SAND2010-6937P

Parallel Transistor-Level Circuit Simulation

Eric R. Keiter, Heidi K. Thornquist, Robert J. Hoekstra, Thomas V. Russo,
Richard L. Schiek, and Eric L. Rankin

Abstract With the advent of multi-core technology, inexpensive large-scale parallel platforms are now widely available. While this presents new opportunities for the EDA community, traditional transistor-level, SPICE-style circuit simulation has unique parallel simulation challenges. Here the Xyce Parallel Circuit Simulator is described, which has been designed from the from-the-ground-up to be distributed memory-parallel. Xyce has demonstrated scalable circuit simulation on hundreds of processors, but doing so required a comprehensive parallel strategy. This included the development of new solver technologies, including novel preconditioned iterative solvers, as well as attention to other aspects of the simulation such as parallel file I/O, and efficient load balancing of device evaluations and linear systems. Xyce relies primarily upon a message-passing (MPI-based) implementation, but optimal scalability on multi-core platforms can require a combination of message-passing and threading. To accommodate future parallel platforms, software abstractions allowing adaptation to other parallel paradigms are part of the Xyce design.

Eric R. Keiter, Heidi K. Thornquist, Thomas V. Russo, Richard L. Schiek, and Eric L. Rankin
Electrical and Microsystems Modeling Department,
Sandia National Laboratories
P.O. Box 5800, Albuquerque, NM 87185-0316,
e-mail: erkeite@sandia.gov, hkthorn@sandia.gov, tvrusso@sandia.gov, rlschiek@sandia.gov, elranki@sandia.gov

Robert J. Hoekstra
Applied Mathematics & Applications Department,
Sandia National Laboratories
P.O. Box 5800, Albuquerque, NM 87185-0318,
e-mail: rjhoeks@sandia.gov

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energys National Nuclear Security Administration under contract DE-AC04-94AL85000.

1 Introduction

At modern technology nodes, analog style SPICE-accurate simulation can be a significant (and prohibitive) development bottleneck. Traditional circuit simulation, originally made popular by the Berkeley SPICE program[26], does not scale well beyond tens of thousands of unknowns, due to the use of direct matrix solver methods.

A number of algorithms for FastSPICE tools have been developed which allow for faster, larger-scale circuit simulation. Often based on circuit-level partitioning algorithms [1, 3, 27, 34], such tools can be applied to much larger problems, but the approximations inherent to such algorithms can break down under some circumstances. In particular, for state-of-the-art modern VLSI design, high levels of integration between functional modules and interconnects are subject to prohibitive parasitic effects, and can render such tools unreliable.

Recent development of inexpensive computer clusters, as well as multi-core technology, has resulted in significant interest for efficient parallel circuit simulation. Parallel "true-SPICE" circuit simulation has been investigated previously, including Frölich [13], who relied on a multi-level Newton approach in the Titan simulator; Basermann [8], who used a Schur-complement based preconditioner; and Peng et al. [28] used a domain decomposition approach and relied on a combination of direct and iterative solvers. Recently, interest has developed around parallel SPICE acceleration using graphical processing units (GPUs) [15].

Parallel circuit simulation requires integration of large and small scale parallelism throughout the entire circuit simulation flow. In this paper, parallel algorithms for circuit simulation, and their implementation in a production simulator, Xyce, are discussed. Xyce [4] is a simulator designed "from-the-ground-up" to be distributed memory-parallel, and is targeted at a spectrum of parallel platforms, from high-end supercomputers, to large clusters, to multi-core desktops. It relies primarily upon a message-passing implementation (MPI) [14], but optimal scalability on multi-core technology can require combined message-passing and threading. Xyce uses software abstractions that allow the simulator to adapt to other parallel paradigms.

2 Background

Circuit simulation adheres to a general flow, as shown in Fig. 1. The circuit, described in a netlist file, is transformed via modified nodal analysis (MNA) into a set of nonlinear differential algebraic equations (DAEs)

$$\frac{dq(x(t))}{dt} + f(x(t)) = b(t), \tag{1}$$

where $x(t) \in \mathbb{R}^N$ is the vector of circuit unknowns, q and f are functions representing the dynamic and static circuit elements (respectively), and $b(t) \in \mathbb{R}^M$ is the input vector. For any analysis type, the initial starting point is this set of DAEs. The

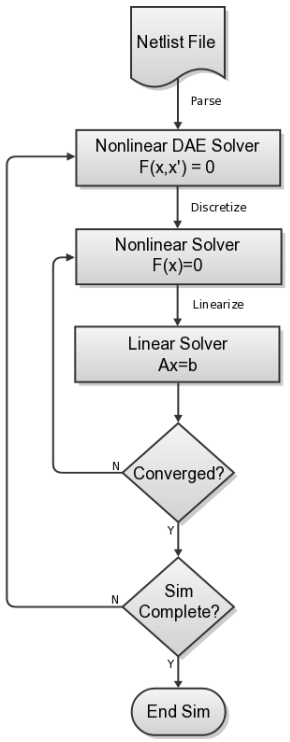

Fig. 1 General circuit simulation flow

numerical approach employed to compute solutions to equation (1) is predicated by the analysis type.

Transient and DC analysis are two commonly used simulation modes, in which the set of equations (1) are solved by numerical integration methods corresponding to the nested solver loop in Fig. 1. For transient analysis, linear systems are of the form:

$$(G+Q/\delta t)\delta x = (b-f)/\delta t \tag{2}$$

involving the conductance matrix $G(t) = \frac{df}{dx}(x(t))$, and the capacitance matrix $Q(t) = \frac{dq}{dx}(x(t))$. For DC analysis, the q terms are not present, so equation (1) is reduced to the nonlinear equation $f(x) = 0$, and the linear system is simplified to:

$$G\,\delta x = -f(x). \tag{3}$$

For transient and DC analysis, the computational expense is in repeatedly solving linear systems of equations given by (2) or (3), respectively. These linear systems are typically sparse, have heterogeneous non-symmetric structure, and are often ill-conditioned. As such, iterative matrix solvers have historically not been the first choice for circuit simulation, and direct sparse solvers [10, 21] have been the industry standard approach. Direct solvers have the advantage of reliability and ease of

use, and for smaller problems direct methods are usually faster than iterative methods. However, direct solvers typically scale poorly with problem size and become impractical when the linear system has hundreds of thousands of unknowns or more.

Despite the problems inherent to circuit matrices, iterative solvers have the potential to be a scalable solution method for large-scale linear systems with lower algorithmic complexity. They are not as easy to use as direct solvers because their effectiveness is dependent upon finding an adequate preconditioner. However, progress has been made on the use of iterative methods in transient circuit analysis; notably Basermann [8] and Bomhof [9], both of whom relied on distributed Schur-complement based preconditioners. Also, multi-grid methods have successfully been applied to power-grid simulation [32]. Recently, a new preconditioning strategy has been developed to generate effective preconditioners for performing transient analysis using matrix structure found during DC analysis [16]. Nonetheless, for conventional transient circuit simulation, iterative matrix solvers have yet to be widely used in production tools.

3 Parallelism Opportunities in Circuit Simulation

Parallelism can be integrated into every step of the circuit simulation flow shown in Fig. 1. Furthermore, at every step, parallelism can be achieved through both coarse-scale (multi-processor) and fine-scale (multi-threaded) approaches. A composition of these two approaches will provide circuit simulation with the best performance impact on the widest variety of parallel platforms.

This section contains a discussion of the parallelism opportunities exploited by the Xyce simulation flow. Section 3.1 will focus on the parallel netlist parser, while section 3.2 describes parallelism in the nonlinear and linear solvers. The majority of the computational time is spent in device evaluations and linear solvers, so section 3.2 will focus on parallelism pertaining to those specific tasks.

3.1 Parallel Netlist Parser

An efficient parallel netlist parser is essential for the simulation of very large circuits. The parser is the gateway through which the devices and network topology are established and, if inefficient in computational time or memory, can easily become the Achilles' heel of a circuit simulator. Developing a netlist parser to work on a wide variety of platforms, from desktops to supercomputers, can be a harrowing task because of parallel file I/O. As such, the netlist parser in Xyce has had to address two file system scenarios: inhomogeneous and distributed, homogeneous file systems.

3.1.1 Inhomogeneous File System

An inhomogeneous file system, often found in networked computer clusters, is the most general type of parallel file system. For such a file system, it cannot be assumed that each node has access to the same directories and files, so the netlist has to be

streamed on one processor. This presents an immediate parallel bottleneck, both in terms of performance as well as memory. The approach taken in Xyce for this sort of file system involves minimally processing the netlist prior to distributing devices and analysis information to the other processors.

Parsing is accomplished in multiple passes of the netlist. In the first pass, the netlist is dynamically flattened and a total device instance count is obtained. This allows an initial, naive, parallel partitioning of devices to be determined based on D/N, where D=number of devices and N=number of nodes. On the subsequent pass, raw netlist data for each device is broadcast sequentially to the remaining nodes or processors. After this process is completed, the devices are instantiated and initialized locally on each node. This approach does not completely remove the parallel bottleneck inherent to streaming the netlist file on one processor. However, it enables the parsing process to be mostly scalable in terms of memory usage.

The initial D/N partition determines the load balance for device evaluation. For large problems, a naive partition (where each device is given an identical weight) is often adequate, as the runtime is dominated by the linear solve at each step. However, a more optimal device evaluation partition can easily be created by using weightings.

3.1.2 Distributed Homogeneous File System

A homogeneous parallel file system is a more common file system on modern parallel architectures. Though less general, it is in principle more scalable. Similar to the heterogeneous case, an initial pass through the netlist is necessary, to determine device count, and also to set file pointers for each node. As each node has full access to the file system, parts of the netlist can be streamed in to each node directly, rather than streamed in on one processor and then communicated. While this has the potential to be scalable in terms of operations as well as memory, the bookkeeping can be prohibitive for very hierarchical netlists. As with the heterogeneous file system, the initial device evaluation partition is determined as part of the parallel netlist parsing process.

3.2 Parallel Approach For Nested Solver Loop

In Xyce, the parallel approach taken for the nested solver loop is designed to account for circuit heterogeneity, so optimal parallel load balance for device evaluation (matrix and residual vector assembly) will likely differ from that of the linear solution phase. Device evaluation and linear solves each happen once per Newton iteration, so over the course of a long run the combined cost of both will comprise the bulk of the wall clock simulation time.

The relative amount of time spent in each phase is problem-dependent. For smaller problems, the device evaluation phase should dominate run time. As the problem size increases, the linear solve phase will dominate, as it should scale superlinearly, while the device evaluations should scale linearly. This is because linear solution methods (whether they be direct or iterative) are generally communication

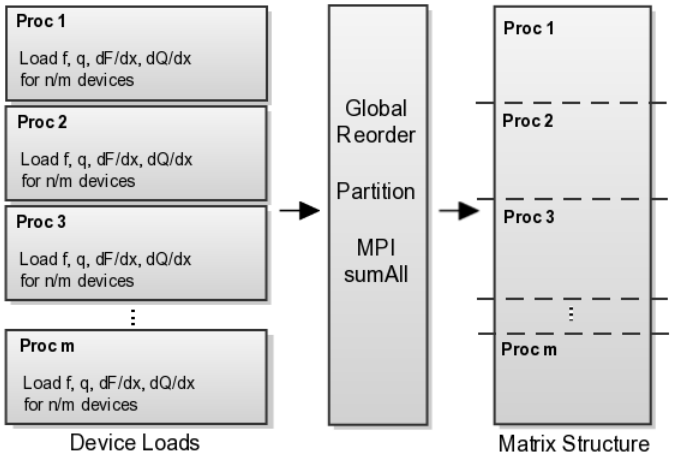

Fig. 2 Different load balance/partitioning for device evaluation and linear solve

intensive, while the communication volume required during the device evaluations is relatively small.

As a result, the device evaluation phase can be balanced by taking into account only the computational work required, while the matrix partitioning has to minimize communication volume. How this communication volume is measured, and how it is optimized is an active area of research for many types of numerical simulation problems. Since the device evaluation and solve phases have different load balance requirements, Xyce has been designed to have completely different parallel partitioning for each. A simplified representation of this is shown in Fig. 2.

3.3 Device Evaluation

In Fig. 2, the left column represents the device evaluation procedure. Communication costs for this are relatively low and insensitive to load balance. At the beginning of each device evaluation, solution and state vector values are needed by each device, and off-processor values are communicated as necessary. At the end of the device evaluation, before final assembly, residual and Jacobian matrix values must be communicated to fill the final linear system. Thus, the communication cost is relatively cheap, and the main metric considered is the computational cost. For many circuits of interest, a naive load balance, in which the total number of devices is evenly divided among the available processors will demonstrate very good parallel scaling. For circuits that are very heterogeneous, weights can be applied to different device types to achieve a better balance.

The middle box in Fig. 2 represents the communication necessary to accommodate both load balances. This is dependent upon the partitioning of the linear system, which is a much more difficult and complex issue. Unlike the device evaluation, a naive partitioning will generally not suffice.

3.4 Linear Solvers for Circuit Simulation

Solving the linear systems of equations, (2) and (3), generated through circuit simulation is often challenging. These linear systems are sparse, typically have heterogeneous non-symmetric structure, and are often ill-conditioned. Direct sparse linear solvers provide a reliable, easy-to-use solution method that is preferred, industry-wide, over iterative solvers. For smaller linear systems, this is understandable, because direct solvers are usually faster than their iterative counterparts. However, when the linear system has hundreds of thousands of unknowns or more, direct solvers become less practical as they suffer from poor scaling. So any advantage gained through a scalable device evaluation procedure, is lost through the linear solver. This situation only becomes more pronounced as the problem, or circuit, size increases.

Iterative solvers, like GMRES [30], are scalable and robust for other types of physical problems, but not generally for circuit simulation. This issue results from the inability to generate effective, general-purpose, scalable preconditioners. The heterogeneity in circuit matrices presents an exceptional challenge for creating a preconditioner that approximates the coefficient matrix well, to accelerate convergence, and is inexpensive to apply and scalable. Xyce takes advantage of several common matrix properties when constructing a preconditioner for equations (2) or (3). In the rest of this section the algorithms that take advantage of these properties will be presented, including: singleton removal, block-triangular permutation, and parallel partitioning. This will be followed by a discussion of the iterative linear solver strategies that are used in Xyce.

3.4.1 Singleton Removal

Conductance and capacitance matrices are sparse, but typically contain dense rows and columns. Such structural matrix features are problematic for parallelism because they have the potential to increase communication costs dramatically. A crucial observation [8] is that the dense rows (or columns) correspond to columns (or rows) with one and only one non-zero entry, called a singleton. Eliminating singleton rows and columns, a standard practice in the original sparse direct methods for circuits (section 6.3.1 of [21]), also eliminates the dense rows and columns. These dense rows and columns typically result from power supply and ground nodes, which are common to digital circuits. Other features such as clock nodes, while not as highly connected as power nodes, still have a high enough connectivity to cause problems. A histogram illustrating the connectivity for a sample integrated circuit (IC) is depicted in Fig. 3.

Dense rows and columns can easily be removed from circuit matrices as pre- and post-solve steps because the corresponding columns or rows will only have one non-zero entry [8]. The procedure, referred to as “singleton removal”, is used by Xyce as a first step for solving equations (2) and (3). The pre-processing step for handling row singletons by removing the independent variable (x_j), is given on the left in Fig. 4. While the post-processing step for handling column singletons, where the variable (x_j) is fully dependent, is given on the right in Fig. 4.

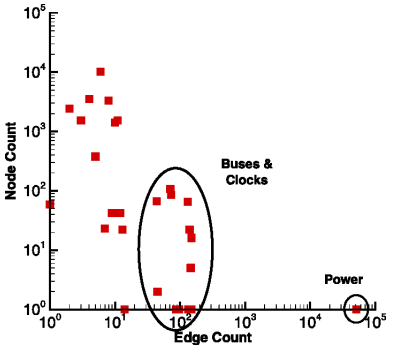

Fig. 3 Example IC connectivity histogram

Singleton removal is a successful strategy, but only when the nodes are connected to ideal sources, and thus can essentially be removed from the system of equations by inspection, as their values are known. In practice, highly connected nodes may be non-ideal due to parasitic elements, and under this circumstance, the singleton removal algorithm can fail to detect them. A mitigation strategy designed to preserve singleton removal, based on multi-level Newton methods is presented in section 4.

$$\begin{bmatrix} & a_{1j} & \\ & \vdots & \\ 0 \cdots & a_{ij} & \cdots 0 \\ & \vdots & \\ & a_{nj} & \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_j \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ \vdots \\ b_i \\ \vdots \\ b_n \end{bmatrix} \qquad \begin{bmatrix} & 0 & \\ & \vdots & \\ a_{i1} \cdots & a_{ij} & \cdots a_{in} \\ & \vdots & \\ & 0 & \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_j \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ \vdots \\ b_i \\ \vdots \\ b_n \end{bmatrix}$$

$$\Rightarrow x_j = b_i / a_{ij} \qquad \Rightarrow x_j = \left(b_i - \sum_{k \neq j} a_{ik} x_k \right) / a_{ij}$$

Fig. 4 Singleton removal for rows (left) and columns (right)

3.4.2 BTF Reordering

The conductance matrix G used during DC analysis can be reducible [10], permutable to a block triangular matrix with small diagonal blocks. Exploiting this block triangular form (BTF) often gives great performance gains both for direct and iterative solvers. Direct solvers only factor the tiny diagonal blocks, handling the off-diagonal blocks in the substitution phase. However, the challenge is in performing transient analysis, when the linear systems are not reducible. Iterative solvers can take advantage of this block structure to generate preconditioners that are effective for both DC and transient analysis [16].

The permutation to block triangular form, via the Dulmage Mendelsohn decomposition, has two steps. First, a maximum matching permutation generates a matrix with a zero-free diagonal. Second, a topological sort finds the strongly-connected components of the associated directed graph. In Xyce, computing the permutation to block triangular form is a serial bottleneck because there is no parallel software

available. However, both steps required to compute this transformation can probably be parallelized. The first step requires a bipartite matching, which is the most difficult to parallelize [24], but most device models produce Jacobian matrices with very few zeros on the diagonal. The second step finds strongly connected components and recent work shows this can be done in parallel [25]. For the time being, computing the BTF permutation in serial is acceptable because the same reordering can be used for each Newton step of the DC and transient analysis. Thus, the cost is amortized over the entire simulation.

3.4.3 Parallel Partitioning

An effective sparse matrix partitioning, where the goal is to reduce the communication in sparse matrix-vector multiplication, is essential to iterative matrix solvers. Furthermore, it has been demonstrated for a variety of physical simulation problems [33, 11] that a good parallel partition can enhance the performance of preconditioned iterative matrix solvers. Graph-based partitioning algorithms have been popular for some time [20] and became computationally feasible for large problems with the advent of multilevel methods [17]. In recent years, hypergraph partitioning has shown a lot of promise [12], in part because it relies upon more accurate metrics for measuring parallel communication costs. This will give an exact estimate of the costs required for a matrix-vector multiply, which is the main communication expense for iterative methods like GMRES. Interestingly enough, hypergraph-based partitioning has also been demonstrated to be an effective algorithm for optimizing circuit layout for path-based delay minimization [18].

Sparse matrix partitioning is performed in Xyce using either a graph or hypergraph partitioner via ParMETIS [19] or Zoltan [12], respectively. Either approach generates a 1D partition, which partitions only the rows, assigning each row to only one processor. Partitioning is computationally expensive, but the graph for conductance and capacitance matrices is static over the course of the simulation. Thus, the partitioning can be reused, amortizing the cost over the entire simulation.

3.4.4 Iterative Linear Solver Strategies

In Xyce, the previously discussed matrix transformations are combined with algebraic preconditioning techniques to generate an iterative linear solver strategy. The two strategies that have proven to be effective in preconditioning circuit matrices will be presented in this section. In both, the linear solver is a block diagonal preconditioned GMRES [30] method, in which the number of "blocks" is the number of cores or processors used. To avoid confusion these "blocks" are referred to as subdomains. A block diagonal preconditioner applies the exact (Jacobi) or approximate inverse of the subdomain matrix as its preconditioner. This preconditioner requires no communication to perform the factorization and in its application, which makes it suitable for parallel computation.

The first strategy is a general-purpose domain decomposition (DD) strategy, and is illustrated in Fig. 5. Domain decomposition uses singleton removal, graph partitioning (ParMETIS [19]) on the resulting symmetrized graph to reduce commu-

nication, and then computes a local fill-reducing ordering (AMD [6]) on the block diagonal before performing a complete or incomplete LU (ILU) factorization. This is used as a preconditioner for GMRES [30].

The second linear solver strategy was recently developed and is based on the block triangular structure that is often found in conductance matrices [16]. When the BTF structure is present, this strategy generates an effective preconditioner for DC analysis that can also be used in transient analysis. The BTF linear solver strategy, as illustrated in Fig. 5, uses singleton removal and then permutes the resulting graph to block triangular form. The strongly connected components are then partitioned, and each subdomain is solved directly using KLU [31] as a preconditioner for GMRES [30].

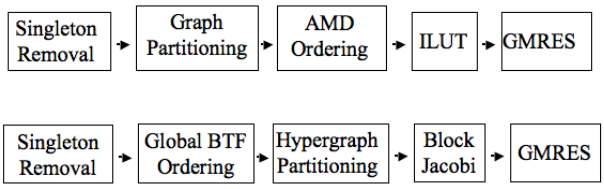

Fig. 5 Domain decomposition (top) and BTF (bottom) linear solver strategies

4 Graph Mitigation using Multilevel Newton Methods

Several of the linear solver steps described in section 3.4 depend upon the circuit or Jacobian matrix having specific structure. For example, the singleton removal algorithm (section 3.4.1) assumes that power and clock node voltages are set with ideal sources, with no intervening parasitic elements. Also, the BTF reordering (section 3.4.2) only works if the circuit or matrix is sufficiently unidirectional, which is not the case for all circuits.

Often, it is possible to mitigate structural problems by strategically partitioning the nonlinear solver, and applying the multilevel Newton method [29]. In general, such methods have been used to enable a simulator to optimally apply ideal solver methods to different phases of a simulation. Peng et al. [28] used a multilevel Newton approach to enhance post-layout transistor-level circuit simulation. In this work, the problem was partitioned into linear and nonlinear partitions. The linear partition (due to post-layout parasitics), produced an SPD matrix that could be efficiently solved using the Conjugate-Gradient (CG) method, while the nonlinear partition produced a much smaller non-SPD matrix that was solved using direct methods.

For parallel circuit simulation, the multilevel Newton method can be applied to break up the graph structure of the circuit, with the goal of preserving the effectiveness of matrix ordering and other pre-processing steps. The method will be described next, followed by its application to non ideal power supplies (section 4.2).

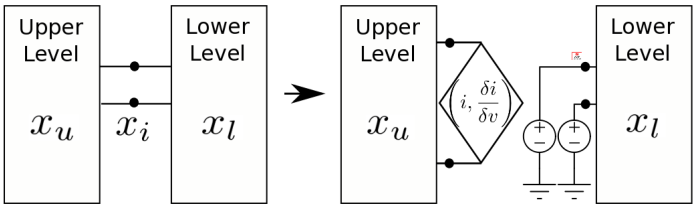

Fig. 6 Simple multilevel Newton decomposition, where x_u, x_l and x_i are the solution variables for the upper level, lower level, and interface, respectively.

4.1 Multilevel Newton Method

The multilevel Newton method has been described by numerous authors [29], and involves partitioning a problem into a tree structure, with a top level and subordinate lower levels. Circuits can often be very naturally partitioned based on subcircuits, but such a partition will not always fulfill the numerical objective which justifies using the method.

An illustration of a multilevel Newton partitioning, consisting of two levels, is shown in Fig. 6. The variables x_u, x_l and x_i represent solution variables for the upper level, lower level, and interface respectively.

The multilevel method treats each level separately, with the lower level undergoing a full Newton solve for each iteration of the upper level. For circuit simulation, using the MNA formulation, most of the solution variables x will be nodal voltages. The interface nodal values x_i will be imposed into each lower level as fixed voltage boundary conditions, or independent voltage sources, as indicated in Fig. 6.

From the point of view of the upper level, the subordinate portions of the circuit, or lower levels, are replaced by macromodels, which provide Ohmic relationships similar to that of conventional compact device models. As such, each lower level subcircuit must provide currents and conductances to fill out their respective matrix stamps in the upper level system.

After each lower level solution has been obtained, that solution is then used to help construct the upper level linear system. Each macromodel provides a matrix stamp that is equal in size to the number of terminal connections, and each entry in that stamp is a conductance term, obtained by solving the following equation:

$$J_{mn} = \frac{\partial f_m}{\partial x_i} = \frac{\partial \hat{f}_m}{\partial x_i} + \frac{\partial f_m}{\partial x_l}\frac{\partial x_l}{\partial x_i} \tag{4}$$

where m is the row index and n is the column index. From the perspective of the top level circuit Jacobian, m corresponds to the KCL equation associated with the circuit node attached to the mth electrode of the lower problem, and n corresponds to the voltage variable for the circuit node attached to the nth electrode. The first term $\hat{f}_m/\partial x_i$ in equation (4) corresponds to the inner problem contribution to the circuit node equation on the matrix diagonal, and is zero if $m \neq n$.

The second term in equation (4) is the product of two vectors, $\partial f_m/\partial x_l$ and $\partial x_l/\partial x_i$. The vector $\partial f_m/\partial x_l$ of the second term of equation (4) corresponds to a sparse row of the lower problem Jacobian, or the derivatives of the terminal KCL equations with respect to the internal variables of the lower problem. The vector $\partial x_l/\partial x_i$ is determined from solving the linear system:

$$J_l \frac{\partial x_l}{\partial x_i} = -\frac{\partial f_l}{\partial x_i} \tag{5}$$

where x_i are interface variables, corresponding to nodal voltages at the terminals to the lower problem, J_l is the Jacobian of the lower level problem, and x_l are lower problem variables. The right hand side of equation (5), $\partial f_l/\partial x_i$ is a sparse vector that corresponds to a column of the full system Jacobian for the derivatives of lower system variables with respect to interface variables.

4.2 Preserving Singleton Removal

As mentioned in section 3.4.1, singleton removal can fail when highly connected nodes are not connected directly to ideal voltage sources. One example of this is illustrated in Fig. 7, in which a parasitic network is attached to both VDD and VSS. Singleton removal depends upon the values being pre-determined and in the example this will no longer be the case. As a result, both linear solution strategies illustrated in Fig. 5 would fail. Additionally, even if singleton removal did not fail, the additional structure created by the power node parasitic network is sufficient to destroy the triangular structure required by the BTF linear solver strategy.

Applying the multi-level Newton method can mitigate the graph problem presented by non-ideal power supplies. The power supply parasitic network on the left side of Fig. 7 is partitioned into one level, while the remaining integrated circuit on the right side is partitioned into another level. From a conceptual standpoint, either partition can be the top or bottom level, but it is more computationally efficient to put the parasitic network on the bottom level as it is (in this example) a relatively small circuit, while the integrated circuit comprises the bulk of the original prob-

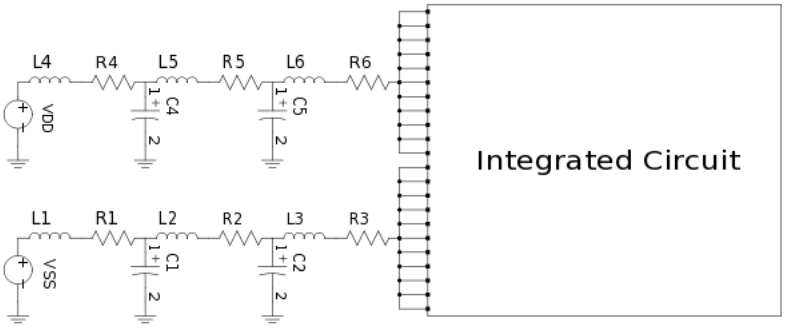

Fig. 7 Power node parasitics example. An RLC network sits between the VDD, VSS sources and the main circuit, so these highly connected nodes cannot be removed with a singleton filter.

lem. By using two-level partitioning, the values of VDD and VSS appear as ideal sources within the partition, and thus these matrix processing steps are preserved. A successful example of this approach is given in section 7.

5 Software

Xyce [4] relies heavily on the open-source Trilinos scientific computing library [23], which was designed for parallel computing. Trilinos provides support for linear algebra data structures as well as preconditioners, linear solvers and nonlinear solvers. Xyce is not directly written to the Trilinos packages and, instead, has an abstract interface that defines the functionality necessary for performing circuit simulation. These software abstractions allow Xyce to use MPI with double precision arithmetic through Epetra or combine and adapt to other parallel paradigms.

Xyce is designed to use abstract interfaces wherever feasible for algorithmic components, so that the implementation of those components may be separated from the implementation of the simulator. Many benefits result from such a decision. This decoupling facilitates code reuse across Xyce and increases algorithmic flexibility. As a result, constituent mechanisms (e.g., nonlinear solvers, linear solvers, preconditioners) can be chosen at runtime. This enables Xyce to be a production simulator, as well as a testbed for parallel algorithm research.

Xyce uses abstract interfaces and runtime polymorphism throughout the simulation code. Much of the higher-level abstractions, relating to the analysis type or time integration methods, have implementations that are contained in Xyce. However, the lower-level abstractions, relating to nonlinear solvers, linear solvers, and

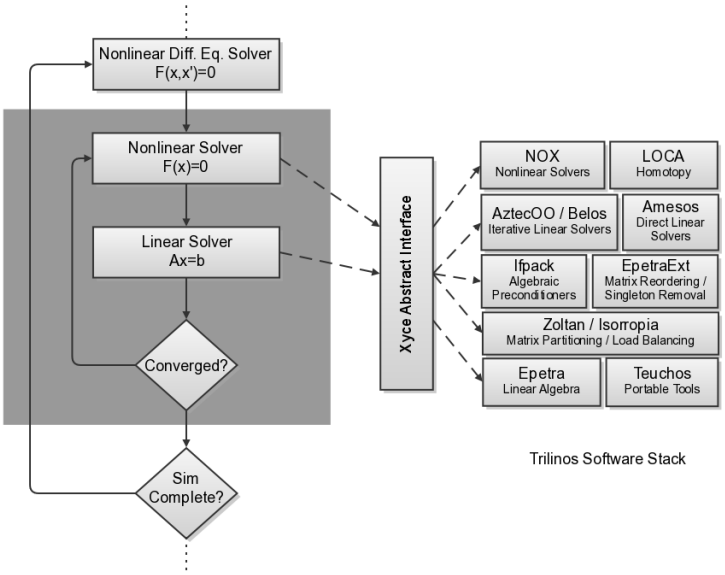

Fig. 8 Xyce simulation flow with interface to Trilinos

basic linear algebra, have interfaces to Trilinos packages. Fig. 8 shows the nested solver loop from the circuit simulation flow diagram (Fig. 1), where the shaded box illustrates the part of Xyce where Trilinos is utilized.

These software abstractions allow Xyce to adapt to future parallel paradigms and arithmetic precision strategies. By default, Xyce uses MPI with double precision arithmetic through Epetra. However, future computational platforms may require the use of other parallel paradigms, such as TBB [5] and CUDA [2], to achieve optimal performance. Furthermore, to address ill-conditioned matrices, it may prove useful to use quad-precision arithmetic in the linear solvers.

These forward-looking computational strategies are the motivation for the newer Trilinos linear algebra packages: templated Petra (Tpetra) and Kokkos. Tpetra provides a templated interface to parallel linear algebra and Kokkos contains the underlying computational kernels enabling platform-dependent optimizations. Several pre-existing Trilinos packages can use TPetra, like NOX, LOCA, Belos, and Teuchos, and many other packages are under development to provide direct solvers and preconditioners using Tpetra.

6 Parallel Linear Solver Strategy Comparison

In this section, Xyce scalability experiments for several different circuits using various linear solvers are presented. The first comparison is an abbreviated version of the results presented in [16], comparing the domain decomposition (DD) and block triangular form (BTF) linear solver strategies with two other direct linear solvers: KLU and SuperLU_DIST (SLUD). KLU is a serial sparse direct linear solver developed specifically for circuit simulation [31]. SLUD is a general purpose (not circuit-specific) distributed memory sparse direct linear solver [22]. The second comparison illustrates the scalability of the two linear solver strategies over an increasing number of processors for the largest test circuit.

Since the goal of developing efficient, scalable linear solvers is to extend the simulation capability to very large ICs, test circuits were chosen that produced large (more than 10^4 unknowns) linear systems. Table 1 partially describes the these circuits, where ckt4 is the chip2 circuit from the well known test suite CircuitSim90 [7] and the others are proprietary integrated circuits.

All computations are performed on a cluster with 2.2 GHz AMD four-socket, quad-core processors with 32 GB DDR2 RAM and an Infiniband interconnect using the OFED software stack. Each node of the machine has a total of sixteen cores, and the user can request anywhere from one to sixteen cores per node. If less than sixteen cores per node are used, the memory is evenly divided between the cores, and more memory is available for each core. For the first comparison, two parallel configurations were considered, 4 cores and 16 cores, on one compute node. For the second comparison, the parallel configurations vary the number of processors per node (ppn) used as well as the number of compute nodes.

Table 1 Circuits: matrix size(N), capacitors(C), MOSFETs(M), resistors(R), voltage sources(V), diodes (D).

Circuit	N	C	M	R	V	D
ckt1	688838	93	222481	176	75	291761
ckt2	434749	161408	61054	276676	12	49986
ckt3	116247	52552	69085	76079	137	0
ckt4	46850	21548	18816	0	21	0
ckt5	25187	0	71097	0	264	0

6.1 Explanation of Tables

For each circuit, results are presented for setup, device evaluation, solve and total times. The "setup time" refers to the time required to read the netlist, setup the circuit topology, allocate devices, resolve parameters, and create the linear system structure. The "device evaluation time" refers to the total time required to compute the Jacobian matrix and residual entries for all the individual devices, and to sum them into the parallel linear algebra structures, for every Newton iteration of every time step. The "solve time" is the total linear solve time for the entire transient simulation. The "total" time is simply the total time for the entire simulation, including setup, device evaluation, and solve times.

In tables 2 and 3, there are a number of entries labeled F_1, F_2, and F_3 representing different failure modes for the tested solvers. The F_1 failure is unique to the BTF linear solver strategy, stemming from the block triangular form having one large irreducible block that makes it impossible to find a suitable partition. The F_2 failure results from Newton's method failing to converge, which ultimately leads to Xyce exiting with a time-step-too-small failure. The F_2 failure can happen with either iterative or direct solvers, but mostly happens with the iterative solvers when the preconditioner is ineffective. The F_3 failure mode denotes circuits that run out of memory, suggesting that the memory requirements of the solver are too high.

6.2 Numerical Results

KLU reliably solves all the test circuits, and is considered the baseline for both comparisons. Table 2 shows the simulation time for three of the five circuits, where the simulations using KLU are run on one core and the others are run on four cores. The BTF linear solver strategy successfully solves all three test circuits and the total simulation time has been reduced relative to serial KLU. The speedups are due to a combination of multiple cores and an improved algorithm, and can thus be superlinear. The DD linear solver strategy successfully solves two of the three test circuits, but is only faster than KLU on ckt5. Using the SLUD solver, the total simulation time is faster than KLU in two of the three test circuits.

Table 3 shows the simulation time for the three largest circuits on sixteen cores. The BTF linear solver strategy, when successful, achieves a substantial speedup and even superlinear speedup for ckt3. However the failure to solve ckt2 is due to its bidirectional structure, which results in a large irreducible block. The DD linear

Table 2 Simulation times in seconds for four cores on a single node

Circuit	Task	KLU (serial)	SLUD	DD	BTF	**Speedup** (KLU/BTF)
ckt3	Setup	131	56	F_2	57	2.3x
	Device Eval	741	568	F_2	562	1.3x
	Solve	6699	2230	F_2	255	26.2x
	Total	7983	2903	F_2	923	8.6x
ckt4	Setup	15	8	8	13	1.2x
	Device Eval	1189	330	264	265	4.5x
	Solve	536	2540	1746	312	1.7x
	Total	1858	2916	2050	619	3.0x
ckt5	Setup	57	21	21	22	2.6x
	Device Eval	801	219	218	221	3.6x
	Solve	346	318	280	67	5.2x
	Total	1360	606	567	360	3.8x

Table 3 Simulation times in seconds for sixteen cores on a single node

Circuit	Task	KLU (serial)	SLUD	DD	BTF	**Speedup** (KLU/BTF)
ckt1	Setup	2396	F_3	207	199	12.0x
	Device Eval	2063	F_3	194	180	11.4x
	Solve	1674	F_3	3573	310	5.4x
	Total	6308	F_3	4001	717	8.8x
ckt2	Setup	2676	F_2	F_2	F_1	
	Device Eval	1247	F_2	F_2	F_1	
	Solve	1273	F_2	F_2	F_1	
	Total	5412	F_2	F_2	F_1	
ckt3	Setup	131	29	F_2	29	4.5x
	Device Eval	741	181	F_2	175	4.2x
	Solve	6699	1271	F_2	84	79.8x
	Total	7983	1470	F_2	306	26.1x

solver strategy does poorly on these larger test circuits, producing convergence failures in all but ckt1. SLUD also has difficulties, running out of memory on ckt1 and causing a convergence failure on ckt2.

The second comparison illustrates the scalability of the two linear solver strategies for ckt1. This comparison also examines the performance of loading values into the Jacobian matrix and the residual vector, which includes the device evaluation. The scaling is done relative to the serial simulation performance, where KLU is used as the linear solver. The simulations were run on 8, 16, 32, and 64 processors (cores) where 4 processors per node (ppn) were used. Thus 2, 4, 8, and 16 nodes were used to perform this study. The plot in Fig. 9 illustrates that the scaling of the Jacobian and residual load are about the same, as one would expect. However, the BTF linear solver strategy is almost twice as fast as the DD approach.

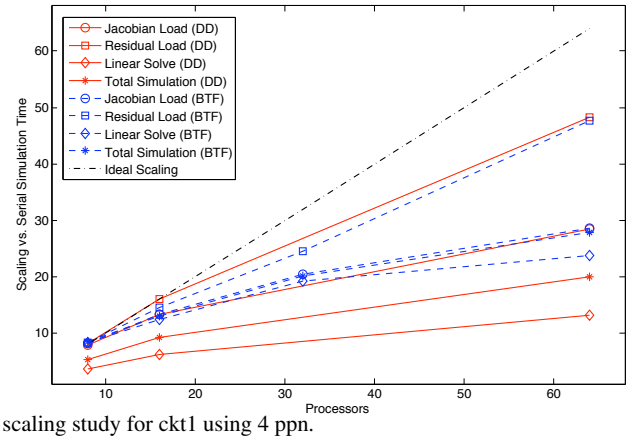

Fig. 9 Xyce scaling study for ckt1 using 4 ppn.

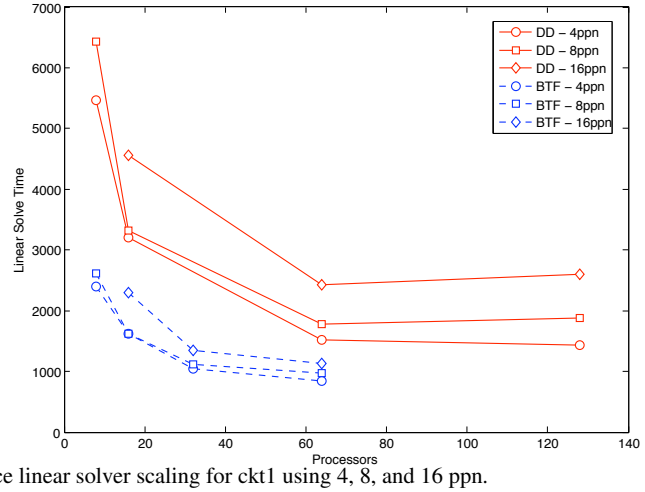

Fig. 10 Xyce linear solver scaling for ckt1 using 4, 8, and 16 ppn.

Relatedly, the BTF approach is a better choice than DD with respect to overall parallel scaling. When the DD strategy is used, the total simulation scaling falls midway between the Jacobian load and linear solve scalings. This indicates that the linear solve is a bottleneck to overall parallel performance, representing a larger fraction of the total runtime. However, with the BTF linear solver strategy, the total scaling is consistent with the Jacobian load scaling, demonstrating that the BTF-based linear solve does not impact the overall simulation scaling.

Fig. 10 shows the total linear solve time for ckt1 over increasing numbers of processors, for three values of processors per node. This plot verifies the performance difference between the BTF and DD linear strategies, independent of the number of processors per node. Furthermore, it illustrates that increasing the number of processors past 32 is not likely to speed up the simulation. For any fixed problem size, this roll-off is to be expected beyond a certain number of processors. However, it should be noted that the overall runtime on 32 processors is approximately twenty times faster than the serial case, which is still a substantial improvement.

7 Graph Mitigation Example

All the results given in the previous section are for circuits which do not have a parasitic network attached to highly connected nodes such as VDD and VSS (power and ground). However, if such a network were to be attached, both the DD and BTF solver strategies would fail. Here, an example is given, in which a power network similar to the one shown in Fig. 7 has been added to ckt3.

The initial consequences of adding the parasitic network are given in Fig. 11. In these figures, the DCOP matrix structure from the modified ckt3 is given, before and after the BTF reordering phase of the BTF solver strategy. The structure has not changed significantly, and the reordered matrix has a single irreducible block. As a result, it is not possible to achieve an effective block-wise load balance in parallel, as the irreducible block will be confined to a single processor.

The graph problems presented by the parasitic network can be mitigated by using the multi-level Newton approach described in section 4. The parasitic network (corresponding to the left side of Fig. 7) in this example is considered the upper level of the problem, while the rest of the integrated circuit is considered to be the lower level. The graph problems are thus confined to the upper level and invisible to the lower. The result of this strategy is given in Fig. 11.

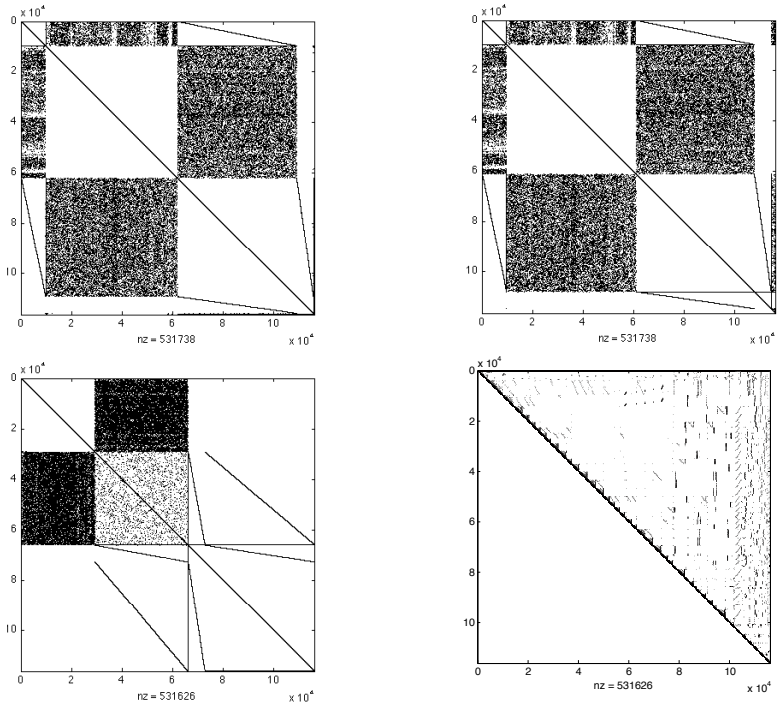

Fig. 11 Matrix pattern for ckt3 with power node parasitics attached before (top, left) and after (top, right) BTF reordering. After the circuit has been partitioned into a multi-level Newton solve, the matrix pattern for the inner solve before (bottom, left) and after (bottom, right) BTF reordering.

8 Conclusion

Parallel circuit simulation requires integration of large and small scale parallelism throughout the entire circuit simulation flow. In this paper, parallel algorithms for circuit simulation, and their implementation in a production simulator, Xyce, have been discussed. Specific attention was given to parallelism issues in the netlist parser, nonlinear solver, and linear solver.

A key design aspect for Xyce is the development of new circuit-specific preconditioners for iterative linear solvers. While not as robust as direct solvers, iterative solver strategies have the potential to enable scalable parallel simulation. The results presented show that the BTF linear solver strategy reduces the total simulation time by up to a factor of twenty compared to the serial solver KLU on 32 processors. However, the strategy only works well when the conductance matrix is reducible, which is not true for some circuits for a variety of different reasons. One graph mitigation strategy, multi-level Newton, has been successful at partitioning an irreducible graph to a reducible one. Future work will include a focus on developing other strategies for graph mitigation. Ultimately, a general parallel tool will require a comprehensive strategy to obtain good parallel performance.

9 Acknowledgments

The authors would like to thank Mike Heroux, Ray Tuminaro, David Day, Erik Boman, and Scott Hutchinson for many helpful discussions.

References

1. *Cadence UltraSim*. `http://www.cadence.com/products/cic/UltraSim_fullchip/`.
2. *NVIDIA CUDA programming guide*. `http://www.nvidia.com/object/cuda.html`.
3. *Synopsys HSIM*. `http://www.synopsys.com/Tools/Verification/AMSVerification/CircuitSimulation/HSIM/`.
4. *Xyce Parallel Circuit Simulator*. `http://xyce.sandia.gov`.
5. *Intel Threading Building Blocks 2.0*. `http://www.intel.com/software/products/tbb/`, March 2008.
6. P. Amestoy, T.Davis, and I.Duff. An approximate minimum degree ordering algorithm. *SIAM J. Matrix Anal. Appl.*, 17(4):886–905, 1996.
7. J. Barby and R. Guindi. Circuitsim93: A circuit simulator benchmarking methodology case study. In *Proceedings Sixth Annual IEEE International ASIC Conference and Exhibit*, Rochester NY, 1993.
8. A. Basermann, U. Jaekel, M. Nordhausen, and K. Hachiya. Parallel iterative solvers for sparse linear systems in circuit simulation, Jan 2005.
9. C. Bomhof and H. vanderVorst. A parallel linear system solver for circuit simulation problems. *Num. Lin. Alg. Appl.*, 7(7-8):649–665, 2000.
10. T. A. Davis. *Direct Methods for Sparse Linear System*. SIAM, 2006.
11. D. M. Day, M. K. Bhardwaj, G. M. Reese, and J. S. Peery. Mechanism free domain decomposition. *Comput. Meth. Appl. Mech. Engrg.*, 182(7):763–776, 2005.

12. K. Devine, E. Boman, R. Heaphy, R. Bisseling, and U. Catalyurek. Parallel hypergraph partitioning for scientific computing. In *Proc. of 20th International Parallel and Distributed Processing Symposium (IPDPS'06)*. IEEE, 2006.
13. N. Fröhlich, B. Riess, U. Wever, and Q. Zheng. A new approach for parallel simulation of vlsi-circuits on a transistor level. *IEEE Transactions on Circuits and Systems Part I*, 45(6):601–613, 1998.
14. W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, Sept. 1996.
15. K. Gulati, J. F. Croix, S. P. Khatr, and R. Shastry. Fast circuit simulation on graphics processing units. In *Proceedings of the 2009 Conference on Asia and South Pacific Design Automation*, pages 403–408. IEEE Press, 2009.
16. Heidi K. Thornquist et al. A parallel preconditioning strategy for efficient transistor–level circuit simulation. In *Proceedings of the 2009 International Conference on Computer–Aided Design*. ACM, 2009.
17. B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. In *Proc. Supercomputing '95*. ACM, December 1995.
18. G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. Multilevel hypergraph partitioning: Applications in VLSI domain. *IEEE Trans. VLSI Systems*, 20(1), 1999.
19. G. Karypis and V. Kumar. ParMETIS: Parallel graph partitioning and sparse matrix ordering library. Technical Report 97-060, CS Dept., Univ. Minn., 1997.
20. B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 49(2):291–307, 1970.
21. K. Kundert. Sparse matrix techniques. In A. Ruehli, editor, *Circuit Analysis, Simulation and Design*. North - Holland, 1987.
22. X. S. Li and J. W. Demmel. SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Trans. Math. Softw.*, 29(2):110–140, 2003.
23. M. Heroux, et. al. An overview of the Trilinos project. *ACM TOMS*, 31(3):397–423, 2005. `http://trilinos.sandia.gov`.
24. F. Manne and R. H. Bisseling. A parallel approximation algorithm for the weighted maximum matching problem. In *Proc. of PMAA'08, LNCS 4967*, pages 708–717. Springer, Berlin, 2008.
25. W. McLendon, B. Hendrickson, S. Plimpton, and L. Rauchwerger. Finding strongly connected components in distributed graphs. *J. of Parallel and Distributed Computing*, 65:901–910, 2005.
26. L. W. Nagel. Spice 2, a computer program to simulate semiconductor circuits. Technical Report Memorandum ERL-M250, 1975.
27. A. R. Newton and A. L. Sangiovanni-Vincentelli. Relaxation based electrical simulation. *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, CAD-3(4):308–330, October 1984.
28. H. Peng and C.-K. Cheng. Parallel transistor level circuit simulation using domain decomposition methods. *ASP-DAC '09: Proceedings of the 2009 Conference on Asia and South Pacific Design Automation*, pages 397–402, Jan 2009.
29. N. B. G. Rabbat, A. L. Sangiovanni-Vincentelli, and H. Y. Hsieh. A multilevel newton algorithm with macromodeling and latency for the analysis of large-scale nonlinear circuits in the time domain. *IEEE Trans. Circuits Syst.*, CAS-26(9):733741, 1979.
30. Y. Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, Philadelphia, PA., second edition, 2003.
31. K. Stanley and T. Davis. KLU: a 'Clark Kent' sparse LU factorization algorithm for circuit matrices. In *SIAM Conference on Parallel Processing for Scientfic Computing (PP04)*, 2004.
32. K. Sun, Q. Zhou, K. Mohanram, and D. C. Sorensen. Parallel domain decomposition for simulation of large-scale power grids. *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 54–59, November 2007.
33. B. Ucar and C. Aykanat. Partitioning sparse matrices for parallel preconditioned iterative methods. *SIAM J. Sci. Comput.*, 29(4):1683–1709, 2007.
34. J. White and A. L. Sangiovanni-Vincentelli. Relax2: A new waveform relaxation approach for the analysis of lsi mos circuits. *in Proc. Int. Symp. Circuits Syst.*, 2:756–759, May 1983.