

There are successful attempts on parallel discrete event simulators, such as TimeWarp[5][10]. It is designed for hardware description languages (HDLs). The key idea it utilizes to deal with parallelism is the complex time roll back mechanism. Another example is SimCluster by Avery Design Systems Inc.[1] It is also for HDLs. Since both systems are at a finer level of abstraction, the communication overhead is significant. Their speedup are far less than linear. Because of the improper levels of abstraction, it is not suitable for system level design esp. for Metropolis[3][2] where the notion of time does not even exist. For system level simulators, no much efforts have been spent on them so far. For example, SystemC is the most widely used system level design language. It models a system with logic threads running in parallel. However, its simulator does not take advantage of the parallelism. The entire design runs in a single process.

The rest of the paper is organized as the following. Section 2 gives an overview of Metropolis; Section 3 shows how the Metropolis simulator is parallelized; Section 4 gives the formal performance analysis of simulation; the performance of Metropolis parallel simulator is given in Section 5; finally draw conclusions.

2 Background

2.1 The Metropolis Environment

Metropolis supports the platform-based design methodology [11][9] in which the orthogonal design aspects are modeled separately, for instance, behavior and architecture, functionality and performance, computation and communication. In this design methodology, a platform is designed with sufficient flexibility to support the implementation of an entire set of products. The product design problem then involves configuring the platform, and deciding which parts of the product's functionality are to be implemented by which platform resources. Typically, designers evaluate several configurations before selecting one that meets design goals.

The Metropolis Design Environment today provides a rich set of tools to analyze a design. The core tool is the Metropolis compiler. It has a typical frontend-backend structure. The compiler frontend is in charge of the syntax and semantics check of the design. Based on the type of the analysis requested, one or more backend tools will be invoked. Currently, the following backend tools are available in Metropolis Design Environment: elaborator, simulator, LOC property checker, static scheduler, SPIN-based formal verification tool, and a refinement verification tool [4].

2.2 Metropolis Meta-Model

Metropolis Meta-Model(MMM) is the specification language used in Metropolis design environment. As its name suggests, it is not restricted to any model of computation (MOC). This gives designers modeling power in that any other MOC's can be modeled in Metropolis, e.g. YAPI (Y-chart Application Programmer's Interface), SDF (Synchronous Data Flow), FSMs.

Metropolis models a design as a network of *processes*. Each process executes an imperative sequential program. Processes communicate in the same way as observed in languages like SystemC [7] and SpecC [6], where a process calls functions declared in *interfaces* that are implemented by a dedicated type of object which we call a *medium*. When an object calls an interface function implemented in another, we say the former is *connected* to the latter. Figure 1 shows a producer-consumer example written in Metropolis MetaModel.

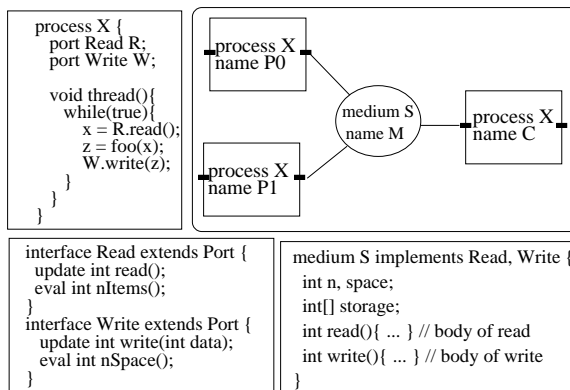


Fig. 1. producer-consumer model

3 Parallelization of Metropolis Simulator

When simulating a system in Metropolis, one more process called simulation manager is automatically added into the system. Its role is to resolve constraints among processes. Conceptually, the Metropolis simulator has two alternating phases. Processes in the system run in one phase. When they come to a point where a constraint resolution is necessary for them to proceed, it switches to the other phase, where simulation manager runs. After the simulation manager finishes its job, it switches back to the process phase, and so on. Note that this simulation algorithm does not imply any underline implementation platforms. As far as it is logically implemented this way, it can produce the consistent behavior of the system.

So, in terms of actual underline implementation platforms, there exist two possibilities. One is single-thread simulation; the other is multi-thread simulation.¹

¹ Precisely speaking, what I really mean here by single-thread is single operating system process plus user level threads; by multi-thread I mean kernel level threads.

3.1 Single-thread Simulation

In this approach, no matter how many processes a system has (including the simulation manager process), they all run in a single operating system process. This is known as user level multi-threading. In general, the advantages of this approach include

- low-cost thread operation
- less system resource consumption
- flexible (application specific) scheduling

The disadvantages of this approach include

- one thread could block the entire process
- cannot take advantages of multiprocessors

Our current implementation of Metropolis simulator is a single-thread one. It is based on SystemC simulation kernel, which is built on top of Qt user-level thread library. Figure 2 shows a sample simulation trace for the system in figure 1. The simulation time we pay is the sum of the time spent on the simulation manager, all processes in user’s system and the context switching time between adjacent process runs. It is obvious that this implementation is not scalable at all. Simulation time always increases as the number of processes increases (assuming total simulation time for one process does not decrease).

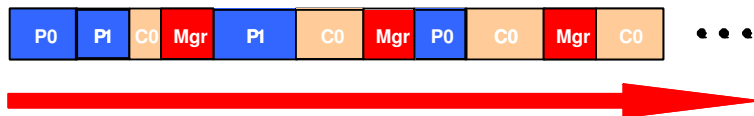


Fig. 2. A sample single thread simulation trace for producer-consumer model

3.2 Multi-thread Simulation

The primary goal of parallelizing Metropolis simulator is to increase simulation speed. At the same time, we hope to make it as portable and scalable as possible.

In order to achieve above goals, I chose SMPs as the target hardware platform, because the way MMM processes communicate, i.e. they connect to the same medium and call the interface functions defined in the medium. It is natural to create one thread for each MMM process, and allocate the medium in the shared memory so that all processes have access to it. Since Metropolis is for system level design and processes are at a high level of abstraction, there will be limited number, e.g. dozens, of them. Therefore, the number of threads is not a problem for SMP machines. In this paper, I use Seaborg nodes, SMP machines each with 16 IBM Power3 processors, as the benchmark platform.

I chose Pthread [8] library to implement the multi-thread simulator for its good performance, portability and scalability. The operating system AIX running on Seaborg nodes implements Pthreads with kernel level threads, which is essential for simulation speed up. Comparing to single-thread simulation, simulation based on kernel level threads consumes a little more system resources, but it does offer big performance improvement by utilizing all available multi-processors. Another benefit gained from kernel level threads is that operating system can automatically balance the loads among all processors. That saves the development efforts, which is otherwise significant in general. For portability, Pthread is also a great choice, because it is an IEEE standard and supported by all vendors. Finally, implementation based on kernel level Pthread is scalable intrinsically. Figure 3 shows a sample multi-thread simulation trace.

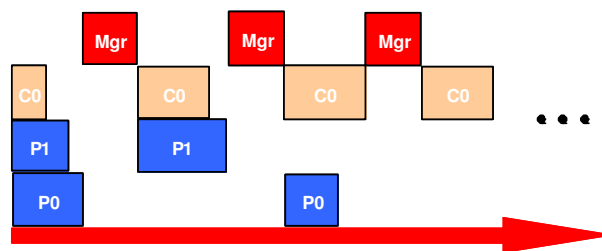


Fig. 3. A sample multi-thread simulation trace for producer-consumer model

3.3 Synchronization in Multi-thread Simulator

There are three kinds of synchronization in multi-thread simulation. Properly distinguishing and handling them are the most challenging part in developing multi-thread simulators.

1. Synchronization between simulation manager and processes
Whenever processes need to resolve constraints, they send requests to simulation manager. Since the constraints are generally related to other processes in the system, simulation manager will wait until all processes in the system to come to such points, and then do scheduling. To implement this mechanism, I introduce a unique mutex protected counter, which counts the number of processes (excluding simulation manager) that have stopped. By doing this, it could bring inefficiency because some of the processes have to wait. If there are more processes than the number of processors, OS can automatically schedule work loads of the processors; otherwise, since there is not enough work to do anyway, we can not do better than letting faster processes wait.

2. Simulation algorithm related synchronization among processes

In order for the simulation manager to resolve constraints, it is needed to record the usage status of media. Since multiple processes could access the same medium simultaneously, mutual exclusion among these processes must be enforced. I implement this by introducing one mutex for each medium. When a process interacts with a medium, it must lock the mutex first to update the usage information of the medium, and then release the mutex upon finishing updating.

3. Synchronization specified by designers among processes

Users can write explicit synchronization code in processes and media. These synchronization code should be respected, even though they may not correctly enforce mutual exclusion among processes. In fact, it is the goal of the simulation to capture such bugs in the design. Unlike simulation algorithm related status update, no extra synchronization will be performed by the simulator. To make the distinction clear, look at the following example. Suppose a process calls an interface function defined in a medium connected to it, the process needs to lock the medium mutex first, update the status (e.g. this interface function is being called by this process) and release the mutex. Then, the body of the interface function will be executed. At the same time, if another process connected to the same medium calls the same function, it needs to lock the mutex, update usage information and release the mutex. Then, the function body can also be executed by the second process unless user specifies mutual exclusion explicitly in their code.

4 Performance Analysis

Regardless of single-thread or multi-thread simulation, simulation speed depends very much on the systems that are being simulated. In the extreme cases, designers can have all processes finish immediately or have all processes run forever, then simulation speed are simply zero and infinity. In order to formally analyze the simulation performance, let's make the following assumptions/simplifications.

- there are M processes in the system
- each process runs the same amount of work W
- each process finishes its work W in N process phases
- simulation manager takes S for both single-thread and multi-thread simulations. $S = \alpha MN$, α is a constant
- context switching takes C for both single-thread and multi-thread simulations
- multi-thread simulation is run on an SMP with X processors

For single thread simulation, total simulation time T_s is

$$T_s = MW + S + (M + 1)NC \tag{1}$$

where $(M + 1)N$ is the number of context switching.

For multi-thread simulation on X -SMP, total simulation time T_m are

$$\begin{cases} T_{m1} = W + S, & \text{if } X > M \\ T_{m2} = S + \lceil \frac{M+1}{X} \rceil (W + NC), & \text{if } X \leq M \end{cases} \quad (2)$$

where $\lceil \frac{M+1}{X} \rceil$ is the maximum number of processes running on one processor (assuming all processes are evenly distributed into all processors by OS).

Therefore, speedup for T_{m1} is

$$T_{p1} = \frac{T_s}{T_{m1}} \quad [X > M] \quad (3)$$

$$= \frac{MW + S + (M + 1)NC}{W + S} \quad (4)$$

$$= \frac{1 + \frac{S}{MW} + \frac{NC}{W}}{\frac{1}{M} + \frac{S}{MW}} \quad \beta = \frac{S}{MW} \quad (5)$$

$$= \frac{1 + \beta + \frac{NC}{W}}{\frac{1}{M} + \beta} \quad (6)$$

Note that β represents the ratio of the time spent on simulation manager to the time spent on all other processes. It determines the speed up like in Amdahl's Law with simulation manager being the sequential part. In the extreme cases, when simulation manager takes no time, i.e. $\beta = 0$, then the speed up is

$$T_{p1} = M + \frac{MNC}{W} \quad (7)$$

It is interesting to notice that as long as $X > M$, the speedup is irrelevant to X . This is because there are always enough processors to hold all processes. The speedup depends on the intrinsic parallelism in the system, not the computing power. Because of the saving of context switching, we got superlinear speedup w.r.t. the number of processes.

On the other hand, if simulation manager dominates the total simulation time, i.e. β is very large. Then,

$$T_{p1} \approx 1 \quad (8)$$

which means no speed up.

Speedup for T_{m2} is

$$T_{p2} = \frac{T_s}{T_{m2}} \quad [X \leq M] \quad (9)$$

$$= \frac{1 + \beta + \frac{NC}{W}}{\frac{1}{M} \lceil \frac{M+1}{X} \rceil + \beta} \quad \beta = \frac{S}{MW} \quad (10)$$

$$\approx \frac{1 + \beta + \frac{NC}{W}}{\frac{1}{X} + \beta} \quad (11)$$

The result is similar to the previous case. When β is zero, the speed up is

$$T_{p2} = X + \frac{XNC}{W} \quad (12)$$

It is superlinear in the number of processors because of the saving on context switching. When β is large enough, speedup is almost 1 or no speedup.

Note that equation 6 and equation 11 have the same form.

$$T = \frac{1 + \beta + \frac{NC}{W}}{\frac{1}{Y} + \beta} \quad (13)$$

where for T_{p1} , $Y = M$, while for T_{p2} , $Y = X$. Figure 4(a) gives some sense of how speedup changes with respect to Y and β . We can see from it that for small β , speedup is changing from linear to sub-linear. When β gets bigger, speedup almost remains the same. Since there is a ceiling operation in equation 10, the exact speedup when $M = 20$ is shown in figure 4(b). The ceiling just slightly affects the speedup.

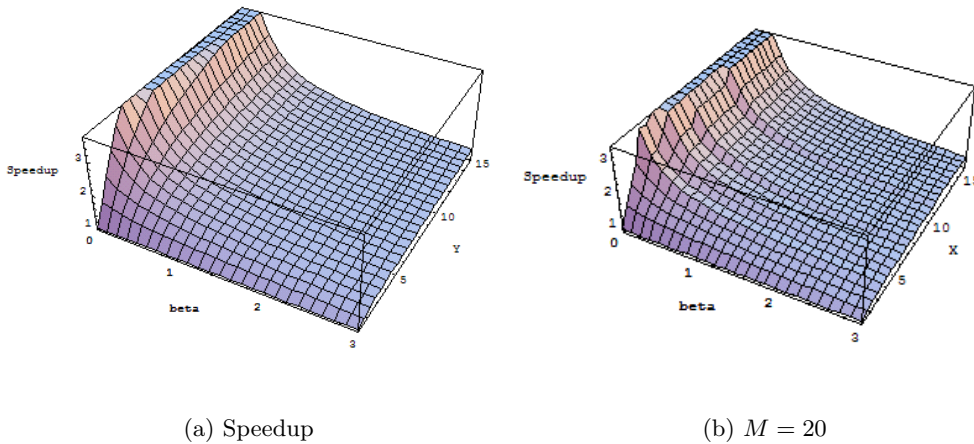


Fig. 4. Speedup w.r.t. $Y = M(X)$ and β

5 Experiment

In this section, I show the timing result of the simulation. The MMM example used here is a simple synthetic one. In this example,

- there are M identical processes

- each process has a fix amount of work to do
- there are no interrelationship among processes

I then ran the simulation on a Seaborg node, which has 16 Power3 processors. One difficulty I met is that there is no way to specify how many processors the simulation should use, or which process runs on which processor.² Considering this limitation, I can not make simulation run on a single processor in order to find out speedup numbers. As an alternative, I compute the total CPU time from all the processors and then divide it by the wall clock (elapsed) time to get speedup numbers. The results are shown in figure 5.

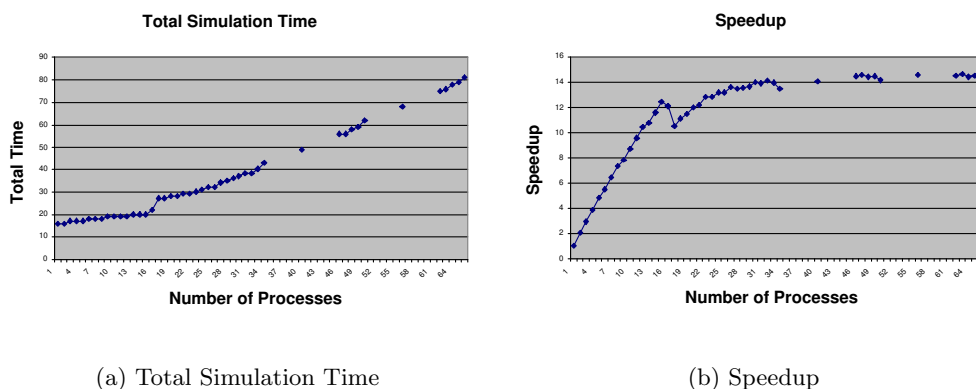


Fig. 5. Multi-thread Simulation

From figure 5(a), we can see that total simulation time have steps. The length of the step is the number of processors in the node, which is 16. When the number of processes becomes larger, steps become less obvious. This might be caused by the competition of processes in the same processor. The speedup trend in figure 5(b) is consistent with the performance analysis described in section 4. When the number of processors (16) is greater than the number of processes ($X > M$), speedup is linear w.r.t. the number of processes as in equation 6; when the number of processors is less than the number of processes ($X \leq M$), speedup is almost a constant, which is equal to the number of processors as in equation 12. Due to the simulation manager and Amdahl's law, speedup is less than ideal $X = 16$, but at around 14.

² I did try what Jason suggested, `pthread_attr_setscope`. However, it turned out that this function affected only the scheduling policy. i.e. `PTHREAD_SCOPE_SYSTEM` lets pthreads compete with other users' or OS' processes/pthreads; `PTHREAD_SCOPE_PROCESS` lets pthreads compete with other pthreads created from the same process. In both cases, OS will use all available processors in the node.

6 Conclusions

In this paper, I present the parallelization of Metropolis simulator. Due to the intrinsic parallel nature of MMM designs, I create one thread for each of the processes. The communication media are allocated in the shared memory so that all processes connected to it can have access to the media. Considering above implementation, SMPs are the best target machines. I chose Seaborg machines as the benchmark platforms. The threads were implemented by pthread library for its great portability, scalability and good performance on Seaborg machines.

Formal performance analysis is given in section 4. The analysis results are consistent with the simulation result shown in section 5. When the number of processors is larger than the number of processes, speedup is linear in the number of processes; when the number of processors is less than the number of processes, speedup saturates at the number of processors in the system.

References

1. Inc. Avery Design Systems. http://avery-design.com/web/avery_simclusterds012003.pdf.
2. Felice Balarin, Luciano Lavagno, Claudio Passerone, Alberto Sangiovanni-Vincentelli, Marco Sgroi, and Yosinori Watanabe. Modeling and designing heterogeneous systems. Technical Report 2002/01, Cadence Berkeley Laboratories, January 2002.
3. Felice Balarin, Yosinori Watanabe, and et al. Metropolis: An integrated environment for electronic system design. *IEEE Computer Society*, April 2003.
4. D. Densmore et al. Microarchitecture development via metropolis successive platform refinement. In *DATE*, 2004.
5. Richard M. Fujimoto. Parallel discrete event simulation. *Commun. ACM*, 33(10):30–53, 1990.
6. D.D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, and S. Zhao. *SpecC: specification language and methodology*. Kluwer Academic Publishers, 2000.
7. T. Grotker, S. Liao, G. Martin, and S. Swan. *System design with SystemC*. Kluwer Academic Publishers, 2002.
8. IEEE. Ieee std 1003.1c-1995.
9. K. Keutzer, A.R. Newton, J.M. Rabaey, and A. Sangiovanni-Vincentelli. System-level design: orthogonalization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(12):1523–1543, December 2000.
10. University of Cincinnati. <http://www.ececs.uc.edu/paw/warped>.
11. Alberto Sangiovanni-Vincentelli. Defining platform-based design. *EEDesign*, February 2002.