Multi-Threaded Circuit Simulation using OpenMP

Mark Zwolinski
School of Electronics and Computer Science
University of Southampton
Southampton SO17 1BJ, UK
Email:mz@ecs.soton.ac.uk

***Abstract*—Circuit-level simulation is a computationally-intensive problem that has proven to be particularly difficult to parallelize. While device evaluation can be performed in parallel in conventional circuit simulators, the execution overhead is high. We show that, by partitioning a circuit, OpenMP can be used to solve sub-circuits in different threads, without compromising accuracy. It is shown that execution time can be reduced proportionally to the number of threads.**

***Index Terms*—Circuit simulation, Parallel algorithms, SPICE.**

I. Introduction

One application that is very demanding of computing resources, but which has proved to be very resistant to parallelization is circuit-level simulation. The basic SPICE-type algorithm [1] has stood the test of time in terms of accuracy. Attempts to accelerate this algorithm have either compromised accuracy (or generality) or have used specialized hardware that has been eclipsed by advances in general purpose processors, e.g. [2].

There is a very clear need to accelerate circuit-level simulation, because of the growing size of circuits and the need to verify them with parametric variations [3], [4]. Instead of designing custom hardware, any such acceleration needs to use standard components – conventional multi-cores or, perhaps, graphics card accelerators. Some recent advances have been made in that direction [5]. Further, for portability, maintainability, etc. any such acceleration should be based upon standards [6].

OpenMP [7] is being adopted as a programming standard for multi-core systems with shared memory across a wide variety of hardware platforms and operating systems. As with many parallel programming standards, early versions of OpenMP assume regular, array-type data structures and enumerated looping constructs. Version 3.0 was published in May 2008 and added a much more general *task* model that is applicable to a much wider range of data structures and iterative or recursive looping schemes.

The simulator described here uses OpenMP to parallelize an existing circuit simulator [8]. Unlike many SPICE-type simulators, the sub-circuit structure is maintained. The original motivation for maintaining this hierarchy was to allow each subcircuit to choose its own time step, but the structure also lends itself to a relatively coarse-grained parallelism. By adding a number of OpenMP directives to the code and making a very small number of other changes to reorganize the existing code, parallel execution can be demonstrated, without loss of accuracy.

It is observed that the creation of threads does incur a computational cost. Therefore there is a minimal size of sub-circuit, below which there is no benefit to using a separate thread. Moreover for small sub-circuits, there is a significant penalty in creating a separate thread. It is shown that for large sub-circuits (1000 transistors), the cost of thread creation is insignificant compared with the computational effort required to evaluate the MOS models and to factorize the matrix equations and that the run time decreases proportionally with the number of processors available.

II. Circuit Simulation

In general terms, the equations for a nonlinear circuit may be expressed as a function [9]:

$$f\left(x, \dot{x}, t\right) = 0 \tag{1}$$

where x is the vector of unknown circuit variables, $\dot{x}$ is the time derivative of x and t is time. This equation cannot be solved analytically and therefore it is discretized in time, such that a nonlinear set of equations is solved at each time point:

$$g\left(x^n\right) = 0 \tag{2}$$

where $x^n = x(t^n)$.

The nonlinear equation (2) is linearized using the Newton-Raphson (N-R) method:

$$J^m x^{m+1} = J^m x^m - g(x^m) \tag{3}$$

where J^m is the matrix of partial derivatives of g with respect to x at iteration m at time point t^n. x^{m+1} is the vector of unknown circuit variables. The iteration proceeds until convergence, $x^{m+1} \approx x^m$.

For simplicity, let us rewrite equation (3) solely in terms of node voltages:

$$G^m v^{m+1} = I^m. \tag{4}$$

Equation (4) can be constructed (at each N-R iteration at each time point) by including the voltages from the previous iteration, v^m, in transistor and other model equations to derive the terminal currents and the partial derivatives of these currents with respect to the terminal voltages to give entries for I^m and G^m, respectively. G^m, known as the Jacobian matrix, can be thought of as a matrix of conductances and I^m as a vector of current sources.

Equation (4) is solved by factorizing G^m into lower and upper triangular matrices, L and U, and forward and back substituting to give x^{m+1}. G is usually very sparse (because in general, electronic components are connected to only 2 or 3 other components) and therefore the solution time is typically $O(N^{1.5})$ or better, where N is the number of circuit nodes.

Calculating the entries of G and I can be done in parallel for each device in the circuit, because there is no interaction between the devices. Techniques exist for the parallel solution of matrices. The device evaluation phase must complete before matrix solution can start and the matrix solution must complete before the device evaluation in the next iteration can begin. So there are two barriers that limit the amount of parallel execution that may be performed, Fig. 1.

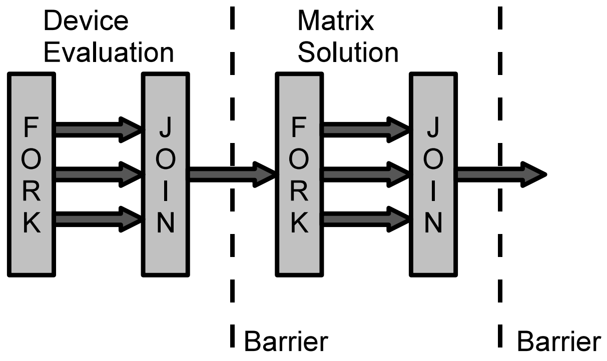

Fig. 1. Parallel thread execution

A. Hierarchy

The idea of maintaining the hierarchical partitioning of a circuit for simulation was first proposed in the mid-1970s [10]. The basic idea is that of node-tearing. From equation (4), the network equations for one sub-circuit may be stated as:

$$\begin{pmatrix} G_{11} & G_{12} \\ G_{21} & G_{22} \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \end{pmatrix} = \begin{pmatrix} I_1 \\ I_2 \end{pmatrix} + \begin{pmatrix} i_1 \\ 0 \end{pmatrix} \tag{5}$$

where v_1 is the vector of voltages for the torn (external) nodes; i_1 is the vector of currents flowing into those torn nodes; v_2 are the internal node voltages; G is the Jacobian matrix for the sub-circuit, with four sub-matrices and I is the vector of current sources. The iteration count, m, has been omitted for clarity. Although equation (5) is stated in terms of nodal analysis for the purposes of this explanation, it may be extended for modified nodal analysis [11]. This matrix equation applies at each Newton-Raphson iteration at each time step in a transient analysis.

By applying partial Gaussian Elimination, the G_{12} sub-matrix may be elminated and the G_{22} matrix transformed to a lower diagonal form:

$$\begin{pmatrix} G'_{11} & 0 \\ G'_{21} & G'_{22} \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \end{pmatrix} = \begin{pmatrix} I'_1 \\ I'_2 \end{pmatrix} + \begin{pmatrix} i_1 \\ 0 \end{pmatrix} \tag{6}$$

In effect, this generates a macromodel of the subcircuit, in terms of its external voltages (v_1) and currents (i_1), with the internal node voltages (v_2) suppressed. This macromodel can then be embedded in a sub-circuit in the next higher level of the hierarchy.

Once the external node voltages (v_1) are known, the internal voltages (v_2) are calculated by a simple back substitution.

The sub-circuit hierarchy can be represented as binary tree. Solving the circuit equations at one N-R iteration at one time point requires two traversals of this tree. This can be done using recursive procedures as illustrated in the following two algorithms.

Algorithm 1 ForwardElim(subcct *ptr)

```
1: if ptr− >child then
2:    ForwardElim(ptr− >child)
3: end if
4: EvaluateDevices(ptr)
5: GaussFore(ptr)
6: if ptr− >sibling then
7:    ForwardElim(ptr− >sibling)
8: end if
```

Algorithm 2 BackSubst(subcct *ptr)

```
1: GaussBack(ptr)
2: if ptr− >child then
3:    BackSubst(ptr− >child)
4: end if
5: if ptr− >sibling then
6:    BackSubst(ptr− >sibling)
7: end if
```

Algorithm 3 Simulation(subcct *maincircuit)

```
1: while t < tMAX do
2:    repeat
3:       ForwardElim(maincircuit)
4:       BackSubst(maincircuit)
5:    until convergence
6:    UpdateTimestep
7: end while
```

The two algorithms are called, in turn, for the main, top-level circuit until convergence is reached at each time point, Algorithm 3. EvaluateDevices calculates the contribution of each device to the sub-circuit matrix equation. GaussFore performs the forward phase of the Gaussian Elimination for each sub-circuit and GaussBack does the back substitution. It can be seen, therefore, that the overwhelming majority of the computation effort is expended in Algorithm 1.

This hierarchical solution approach has been implemented in a circuit simulator. If all subcircuits use a common timestep, the results obtained from a hierarchically partitioned simulation are mathematically the same as for a non-partitioned circuit. There may, however, be numerical differences because of a different evaluation order. It should also be noted that in order to perform the internal node supression, Gaussian Elimination is used, in contrast to LU factorization, as in SPICE.

III. Simulator Acceleration

The application of OpenMP to the hierarchical circuit simulator is motivated by a simple observation: the processing for one sub-circuit can be done at the same time as that for any of its siblings. Therefore, in principle, a new thread can be created for each sibling at each level of the hierarchy. It is, however, true that a child must be processed before its parent during the Forward Elimination phase (Algorithm 1). Therefore, there is no useful purpose in creating a new thread for the first child of any parent.

As has been noted, there is a cost to creating a new thread. The application of OpenMP has therefore been restricted to the Forward Elimination phase. This allows parallelization of both the model evaluation and marix factorization.

Algorithm 1 is therefore rewritten as Algorithm 4.

Algorithm 4 ForwardElim(subcct *ptr)

```
 1: if ptr− >sibling then
 2:     #pragma omp task
 3:     ForwardElim(ptr− >sibling)
 4: end if
 5: if ptr− >child then
 6:     ForwardElim(ptr− >child)
 7: end if
 8: EvaluateDevices(ptr)
 9: GaussFore(ptr)
10: #pragma omp taskwait
```

As can be seen, the changes are minimal. The call to process any sibling is made at the start of the routine. This does not affect the functionality in any way. A breadth-first, rather than a depth-first traversal is made, but children are always processed before their parent. The change is made to allow a new thread to be created at the start of the algorithm, so that it will execute concurrently with the remainder of the routine.

The OpenMP directive `#pragma omp task` is used to indicate that the call to ForwardElim for the sibling should be executed as a separate thread. Because this call will be executed for all the siblings at one level, all siblings would therefore be processed concurrently in separate threads. It is possible to attach attributes to the OpenMP directives to indicate the data scope and hence to protect data against corruption by other threads. In this case, because of the design of the data structures and because of the way in which models are evaluated and matrix values are updated, there is no interaction between siblings and hence there is no need to add extra attributes. Data from siblings is collected by their parent and hence any interaction between siblings occurs after they have all completed their execution.

A second OpenMP directive is needed at the end of the routine to ensure synchronization. `#pragma omp taskwait` causes the calling routine to wait until any threads that it has created have completed. Omitting this directive could allow processing to start on the parent before the children have completed and hence lead to incorrect or corrupted data.

In addition to these two directives, the two OpenMP directives `#pragma omp parallel` and `#pragma omp serial` need to be included in the main calling routine to set up parallel regions and to ensure that the timing and N-R loop control statements are only executed once, respectively.

In summary, therefore, the recasting the existing hierarchical circuit simulator as a multi-threaded requires the addition of just *four* OpenMP directives and minor rearrangement of some pieces of code.

TABLE I
Run times for multiplier

Threads	Run Time (s)
1	59.2
2	81.8
3	81.3
4	71.8
5	64.5
6	56.4
7	47.9
8	40.6

IV. Results

OpenMP 3.0 is included in gcc 4.4 [12]. An evaluation version of a commercial compiler was also used, which gave similar results, but which are not reported here. All simulations were run on a Mac Pro with two quad-core, 2.8GHz processors under Mac OS 10.5.

The number of threads used by a program can be set by the environment variable `OMP_NUM_THREADS`. By default, this is equal to the number of cores, in this case 8. In all the experiments reported below, `OMP_NUM_THREADS` was set to between 1 and 8 to emulate different numbers of available cores. There is, of course, a small overhead for including the OpenMP code, even if it is not used, but this is not quantified.

In all cases, the execution time is measured using the `time` command. The “real” time is recorded, as this measures the elapsed time, which is what parallel processing is designed to reduce.

A. Example 1

The first example is an analog multiplier wth 33 MOS transistors. 8 instances of the circuit were simulated in parallel. It might be argued that this is a case of trivial parallelism, but as the 8 instances are included in the matrix for the main circuit (in their reduced form), the 8 instances form one simulation. The run times for a transient analysis of 500ms using 1 to 8 threads are given in Table I.

It can be seen from Table I that the run time actually increases as more threads become available and that the break-even point is not reached until 6 threads are available. When 8 threads are available, the run time is about 68% of that for one thread, or a speed-up of nearly 1.5 times. It must be pointed out that there is no load balancing. With, for example, 4 threads,

TABLE II
RUN TIMES FOR PCHIP

Threads	Run Time (s)
1	68.0
2	60.9
3	54.0
4	46.8
5	38.9
6	30.7
7	23.0
8	15.2

the first 3 sub-circuits each has its own thread, while the final 5 sub-circuits are all processed on the final thread. So these figures are perhaps more encouraging than might first appear. The second observation is that there is clearly a significant overhead to thread creation. This explains the peak at 2 and 3 threads, but also indicates that sub-circuits need to have at least a few tens of transistors for any saving to be significant.

B. Example 2

The second example is taken from the CircuitSim90 [13] collection of benchmark circuits. The pchip circuit has 1029 transistors. The input and output buffers were not considered. Eight instances of the circuit were used, but this time they were chained together, to avoid any suggestion of trivial parallelism. The run times for the operating point analysis are given in Table II and plotted in Fig. 2.

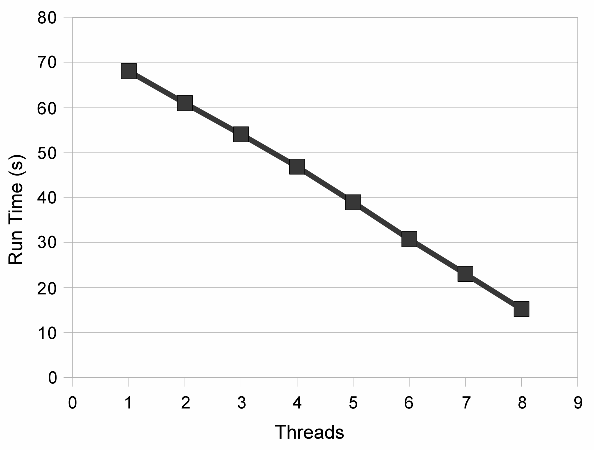

Fig. 2. Run time vs. No. Threads

The trend in Fig. 2 clearly shows that the run time decreases monotonically with the number of available threads. In this case, the complexity of the computation far outweighs the cost of thread creation, so there is no peak at 2 or 3 threads. Again, there is no load balancing, so, in effect, this shows the time required to process 8 sub-circuits down to one sub-circuit per thread. The speed-up is now 4.47 times for 8 threads.

C. Discussion

The results for the second example clearly demonstrate that it is possible to achieve a significant speed-up in circuit simulation by partitioning the circuit and distributing the processing across threads running on different cores. The examples given here use replicated sub-circuits to demonstrate the concept, but ideally a circuit already partitioned into multiple sub-circuits would be used. There are very few (if any) such benchmarks available. Thus, some form of automatic partitioning could be employed. This is, however, another manifestation of the clique problem, which is known to be NP-complete.

By working at the sub-circuit level, both device model evaluation and matrix factorization are performed in parallel. Amdahl's Law [14] still applies, of course. Within each Newton-Raphson iteration, there is still a need to implement part of the processing as a single thread - the factorization and solution of the main circuit's matrix, if nothing else. This will therefore limit the speed-up that can be achieved and this will always be less than the maximum implied by the number of processors.

V. CONCLUSIONS

We have presented an approach to exploiting parallelism on multi-core, shared memory systems in the context of circuit-level simulation. By taking a circuit-level simulator that maintains a hierarchy of sub-circuits internally, we use the new OpenMP standard to fork sub-circuit processing off to threads, each of which is running on a separate processor. Provided that each sub-circuit is complex enough for its processing to outweigh the cost of creating a new thread, we have demonstrated significant speed-up over a single threaded simulation.

REFERENCES

[1] L. Nagel, *SPICE2: A Computer Program to Simulate Semiconductor Circuits*. University of California, Berkeley, USA, 1975.
[2] R. Saleh, K. Gallivan, M.-C. Chang, I. Hajj, D. Smart, and T. Trick, "Parallel circuit simulation on supercomputers," *Proceedings of the IEEE*, vol. 77, no. 12, pp. 1915–1931, Dec 1989.
[3] Y. Ye, F. Liu, S. Nassif, and Y. Cao, "Statistical modeling and simulation of threshold variation under dopant fluctuations and line-edge roughness," *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*, pp. 900–905, June 2008.
[4] D. Sylvester, K. Agarwal, and S. Shah, "Variability in nanometer cmos: Impact, analysis, and minimization," *Integration, the VLSI Journal*, vol. 41, no. 3, pp. 319 – 339, 2008.
[5] W. Dong, P. Li, and X. Ye, "Wavepipe: Parallel transient simulation of analog and digital circuits on multi-core shared-memory machines," *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*, pp. 238–243, June 2008.
[6] T. Mattson and M. Wrinn, "Parallel programming: Can we PLEASE get it right this time?" *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*, pp. 7–11, June 2008.
[7] http://www.openmp.org.
[8] M. Zwolinski and K. Nichols, "The design of a hierarchical circuit-level simulator," *Electronic Design Automation*, 1984.
[9] V. Litovski and M. Zwolinski, *VLSI Circuit Simulation and Optimization*. Chapman and Hall, 1997.
[10] N. Rabbat and H. Hsieh, "A latent macromodular approach to large-scale sparse networks," *Circuits and Systems, IEEE Transactions on*, vol. 23, no. 12, pp. 745–752, Dec 1976.
[11] C.-W. Ho, A. Ruehli, and P. Brennan, "The modified nodal approach to network analysis," *Circuits and Systems, IEEE Transactions on*, vol. 22, no. 6, pp. 504–509, Jun 1975.
[12] http://gcc.gnu.org/.
[13] http://www.cbl.ncsu.edu:16080/benchmarks/.
[14] G. Amdahl, "Validity of the single processor approach to achieving large-scale computing capabilities," in *AFIPS Conference Proceedings, (30)*, 1967, pp. 483–485.