Linear circuit analysis based on parallel asynchronous fixed-point method

Manuel Marin, David Defour, Federico Milano

Linear circuit analysis based on parallel asynchronous fixed-point method

Manuel Marin, *Student Member, IEEE*, David Defour, and Federico Milano, *Senior Member, IEEE*

***Abstract*—Time-domain circuit analysis and simulation is commonly performed through intrinsically serial methods, such as those implemented in the popular software package SPICE. However, the practical utilization of these methods is limited to circuits of a certain size, as for larger problems the simulation time becomes prohibitive. Parallelization of such serial routines has been proposed as the main alternative to accelerate the analysis and overcome the problem, even if the speed-up that can be achieved by this strategy is bounded, according to the Amdahl's law. Currently, there is a lack of intrinsically parallel methods which would allow to detach the efficacy of the solution from the problem size. In this article, we develop a new theoretical approach to circuit analysis from an intrinsically parallel point of view. We propose a fixed-point method and determine its convergence condition according to the theory of asynchronous iterations. We also perform a series of tests that show that our method is faster in most cases than those based on the traditional intrinsically serial approach. In particular, we obtain an empirical proof that our approach is independent of the problem size, offering great opportunities for scalability.**

***Index Terms*—Linear circuits, time-domain analysis, fixed-point arithmetic, parallel algorithms, iterative method.**

I. Introduction

CIRCUIT simulation has become an extremely important part in integrated circuit evaluation and design and a continuously growing field [1], [2], especially after the introduction of the SPICE simulator [3]. Meanwhile, as the number of components in modern applications continues to expand, there is a constant necessity to analyze larger and larger circuits, eventually without increasing the simulation time [4], [5]. Most attempts to achieve such goal have consisted of taking one of the existing methods, which are intrinsically serial, and parallelize part of it in order to obtain a speed-up [6]–[11]. Nevertheless, in doing this, there is always a fraction of the algorithm that remains sequential, which according to the Amdahl's Law determines that the speedup to be achieved is fairly limited [12]. This would not be the case of an algorithm that is intrinsically parallel; however, the task of developing such algorithm requires to reformulate the way in which we analyze circuits. In this article, we present a novel approach to accelerate circuit analysis and simulation which instead of parallelizing existing, intrinsically serial routines, proposes a new, intrinsically parallel method allowing to detach the complexity of the solution from the problem size. The proposed model attempts to change circuit simulation paradigm and is inspired from the circuit component dynamics that can be observed in real life.

The transient response of a circuit is typically obtained as a series of system states computed for several time instants. Computing one state involves the resolution of the circuit equations evaluated at that point. In SPICE, this is done in two sequential steps, *load* and *solve*. During the load step, the circuit equations are obtained from evaluating component models; during the solve phase, a linear system is solved. As component models are generally non-linear, the resolution of the system equations usually involves the repetition of the two steps in an iterative procedure, such as the Newton-Raphson method. In some applications, the load phase takes up to 75% of the total time, although this ratio tends to decrease with the circuit size.

The load phase is easy parallelizable: it is done by distributing the evaluation of several independent components among independent processors. Some successful examples of this, involving Field-Programmable Gate Arrays (FPGA) and Graphic Processing Units (GPU) are presented in [6], [7]. The solve phase, on the other hand, requires a more delicate approach. The original version of SPICE uses sparse matrix routines to solve the linear system, and PARASPICE is one of the first attempts of parallelizing such routines [8]. More recently, in [9], the same approach is revisited using the KLU algorithm [13] and FPGA. Other researchers have proposed relaxation techniques (e.g., Gauss-Seidel, Gauss-Jacobi) as an alternative, more parallel-friendly approach to matrix factorization. A survey of those techniques, including timing simulation, iterative timing simulation and waveform relaxation is done in [10]. In [11], a novel approach consisting of decomposing the domain and distributing it into several parallel processors is presented. However, at the end, sequentiality persists in all these models, thus, limiting the maximum speed-up according to Amdahl's Law.

The approach that we develop in this article attempts to avoid sequentiality by mimicking the actual behaviour of an electrical circuit, where each component evolves in a continuous cycle, permanently reevaluating its state regardless of external factors. To efficiently model such a process, we apply the concept of *team algorithms*, developed in [14]. In this concept, the value of a critical unknown is computed through running several different algorithms and forming convex combinations of the results; the obtained value is used as starting guess in a new cycle, until all the algorithms reach an agreement on the unknown's value. An analog idea is presented in [15] under the name of *agreement algorithm*.

M. Marin is with Université de Perpignan Via Domitia, DALI and Université Montpellier 2, LIRMM, France, and also with the School of Electrical, Electronic and Communications Engineering of the University College Dublin, Dublin, Ireland (e-mail: manuel.marin@univ-perp.fr).

D. Defour is with Université de Perpignan Via Domitia, DALI and Université Montpellier 2, LIRMM, France (e-mail: david.defour@univ-perp.fr).

F. Milano is with the School of Electrical, Electronic and Communications Engineering of the University College Dublin, Dublin, Ireland (e-mail: federico.milano@ucd.ie).

Similarly, in the real operation of an electrical circuit, we have a number of electrical components connected through a common node; the components impose their dynamics on the node, affecting and modifying the node's voltage, until it becomes stable after a certain number of interactions. We can argue that this kind of physical phenomena that occur in real life are intrinsically parallel.

This paper intends to bring the following main contributions to the existing literature:

1) It proposes a novel method for time-domain transient circuit simulation, which is based on an intrinsically parallel iteration and so is unaffected by the Amdahl's argument. A proof of convergence of the iterative method is given for the case of linear component models.
2) It presents an empirical comparison of the proposed method with traditional approaches based on matrix factorization. In particular, it is shown that the number of iterations performed by our method is independent from the problem size, allowing performance to scale with the amount of computing units available.

The remaining of the article is organized as follows. In Section II, we offer a discussion on several alternatives to implement an iterative process on a parallel computer, which serves as theoretical background for the following sections. In Section III, we present our model of intrinsically parallel circuit simulator and our convergence result. Implementation details and the results of a series of tests, along with comparisons with existing circuit simulators based on the traditional, serial approach, are given in Section IV. Finally, Section V concludes the paper.

II. Background

Usually, iterative processes are implemented on a parallel computer by assigning each component of the system state vector to an independent processor. From that point on, different synchronization schemes lead to different types of iteration. The first one that we want to revisit is the *synchronous iteration*, defined as follows.

Definition 1 (Synchronous iteration)**.** Let $\boldsymbol{x} \in \mathbb{R}^n$ and $\boldsymbol{f}\colon \mathbb{R}^n \to \mathbb{R}^n$. Given an initial value $\boldsymbol{x}^0$, the series defined by:

$$\boldsymbol{x}^{k+1} = \boldsymbol{f}(\boldsymbol{x}^k), \quad k \in \mathbb{N}, \tag{1}$$

is termed a synchronous iteration associated to $F(\cdot)$.

In this model the vector is updated as a block, i.e., all the components are updated before the process moves to a next iteration. In order to implement it, a synchronization step is needed at the end of each cycle. In this synchronization step, processors that have finished their update wait for the others to catch up. The idle time will depend on a series of factors, e.g., load balancing, processor frequency, communication network and memory access patterns [16].

An alternative to the synchronous model is the so-called *asynchronous iteration*, in which individual processors are always allowed to work, independently of the state of others. Typically, one processor will update the component assigned to it using the most recent data available, then communicate its action to other processors and restart the cycle, without waiting. In consequence, only certain vector components are updated at each asynchronous iteration, and some of the components used in these updates have values that do not correspond to the very last iteration. In order to specify the model, we need to introduce the *update function* and the *delay function*.

The update function, noted $\mathcal{U}(\cdot)$, receives the iteration counter $k \in \mathbb{N}$, and returns a subset of $\{1, \ldots, n\}$ indicating the list of processors that will update their components during iteration k. The delay function, noted $d(\cdot)$, receives the indices $i, j \in \{1, \ldots, n\}$ and the iteration counter $k \in \mathbb{N}$, and returns the delay of processor j with respect to processor i at iteration k.

Definition 2 (Asynchronous iteration)**.** Let $\boldsymbol{x} \in \mathbb{R}^n$ and $\boldsymbol{f}\colon \mathbb{R}^n \to \mathbb{R}^n$. For $k \in \mathbb{N}, i, j \in \{1, \ldots, n\}$, let $\mathcal{U}(k) \subseteq \{1, \ldots, n\}$ and $d(i, j, k) \in \mathbb{N}_0$, such that:

$$d(i, j, k) \geq 0, \qquad \forall i, j, k, \tag{2a}$$

$$\lim_{k \to \infty} d(i, j, k) < \infty, \qquad \forall i, j, \tag{2b}$$

$$|\{k\colon i \in \mathcal{U}(k)\}| = \infty, \qquad \forall i. \tag{2c}$$

Given an initial value $\boldsymbol{x}^0$, the series defined by:

$$x_i^{k+1} = \begin{cases} f_i(x_1^{k-d(i,1,k)}, \ldots, x_n^{k-d(i,n,k)}) & \text{if } i \in \mathcal{U}(k), \\ x_i^k & \text{if } i \notin \mathcal{U}(k), \end{cases} \tag{3}$$

is termed an asynchronous iteration associated to $\boldsymbol{f}(\cdot)$, with update function $\mathcal{U}(\cdot)$ and delay function $d(\cdot)$.

The assumption in (2a) states that only values computed in previous iterations are used in any update. The one in (2b) states that newer values of the components are eventually used. Finally, the assumption in (2c) states that no component ceases to be updated during the course of the iteration. In the case of a shared memory machine, the two latter assumptions become equivalent, as there are no delay associated to the communications. Indeed, as soon as a component is updated, its new value becomes visible and available to all the processors. Also; note that a synchronous iteration is a special case of asynchronous iteration with $\mathcal{U}(k) = \{1, \ldots, n\}$ and $d(i, j, k) = 0$, for all i, j, k.

Regarding convergence of asynchronous iterations, several results have been published. In particular, when the application function $\boldsymbol{f}(\cdot)$ is linear we have the following theorem by Chazan and Miranker [17].

Theorem 1 (Sufficient condition for convergence in the asynchronous case [17])**.** *Let $\boldsymbol{x} \in \mathbb{R}^n$, and $\boldsymbol{f}\colon \mathbb{R}^n \to \mathbb{R}^n$ be a linear application, i.e.,*

$$\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{L}\boldsymbol{x} + \boldsymbol{b}, \quad \boldsymbol{L} \in \mathbb{R}^{n \times n}, \boldsymbol{b} \in \mathbb{R}^n. \tag{4}$$

Let $|\boldsymbol{L}|$ denote the matrix of absolute values of the entries of $\boldsymbol{L}$, and $\rho(|\boldsymbol{L}|)$ its spectral radius (see Appendix A for a definition of spectral radius). If

$$\rho(|\boldsymbol{L}|) < 1, \tag{5}$$

then the asynchronous iteration $\boldsymbol{x}^k, k \in \mathbb{N}$ associated to $\boldsymbol{f}$ converges to $\boldsymbol{x}^*$, the unique fixed point of $\boldsymbol{f}$, regardless of the

selection of $\mathcal{U}(\cdot)$, $d(\cdot)$ and $\boldsymbol{x}^0$. Furthermore, if $\rho(|\boldsymbol{L}|) \geq 1$, then there exists $\mathcal{U}(\cdot)$, $d(\cdot)$ and $\boldsymbol{x}^0$ such that $\boldsymbol{x}^k$ does not converge to $\boldsymbol{x}^*$.

In some cases, if the assumption in (5) is not met, one can still build a series that converges to a fixed point x^*. However this requires additional assumptions on the update and shift functions, leading to a new kind of iteration known as *partially asynchronous*.

Definition 3 (Partially asynchronous iteration)**.** Consider Definition 2 of an asynchronous iteration. Replace the assumptions in (2b) and (2c) by the following:

$$\exists \bar{d} \in \mathbb{N}\colon d(i,j,k) \leq \bar{d}, \qquad \forall i,j,k, \tag{6a}$$

$$\exists \bar{s} \in \mathbb{N}\colon i \in \bigcup_{s=1}^{\bar{s}} \mathcal{U}(k+s), \qquad \forall i,k. \tag{6b}$$

$$d(i,i,k) = 0, \qquad \forall i,k, \tag{6c}$$

The series x^k is now termed a partially asynchronous iteration.

The assumption in (6a) establishes that not only newer values of the components are eventually used, but each of these values is used before $\bar{d}$ iterations have passed from their calculation. The assumption in (6b), in turn, states that each component is updated at least once in every $\bar{s}$ consecutive iterations. These assumptions require to introduce a synchronization step every once in a while (for example, every $\bar{d}$ or $\bar{s}$ iterations). In the case of a shared memory machine, only one of the two is needed and $\bar{d} = \bar{s}$. The assumption in (6c) establishes that every component is updated using its last calculated value. In practical terms this is equivalent to having each component assigned to only one processor.

Now we can enunciate the following result, regarding convergence of partially asynchronous iterations.

Theorem 2 (Sufficient condition for convergence in the partially asynchronous case [18])**.** *Let $\boldsymbol{x} \in \mathbb{R}^n$, and $\boldsymbol{f}\colon \mathbb{R}^n \to \mathbb{R}^n$ be a linear application, i.e.,*

$$\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{L}\boldsymbol{x} + \boldsymbol{b}, \quad \boldsymbol{L} \in \mathbb{R}^{n\times n}, \boldsymbol{b} \in \mathbb{R}^n. \tag{7}$$

Also, let $\boldsymbol{g}\colon \mathbb{R}^n \to \mathbb{R}^n$, defined by:

$$\boldsymbol{g}(\boldsymbol{x}) = (1-\alpha)\boldsymbol{x} + \alpha(\boldsymbol{L}\boldsymbol{x} + \boldsymbol{b}), \tag{8}$$

where $\alpha \in \mathbb{R}$ and $0 < \alpha < 1$.

If $\boldsymbol{L} = (l_{ij})$ is irreducible (see Appendix A for a definition of irreducibility) and

$$\sum_{j=1}^{n} |l_{ij}| \leq 1, \quad \forall i, \tag{9}$$

then the partially *asynchronous iteration $\boldsymbol{x}^k, k \in \mathbb{N}$, associated to $\boldsymbol{g}(\cdot)$ converges to $\boldsymbol{x}^*$, fixed point of $\boldsymbol{f}(\cdot)$.*

In [19], Lubachevsky and Mitra presented a special case in which convergence occurs for $\alpha = 1$, at a linear (geometric) rate. They also showed that the average rate of convergence per iteration in the long-term, is low-bounded by an expression which combines the problem size n, the measurements of asynchronism $\bar{d}$ and $\bar{s}$, the entries of matrix $\boldsymbol{L}$ and the fixed point $\boldsymbol{x}^*$, as follows:

$$(1-\sigma^r)^{1/r}, \tag{10}$$

where

$$r = 1 + \bar{d} + (n-1)(\bar{s} + \bar{d}), \tag{11a}$$

$$\sigma = \min_{i,j}{}^{+} \left(\frac{x_i^* l_{ij}}{x_j^*} \right), \tag{11b}$$

and $\min^{+}(\cdot)$ refers to the minimum of the positive elements. This means that the error cannot be reduced by a factor greater than $(1-\sigma^r)^{1/r}$ in any average iteration. Note that as σ is always lower than 1, for increasing values of n, $\bar{d}$ and $\bar{s}$, the rate of convergence approaches 1, i.e., the convergence becomes sub-linear.

Assume that we are in the synchronous case, in which $\bar{d} = \bar{s} = 0$. Then from equation (11a) we have $r = 1$, so the expression in (10) becomes simply

$$1 - \sigma. \tag{12}$$

In other words the rate of convergence is unaffected by the size of the problem, n.

Now assume that we are in a shared memory environment, that only allows for the slightest desynchronization (1 iteration) between processors, i.e., $\bar{d} = \bar{s} = 1$. In this case we have $r = 2n$. Let's see how this affects the convergence rate: Figure 1 shows the bound on the convergence rate (lower is better) as a function of the problem size, for different values of the parameter σ. We observe that for very small problems ($n < 2$) the convergence is already sub-linear. Only for 'very good' values of σ (e.g., 0.9) the problem can grow, although very slightly, without the convergence rate being affected so much.

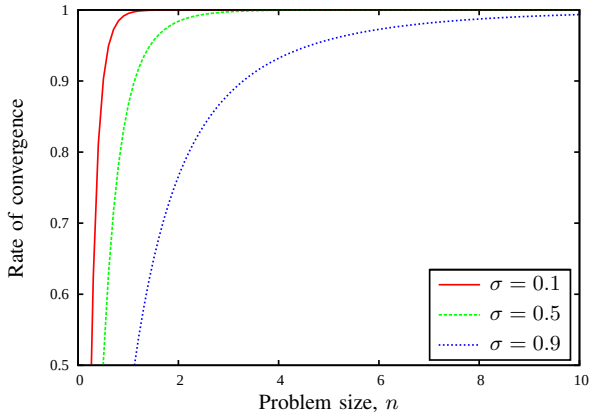


Fig. 1: Bound on the convergence rate as a function of the problem size, for different values of the parameter σ.

This may be a strong reason to prefer the synchronous approach over the partially asynchronous. However, depending on the implementation issues, the partially asynchronous strategy may become advantageous in certain cases, e.g., when the costs of synchronization are relatively high, or when the number of iterations needed to converge is relatively low.

III. Model description

Following the precedent discussion on parallel iterations we now introduce our model of an intrinsically parallel circuit simulator, through a top-down approach. Consider Table I as a synthesis of the variables and parameters of the model, which we introduce gradually.

TABLE I: List of variables and parameters of the model.

Symbol	Type	Definition
$\mathcal{N}$	Fixed	Set of circuit's nodes.
$\mathcal{C}$	Fixed	Set of circuit's components.
$\mathcal{I}_h$	Fixed	Set of influencers of node h.
v_h	Variable	Voltage at node h.
ϕ_h	Variable	Sum of current injections on node h.
ψ_{hk}	Variable	Voltage at node h according to the influencer k.
i_{hk}	Variable	Current flowing from node k to node h.
r_{hk}	Fixed	Resistance of the component situated between nodes h and k.
$\hat{v}_{hk}$	Fixed	Voltage source of the component situated between nodes h and k.
z_h	Fixed	Weight within node h of its own voltage.
w_{hk}	Fixed	Weight within node h of the voltage determined by the influencer k.

The general layout of a discrete-time circuit simulator is given in Algorithm 1. Instruction 5 is the crucial step, in which the system state is updated. Here is where our approach differs from SPICE and such classic techniques, in that the method applied to compute the state consists of an intrinsically parallel iteration. We will now describe this method and the structures that support it.

Algorithm 1 Discrete-time simulator.

Input: Circuit specification including initial state $\boldsymbol{v}_0$, time step length Δt and total time T.
Output: Circuit state $\boldsymbol{v}(t)$, at $t = 0, \Delta t, 2\Delta t, \ldots \leq T$.
1: $t \leftarrow 0$
2: $\boldsymbol{v}(t) \leftarrow \boldsymbol{v}_0$
3: **repeat**
4: $\quad t \leftarrow t + \Delta t$
5: $\quad$ calculate new $\boldsymbol{v}(t)$
6: **until** $t \geq T$

A. Circuit model

The circuit is represented as a set of nodes and components, $\mathcal{N}$ and $\mathcal{C}$, respectively. Each component is connected between two nodes, and each node has at least two components connected on it. The circuit is *well defined* if there are components connecting all the nodes in a closed chain.

For each non-ground node $n_h \in \mathcal{N}$, we define a set of *influencers* $\mathcal{I}_h \subset \mathcal{N}$, that groups all the nodes separated from n_h by exactly one component. Some authors call these the *fanin* nodes [10].

Consider the circuit in Fig. 2 as illustration. In this case $\mathcal{N} = \{n_A, n_B, n_C, n_D\}$ and $\mathcal{C} = \{c_0, c_1, c_2, c_3, c_4\}$. The circuit is well defined since c_0, c_1, c_3 and c_4 connect all the nodes in a closed loop. The set of influencers of n_B is $\mathcal{I}_B = \{n_A, n_C\}$, the set of influencers of n_C is $\mathcal{I}_C = \{n_A, n_B, n_D\}$ and the set of influencers of n_D is $\mathcal{I}_D = \{n_A, n_C\}$.

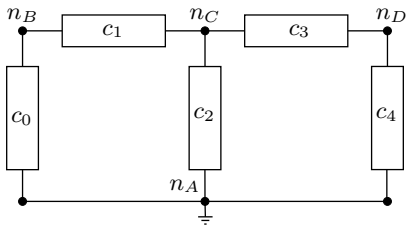


Fig. 2: Example circuit.

The iteration is performed in two steps. In the first step, every component reads, from each of its terminal nodes, the voltage v and the sum of current injections ϕ. Using these values it calculates a new voltage at each node, ψ, and a new current injected to each node, i. Finally it returns these values to the respective nodes. Figure 3 illustrates this procedure for c_1 in our example, which is connected between n_B and n_C.

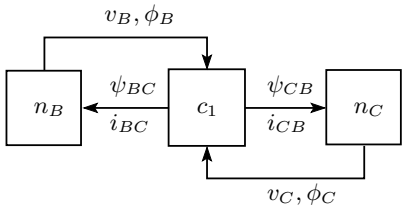


Fig. 3: Component-level procedure.

In the second step every node reads, from each one of the components connected to it, the voltage ψ and the current injection i. Using these values it computes the actual voltage v and the sum of current injections ϕ. Finally it returns these values to all those components to restart the cycle. Figure 4 illustrates this procedure for n_C in our example, whose set of influencers is $\mathcal{I}_C = \{n_A, n_B, n_D\}$.

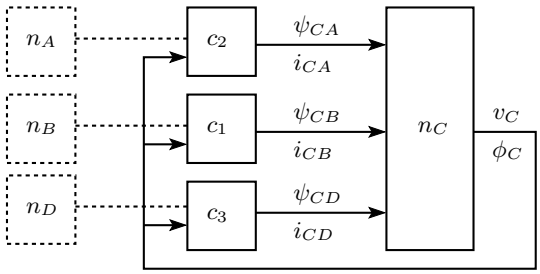


Fig. 4: Node-level procedure.

B. Component and node models

Now we will enter the boxes of components and nodes and describe how their inputs are converted into outputs.

1) Component: We represent all components as a voltage source in series with a resistance. This *universal component* model allows us to describe the behaviour of any of the following: voltage source, resistor, inductor and capacitor. The benefits of using a universal component model will be highlighted later in Section IV, when we discuss implementation issues.

Consider c_{hk}, a component connected between n_h and n_k. For this component, we define two parameters: $\hat{v}_{hk}$, corresponding to the voltage source, measured from n_h to n_k, and r_{hk}, corresponding to the resistance. Note that $\hat{v}_{hk} = -\hat{v}_{kh}$ and $r_{hk} = r_{kh}$. Next we show how the parameters $\hat{v}_{hk}$ and r_{hk} are obtained depending on the specific component type.

a) Voltage source: This component consists of an independent voltage source β of maximum current μ. Here, $\hat{v}_{hk}$ takes the value β, and the resistance r_{hk} is set to a value sufficiently small to ensure that the voltage drop across its two terminals is negligible, i.e., lower than a positive, small real number ϵ:

$$0 \leq r_{hk}\mu < \epsilon. \tag{13}$$

We achieve this by setting $r_{hk} = \frac{\epsilon}{2\mu}$.

b) Resistor: This component consists of a constant resistance ϱ. Here $\hat{v}_{hk}$ is set to zero and r_{hk} takes the value ϱ.

c) Inductor: This component consists of a constant inductance λ. From Dommel [20], we know that the dynamic behaviour of an inductance during a time interval $[t, t + \Delta t]$, can be emulated by an equivalent circuit of a voltage source in series with a resistance. This gives the values of r_{hk} and $\hat{v}_{hk}$ in this case:

$$r_{hk} = \frac{2\lambda}{\Delta t}, \tag{14a}$$

$$\hat{v}_{hk}(t) = v_k(t) - v_h(t) - r_{hk} i_{hk}(t), \tag{14b}$$

where t is the simulation clock.

d) Capacitor: A constant capacitance γ. We follow the same procedure as for the inductor. Now:

$$r_{hk} = \frac{\Delta t}{2\gamma}, \tag{15a}$$

$$\hat{v}_{hk}(t) = v_h(t) - v_k(t) + r_{hk} i_{hk}(t), \tag{15b}$$

where t is the simulation clock.

TABLE II: Model parameters per component type.

Component	$\boldsymbol{r_{hk}}$	$\boldsymbol{\hat{v}_{hk}}$
Voltage source	$\frac{\epsilon}{2\mu}$	β
Resistor	ϱ	0
Inductor	$\frac{2\lambda}{\Delta t}$	$v_k(t) - v_h(t) - r_{hk} i_{hk}(t)$
Capacitor	$\frac{\Delta t}{2\lambda}$	$v_h(t) - v_k(t) + r_{hk} i_{hk}(t)$

Table II synthesizes the expressions of $\hat{v}_{hk}$ and r_{hk} for each component type. Once these parameters are obtained, the outputs of c_{hk} are computed as follows:

$$\psi_{hk} = v_h + r_{hk}\phi_h, \tag{16a}$$

$$i_{hk} = \frac{v_k - v_h - \hat{v}_{hk}}{r_{hk}}. \tag{16b}$$

These equations correspond to the Ohm's Law and the Kirchhoff's Current Law applied on c_{hk}, n_h and n_k. (See Appendix B for the deduction of these equations.) Note that when the sum of current injections on n_h is equal to zero, i.e., $\phi_h = 0$, we have from equation (16a) $\psi_{hk} = v_h$. In other words, the voltage according to the influencer k is the same as the voltage according to the node itself. This corresponds to a condition of convergence at component-level.

2) Node: Consider now n_h and its set of influencers $\mathcal{I}_h$. For this node, we define positive real numbers z_h and $w_{hk}, k \in I_h$ such that

$$z_h + \sum_{k \in I_h} w_{hk} = 1. \tag{17}$$

These numbers correspond to the weights of a convex combination that will be used to update v_h. In Subsection III-C we show that there is a condition to respect when selecting these weights, in order to ensure convergence.

Once the weights are chosen, the outputs of n_h are determined as follows:

$$v'_h = z_h v_h + \sum_{k \in \mathcal{I}_h} w_{hk}\psi_{hk}, \tag{18a}$$

$$\phi_h = \sum_{k \in \mathcal{I}_h} i_{hk}. \tag{18b}$$

Equation (18a) states that a new value of the voltage is computed as a convex combination of its own value, available from the previous iteration, and the values according to all its influencers. Equation (18b) is self-explanatory.

Note that when all the influencers agree on the value of the voltage at n_h, i.e., $\psi_{hk} = v_h, \forall k \in \mathcal{I}_h$, we have from equation (18a) $v'_h = v_h$. This corresponds to a condition of convergence at node-level. When this condition is satisfied at all the nodes, the iteration converges at system-level. In the next subsection we will determine whether this situation is ever reached, and which assumptions need to be made in order to ensure convergence, regarding the theoretical background presented earlier in Section II.

C. Convergence study

Let n_G be the ground-node and $\mathcal{N}'$ the set of all nodes in the circuit minus the ground, i.e., $\mathcal{N}' = \mathcal{N} \setminus \{n_G\}$. Let $m = |\mathcal{N}'|$, and $\boldsymbol{v} \in \mathbb{R}^m$ be the vector of voltages at all non-ground nodes. We define the function $\boldsymbol{f} : \mathbb{R}^m \to \mathbb{R}^m$ as follows:

$$f_h(v) = v_h + \sum_{k \in \mathcal{I}_h} \sum_{j \in \mathcal{I}_h} w_{hk} r_{hk} \frac{v_j - v_h - \hat{v}_{hj}}{r_{hj}}, h \in \mathcal{N}'. \tag{19}$$

This expression is obtained by combining equations (16a), (16b), (17), (18a) and (18b) and defining $\boldsymbol{v}' = \boldsymbol{f}(\boldsymbol{v})$. Then, $\boldsymbol{f}(\cdot)$ becomes the iteration function. This is a linear application, i.e.,

$$\boldsymbol{f}(\boldsymbol{v}) = \boldsymbol{L}\boldsymbol{v} + \boldsymbol{b}, \tag{20}$$

where $\boldsymbol{L}$ and $\boldsymbol{b}$ are given by:

$$
\begin{aligned}
l_{hh} &= 1 - \sum_{k\in\mathcal{I}_h}\sum_{j\in\mathcal{I}_h}\frac{w_{hk}r_{hk}}{r_{hj}},\\
l_{hj} &= \begin{cases}\sum\limits_{k\in\mathcal{I}_h}\frac{w_{hk}r_{hk}}{r_{hj}} & \text{if } j\in\mathcal{I}_h,\\ 0 & \text{otherwise.}\end{cases}\\
b_h &= -\sum_{k\in\mathcal{I}_h}\sum_{j\in\mathcal{I}_h} w_{hk}r_{hk}\frac{\hat{v}_{hj}}{r_{hj}}.
\end{aligned}
\tag{21}
$$

The following theorem establishes that a totally asynchronous iteration associated to $\boldsymbol{f}(\cdot)$ is not certain to converge.

Theorem 3. *Let $\boldsymbol{v}^k, k\in\mathbb{N}$ be an asynchronous iteration associated to $\boldsymbol{f}(\cdot)$, defined in (20). Then, there exist an update function $\mathcal{U}(\cdot)$, a delay function $d(\cdot)$ and an initial guess $\boldsymbol{v}^0$ such that $\boldsymbol{v}^k$ does not converge to a fixed point.*

Proof. To apply Theorem 1 in Section II, we just need to prove that $\rho(|\boldsymbol{L}|)\geq 1$, where $\boldsymbol{L}$ is given by (21). Now, as the sum of the elements in any row of $\boldsymbol{L}$ is equal to 1, then 1 is an eigenvalue of $\boldsymbol{L}$ and $\rho(\boldsymbol{L})\geq 1$. And by applying Theorem 5 in Appendix A, $\rho(|\boldsymbol{L}|)\geq 1$. □

This means that to ensure convergence we need further assumptions. The following theorem establishes a sufficient condition for convergence.

Theorem 4. *Let $\boldsymbol{v}^k, k\in\mathbb{N}$ be a partially asynchronous iteration associated to $\boldsymbol{g}(\cdot)$ defined by:*

$$
\boldsymbol{g}(\boldsymbol{v}) = (1-\alpha)\boldsymbol{v} + \alpha(\boldsymbol{L}\boldsymbol{v}+\boldsymbol{b}),
\tag{22}
$$

where $0<\alpha<1$ and $\boldsymbol{L}$ and $\boldsymbol{b}$ are given by (21). Let the weights z_h and $w_{hk}, k\in I_h$ satisfy

$$
1 - \sum_{k\in\mathcal{I}_h}\sum_{j\in\mathcal{I}_h}\frac{w_{hk}r_{hk}}{r_{hj}} \geq 0, \quad \forall h.
\tag{23}
$$

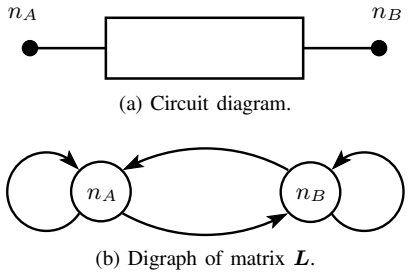
Then $\boldsymbol{v}^k$ converges to $\boldsymbol{v}^*$, fixed point of $\boldsymbol{f}(\cdot)$.

Proof. We will use Theorem 2 in Section II. Since we are assuming (23), we have $l_{hh}\geq 0$ (note that $l_{hk}\geq 0$ by definition) and then

$$
\sum_{k=1}^{n}|l_{hk}| = \sum_{k=1}^{n} l_{hk} = 1, \quad \forall h.
$$

So $\boldsymbol{L}$ verifies the assumption in (9).

It remains to prove that $\boldsymbol{L}$ is irreducible. According to Theorem 6 in Appendix A, we need to look at the digraph $\boldsymbol{D}(\boldsymbol{L})$ and see if it is strongly connected. This digraph can actually be obtained from the circuit diagram, as follows. For each node in the circuit, we place a vertex in $\boldsymbol{D}$. Next, for each component we place a pair of anti-parallel arcs, connecting the two vertices corresponding to its terminals. Finally, we place an arc from each vertex to itself. Figure 5 illustrates this transformation process. As result, the directed arcs going into each vertex in the graph identify that node's influencers. And as the circuit is well defined, this graph is strongly connected. □



(a) Circuit diagram.

(b) Digraph of matrix $\boldsymbol{L}$.

Fig. 5: Equivalence between the circuit diagram and the digraph of the application matrix $\boldsymbol{L}$. Some authors call this the *dependency graph* [10].

IV. Implementation and tests

Now that we have described our model, we can complement Algorithm 1 in Section III with the details of the proposed intrinsically parallel iteration. Let $\mathcal{C}'\subset\mathcal{C}$ be the set of all inductors and capacitors in the circuit. Algorithm 2 details the implementation of the method using synchronous iterations. The totally asynchronous version can be obtained by eliminating the instruction in line 8; the partially asynchronous, by replacing that instruction by "synchronize processors every $\bar{d}$ iterations."

Algorithm 2 Intrinsically parallel discrete-time circuit simulator, synchronous case.

Input: Circuit specification including set of nodes $\mathcal{N}$, components $\mathcal{C}$, sets of influencers $\mathcal{I}_{(\cdot)}$, node and component parameters and initial state $\boldsymbol{v}_0$; iteration function $\boldsymbol{f}(\cdot)$; time step length Δt and total time T.
Output: Circuit state $\boldsymbol{v}(t)$, at $t=0,\Delta t, 2\Delta t,\ldots\leq T$.

```
1:  t ← 0
2:  v(t) ← v_0
3:  repeat
4:      t ← t + Δt
5:      for all i ∈ N in parallel do
6:          repeat
7:              v_i ← f_i(v)
8:              synchronize processors
9:          until convergence
10:     end for
11:     save current state v(t)
12:     for all h ∈ C' in parallel do
13:         update v̂_h(t)
14:     end for
15: until t ≥ T
```

Note that in instruction 7 of Algorithm 2 all the processors update their coordinate of vector v, by evaluating the iteration function, $F()$. As we are using a universal component model, the arithmetic operations involved in this function evaluation are actually the same for all processors; only the data differs. This is an ideal situation for the implementation of the model on architectures of the class SIMD (Single Instruction Multiple Data) such as GPU, where such configuration allows for the most data-level parallelism to be exploited.

Next we will present different implementations of Algorithms 1 and 2 along with the results of some tests. We use GPU to implement and run our algorithm, and CPU to implement and run the algorithms from the classic approach. (For a detailed review of GPU architecture, see reference [21].)

Our computing environment is synthesized in Table III.

TABLE III: Computing environment used in the tests.

Hardware	Clock rate (GHz)	Number of cores
Xeon X560 CPU	2.67	12 (1 used)
GeForce GTX 560 GPU	1.62	336
GeForce GTX 480 GPU	1.40	480
GeForce GTX 680 GPU	1.06	1,536

Software	Version
GCC	4.8.2
CUDA	6.0
LAPACK	3.5.0
KLU	1.2.1

A. Implementation alternatives

1) Intrinsically parallel, synchronous (IPS): We first considered Algorithm 2 and implemented it in CUDA C++ [22], in order to run it on our Nvidia GPUs. Generally speaking, CUDA is a platform and language for General Purpose GPU programming (GPGPU) allowing to declare and specify *kernels*, which are executed concurrently by thousands of threads on the GPU. The threads are grouped into blocks and distributed to several CUDA cores, where they can make use of the several GPU resources (e.g., parallel floating-point units, different memory layers, etc.)

The IPS implementation revolves around two classes, *Component* and *Node*; both of them are implemented as "structure of arrays," so the program can benefice of coalesced memory accesses (i.e., adjacent GPU threads reading or writing consecutive locations in GPU memory), which is a standard lever for performance. The implementation has two kernels, *iterate()* and *update()*, for performing instructions 7 and 13 in Algorithm 2, respectively. These kernels operate at component-level, i.e., the program launches as many threads as components in the circuit and each thread performs the calculations associated to one component. In this way we have a constant number of operations (each component is linked to exactly two nodes), as opposed to a variable number (each node could be linked to any arbitrary number of components) per iteration. This improves the regularity of our solution which is another lever for performance.

Note that CUDA does not provide any explicit block-level mechanism of synchronization. This means that threads from different blocks can get out of phase when running a sequential loop. Therefore in order to ensure synchronization, we let the CPU control the loop in instructions 6 to 9. At every iteration of the loop, the CPU launches the *iterate()* kernel onto the GPU, so the GPU computes a new approximation of the bus voltages and evaluates the condition of convergence. The CPU waits for the kernel to finish before progressing into the next iteration. The synchronization step in line 8 is implicit in this model.

2) Intrinsically parallel, partially asynchronous (IPPA): Next we considered the modified version of Algorithm 2, in which synchronization between processors is only enforced after a given number of asynchronous iterations. This number is passed as a parameter to the kernel, so the GPU can iterate that many times before returning the control to the CPU. When this parameter is set to one, we obtain the IPS implementation, i.e., the program performs only one (asynchronous) iteration before synchronizing all processors.

3) Classic approach: Last, for fair comparison purposes, we implemented two versions of the classic circuit simulator (see Algorithm 1 in Section III) that relies on matrix factorization to compute the bus voltages. One of these versions utilizes the dense solver LAPACK [23]; the other one uses the optimized sparse solver KLU [13]. In both cases, the matrix is fully re-factorized at each time step to simulate the case in which there are nonlinearities in the component models.

Both implementations run on single-core CPU. However, it is worth noting that some parallel versions of LAPACK optimized for GPU can yield a speed-up of one order of magnitude [24]. In KLU, in turn, parallelization seems constrained by the Gilbert-Peierls phase of the algorithm [25].

B. Tests and results

Next, in order to evaluate our method (IPS and IPPA implementations) in comparison to the classic approach to circuit simulation (LAPACK and KLU implementations), we randomly generated test circuits with sizes from 500 to above 10,000 nodes. Then we asked our programs to simulate one second of real time operation of these circuits, with a time step of 5 milliseconds, and collected several performance figures. (By default, we use the GTX 680 GPU.)

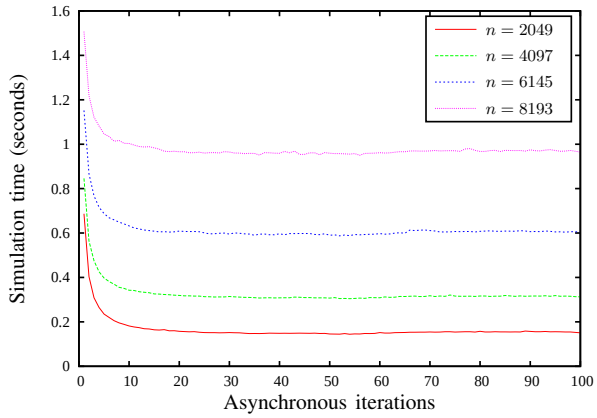
To generate the test circuits we developed a dedicated C++ program, which uses the Boost library implementation of the Mersenne twister [26] as a pseudo-random number generator. The strategy used in generating the circuits is the following: first, we generate a random circuit of 16 components that we call a *cluster*; second, we increase the number of nodes gradually by copying this cluster several times and linking random nodes between the copies; third and final, a random node in every cluster is linked to ground. The idea behind this method is to really isolate the effect of the problem size from all the other factors that could impact on performance (e.g., topology, numerical values, etc.) A similar approach for measuring the performance of the Spice simulator is presented in [10].

1) IPS v/s IPPA: Our first experiment consisted in measuring the effect of asynchronicity over performance, by comparing the IPS and IPPA implementations.

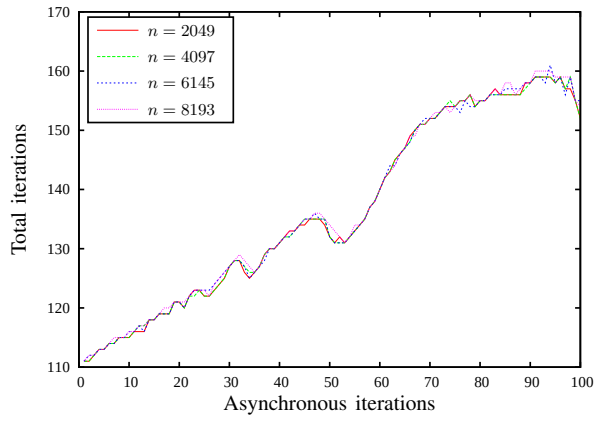
Figure 6(a) shows the execution time of IPS and IPPA on four of the test circuits. On the horizontal axis, we have the number of asynchronous iterations performed by the kernel in IPPA (the first point in these curves corresponds to the output of IPS, i.e., the case where the number of asynchronous

iterations is set to one). The different curves represent different values of the problem size, n. Clearly, there is a benefit in allowing asynchronism. Furthermore, in this experiment all the four test circuits behave in a very similar way: the simulation time decreases as the number of asynchronous iterations starts increasing, until a certain point after 20 iterations where the gains seem to stall.

Figure 6(b) shows, for the same experiment as before, the average number of iterations executed per time step. Here we observe that the more asynchronous iterations the model is allowed to perform, the more total iterations are needed to converge. In other words, asynchronism has a negative impact on the rate of convergence, which is consistent with the theory exposed in Section II. However, even if that is the case, the overall effect of asynchronism is positive as the cost of these 'extra' iterations is compensated by having to perform fewer global synchronization steps on the CPU side.



(a)



(b)

Fig. 6: Effect of asynchronism on performance: (a) simulation time and (b) average number of iterations per time step, as functions of the number of asynchronous iterations allowed.

2) IPPA v/s KLU and LAPACK: Next we compared IPPA with the classic circuit simulator, represented by the LAPACK and KLU implementations. To maximize performance based in our previous observations, we set the number of asynchronous iterations in IPPA to 40 (see Figure 6(a)).

Figure 7 shows the execution time on a logarithmic scale of IPPA, LAPACK and KLU as a function of the problem size, n. We observe that the IPPA implementation is faster than both the LAPACK and KLU implementations on almost all the considered range. However, as n grows, the simulation time increases faster in IPPA than in KLU.

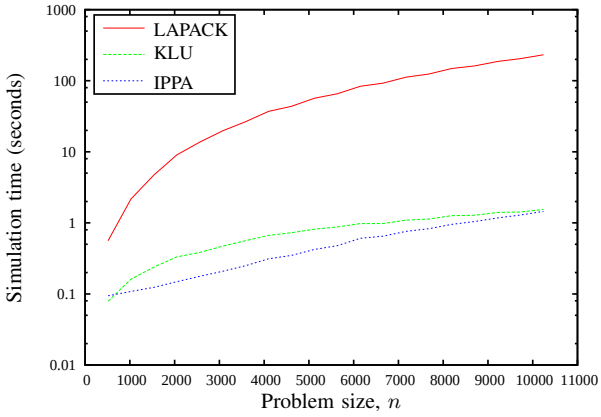


Fig. 7: Related performance of IPPA, LAPACK and KLU.

This last situation is most certainly an effect of our implementation not being tuned to yield maximum GPU performance just yet. To illustrate how much room is there for improvement, Fig. 8 presents the IPPA simulation time (left vertical axis) along with the average number of iterations performed per time step (right vertical axis). The horizontal axis contains the problem size, n. Whereas the simulation time grows exponentially with the problem size, the number of iterations seems to be unaffected by n. In other words, it is only the concurrent access to shared resources, which become scarce as the occupancy grows, that prevents maximum performance to be achieved and increases the simulation time more than it can be explained by the amount of work effectively performed by each thread. If the application was tuned to keep every thread working at all times, then it would be only a matter of having enough computing cores for the simulation time to become independent from the problem size. Of course this may be impossible to achieve in practice, as the resources of any computing architecture are essentially limited. However, there are many techniques that can be implemented in order to 'mask' the scarcity of resources as the occupancy grows [22].

As it is implemented today, we can already observe how the performance of our application scales with the amount of available resources. Figure 9 shows the simulation time of IPPA on different GPU architectures. Notice how, as the number of cores in the architecture grows, the time needed to complete the analysis decreases accordingly. Additionally, the speedup is higher for higher problem sizes.

Our current implementation of IPPA uses GPU's global memory, which is *off-chip*, to store and load all the relevant data during the course of the iteration. The advantage of doing so is that all the threads can access the data at all times. However, GPU also provides *on-chip* memory which is only shared by threads within the same thread-block; the access to this shared memory is about one hundred times faster

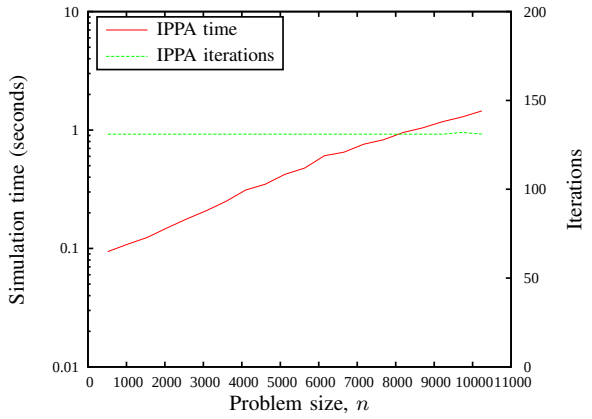


Fig. 8: Simulation time and number of iterations performed by IPPA.

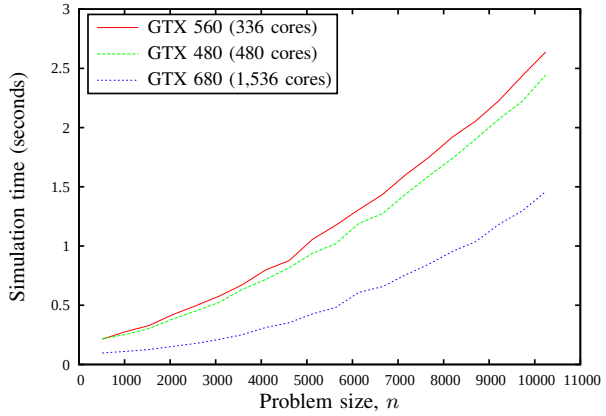


Fig. 9: Performance comparison of different GPU architectures running IPPA.

than the access to global memory. Thus, we can conceive an optimization strategy where we assign subsets of strongly-connected components in the circuit, to threads within the same block on the GPU. Then we let these threads perform a number of local iterations on shared memory, before committing their results to global memory for their integration with those from other blocks. This technique will reduce the number of accesses to global memory by a factor equal to the number of iterations performed locally; however, it might also damage the convergence of our iteration. The full study and implementation of this idea will be addressed in future works.

V. Conclusion

We have presented a model for time-domain transient circuit simulation which successfully competes in terms of performance with available state-of-the-art solutions. Furthermore, our approach is intrinsically parallel so it can be efficiently implemented on today's parallel machines. Our tests using a GPU implementation of the model show that the amount of parallel iterations needed to compute the result is independent of the number of nodes in the circuit. However, in order to fully exploit this interesting property, the current implementation must be tuned to achieve maximum utilization of the GPU architecture.

The optimization of the current GPU implementation according to the previous point, as well as the implementation of the proposed method on other parallel architectures, will be addressed in future works. We will also expand our model and investigate convergence issues and applicability of the proposed solution in the non-linear case.

Appendix A
Mathematical Reminders

In this section we present some classical definitions and mathematical results, for the sake of completion.

Definition 4 (Spectral radius)**.** Let $\boldsymbol{A} \in \mathbb{C}^{n\times n}$ with eigenvalues $\lambda_i, 1 \leq i \leq n$. Then,

$$\rho(\boldsymbol{A}) := \max_{1\leq i\leq n} |\lambda_i|, \tag{24}$$

is the spectral radius of $\boldsymbol{A}$.

Theorem 5 (Bound for the spectral radius [27])**.** *Let $\boldsymbol{A} \in \mathbb{C}^{n\times n}$, and let $|\boldsymbol{A}|$ be the matrix of absolute values of the entries of $\boldsymbol{A}$. Then $\rho(\boldsymbol{A}) \leq \rho(|\boldsymbol{A}|)$.*

Definition 5 (Reducible matrix)**.** Let $\boldsymbol{A} \in \mathbb{C}^{n\times n}$. $\boldsymbol{A}$ is called reducible if there exists a permutation matrix $\boldsymbol{P}$ such that $\boldsymbol{P}^T\boldsymbol{A}\boldsymbol{P}$ is of the form

$$\begin{bmatrix} \boldsymbol{A}_1 & \boldsymbol{A}_{12} \\ \boldsymbol{0} & \boldsymbol{A}_2 \end{bmatrix},$$

where $\boldsymbol{A}_1$ and $\boldsymbol{A}_2$ are square matrices of size at least 1. If $\boldsymbol{A}$ is not reducible, then $\boldsymbol{A}$ is called irreducible.

Definition 6 (Digraph of a matrix)**.** Let $\boldsymbol{A} = (a_{ij}) \in \mathbb{C}^{n\times n}$. The digraph $\boldsymbol{D}$ with vertices $1, \ldots, n$, in which there is an arc (i, j) if and only if $a_{ij} \neq 0$, is called the digraph of $\boldsymbol{A}$. We note $\boldsymbol{D} = \boldsymbol{D}(\boldsymbol{A})$.

Definition 7 (Strong connection)**.** Let $\boldsymbol{D}$ be a digraph and i, j any pair of vertices. If there are directed walks from i to j and from j to i, then $\boldsymbol{D}$ is strongly connected.

Theorem 6 (Interpretation of irreducibility in terms of the digraph of the matrix [27])**.** *Let $\boldsymbol{A} \in \mathbb{C}^{n\times n}$. Then $\boldsymbol{A}$ is irreducible if and only if $\boldsymbol{D}(\boldsymbol{A})$ is strongly connected.*

Appendix B
Deduction of the Component Model Equations

Assuming the steady-state is attained, in which case $v_h = \psi_{hk}$, we apply Ohm's Law on c_{hk}:

$$\psi_{hk} = v_h = v_k - r_{hk}i_{hk} - \hat{v}_{hk}. \tag{25}$$

Next we apply Kirchhoff's Current law on n_h:

$$\phi_h = \sum_{j\in\mathcal{I}_h} i_{hj} = \sum_{\substack{j\in\mathcal{I}_h \\ j\neq k}} i_{hj} + i_{hk} = 0 \tag{26}$$

We clear i_{hk} from (26) and replace it in (25):

$$\psi_{hk} = v_k + r_{hk} \sum_{\substack{j\in\mathcal{I}_h \\ j\neq k}} i_{hj} - \hat{v}_{hk}. \tag{27}$$

Expanding, and applying (25):

$$\psi_{hk} = v_k + r_{hk} \sum_{j \in \mathcal{I}_h} i_{hj} - r_{hk} i_{hk} - \hat{v}_{hk}$$
$$= v_h + r_{hk} \sum_{j \in \mathcal{I}_h} i_{hj}. \quad (28)$$

Finally, taking ϕ_h from (26) and replacing it in (28) we obtain the first formula:

$$\psi_{hk} = v_h + r_{hk} \phi_h. \quad (29)$$

And clearing i_{hk} from (25) we obtain the second formula:

$$i_{hk} = \frac{v_k - v_h - \hat{v}_{hk}}{r_{hk}}. \quad (30)$$

REFERENCES

[1] A. Brambilla and D. D'Amore, "The simulation errors introduced by the spice transient analysis," *Circuits and Systems I: Fundamental Theory and Applications, IEEE Transactions on*, vol. 40, no. 1, pp. 57–60, Jan 1993.

[2] A. Brambilla and P. Maffezzoni, "Envelope following method for the transient analysis of electrical circuits," *Circuits and Systems I: Fundamental Theory and Applications, IEEE Transactions on*, vol. 47, no. 7, pp. 999–1008, Jul 2000.

[3] L. W. Nagel and D. O. Pederson, *SPICE: Simulation program with integrated circuit emphasis*. Electronics Research Laboratory, College of Engineering, University of California, 1973.

[4] H. Elwan and A. M. Soliman, "Low-voltage low-power cmos current conveyors," *Circuits and Systems I: Fundamental Theory and Applications, IEEE Transactions on*, vol. 44, no. 9, pp. 828–835, Sep 1997.

[5] F. Awwad, M. Nekili, V. Ramachandran, and M. Sawan, "On modeling of parallel repeater-insertion methodologies for soc interconnects," *Circuits and Systems I: Regular Papers, IEEE Transactions on*, vol. 55, no. 1, pp. 322–335, Feb 2008.

[6] K. Gulati, J. F. Croix, S. P. Khatr, and R. Shastry, "Fast circuit simulation on graphics processing units," in *Proceedings of the 2009 Asia and South Pacific Design Automation Conference*, ser. ASP-DAC '09. Piscataway, NJ, USA: IEEE Press, 2009, pp. 403–408. [Online]. Available: http://dl.acm.org/citation.cfm?id=1509633.1509733

[7] N. Kapre and A. DeHon, "Performance comparison of single-precision spice model-evaluation on fpga, gpu, cell, and multi-core processors," in *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*. IEEE, 2009, pp. 65–72.

[8] G.-C. Yang, "Paraspice: a parallel circuit simulator for shared-memory multiprocessors," in *Design Automation Conference, 1990. Proceedings., 27th ACM/IEEE*. IEEE, 1990, pp. 400–405.

[9] N. Kapre and A. DeHon, "Parallelizing sparse matrix solve for spice circuit simulation using fpgas," in *Field-Programmable Technology, 2009. FPT 2009. International Conference on*. IEEE, 2009, pp. 190–198.

[10] A. R. Newton and A. L. Sangiovanni-Vincentelli, "Relaxation-based electrical simulation," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 3, no. 4, pp. 308–331, 1984.

[11] H. Peng and C.-K. Cheng, "Parallel transistor level circuit simulation using domain decomposition methods," in *Design Automation Conference, 2009. ASP-DAC 2009. Asia and South Pacific*. IEEE, 2009, pp. 397–402.

[12] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference*. ACM, 1967, pp. 483–485.

[13] T. A. Davis and E. Palamadai Natarajan, "Algorithm 907: Klu, a direct sparse solver for circuit simulation problems," *ACM Trans. Math. Softw.*, vol. 37, no. 3, pp. 36:1–36:17, Sep. 2010. [Online]. Available: http://doi.acm.org/10.1145/1824801.1824814

[14] S. Talukdar, S. Pyo, R. Mehrotra, C.-M. U. D. R. Center, and E. P. R. Institute, *Designing Algorithms and Assignments for Distributed Processing*, ser. EPRI report. Electric Power Research Institute, 1983. [Online]. Available: http://books.google.fr/books?id=YSo3nQEACAAJ

[15] J. N. Tsitsiklis, "Problems in decentralized decision making and computation." DTIC Document, Tech. Rep., 1984.

[16] A. Frommer and D. B. Szyld, "On asynchronous iterations," *JOURNAL OF COMPUTATIONAL AND APPLIED MATHEMATICS*, vol. 123, pp. 201–216, 2000.

[17] D. Chazan and W. Miranker, "Chaotic relaxation," *Linear algebra and its applications*, vol. 2, no. 2, pp. 199–222, 1969.

[18] P. Tseng, D. P. Bertsekas, and J. N. Tsitsiklis, "Partially asynchronous, parallel algorithms for network flow and other problems," *SIAM Journal on Control and Optimization*, vol. 28, no. 3, pp. 678–710, 1990.

[19] B. Lubachevsky and D. Mitra, "A chaotic asynchronous algorithm for computing the fixed point of a nonnegative matrix of unit spectral radius," *Journal of the ACM (JACM)*, vol. 33, no. 1, pp. 130–150, 1986.

[20] H. W. Dommel, "Digital computer solution of electromagnetic transients in single and multiphase," *Networks IEEE*, vol. 88, no. 4, pp. 388–399, 1969.

[21] S. Hong and H. Kim, "An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness," in *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3. ACM, 2009, pp. 152–163.

[22] C. Nvidia, "Compute unified device architecture programming guide," 2007.

[23] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*, 3rd ed. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1999.

[24] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra, "Dense linear algebra solvers for multicore with gpu accelerators," in *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 1–8.

[25] J. R. Gilbert and T. Peierls, "Sparse partial pivoting in time proportional to arithmetic operations," *SIAM Journal on Scientific and Statistical Computing*, vol. 9, no. 5, pp. 862–874, 1988.

[26] J. Siek, L.-Q. Lee, and A. Lumsdaine, "Boost random number library," *Software library. URL http://www. boost. org/users/download/. Accessed*, 2012.

[27] R. A. Horn and C. R. Johnson, *Matrix analysis*. Cambridge university press, 2012.

Manuel Marin received his M.Sc. degree in electrical engineering from Pontificia Universidad Católica de Chile, Santiago, Chile, in 2005 and his M.Sc. degree in computer science from Université de Perpignan, Perpignan, France, in 2011. Since 2012, he is a Ph.D. student in the Faculty of Computer Science at Université de Perpignan, and since 2014, also in the School of Electrical, Electronic and Communications Engineering at University College Dublin as part of a co-tutelle program. His research activities concentrate in multi-core computer architectures, power flow simulation and computer arithmetic.

David Defour received his M.Sc. degree from Université Montpellier 2, Montpellier, France, in 2000 and his Ph.D. degree in computer science from Ecole Normale Supérieure de Lyon, Lyon, France, in 2003. Since 2004, he is an associate professor in the Faculty of Computer Science at Université de Perpignan, Perpignan, France. His fields of interest include general purpose GPU programming (GPGPU), computer arithmetic and computer architecture and microarchitecture.

Federico Milano (S'09) received from the University of Genoa, Italy, the Electrical Engineering degree and the Ph.D. degree in 1999 and 2003, respectively. From 2001 to 2002 he was a Visiting Scholar at the University of Waterloo, Canada. From 2003 to 2013, he was with the University of Castilla-La Mancha, Spain. In 2013, he joined the University College Dublin, Ireland, where he is currently Associate Professor. His research interests include power system modelling, stability and control.