Parallel Large-scale MOSFET Circuit Simulation using Multi-core CPU and Time-saving Techniques

Chun-Jung Chen*, Tai-Ning Yang, Chih-Jen Lee, Li-Ping Chou, Sheng-Hsuan Hsu, Allen Y. Chang, and Chung-Hsin Liu.

Department of Computer Science, Chinese Culture University, Taipei, Taiwan

*Corresponding author’s e-mail: teacherchen62@yahoo.com.tw

Abstract. In this paper, we investigate the acceleration properties of fine-grain parallel large-scale MOSFET circuit simulation using multi-core CPU. Several CPU have been tested for understanding maximum possible accelerations. A parallel circuit simulator, MOSTIME, is used to simulate experimental circuits. Performing tests on CPU and MOSTIME, we find that more advanced accelerations on large-scale fine-grain parallel circuit simulation would rely on time-saving software techniques and GPU (Graphic Processing Unit) cards. A newly implemented time-saving technique for MOSTIME has been tested to show promising effects.

1. Introduction

In this paper, we discuss large-scale fine-grain parallel circuit simulation [1][2] using multi-core CPU [3]. The major purpose is to discuss the actual possible speedup of parallel large-scale circuit simulators using available multi-core CPU and to fund hopeful strategies for further acceleration of the simulations. We use MOSTIME [3][4], a parallel large-scale MOSFET circuit simulator, as the experimental simulator. There exist several circuit simulation algorithms in MOSTIME, including ITA (Iterated Timing Analysis) [2] and Nonlinear Relaxation (NR) [4]. The classical Direct approach that is used by SPICE has been coded in MOSTIME, too. Moreover, a specific Direct approach modified in the way that inner induced nonlinear equations are solved by Nonlinear Relaxation (NR) rather than Newton-Raphson iteration is available, too. This version of Direct approach is called NR-Direct approach in this paper [4]. We note that NR is also used by ITA, in which ITA cleverly arranges such that only active subcircuits (group of circuit nodes) would be calculated. Note that MOSTIME does perform parallel treatments on NR. In the “parallel NR,” each nonlinear equation (or a group of nonlinear equations) is (are) dispatched to a core of the multi-core CPU to be solved in parallel manner. Detailed numerical algorithms will be illustrated in the following Section 2.

To understand the possible maximum speedup of circuit simulation by using multi-core CPU, we do some experiments by running small time-consuming programs. We record various used CPU time into a table and analysis it. We find that, due to some hardware limitations, e.g. memory access bottle neck, floating point processing bottle neck, and so on, the speedups are not proportional to the number of adopted cores. Moreover, seems that the speedup has been bounded to a maximum value (about 4). In respect to this phenomenon, we would make reasonable suggestions for further acceleration enhancements.

To understand the parallel computing performance of MOSTIME, we have recorded the real exhausted CPU time for different stages of codes. These stages include Nonlinear equation solving stage, subcircuit simulation stage, integral equation discretization stage, input deck processing stage and so on.
With these recorded time, we can understand time ratios of different stages. Through this, we could get the parallel processing portion ratio (the only parallel processed portion of MOSTIME is the code for Nonlinear equation solving) and then use Amdahl’s law to know how good MOSTIME’s speedup could be achieved. This understanding help us designing hopeful strategies for further acceleration.

We have raise strategies for further acceleration of parallel circuit simulations. One strategy is to reduce the time complexity of parallel numerical algorithm. Following it, we devise a new software technique, the Parallel SOD (Simulation-on-demand) on NR. This technique has been added to NR-Direct approach in MOSTIME and tested (to show its good effect).

The structure of this paper is as follows. Section 2 describes the used numerical algorithms of this paper, which is followed by a section describing the “intrinsic” parallel power tests on several multi-core CPU and illustrating the accelerating property of multi-core CPU. Then, Section 4 follows, which discusses the accelerating situation of various stages of MOSTIME and suggests strategies for further speedup. A new time-saving software technique (to reduce time complexity) has been raised and implemented. In Section 5, experimental results of implementation codes are compared and discussed. Finally, a conclusion section follows.

2. Numerical Methods in MOSTIME

In this section, we will describe related numerical methods used in this paper. A circuit performed by circuit simulation could be described as a system of time-varying differential equation $F$:

$$F(Y(t), \dot{Y}(t), U, t) = 0$$

In which $Y$ is the circuit variable, $U$ is primary inputs, $t$ is the time, and “.’.” means time differentiations.

We discretize (1) at a time point (say $t_{n+1}$), and $t_n$ is the previous time point), deriving a system of Nonlinear equation $G$.

$$G(Y(t_{n+1}), Y(t_n), U(t_{n+1})) = G(Y_{n+1}, Y_n, U_{n+1}) = 0$$

In Direct approach, this equation is solved by Newton Raphson Iteration. However, Nonlinear Relaxation (NR), the major role of this paper, can do the same thing. Algorithm 1 represents it.

Algorithm 1 (Nonlinear Relaxation):

```plaintext
NonLinRelax () {
    $V^* = \text{Initial value for } V$;
    do {
        $V = V^*$;
        L0: for(i = 1; i <= N; i++) { // N is circuit node number
            Solve $G_i$ for $V^*$, by Newton Raphson Iteration;
        }
        L2: while($|V - V^*| > \varepsilon$);
    }
}
```

Algorithm 2 (Clustering Nonlinear Relaxation):

```plaintext
ClusteringNonLinRelax () {
    $V^* = \text{Initial value for } V$;
    do {
        $V = V^*$;
        L0: for(i = 1; i <= M; i++) { // M is subcircuit number
            Solve $S_i$, for $V_i^*$, using Newton Iteration;
        }
        L2: Collect all $V_i^*$ and store them into $V^*$;
        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). At label L2, the iteration continues if the value difference of solutions between adjacent iterations is not small enough. There are Gauss-Seidel (GS) version and Gauss-Jacobi (GJ) versions for loop at label L0 [2] (the relaxation process). GJ version is suitable for parallel computing but is less efficient. To use GJ means the newest $V_i^*$ won’t be referred by other nodes at the same iteration.

Algorithm 1 doesn’t utilize the partition/grouping strategy [1][2] (which groups coupling equations into one subcircuit to be solved together). However, partition strategy is important for relaxation-based algorithms. In MOSFET circuits, nodes couple to others frequently. If coupled nodes were solved separately, like Algorithm 1, the convergence process would be very slow. Therefore, partitioning is necessary. Assume that all $N$ equations are partitioned into $M$ clusters (subcircuits), and the $i^{th}$ cluster is called $S_i$. Circuit variables inside $S_i$ are $v_i$. All The nonlinear equations of a cluster are shown as:
Table 1. Used CPU time of various stages of MOSTIME in simulating ALU8.

| Stages of Codes | Used CPU Time and Ratio |
|-----------------|-------------------------|
|                 | NR-Direct               | Parallel NR-Direct |
| 1: Codes other than those of NR-Direct | 1.006 | 5.4% | 0.899 | 13.8% |
| 2: Inside NR-Direct, codes other than those for NR | 0.259 | 1.4% | 0.372 | 5.7% |
| 3: Inside NR-Direct, codes other than those for NR (especially for step integral, model update, and so on) | 0.395 | 2.1% | 0.478 | 7.3% |
| 4: Inside NR, codes other than those for label L0 | 0.593 | 3.2% | 0.846 | 13.0% |
| 5: Codes of label L0 of NR | 16.22 | 87.8% | 3.93 | 60.2% |

*:CPU time is in Intel i7 855U second. #: Algorithm 1 or Algorithm 2

\[ S_r(V_r, V_s, U) = S_r(v_{r_{i-1}}, v_r^*, v_{i-1:M}, V_s, U) = 0 \]  

In which \( V_o \) represents \( V(Y_n) \). Algorithm2 uses partitioning. In label L1, \( S \) is solved by standard Newton Raphson Iteration, and all sub-vector of \( V^* \), \( v_i^* \), are collected and stored into \( V^* \) at L2. Note that label L0 is parallel computed in MOSTIME.

MOSTIME contains NR-Direct Approach, the same as standard Direct Approach except that Nonlinear equations of each time points, (2), are solved by Algorithm 2 rather than Newton Raphson Iteration. Note that one famous algorithm for Fast SPICE is ITA [1]. ITA also uses NR to solve Nonlinear equations. ITA uses the Selective-tracing scheme [2] to cleverly select subcircuits in (2) for calculation. That is, ITA could be called “Selective-tracing-Direct Approach” in respect to NR-Direct Approach.

3. Acceleration Properties of Multi-core CPU

There exist many multi-core CPU. We want to know their “intrinsic” potentials for accelerating numerical algorithms like those of parallel circuit simulations. In our parallel circuit simulation program, we want to know used CPU time of various stages. To achieve this, we practically record the used CPU time of different portions of MOSTIME. The used example circuit is ALU8 (8-staged ALU). The used CPU time of stages are shown in Table 1. There are two algorithms used, in which Parallel NR-Direct uses 8 threads of Intel i7 855U CPU to compute. Note that only Stage 5 (for solving Nonlinear equations) is accelerated by parallel computing, and it is clearly the major portion of the entire used CPU time.

We want to design small intrinsic testing programs to test the acceleration properties of CPUs. Considering the content of Nonlinear equation solving process, we design versions that have floating point calculations as well as many memory (on local portions, i.e. arrays) access. The two programs are listed in Algorithm 3 and Algorithm 4, respectively. We call them Intrinsic Testing #1 and Testing#2. The former one has few memory access and is almost all composed of floating point operations. It is “mathematic-intensive” type. The later one, Testing#2, is composed of some floating point operations as well as many memory access (on local arrays) and is “memory-intensive” type. We think Testing#2 might be more similar to codes for Nonlinear equation solving process.

We use these two programs to check speedup situations of various multi-core CPU. Both sequential version and parallel version are tested to get speedups, whose results are shown in Table 2.

Algorithm 3 (Intrinsic Testing#1):

```c
#define PTS 20000
#define REPEAT_N 800
#define MXP_M 120
//next line, parallel=1 would enable parallel computing
#pragma omp parallel for if(parallel)
for(p = 0; p < PN; p++) {
  acc2[p] = 0.0;
  for(x2[p] = 0; x2[p] <= 2000*PI; x2[p] += 2*PI/PTS) {
    acc2[p] += (sin(x2[p]) + cos(x2[p]) + tan(x2[p]));
  }
  for(i = 0; i < PN; i++) acc += acc2[i];
}
```

Algorithm 4 (Intrinsic Testing#1):

```c
#pragma omp parallel for if(parallel)
for(p = 0; p < PN; p++) {
  ar[p][0] = p+1;
  for(k2[p] = 1; k2[p] <= REPEAT_N; k2[p]++)
    for(j2[p] = 1; j2[p] < PTS_M; j2[p]++)
      ar[p][j2[p]] = ABS((int)(pow((double)ar[p][j2[p]-1], 1.045) * 7.5)) % 1000;
}
```

```
for(p = 0; p < PN; p++)
  ar[p][0] = ar[p][PTS_M-1];
```
Table 2. Speedup\,* results of Intrinsic Testing programs, executed by several multi-core CPU

| Program     | CPU name & #cores                  | CPU threads | Used thread # |   |   | 16 | 32 |
|-------------|------------------------------------|-------------|---------------|---|---|----|----|
| Testing#1   | Intel i7 855U, 4 cores             | 8           | 0.97          | 1.20 | 1.62 | 2.28 |N/A |N/A |
|             | Intel i7 8565U, 4 cores            | 8           | 0.97          | 1.04 | 1.30 | 1.55 |N/A |N/A |
|             | Intel Xeon CPU E3-1245 v6, 4 cores | 8           | 0.96          | 1.07 | 1.61 | 1.97 |N/A |N/A |
|             | Intel Xeon Silver 4216, 16 cores   | 32          | 0.95          | 0.99 | 1.24 | 1.90 |2.04 |3.1 |
| Testing#2   | Intel i7 855U, 4 cores             | 8           | 0.99          | 1.68 | 2.42 | 3.67 |N/A |N/A |
|             | Intel i7 8565U, 4 cores            | 8           | 0.98          | 1.83 | 2.53 | 4.05 |N/A |N/A |
|             | Intel Xeon CPU E3-1245 v6, 4 cores | 12          | 0.99          | 1.74 | 2.73 | 3.97 |N/A |N/A |
|             | Intel Xeon Silver 4216, 16 cores   | 32          | 0.99          | 1.32 | 1.82 | 2.50 |3.17 |3.98 |

\* Speedup is the ratio of sequential used CPU time and parallel used CPU time.

We find that the overall speedup doesn’t show the growth rate proportional to the number of used threads. The real acceleration performance in a multi-core CPU seems to be limited by the hardware structures of CPU. For example, the instruction code feeding, memory access control, floating point process, and so on might be the bottleneck of multi-core CPU. On the other hand, we have also tested Testing#1 by using several GPU card and find linear relationship between speedup and the number of used threads. Although the computation power of one thread of GPU is less than 10% of that of CPU (by our tests performed on GPU cards: Tesla T4, GTX1050Ti, GT1080Ti, and CPU: Intel i7), the number of GPU threads could be several thousands. If computations are regular, GPU could derive good accelerations.

By the way, checking Stage 5 of Table 1, we find its speedup has reached 4.13 (16.22/3.93), which means the parallel performance of real codes in MOSTIME has reached a good level.

4. Suggested Strategies for Further Acceleration of Parallel Circuit Simulation

In this section, we consider the practical speedup situation of large-scale circuit simulation and raise useful suggestions. What we discuss is fine-grain parallel circuit simulation, and the scope of course-grain is another story.

We have used testing programs, Testing#1 and Testing#2, to derive Table 2. In addition, GPU has also be tested but not recorded in tables. This table shows that only maximum about 4 times of speedup could be provided. In MOSTIME, the accelerated code stage is that for solving Nonlinear equations. Referring to Table 1, we find that this stage, Stage 5, gets speedup 4.13, which is about the best speedup of using multi-core (refer to Table 2). This result justifies that MOSTIME's implementation for accelerating the Nonlinear equations is very successful.

We are curious about that how good the practical speedup could be? In considering this, the portions of calculations that could not be accelerated need to be considered. A well-known law called Amdahl's Law can solve this problem. It is shown as follows:

\[ S(N) = \frac{1}{(1-p) + (p/N)} \]  

Where \( p \) is the percent of codes that could be parallel computed, \( S \) is the speedup, and \( N \) is the “true speedup” for \( p \). The existence of \((1-p)\) has clear effects on the maximum speedup could derived. Checking Table 1, we find that MOSTIME’s \( p \) (in solving ALU8) is 87.8%, which means that the maximum possible speedup \((N = \infty)\) is 8.2 \((1/(1-0.878))\). In real case of running ALU8 in Table 1 (thread\# = 8, CPU time = 18.47 and 5.528 for the two algorithms) we gets \( S = 2.83 \). Note thread number is not equal to \( N \) of equation (4). To further increase this number, we need to increase \( N \) or reduce \( p \).

Now we discuss strategies that could further increase speedup of fine-grain parallel circuit simulation. At first, it's important to shrink the portions of code that can't be parallel computed, i.e. \( 1-p \) in (4). Even 10% of such kind of portion could lead to maximum 10 times speedup only. Secondary, to increase \( N \) in (4), which means to increase the real speedups on running Intrinsic Testing programs. Table 2 has told us that multi-core CPU could only provide about 4 for \( N \). To derive better \( N \), we could use GPU.
GPU can accelerate regular computations (just like those for Nonlinear equations) with the speedup proportional to the number of threads. Moreover, the hardware of GPU is more “scalable” (meaning easy to increase thread number), in which thousands of threads inside a device is not rare. Third, to reduce the amount of overall computations is important, too. We habitually consider that parallel computation can accelerate the computation brutally and therefore time-saving software technique is not important. However, through the discussions above, such time-saving techniques are still important.

To sum up, there are two strategies for further accelerating fine-grain parallel circuit simulations. First one is both to use GPU instead of multi-core CPU and to enlarge $p$, in Amdahl’s law, as much as possible. Second strategy is to adopt more time-saving techniques for parallel computations. For the second strategy, we have two ideas. One is to use SOD (Simulation-on-demand) to bypass useless subcircuit calculations, and another one is to utilize Selective-tracing scheme of ITA. Using SOD is an effective method, which has been implemented in MOSTIME. This method needs to analysis the signal flow graph (a directed graph whose vertices represent subcircuits and directed edges represent fan-in/fan-out relationships among subcircuits) of the partitioned circuits. It perform a backward traversal from the “wanted” primary output, marking contributed subcircuits. The marked subcircuits are then called “useful” and will be calculated in simulation. Another idea is to utilize ITA, which cleverly controls the number of subcircuit computed in the NR algorithm and reduces the computation amount.

5. Experimental Results
In this section, we demonstrate some experimental results derived by running MOSTIME. This circuit simulator executes on a workstation with Intel i7 855U CPU. This CPU has 4 cores and 8 threads. MOSTIME owns several circuit simulation algorithms. We want to test ITA, NR-Direct approach, NR-Direct+parallel approach, and NR-Direct+parallel+SOD approach. We note that the last algorithm includes the newly implemented time-saving technique. All algorithms use the same MOSFET model. Due to different developing periods, ITA algorithm is quite different from other algorithms, majorly in used data structures. These differences affect the running speed. Parallel codes are controlled by OpenMP. We record the used CPU time and compare obtained waveforms of these four algorithms.

The circuits to be tested include two types: ALU (Arithmetic Logic Unit) and SCT (Synchronous Counter). Both these circuits can be formed by cascading single stages. The schematic of one stage of these circuits are shown in Figure 1 (a) and (b) respectively. In Figure (a), $V_a$ and $V_b$ means input data, Logic pin selects the arithmetic/logic functions, Or_sub pin selects Or/And or Subtraction/Addition functions. In Figure (b), a master-slave JK Flip Flop is shown, whose $J$ and $K$ have been connected together to form the function of T Flip Flop.

ALU type circuits include ALU1, ALU4, and ALU8, which have one, four, and eight stages of Figure 1(a) respectively. We note that $V_a$ and $V_b$, input data pins, of all stages are connected together. Other control signals like Logic and Or_sub are connected together also. In order to demonstrate the capability of SOD, we only assign $S0$ (Sum of 1st bit) of ALU-type circuits and $Q0$ (counter's 1st bit) of the SCT-type circuit as the “wanted” outputs. The simulated waveforms of ITA, NR-Direct+Parallel, and NR-Directed+Parallel+SOD are plotted together for comparison purpose. Figure 1 (c) shows this comparison of ALU8. We can't distinguish the three different waveforms, since they match others so well.

Now we check the simulation speeds of these four algorithms. Table 3 records the used CPU time of these algorithms in order. The first and second algorithms are similar, in which they are different only in the way treating NR (for solving Nonlinear equations) algorithm. ITA uses Selective-tracing scheme to select subcircuits for calculation, and, on the other hand, NR-Direct just brutally calculate all subcircuits in the GJ fashion. Due to different implementations (different data structures, codes, and so on), NR-Direct shows better simulation speed than ITA. The third algorithm is NR-Direct+parallel. The used CPU provides 8 threads, and the used CPU time all have been reduced. For the later three bigger circuits, roughly two times speedup have been derived. Then, we go checking the last algorithm that uses NR-Direct, parallel computing, and SOD. The simulation times have been further reduced. Since the wanted output is at the first bit only, the time-saving phenomenon is quite clear. Especially in ALU8,
the overall speedup with respect to NR-Direct is 7. These results justify that SOD is quite effective for parallel computation also.

Figure 1. (a) Schematic of 1-bit ALU (b) Schematic of 1-bit SCT (c) Waveform comparison of ALU8, S0 is the output of 1st bit.

| Circuit | ITA | NR-Direct | +Parallel | +SOD |
|---------|-----|-----------|-----------|------|
| ALU1    | 1.40| 1.40      | 1.01      | 1.02 |
| ALU4    | 8.69| 5.12      | 2.81      | 1.56 |
| ALU8    | 17.47| 9.34      | 4.33      | 1.37 |
| SCT4    | 3.40| 2.66      | 1.47      | 0.82 |

*CPU time is in Intel i7 855U second.

6. Conclusion
This paper discusses the acceleration of large-scale MOSFET circuit simulation using fine-grain parallel computations. The multi-core CPU and GPU are considered and MOSTIME is used as the experimental tool to simulate some MOSFET circuits. We find that multi-core CPU has performance limitation and GPU might be the better choice for further acceleration. Amdahl's law shows that to reduce the portion of codes that can't be parallel computed is important. This paper also concludes that time-saving techniques are also quite beneficial for parallel computation, and an efficient time-saving technique has been raised, implemented and tested in MOSTIME for justification. For the fine-grain parallel circuit simulation, parallel ITA and using GPU would be good directions to further increase the speedup of parallel circuit simulations.

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, Jen-Dong Sun, and Sheng-Hsuan Hsu, “Distributed Circuit Simulation using Combined Simulation Method,” The 2019 International Conference on Wireless Communications and Smart Grid (ICWCSG2019), 2019/7/26-28, Guilin, China.