Distributed Circuit Simulation using Combined Simulation Method

Chun-Jung Chen*, Tai-Ning Yang, Chih-Jen Lee, Jen-Dong Sun, and Sheng-Hsuan Hsu

Department of Computer Science of Chinese Culture University, #55 Hwagng Road, Taipei, Taiwan.
*Corresponding author’s e-mail: teacherchen62@yahoo.com.tw

Abstract. In this paper, we describe methods to utilize the multi-core CPU as well as computers connected by internet to perform large-scale MOSFET circuit simulations. That is, both distributed and parallel computing are utilized to do the simulation. Parallel Newton Relaxation and Combining Simulation Method (CSM) are our numerical tools to accomplish parallel and distributed computing respectively. We note that both coarsed and fine grained parallel computing strategies are used in this paper. All proposed methods have been implemented and tested. Experimental results justify pleasing effects of all proposed methods.

1. Introduction

We know that circuit simulation [1, 2, 3] is very important in IC design community and it is also very time-consuming. These years, the multi-core CPU becomes very popular, and all computers have been connected onto internet. Hence, in this paper, we propose parallel and distributed-based computing methods for large-scale MOSFET circuit simulation, in which the multi-core CPU and computers connected by internet are used as the corresponding hardware respectively.

In the distributed computing respect of this research, we will ask distributed processors (computers on internet) to simulate distinct portions of the simulated circuit and then combine their resulted waveforms. The simulated circuit is divided into sub-tasks and be solved by distributed computers. The final waveforms are obtained by combining waveforms of these sub-tasks. This strategy is called Combining Simulation Method (CSM) [3]. Note that the Simulation-on-Demand (SOD), i.e. only to simulate subcircuits contributing to wanted outputs, is used in CSM to partition simulation tasks.

In the parallel computing respect of this research, we use multi-core CPU to compute subcircuits in parallel in one computer. In each distributed computer, a sub-task (a portion of the whole simulated circuit) is solved by Direct approach [1] (which is used by SPICE ). But the Direct approach is modified in the way solving inner induced nonlinear equations: we use parallel Nonlinear Relaxation, which is used by Iterated Timing Analysis (ITA) [1], to solve the nonlinear equations. We call the code for this nonlinear equation solver ParallelNL sub-program in this paper. In ParallelNL, each nonlinear equation (or a group of nonlinear equations) is (are) dispatched to a thread of the multi-core CPU to be solved in parallel.

Hence, in this paper, CSM and Nonlinear Relaxation are the numerical tools for distributed and parallel computing respectively. All mentioned methods have been implemented in MOSTIME [3][4]. A major computer (called master computer) handles CSM algorithm and sends sub-tasks to distributed
computers (called slave computers). Slave computers then use Direct approach and ParallelNL to solve the given sub-tasks. By the way, the master and slaver versions of MOSTIME are executed by different kind of computers respectively. The running results, including waveforms and used CPU times, have been recorded and compared with those of ITA (in MOSTIME). We find that calculated waveforms of new method are quite accurate and the obtained time speedups are pleasing.

The outline of this paper is as follows. In the following section, the used numerical methods are described. Then a section follows to describe the real implementation-related matters. Section 4 shows experimental results, and finally a conclusion section follows.

2. Major Numerical Methods

2.1. ParallelNL

In this section, we describe used numerical methods of this paper. An entire circuit simulation algorithm will be illustrated here. A solved circuit could be described as:

\[ F(Y(t), Y'(t), U(t)) = 0 \]  \hspace{1cm} (1)

In which \( Y \) is the circuit variable, \( U \) is input signal sources, \( t \) is the time, and “.” means time differentiations. (1) is handled by integral formula at a time point (says \( t_{n+1} \), and \( t_n \) is the previous time point) to derive a system of nonlinear equation.

\[ G(Y(t_{n+1}), Y(t_n), U(t_{n+1})) = G(Y_{n+1}, Y_n, U_{n+1}) = 0 \]  \hspace{1cm} (2)

The major difference between our method and Direct approach is the way solving (2), the system of nonlinear equations. The following pseudo program shows the Nonlinear Relaxation algorithm used by us.

Algorithm 1 (Nonlinear Relaxation):

\[ NonLinRelax() \{
  V^* = \text{Initial value for } V;
  \text{do} \{ \\
    V = V^*;
    L0: \text{for}(i = 1; i <= N; i++) \{ \\
      L1: \text{Solve } G_i \text{ for } V^*_i \text{ by using Newton Iteration; } \\
    \}
    L2: \text{while}(|V - V^*| >= \varepsilon);
  \}
\]

In Algorithm 1, \( V \) represents \( Y_{n+1} \) of (2), \( V^* \) is the value of the newest relaxation iteration, \( N \) is the number of equations, and \( \varepsilon \) is a small error tolerance value. \( G_i \) is one equation of (2). In Label L2, the iteration is controlled by checking the value difference of solutions between adjacent iterations. There are Gauss-Seidel (GS) version and Gauss-Jacobi (GJ) versions for this algorithm [3]. Obviously, the GJ version is suitable for parallel computing. Algorithm 1 uses GJ version (which means the newest \( V_i \) is not “propagated” to other equations). Therefore, in this algorithm one equation could be solved by one thread of multi-core CPU independently. We know that GJ version of Nonlinear Relaxation converges slower than GS version of Nonlinear Relaxation. But GJ version is irreplaceable in parallel computing.

We note that Algorithm 1 doesn't utilize the partition/grouping strategy [1][2] that groups coupling equations into one sub-circuit to be solved together. Partition strategy is very important for Relaxation-based algorithms. In a MOSFET circuits, nodes usually couple to others, e.g. two nodes of a CMOS NAND gate couple to each other bi-directionally. If we use Algorithm 1 to solve for the transient solution of a NAND gate, the convergence speed of relaxation process would be slow. If the partition strategy is used, the two nodes would be assigned to a sub-circuit to be solved together to cancel the
problem of slow convergence. Assume that all $N$ equations have been divided into $M$ clusters (subcircuits), and the nonlinear equations of the $i^{th}$ cluster are collected in $CG_i$. Circuit variables inside $CG_i$ are $CV_i$. The nonlinear equation of a cluster is shown as:

$$CG_i(CV, CV_o, U) = 0$$

Algorithm 2 (Clustering Nonlinear Relaxation):

ClusteringNonLinRelax () {
    $CV^* = \text{Initial value for } CV$;
    do {
        $CV = CV^*$;
        for ($i = 1; i <= M; i++$) {
            L1: Solve $CG_i$ for $CV^*$ by using Newton Iteration;
        }
    } while ($|CV - CV^*| \geq \varepsilon$);
}

We list the algorithm with partitioning strategy in Algorithm 2. As we mentioned above, the GJ-version is used in order to adopt parallel computing. This algorithm looks like Algorithm 1 except that a group of equations (cluster) is solved together rather than one equation. In Label L1, $CG_i$ has been solved for $CV_i$ by using Newton Raphson iteration.

One famous algorithm for Fast SPICE is ITA [1]. ITA is a more advanced version of Algorithm 2. Due to the close relationship of Algorithm 2 and ITA (so ITA is also used to compare the performance of our methods in next section) we describe ITA here. The simulated circuit is partitioned into $M$ subcircuits. The $i^{th}$ one is as follows:

$$F_i(Y_i(t), \dot{Y}_i(t), E_i(t), \dot{E}_i(t), t) = 0$$

(4)

Here, $E_i$ contains circuit variables outside the $i^{th}$ subcircuit, which is called Decoupling Vector or input variables ($U$ is removed since it is a global variable). This equation could be simplified as:

$$f(y(t), \dot{y}(t), w(t), w(t), t) = 0$$

(5)

Where $y$ is the variable of a subcircuit, $w$ is the input variable. This equation could be handled by integral formulas to obtain the subcircuit’s nonlinear equation:

$$g(y(t_{n+1}), y(t_n), w(t_{n+1}), w(t_n)) =$$

$$g(y_{n+1}, y_n, w_{n+1}, w_n) = 0$$

(6)

We can find that (6) is one partitioned sub-system of (2), and it is solved by Newton Raphson Iteration. The ITA’s pseudo code is described in Algorithm 3.

Algorithm 3 (ITA Circuit Simulation Algorithm):

/* Simulation duration is $T_{begin}$ ~ $T_{end}$ */

ITA ($T_{begin}$, $T_{end}$) {
    Put subcircuits connected to primary input into $E(T_{begin})$;
    // $E$ is an priority queue, its elements are basic queues
    L0: while ($E$ is not empty) {
        $t_{n+1}$ = the smallest event-time in $E$;
L1: for ($k = 1; E(t_{n+1})$ is not empty; $k++$) {
  // $k$ is the relaxation index
  Clear TMP; // TMP is a basic queue
L2:  for (each subcircuit $a$ in $E(t_{n+1})$) {
    // solve (5) for solution of a new time point
    Solve (5) of $a$ at $t_{n+1}$ for transient responses;
    if (can’t solve $a$ or solved result is unacceptable) {
      Shrink the time step of $a$;
    }
    if ($a$ has been converged) { // converged
      Estimate next solving time $t_{next}$ and add $a$ into $E(t_{next})$;
    }
    else { // not converged
      Add $a$ into TMP;
      // here is the Selective-tracing Scheme
L3:    for (each fan-out subcircuit $w$ of $a$) {
        Add $w$ into $E(t_{n+1})$;
      }
    }
  }
}
$E(t_{n+1}) = TMP$;
}

The loop iterating for all time points is shown at Label L0. The Nonlinear Relaxation iteration is shown at Label L1. ITA uses queues to store “active” subcircuits of a time point, and only calculates subcircuits in the queue. This design is shown at Label L2. ITA utilizes this Selective-tracing scheme [2] to dynamically traces and selects active sub-circuits, shown at Label L3. ITA only calculates subcircuits that are active in current time point ($t_{n+1}$). We note such “simulation-on-active” (SOA) characteristic might be utilized by our parallel nonlinear equation solver in the future.

2.2. CSM
The basic idea of CSM is to use slave processors to simulate different portions of the simulated circuit, and then combine the obtained sub-waveforms. There is a calculation procedure in the master computer that analyzes the simulated circuit, divides the circuit, sends the divided portions to slave computers, waits for the termination of slave computers, and then combines obtained sub-waveforms. The slave computers just simulate portions of the analyzed circuits and generate portions of the wanted waveforms. For the success of CSM, a good “dividing” method is critical. We call all subcircuits affect (or contribute) to a wanted node $x$ the burden subcircuits of $x$, burden($x$). We find that SOD (Simulation on Demand) is a good way to divide the circuit for CSM. For, burden subcircuits of wanted nodes are usually different. SOD could simulate burden subcircuits only. The master computer divides wanted outputs into sub-sets and send them to various slave computers. Slave computers then use SOD to calculate the waveforms of a portion of the simulate circuit. We could call this method the SOD-based CSM.
3. The Distributed Computing and Task Partitioning

3.1. Distributed Computing
In [3], the CSM is constructed by using the multi-core CPU. To construct the distributed computing-based CSM, we need to implement a function to pass files between computers connected by internet at first. We use TCP/IP functions such as recv(), make_contact(), and so on to build such connection functions. The slave computers wait for decks sent by master computer all the time, and execute circuit simulation on the received deck immediately. Using these “self-made” communicating functions, the master computer sends decks to slave computers and receives result waveforms from slave computers.

3.2. Task Partitioning
In [3], there are methods raised for wanted output nodes partitioning in SOD-based CSM. In [3] they find burden subcircuits those are independent to others as possible, and then put them into different processors to be simulated in parallel. The partition problem of CSM becomes go partitioning the output nodes. To automatically partition output nodes, the “relationships” among output nodes need to be analyzed. They find that the relationship is the “overlap situation” of burden subcircuits of output nodes. They then define Overlap Degree to indicate such overlap situations. Overlap Degree from output node x to output node y (OD(x, y)) is defined as: #overlap burden subcircuits / #burden subcircuits of x. The bigger the OD(x, y) is, the more likely that x could be grouped with y and then be simulated by the same slave computer. Hence, the partitioning steps for wanted nodes are process of checking the Overlap Degree of all pair of wanted nodes. If the Overlap Degree of a pair of nodes is big, the two wanted nodes were grouped together. Finally, sets of wanted nodes are obtained, which are sent to slave computers.

However, we find that the calculation power of slave computers affect the load balancing of CSM. To achieve better load balancing, both partitioning for wanted output nodes and slave computation power consideration need to be considered at the same time. We leave this to the further research. Therefore, in this paper we use simple partitioning for wanted nodes rather than that used by [3].

4. Experimental Results
We have implemented all proposed methods in our simulator MOSTIME [3][4]. Our practical environment includes: Visual Studio 2008, OpenMP API, desktop and note book computers equipped with Intel i7 CPU, and so on. There exist a master computer and two slave computers in our environment. The two slave computers use Intel Xeon E3-1245 v6 and Intel i7 8550u respectively. We note that slave computers compute the simulations mainly, but the recorded CPU time is in the unit of the master computer. We use a text file (called IPfile.txt) to store the IP addresses of slave computers. The two slave computers have their own IP and use different port numbers.

Several circuits have been simulated, whose specifications and brief descriptions are listed in Table 1. Wanted nodes of these circuits are just simply partitioned by their appearance sequence. Since we have only two slave computers, we could arrange the sequence of wanted nodes such that they could be partitioned well, i.e. making each slave computer simulates unique and independent portions of the simulated circuit.

We check the quality of simulation accuracy at first. Schematic and waveforms of ALU (Arithmetic Logic Unit) are plotted in Fig. 1. We can find that waveforms have matched each other very well. Next, we check the computation speeds recorded in Table 2. There are several algorithms, which are ITA (Algorithm 3), NL (Algorithm 1), NL+SOD, Distributed (CSM with SOD), and Distributed+Parallel (slave computer uses ParallelNL, Algorithm 2). There are several independent portions in these circuits, e.g. “INV100x2” has two independent 100-staged inverter chains. As we mention above, these independent portions are just assigned to different slave computers. Used CPU times and their ratios (compared with the CPU time of NL+SOD) are listed in Table 2. The ratio values could be used to see the performance of proposed algorithms. We could find that Distributed version improves NL+SOD version.
Moreover, Distributed+Parallel version further improves Distributed version in all circuits except two inverter chain circuits. Inverter chains have data dependency on all subcircuits, so parallel computing on several subcircuits at the same time doesn’t earn benefits. Finally, we could say that new techniques fasten the simulation speed clearly. The distributed and parallel computing methods proposed in this paper are successful.

5. Conclusion

In this paper, we have presented techniques to utilize the popular and powerful multi-core CPU and internet for large-scale MOSFET circuit simulation. Our fundamental techniques are CSM and ParallelNL. The real implementation on multi-core CPU computers connected by internet has been tested, and several experimental results justify that proposed techniques provide good parallel-computing efficiencies. The Simulation-on-Active property of ITA algorithm and more detailed partition method for CSM might be topics in our future research.

![Figure 1. (a) Schematic of 1-bit ALU. (b) Waveform comparison for circuit ALU4x2, which has two 4-bit ALUs.](image)

### Table 1. Specifications of tested circuits

| Ckt.     | Description                      | Wanted output nodes |
|----------|----------------------------------|---------------------|
| INV100x2 | Two 100-staged inverter chain    | 2                   |
| INV100x4 | Four 100 staged inverter chain   | 4                   |
| ALU2x4   | Four 2-bit ALU                   | 8                   |
| ALU4x2   | Two 4-bit ALU                    | 8                   |
| ALU8x2   | Two 8-bit ALU                    | 8                   |
| PMUL4    | A 4-bit parallel multiplier      | 8                   |

### Table 2. Used CPU time\(^{\text{c}}\) and ratios\(^{\text{d}}\) of various algorithms

| Ckt.     | ITA      | NL       | NL+SOD   | Distributed\(^{e}\)   | Distributed+Parallel\(^{d}\) |
|----------|----------|----------|----------|-----------------------|-----------------------------|
| INV100x2 | 31       | 5.78/100%| 5.87/101%| 5.71/98%              | 6.5/112%                    |
| INV100x4 | 33.8     | 11.5/100%| 11.5/100%| 7.06/61%              | 7.44/64%                    |
| ALU2x4   | 28.2     | 48.4/100%| 43.1/89% | 14.4/29%              | 11.4/23%                    |
| ALU4x2   | 37.4     | 22.3/100%| 20.9/93% | 17.2/77%              | 14.6/65%                    |
| ALU8x2   | 157      | 135/100% | 57.6/42% | 24.5/18%              | 23.5/17%                    |
| PMUL4  | 97.2 | 64.3/100% | 63.6/98% | 43.9/68% | 22.3/34% |
|--------|------|-----------|----------|----------|----------|

%: CPU time unit is in Intel i7 8550 seconds

#: All percent values are ratios of CPU time compared with those of NL version

*: Which is CSM

$: Which is CSM + ParallelNL

References

[1] A. R. Newton and A. L. Sangiovanni-Vincentelli, “Relaxation-based electrical simulation,” IEEE Trans on CAD, Vol. CAD-3, pp. 308-311, Oct. 1984.

[2] R. A. Saleh and A. R. Newton, “The exploitation of latency and multirate behavior using nonlinear relaxation for circuit simulation,” IEEE Trans., Computer-aided Design, vol. 8, pp. 1286-1298, December 1989.

[3] Chun-Jung Chen, and Tien-Hao Shih, “Parallel Circuit and Interconnect Simulation Using Multi-core PC,” 2012 IEEE High Performance Extreme Computing Conference, Sep. 11-12, 2012, Waltham, MA, USA.

[4] Chun-Jung Chen, Tai-Ning Yang, Chih-Jen Lee, and Jen-Dong Sun, "Parallel Iterated Timing Analysis Algorithm for MOSFET Circuit Simulation," 5th International Conference on Mechanical, Control, and Electronic Information (ICMCEI2017), Taian, China, 2017, Jul. 8-9.