Desktop Supercomputing: Simulation With Multi- and Many-Core Processors Invited Paper

Mark Zwolinski

***Abstract*—Processor speed has remained constant for several years, but the number of CPUs per chip has increased. Furthermore, graphics cards now include tens of processors. Using these resources for scientific computing is a new challenge. A number of standards have appeared that simplify much of the mechanics of writing parallel programs. The fundamental challenge of exactly how to exploit parallelism remains. This paper shows how two technologies, OpenMP and OpenCL, have been used to accelerate different aspects of circuit simulation.**

***Keywords*—Circuit simulation, Parallel algorithms, SPICE.**

I. Introduction

In 1965, Gordon Moore [1] observed that the number of transistors on an integrated circuit was doubling each year. Although subject to some revision, notably that *performance* doubles every 18 months, "Moore's Law" has become a self-fulfilling prophecy. In recent years, there has been an important qualification to this law. The number of transistors continues to increase at the same rate as before, but clock speeds have stalled at less than 4 GHz. While clock speed should not be used as an absolute measure of performance, it is clear that the throughput of individual CPUs is increasing much more slowly than the transistor count.

The explanation for this discrepancy is, of course, that the number of CPU cores per integrated circuit is increasing. Ideally, therefore, the throughput per chip is continuing to increase in line with Moore's Law. In practice, this speed increase can only be achieved if the applications are trivially parallel – in other words, if there is no communication between concurrent processes.

The number of cores per chip is currently 4 to 6 for Intel and AMD devices. A PC or server might contain four such ICs, allowing perhaps 16 cores to share memory. High Performance Computing (HPC) systems include many (tens of thousands) of such servers, which communicate through message passing protocols. On the other hand, graphics cards include tens or hundreds of small, dedicated processors (Graphics Processing Units or GPUs). Each processor is capable of floating-point operations. Thus, graphics cards can be utilised for general purpose numerical programming (General Purpose GPUs or GPGPUs).

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

While transistor counts have been growing as anticipated by Moore's Law, the productivity of designers has been advancing more slowly. There is currently a "design gap" between the output of designers and the productivity expected for fulfillment of the Moore's Law prophecy. Simulators are important tools for bridging the design gap, but to date, most simulators for electronics design have been written with a single execution thread. This presents the designers of simulation tools with a new challenge. For many years, it has been possible to rely on increasing CPU speed to drive simulator performance. That is no longer possible and simulators must now be designed to exploit concurrency.

Three standards have emerged for the three types of parallelism introduced above. MPI (Message Passing Interface) [2] provides a mechanism for loosely-coupled processes to communicate. OpenMP [3] allows concurrent threads on parallel processors to communicate through shared memory. OpenCL [4] is a new standard that allows a programmer to exploit the power of GPUs. The three technologies can be mixed in the same software, allowing both homgeneous and heterogeneous systems to be built.

In this paper, the three programming technologies will be briefly described in the next section. An example of parallelising a simulator program using OpenMP will then be described. Finally, an example of the use of OpenCL will be given.

II. Parallel Programming

Parallel programming has always been difficult and remains so. In this section, we will briefly look at three standards that assist with the mechanics of parallel programming. None of these approaches is a solution to the problem of *how* to convert a sequential algorithm into a parallel form.

It is possible to combine two or more of these standards in a single application, in order to make full use of the available resources.

A. MPI

MPI [2] is the standard used in High-Performance Computing (HPC). As the name implies, MPI provides a standard method for passing messages between processes running on concurrent processors. Most of the functionality is provided by daemon processes running on each processor, thus, from a programmer's point of view, it is only necessary to include funtion calls, such as `MPI_Send` and `MPI_Recv`.

Because MPI relies on message passing that is slow and unpredictable with respect to time, it is only effective if the application is either sufficiently decoupled or sufficiently large that the overheads are not significant compared with the computation. (This is generally true of all parallel processing, but the speed of message passing is particularly significant.) Thus MPI is suited to simulations of large physical systems. In general, however, applications such as circuit simulation do not map easily to MPI.

B. OpenMP

OpenMP [3] is an applications programming interface for implementing shared memory, parallel programming. Within a program, a number of threads may be created, to run in parallel on separate cores. The shared memory model is particularly relevant to the multi-core processor systems that are now appearing as workstations.

A significant advantage of OpenMP over other coding styles is that it does not necessarily require major rewriting of existing code. The basic OpenMP model is that code is annotated with directives to show parallel sections.

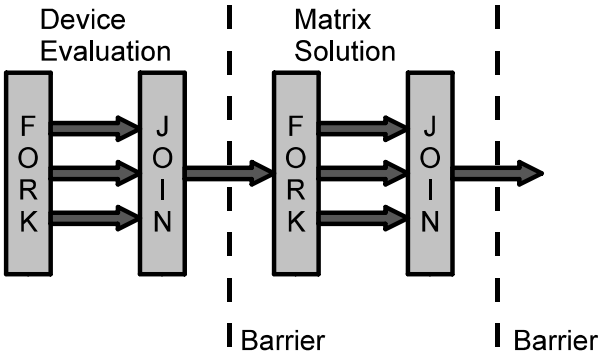

Fig. 1. Parallel thread execution

The execution model is that shown in Fig. 1. Threads are forked and joined according to the directives given by the programmer. For many applications this model of parallel sections interleaved with single threaded sections is appropriate. However, creating a new thread will take a certain amount of time. Therefore simply adding directives to existing code, without considering the overall program flow is unlikely to achieve a major speed-up.

The latest version of OpenMP, version 3.0, was published in May 2008. A significant enhancement is the ability to label arbitrary loops and function calls with a `task` directive. For example, each element of a linked list of indeterminate length could be processed by a different thread using the `task` directive:

```
#pragma omp parallel
#pragma omp single
for (p=start; p; p=p->next)
#pragma omp task
  task(p);
```

The first directive, `#pragma omp parallel` is needed to set up the parallel environment. `#pragma omp single` specifies that the `for` loop incrementing is only done once, while the tasks are forked off to individual threads with the third directive. In this example, there is no need to qualify the parameter passed to each task as there is (apparently) no interaction between elements in the list.

The `task` directive can be applied to any statement, although the caveat about thread creation costs clearly applies. In particular, it can be applied to functions that process arbitrary data structures such as linked lists or trees.

It should also be noted that execution continues along the main thread at the same time as any forked threads. If the main thread completes a task before any forked threads, execution will proceed to the next statement. It may, therefore, be necessary to specify an explicit join point. In OpenMP 3.0, this is done with the `#pragma omp taskwait` directive.

C. OpenCL

In recent years, there has been a trend to move graphics processing onto specialised graphics cards. These contain 10 or more small-scale processors, each capable of floating-point operations. Attention has turned to the possible use of these Graphical Processing Units (GPUs) for numerical processing. The leading vendors have each produced their own development kits, but in 2009, a common programming interface – OpenCL [4] – was released. OpenCL is now an important part of Apple's Mac OS X.

The architecture of GPUs has been used in more powerful General Purpose GPUs (GPGPUs), that have a larger number of processors and which may not even include graphics outputs. Examples include the NVidia Tesla range [5] and the ATI/AMD Firestream cards [6].

Each processing unit on a graphics card has a limited amount of memory and limited processing power. The OpenCL language is a subset of C, designed to allow part of a problem to run on each processor. For example, a function to square the elements of a vector can be written as:

```
__kernel void square(
   __global float* input,
   __global float* output,
   const unsigned int count)
```

```
{
    int i = get_global_id(0);
    if(i < count)
        output[i] = input[i] * input[i];
}
```

Because OpenCL is intended to be portable between different GPGPUs, the *kernel* code is compiled "on the fly". The programming interface, therefore, consists of routines to determine the hardware resources, to set up the computing environment, including input and output buffers, and to compile and run the kernel code.

The use of OpenCL is not limited to GPGPUs. The CPU of a system, perhaps with multiple cores, can be used an an OpenCL resource, running the same kernel code. As presently implemented, this is unlikely to be efficient, but could, in principle, obviate the need for OpenMP.

III. Multi-Threaded Circuit Simulation Using OpenMP

A. Hierarchical Circuit Simulation

In general terms, the equations for a nonlinear circuit may be expressed as a function [7]:

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

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:

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

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

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

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

where A^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, $x^{m+1} \approx x^m$.

Calculating the entries of A^m and b^m 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.

A different approach is to partition the circuit and to solve each partition in parallel. The idea of maintaining the hierarchical partitioning of a circuit for simulation was first proposed in the mid-1970s [8]. The basic idea is that of node-tearing.

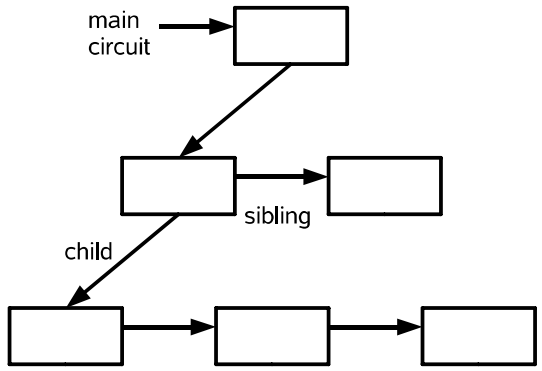

Fig. 2. Sub-circuit hierarchy

The sub-circuit hierarchy can be represented as binary tree, Fig. 2. 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.

B. 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.

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.

TABLE I
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

C. Results

This example is taken from the CircuitSim90 [9] 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 number of threads can be set by the environment variable `OMP_NUM_THREADS`. By default, this is equal to the number of cores, in this case 8.The run times for the operating point analysis are given in Table I and plotted in Fig. 3.

The trend in Fig. 3 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. 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 4.47 times for 8 threads.

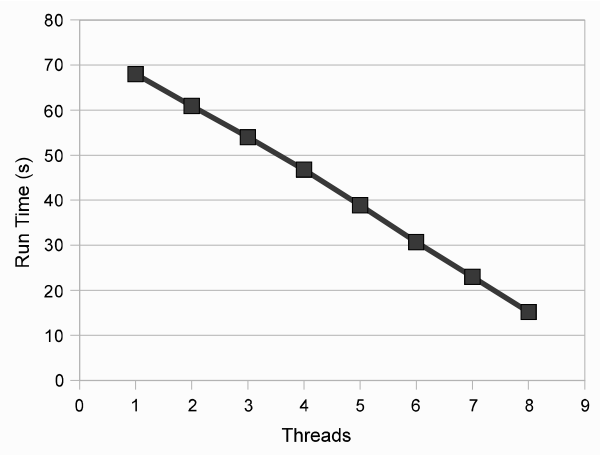

Fig. 3. Run time vs. No. Threads

IV. Matrix Factorisation Using OpenCL

In circuit simulation, at each N-R iteration, equation (3) is solved by factorizing A^m into lower and upper triangular matrices, L and U, and forward and back substituting to give x^{m+1}. J 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. On the other hand, in circuit simulation, the matrix is asymmetric (because circuits have gain), so methods such as Cholesky decomposition are not appropriate.

Crout's algorithm [10] is used to factorise a matrix. Implicitly $l_{ii} = 1, i = 1, \ldots, N$,

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj}, i = 0, \ldots, j \tag{4}$$

and

$$l_{ij} = \frac{1}{u_{ii}} \left[a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj} \right], i = j+1, \ldots, N-1. \tag{5}$$

It can be seen that there is dependency between the two computations. Thus matrix factorisation is usually performed in a serial manner.

In order to parallelise the process, it is necessary to divide the matrix into sub-matrices [11]. These sub-matrices are then coupled together in a final step. Each of the sub-matrices can be factorised in parallel. This is exactly equivalent to thinking of the circuit being partitioned into sub-circuits, as in Figure 2.

Figure 4 shows the speed increase that can be achieved for large matrices. The experiment was performed on an NVidia Tesla card. The test data was a diagonally-banded matrix. In fact, the example was coded using the precursor to OpenCL – CUDA. It can be seen that the GPU version of the code is about 13 times faster for matrices of dimension 8000. The cross-over occurs at about 500; below that the overhead required to move data on and off the GPU tends to dominate.

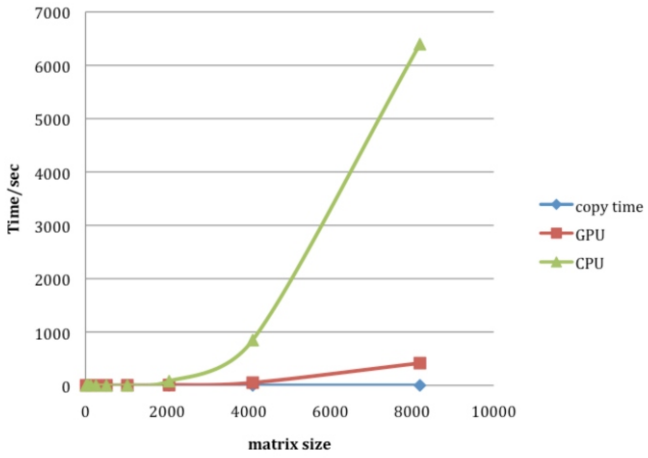

Fig. 4. GPU vs. CPU Run Time

V. Conclusions

Parallel programming remains one of the most significant challenges in the development of new EDA tools. While new technologies allow multiple CPUs and GPUs to be exploited, they do not solve the problem of how to partition a problem. Nevertheless, by using these technologies, either singly or together, we now have the opportunity to simulate much larger systems on a desktop machine than would be possible using a single CPU.

Acknowledgements

The results for matrix factorisation were obtained by Wang Yuyang as part of his MSc dissertation project.

References

[1] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, no. 8, 1965.
[2] http://www.open-mpi.org/.
[3] http://www.openmp.org.
[4] http://www.khronos.org/opencl/.
[5] http://www.nvidia.com/object/tesla_computing_solutions.html.
[6] http://www.amd.com/us/products/technologies/stream-technology/Pages/stream-technology.aspx.
[7] V. Litovski and M. Zwolinski, *VLSI Circuit Simulation and Optimization*. Chapman and Hall, 1997.
[8] 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.
[9] http://www.cbl.ncsu.edu:16080/benchmarks/.
[10] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery, *Numerical Recipes in C*, 2nd ed. Cambridge, UK: Cambridge University Press, 1992.
[11] C.-C. Chen and Y.-H. Hu, "Parallel LU factorization for circuit simulation on an MIMD computer," *Proceedings of the 1988 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pp. 129–132, Oct 1988.