Asynchronous Power Flow on Graphic Processing Units

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

***Abstract*—Asynchronous iterations can be used to implement fixed-point methods such as Jacobi and Gauss-Seidel on parallel computers with high synchronization costs. However, they are rarely considered in practice due to the low convergence rate. This paper describes an implementation on GPUs of a novel Power Flow analysis model using asynchronous iterations. We present our model for the solution of the Power Flow analysis problem, prove its convergence and evaluate its performance for a GPU execution.**

***Index Terms*—Asynchronous iterations, fixed-point method, power flow analysis, graphic processing units.**

I. Introduction

Power Flow (PF) analysis refers to the steady-state analysis of AC power networks, widely used in several applications for power system operation and planning [1]. Traditional PF analysis tools rely on matrix factorization using highly optimized serial direct solvers [2], [3], [4]. More recently, a parallel-friendly PF analysis tool has also been introduced, based on an analogical representation of the system [5]. The real power system is mapped onto a CMOS integrated circuit, which allows one to physically simulate the system at a much higher frequency. The efficiency of this approach is linked with the intrinsic parallelism of the physical system, where every element of the network is acting simultaneously. The main drawback is the lack of flexibility since for every system that one wants to simulate, a new analogue circuit has to be built.

In this paper we present a software implementation of the same idea presented in [5], so that any arbitrary system can be analyzed without extra effort. This allows us to propose an intrinsically parallel PF analysis model. Our approach is based on fixed-point iterations performed individually for every element in the system. Results of succesive updates are exchanged until an equilibrium point is reached, as in the concept of *team algorithms* [6], [7]. The approach is well-suited for parallel implementation using *asynchronous iterations*, which have also been applied to the network flow problem [8], [9].

The main reason why asynchronous iterations are not usually preferred over synchronous ones is that (i) they require stronger assumptions in order to converge, and (ii) convergence rate is smaller. However, massively parallel computing environments can benefit from the asynchronous approach whenever the cost of synchronization becomes too high. For instance, in the Graphic Processing Unit (GPU) synchronization between threads from different blocks can consume up to 50% of the total execution time [10]. In order to target GPU architectures, we study convergence properties of our model and provide a sufficient condition for convergence.

The remainder of the paper is organized as follows. Section II introduces fixed-point iterations and presents several alternatives for parallel implementation, including synchronous, asynchronous and partially asynchronous frameworks. In Section III, the proposed intrinsically parallel method for PF analysis is presented and its convergence properties are studied. Section IV describes the implementation and evaluation of the proposed method on a set of benchmarks as well as comparisons with existing approaches. Finally, Section V draws conclusions and outlines future work.

M. Marin is with Université de Liège, Institut Montefiore, Liège, Belgium (e-mail: mmarin@ulg.ac.be).

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).

II. Fixed-point iterations on a parallel computer

Let us first recall the concept of fixed-point iterations defined as follows.

Definition 1 (Fixed-point iteration)**.** Let $\boldsymbol{f}\colon \mathbb{R}^n \to \mathbb{R}^n$ and $\boldsymbol{x} \in \mathbb{R}^n$. Then, $\boldsymbol{x}$ is a fixed-point of $\boldsymbol{f}(\cdot)$ if and only if $\boldsymbol{x} = \boldsymbol{f}(\boldsymbol{x})$. Furthermore, let $\boldsymbol{x}^{(0)} \in \mathbb{R}^n$. Then, the iteration given by:

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{f}(\boldsymbol{x}^{(k)}), \quad k = 0, 1, \ldots, \tag{1}$$

is called a fixed-point iteration on $\boldsymbol{f}(\cdot)$.

Examples of fixed-point iterations are the Jacobi and Gauss-Seidel methods for solving systems of linear equations. Note that these methods can be succesfully applied to PF analysis by re-writing the (non-linear) PF equations in a suitable way [1]. In this article, we consider the PF problem in its original non-linear form.

Fixed-point iterations can be implemented in parallel by splitting the set of input element among threads. Each thread updates its respective elements and communicates its results to others with or without synchronization.

A. Synchronous iterations

With synchronous iterations, all the elements are updated by block before the process moves as a whole into the next step. The above requires to place a synchronization point at the end of each iteration. As a consequence, threads that finish their updates earlier are forced to wait for the others to catch up. The reason for threads having different processing times are various. Some of them may be intrinsic to the algorithm, e.g., irregular memory access patterns, load imbalance. Some

others may be related to the architecture, e.g., differences in processor frequency, communication network constraints [11].

B. Asynchronous iterations

An alternative to the synchronous approach is the so-called asynchronous iteration, where individual threads do not wait for others and simply go ahead with their work. This means that at every asynchronous iteration only a subset of the elements are updated. No synchronization point is explicitly added to the program. As a consequence, some of the updates may be computed with data from several iterations back.

An asynchronous iteration is defined in terms of an *update function* and a *delay function*. The update function, noted $\mathcal{U}(\cdot)$, receives the iteration counter $k \in \mathbb{N}$, and returns a subset of $\{1,\dots,n\}$ indicating the list of threads that update their elements at iteration k. The delay function, noted $d(\cdot)$, receives the indices $i,j \in \{1,\dots,n\}$ and the iteration counter $k \in \mathbb{N}$, and returns the delay of thread j with respect to thread i at iteration k. This delay represents the number of iterations performed since the value of x_j was last made visible to i.

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

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

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

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

In addition, let $\boldsymbol{x}^{(0)} \in \mathbb{R}^n$. Then, the iteration defined by:

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

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

Assumptions in (2a), (2b) and (2c) ensure that the process is well defined. Specifically, 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 elements are eventually used. Finally, the assumption in (2c) states that no element ceases to be updated during the course of the iteration. In practical terms, if the updates are made visible as soon as they are computed, then the third assumption implies the second. This is the case, for example, of an application that uses GPU global memory to store the data. Note that a synchronous iteration is a special case of asynchronous iteration with $\mathcal{U}(k) = \{1,\dots,n\}$ and $d(i,j,k) = 0$, for all i,j,k. Due to the peculiarities introduced above, asynchronous iterations converge differently from synchronous ones. The following result considers the case of a linear application.

Theorem 1 (Necessary and sufficient condition for convergence of linear asynchronous iterations [12])**.** *Let $\boldsymbol{f}\colon \mathbb{R}^n \to \mathbb{R}^n$, given by,*

$$\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{L}\boldsymbol{x} + \boldsymbol{b}, \tag{4}$$

where $\boldsymbol{L} \in \mathbb{R}^{n\times n}$, and $\boldsymbol{b} \in \mathbb{R}^n$. Let $|\boldsymbol{L}|$ denote the matrix of absolute values of the entries of $\boldsymbol{L}$, and $\rho(|\boldsymbol{L}|)$ its spectral radius. If

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

then any asynchronous iteration on $\boldsymbol{f}(\cdot)$ converges to $\boldsymbol{x}^*$, the unique fixed-point of $\boldsymbol{f}(\cdot)$, 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 the associated asynchronous iteration does not converge to $\boldsymbol{x}^*$.

Proof. See [12]. □

C. Partially asynchronous iterations

In some circumstances, convergence of asynchronous iterations can be facilitated by making threads synchronize every once in a while.

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

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

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

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

Then, the iteration given by equation (3) is now termed a partially asynchronous iteration.

The assumption in (6a) establishes that not only newer values of the vector elements 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 element is updated at least once in every $\bar{s}$ consecutive iterations. Finally, the assumption in (6c) establishes that every element is updated using its own last calculated value. In practical terms, the first two assumptions can be met by performing a synchronization step every l iterations, where l is the minimum of $\bar{d}$ and $\bar{s}$. The third assumption can be met by having each element assigned to only one thread.

As mentioned above, partially asynchronous iterations converge 'more easily' than totally asynchronous ones. The following result establishes a convergence criteria in the linear case, with softer assumptions than the ones in Theorem 1.

Theorem 2 (Sufficient condition for convergence of linear partially asynchronous iterations [13])**.** *Let $\boldsymbol{f}\colon \mathbb{R}^n \to \mathbb{R}^n$, given by,*

$$\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{L}\boldsymbol{x} + \boldsymbol{b}, \tag{7}$$

where $\boldsymbol{L} \in \mathbb{R}^{n\times n}$, and $\boldsymbol{b} \in \mathbb{R}^n$. Let $\boldsymbol{g}\colon \mathbb{R}^n \to \mathbb{R}^n$, given by:

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

where $\alpha \in (0,1)$.

If $\boldsymbol{L} = (l_{ij})$ is irreducible and

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

then any partially *asynchronous iteration on* $\boldsymbol{g}(\cdot)$ *converges to* $\boldsymbol{x}^*$*, fixed-point of* $\boldsymbol{f}(\cdot)$.

Proof. See [13]. □

In [14], the authors presented a special case in which convergence occurs for $\alpha = 1$, at a linear (geometric) rate. They also determined a lower bound for the average convergence rate (per iteration) in the long-term, τ, as follows:

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

with

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

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

where $\min^+(\cdot)$ refers to the minimum of the positive elements. This means that the error is scaled at most by a factor of $(1-\sigma^r)^{1/r}$ in any average iteration. As σ is always lower than 1, the convergence rate approaches 1 for increasing values of n, $\bar{d}$ and $\bar{s}$. That is to say, the convergence becomes sub-linear.

For illustration purposes, consider that $\bar{d} = \bar{s} = 0$ (synchronous iteration). In this case, $r = 1$ and $\tau = 1 - \sigma$. In other words, the convergence rate is unaffected by the problem size n and only depends on σ. Now consider that $\bar{d} = \bar{s} = 1$. This corresponds to a partially asynchronous iteration with a synchronization step every other iteration. In this case, $r = 2n$, which means that convergence slows down as the problem grows. Figure 1 shows the lower bound on the convergence rate as a function of the problem size n. The different curves represent different values of σ. Observe that, unless σ is very high, convergence becomes sub-linear very quickly.

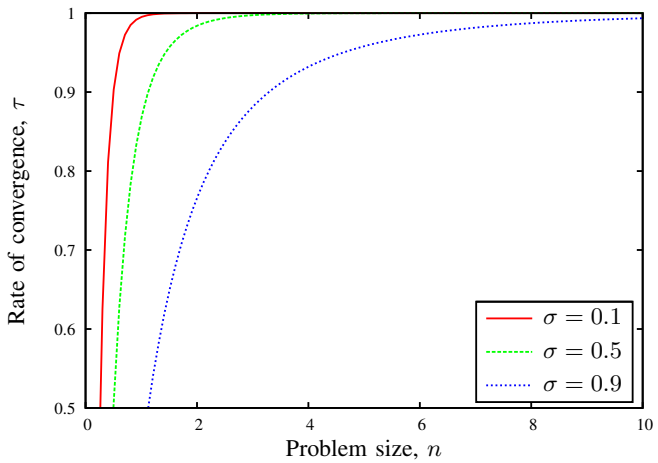

Fig. 1. Bound on the convergence rate as a function of the problem size, for different values of the parameter σ.

This might be a strong reason to prefer the synchronous approach over the partially asynchronous. However, the partially asynchronous approach can be a valid alternative in some cases, e.g., if the costs of synchronization are relatively high, or when the number of iterations required to converge is relatively low.

III. Proposed method for AC circuit analysis

This section presents how to perform PF analysis based on partially asynchronous iterations. The method is designed to be implemented on Single Instruction Multiple Thread (SIMT) architectures such as GPUs, by exploiting regularity. The method is presented through a top-down approach. The network model is described first, specifying how the different parts communicate and interact. Next, the models of the different elements that compose the power system are specified in their atomic behaviour. Finally, the convergence properties of the proposed method are studied.

A. Network model

The proposed methodology distinguishes two fundamental units in the power system: components and nodes. The components are the electrical devices, and the nodes are the points of interconnection. For sake of simplicity and also to facilitate illustration of the method, the following three assumptions are made:

1) Only dipole components, i.e., components connected to two nodes, are present.
2) There is no more than one component connected between any two nodes.
3) Every node is connected to at least two components.

The circuit is *well defined* if there are components connecting all the nodes in a closed path. Node h in the circuit is noted n_h. The component connected between n_h and n_m is noted c_{hm}. The set of all nodes in the circuit is noted $\mathcal{N}$, and the set of components, $\mathcal{C}$.

In addition, a set of *influencers*, $\mathcal{I}_h \subset \mathcal{N}$, is defined for each non-ground node $n_h \in \mathcal{N}$. This set contains all the nodes separated from n_h by exactly one component. These are also known as the *fanin* nodes [15].

For illustration purposes, consider the circuit in Fig. 2. In this case, $\mathcal{N} = \{n_0, n_1, n_2, n_3\}$ and $\mathcal{C} = \{c_{01}, c_{12}, c_{02}, c_{23}, c_{03}\}$. The circuit is well defined, since c_{01}, c_{12}, c_{23} and c_{03} connect all the nodes in a closed path. In addition, $\mathcal{I}_1 = \{n_0, n_2\}$, $\mathcal{I}_2 = \{n_0, n_1, n_3\}$, and $\mathcal{I}_3 = \{n_0, n_2\}$.

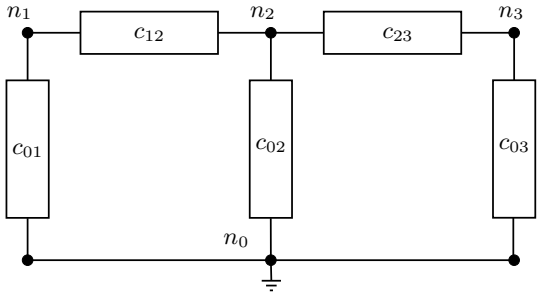

Fig. 2. Example 4-node circuit.

In the proposed methodology, $\mathbb{v}_h$ denotes the voltage at node n_h, and $\mathbb{u}_{hm}$ denotes the voltage at node n_h 'according' to component c_{hm}. The two are different during the course of the iteration, and become equal at convergence. In addition,

$\mathbb{s}_{hm}$ denotes the power injected to n_h by component c_{hm}, and $\Delta\mathbb{s}_h$ denotes the power mismatch at n_h.

At every iteration k, a component c_{hm} obtains $\mathbb{v}_h^{(k)}$ and $\Delta\mathbb{s}_h^{(k)}$ from node n_h, and returns $\mathbb{u}_{hm}^{(k)}$ and $\mathbb{s}_{hm}^{(k)}$ to it. A similar exchange occurs with node n_m at the other end of the component. Next, at the same iteration k, node n_h receives $\mathbb{u}_{hj}^{(k)}$ and $\mathbb{s}_{hj}^{(k)}$ from each component c_{hj} connected to it, and returns $\mathbb{v}_h^{(k+1)}$ and $\Delta\mathbb{s}_h^{(k+1)}$ to all of them. Then, the process moves into a next iteration. Figure 3 illustrates the above for component c_{12} and node n_2 in the example 4-node circuit.

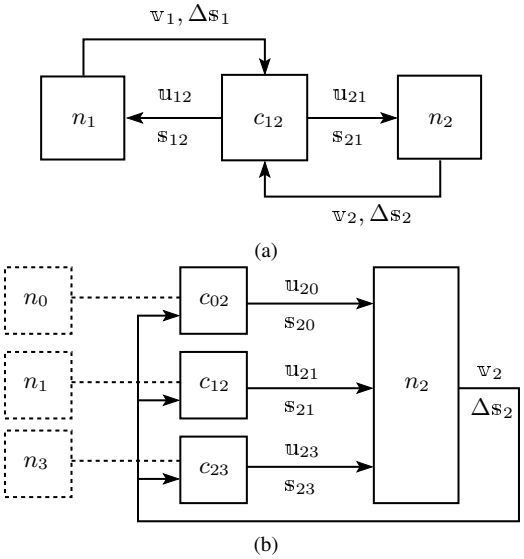

Fig. 3. Network model: (a) Component; and (b) Node.

B. Component and node models

1) Component model: For the purposes of the proposed method, only three types of component are required: the generator, the branch and the shunt. The use of a universal model improves instruction regularity of the GPU implementation, as discussed in Section IV. The impedance of component c_{hm} is noted z_{hm}.

a) Generator: It is connected between the ground and one node, say n_h. The generator is there to ensure that the reactive power mismatch at the node is equal to zero. In this sense, the model assumes that generators are able to provide as much reactive power as needed in order to maintain the voltage. The slack generator also does the same for real power. Accordingly, the iteration is the following: for non-slack generators,

$$\mathrm{Im}(\mathbb{s}_h^{(k)}) = 0. \tag{12}$$

And for the slack generator,

$$\mathbb{s}_h^{(k)} = 0. \tag{13}$$

b) Branch: It is connected between two non-ground nodes, say n_h and n_m. It consists of a constant impedance, z_{hm}. The branch iteration is defined as follows:

$$\mathbb{u}_{hm}^{(k)} = \mathbb{v}_h^{(k)} + z_{hm}\left(\frac{\Delta\mathbb{s}_h^{(k)}}{\mathbb{v}_h^{(k)}}\right)^*, \tag{14a}$$

$$\mathbb{s}_{hm}^{(k)} = \mathbb{v}_h^{(k)}\left(\frac{\mathbb{v}_m^{(k)} - \mathbb{v}_h^{(k)}}{z_{hm}}\right)^*. \tag{14b}$$

c) Shunt: It is connected between the ground and one node, say n_h. It represents an impedance to ground, noted z_{h0} (the subscript '0' indicates the ground). The shunt iteration is similar to the branch one. The main difference is that $\mathbb{v}_m$ in equation (14b) is replaced by zero (the ground voltage). The iteration is the following:

$$\mathbb{u}_{h0}^{(k)} = \mathbb{v}_h^{(k)} + z_{h0}\left(\frac{\Delta\mathbb{s}_h^{(k)}}{\mathbb{v}_h^{(k)}}\right)^*, \tag{15a}$$

$$\mathbb{s}_{h0}^{(k)} = -\frac{|\mathbb{v}_h^{(k)}|^2}{(z_{hm})^*}, \tag{15b}$$

where $|\cdot|$ indicates the absolute value.

These equations are obtained by applying the Ohm Law on component c_{hm}, and the Kirchhoff Current Law on node n_h. Note that if the power mismatch at n_h is equal to zero (i.e., $\Delta\mathbb{s}_h^{(k)} = 0$), then $\mathbb{u}_{hm}^{(k)} = \mathbb{v}_h^{(k)}$. That is to say, the voltage according to component c_{hm} is the same as the one according to node n_h itself. This corresponds to a condition of convergence at component level.

2) Node model: The node is simply modelled as a point where different components exchange data and consolidate their results. The node iteration is defined as follows:

$$\mathbb{v}_h^{(k+1)} = \sum_{m\in\mathcal{I}_h} w_{hm}\mathbb{u}_{hm}^{(k)}, \tag{16a}$$

$$\Delta\mathbb{s}_h^{(k+1)} = \sum_{m\in\mathcal{I}_h} \mathbb{s}_{hm}^{(k)}, \tag{16b}$$

where w_{hm} is defined as the weight of component c_{hm} within node n_h. These weights satisfy

$$\sum_{m\in I_h} w_{hm} = 1. \tag{17}$$

In other words, the voltage at each node is updated with the weighted sum of the voltages computed by each component connected to it. Note that if convergence is attained at component level (i.e., $\mathbb{u}_{hm}^{(k)} = \mathbb{v}_h^{(k)}, \forall m \in \mathcal{I}_h$), then $\mathbb{v}_h^{(k+1)} = \mathbb{v}_h^{(k)}$. This happens when all the components agree on the value of the node voltage, and corresponds to a condition of convergence at node level.

A second node iteration is added to allow PQ loads to become constant impedances, if the voltage at the node is outside specific limits. This is a technique used in traditional PF analysis [1]. In the proposed method, the power injected to n_h by loads is noted $\mathbb{s}_{L,h}$. If the voltage is outside the limits, this variable is scaled by a factor depending on the current voltage and the violated limit. Then, the power mismatch at the node is adjusted to reflect the change in the load. In conclusion,

the iteration is composed of two parts. The first updates the power injected by loads, as follows:

$$\mathbb{s}_{L,h}^{(k+1)} = \begin{cases} \left(\frac{v_h^{(k)}}{v_h^{min}}\right)^2 \mathbb{s}_{L,h}^{(k)} & \text{if } v_h^{(k)} < v_h^{min}, \\ \left(\frac{v_h^{(k)}}{v_h^{max}}\right)^2 \mathbb{s}_{L,h}^{(k)} & \text{if } v_h^{(k)} > v_h^{max}, \\ \mathbb{s}_{L,h}^{(k)} & \text{otherwise,} \end{cases} \tag{18}$$

where v_h^{min} and v_h^{max} are the voltage limits at n_h. Then, the second part updates the power mismatch, as follows:

$$\Delta\mathbb{s}_h^{(k+1)} = \Delta\mathbb{s}_h^{(k)} - \mathbb{s}_{L,h}^{(k)} + \mathbb{s}_{L,h}^{(k+1)}. \tag{19}$$

The use of the above iteration reduces instruction regularity, as it needs to be performed only by nodes. However, it may be required for convergence of the method and involves a limited number of operations which can be acceptable in a SIMT execution context.

C. Convergence study

This section investigates the convergence properties of the proposed iterative method for two different synchronization schemes.

Let n_0 be the ground-node, and $\mathcal{N}'$ the set of all nodes in the circuit minus the ground. That is to say, $\mathcal{N}' = \mathcal{N} \setminus \{n_0\}$. Also, let $\mathbb{v} \in \mathbb{C}^{n-1}$ be the vector of voltages at all non-ground nodes. The iteration function, $\mathbb{f} : \mathbb{C}^{n-1} \to \mathbb{C}^{n-1}$, is given by:

$$\mathbb{f}_h(\mathbb{v}) = \mathbb{v}_h + \sum_{m\in\mathcal{I}_h}\sum_{j\in\mathcal{I}_h} w_{hm} z_{hm} \frac{\mathbb{v}_j - \mathbb{v}_h}{z_{hj}}, \quad h \in \mathcal{N}'. \tag{20}$$

This expression is obtained by combining equations (14a), (14b), (16a), (16b) and (17). Equation (20) is linear, thus, it can be written as follows:

$$\mathbb{f}(\mathbb{v}) = \boldsymbol{L}\mathbb{v}, \tag{21}$$

where $\boldsymbol{L}$ is given by:

$$\begin{aligned} l_{hh} &= 1 - \sum_{m\in\mathcal{I}_h}\sum_{j\in\mathcal{I}_h} \frac{w_{hm} z_{hm}}{z_{hj}}, \\ l_{hj} &= \begin{cases} \sum\limits_{m\in\mathcal{I}_h} \frac{w_{hm} z_{hm}}{z_{hj}} & \text{if } j \in \mathcal{I}_h, \\ 0 & \text{otherwise.} \end{cases} \end{aligned} \tag{22}$$

However, one can notice that a totally asynchronous iteration on $\boldsymbol{f}(\cdot)$, as given by (21), is not guaranteed to converge. This is linked with the fact that the sum of the elements in any row of $\boldsymbol{L}$, where $\boldsymbol{L}$ is given by (22), is equal to 1, then 1 is an eigenvalue of $\boldsymbol{L}$ and $\rho(\boldsymbol{L}) \geq 1$. Since for every matrix A, $\rho(|\boldsymbol{A}|) \geq \rho(\boldsymbol{A})$, then $\rho(|\boldsymbol{L}|) \geq 1$.

This implies that to ensure convergence of the proposed iteration, further assumptions are needed. The following theorem establishes a sufficient condition for convergence based on partially asynchronous iterations.

Theorem 3. *Let $\mathbb{g}(\cdot)$ defined by $\mathbb{g}(\mathbb{v}) = (1-\alpha)\mathbb{v} + \alpha\mathbb{f}(\mathbb{v})$, where $0 < \alpha < 1$ and $\mathbb{f}(\cdot)$ given by (20). Let the weights, $w_{hm}, m \in I_h$, satisfy the following:*

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

Then, any partially asynchronous iteration over $\mathbb{g}(\cdot)$ converges to $\mathbb{v}^*$, fixed-point of $\mathbb{f}(\cdot)$.

Proof. See [16]. □

IV. Case Study

The proposed method for PF analysis has been tested on the Nvidia Kepler GPU architecture [17]. It has been evaluated on a series of benchmarks, with the aim of exploring convergence, performance and scalability issues.

A. Implementation

Regularity is one of the most powerful leverages for GPU performance [18]. Algorithm 1 presents a regular implementation of the proposed method. In order to ensure asynchronous evaluation of each component, we set the number of blocks to be smaller that the number of blocks that can be handled in parallel by the hardware. Therefore all threads are continuously running until convergence of the system. Threads are handling the set of components in a round-robin manner, which leads to a similar amount of work per thread as components are always connected to either one or two nodes. If threads were assigned to nodes instead, then some threads would perform much more work than others, as nodes can be connected to any number of components.

Instructions 5 and 6 calculate the voltage at its component's terminals and updates the appropriate coordinates on vector $\mathbb{v}$. Instructions 7 and 8 calculate the power injections at each terminal and updates the appropriate coordinates on vector $\Delta\mathbb{s}$. The process is repeated after synchronizing all threads, until convergence is reached.

Algorithm 1 Intrinsically parallel algorithm for PF analysis.

Input: Vector of initial power mismatches, $\Delta\mathbb{s}$; component model parameters; tolerance for the error, ϵ, maximum number of asynchronous iterations, K.
Output: Vector of node voltages, $\mathbb{v}$.

1: $\mathbb{v} = \mathbf{1}\angle\mathbf{0}$ # Initial guess
2: **for all** h, m: $c_{hm} \in \mathcal{C}$ **in parallel do**
3: $k = 0$
4: **repeat**
5: calculate $\mathbb{u}_{hm}$ and $\mathbb{u}_{mh}$
6: update $\mathbb{v}_h$ and $\mathbb{v}_m$
7: calculate $\mathbb{s}_{hm}$ and $\mathbb{s}_{mh}$
8: update $\Delta\mathbb{s}_h$ and $\Delta\mathbb{s}_m$
9: $k \leftarrow k + 1$
10: **if** $k > K$ **then**
11: synchronize threads
12: $k = 0$
13: **until** $|\Delta\mathbb{s}_h|, |\Delta\mathbb{s}_m| < \epsilon$

As discussed in [19], coalesced global memory accesses can drastically improve GPU performance by reducing the number of memory transactions. In the proposed implementation, the above is achieved by using a 'structure of arrays' approach. In addition, performance can benefit of shared memory for recurrent memory operations. However, shared memory is

only accessible to threads within one thread-block, whereas global memory is accessible to all threads. Accordingly, using shared memory in the proposed method increases the delay between threads, which affects convergence as discussed in Section III-C. The point where shared memory usage brings performance improvement is highly dependent of the problem parameters and setting this point could benefit from previous research conducted on *flexible communication* [20].

B. Evaluation

The implementation is evaluated using randomly generated benchmarks from 1,024 to 8,192 nodes, which correspond to real size networks. The benchmarks are obtained using the random radial network generator provided by Dome [3]. We used a Xeon E5645 CPU, with 12 cores at 2.4Ghz (1 used), a Tesla K40c, with 2 880 cores at 0.7 Ghz and a GeForce GTX 680 with 1 536 cores at 1.06Ghz. All source code is compiled using G++ 4.8.2 and CUDA 6.5.

1) Convergence: As discussed in Section II-C, partially asynchronous fixed-point iterations typically exhibit a sub-linear rate of convergence. This means that reducing the convergence tolerance below a small threshold may require a large number of iterations. Figure 4 illustrates the relation between the convergence error and the number of iterations for the proposed method using the K40c GPU. It shows the mismatch (on a logarithmic scale) as a function of the maximum number of iterations performed by any component. The different curves represent different problem sizes. Note that the behaviour is very similar on all 6 benchmarks, suggesting that convergence is independent from the problem size. However, the convergence rate for this experiment depends on the targeted accuracy. For example, reducing the error from 10^{-3} to 10^{-4} requires about 555 iterations in average. This corresponds to a convergence rate of 0.995. The convergence rate has an asymptotic limit of 1 when the targeted accuracy is increasing which is coherent with the theoretical study of Section II-C,

The farther the method is from the exact solution, the faster it approaches that solution. This is the opposite of the Newton-Raphson (NR) method where the convergence rate is improving at each iteration. From this point of view, it may be interesting to combine both approaches into a hybrid solution that switches methods depending on the error. For example, the proposed method can be used to obtain a first solution fast, and the NR method can be used to improve that solution afterwards.

2) Performance: We compared the performance of the proposed approach with serial NR provided by Dome [3]. The Dome's implementation uses KLU, a highly optimized library for sparse matrix factorization, particularly suited to deal with matrices from circuit analysis [21]. The proposed method is tested on the Tesla K40c GPU, and the NR on the Xeon E5645 CPU.

Figure 5 shows the execution time of both methods (on a logarithmic scale) as a function of the problem size, n, for a targeted error of 10^{-4}. The NR execution time grows exponentially with the problem size, which is consistent with

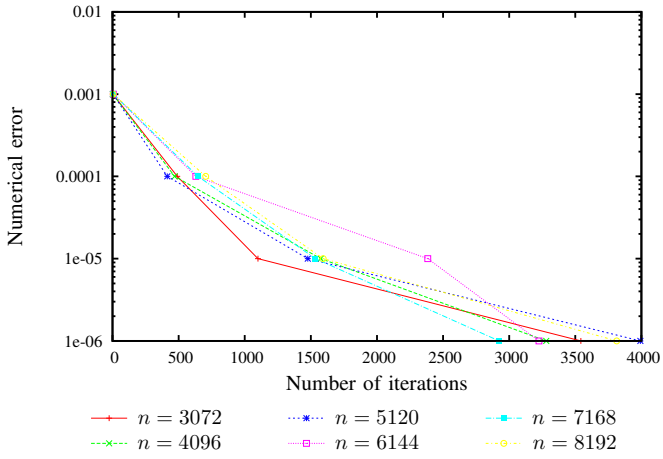

Fig. 4. Numerical error as a function of the number of iterations on the K40c GPU.

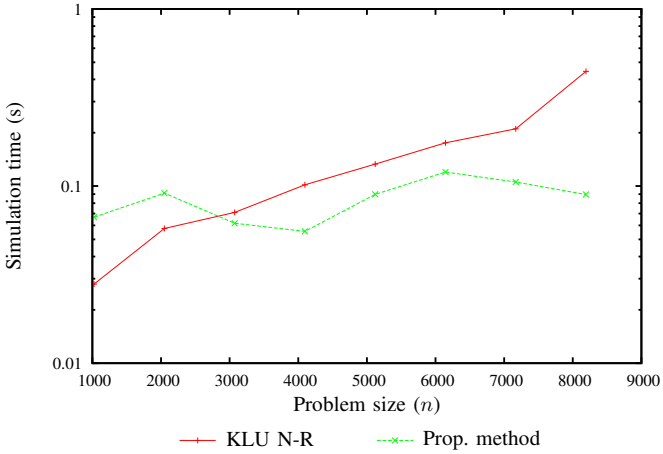

Fig. 5. Related performance of N-R and the proposed method (accuracy: 10^{-4}).

precedent results [1]. In the proposed method, in turn, the execution time seems less affected by the size, and exhibits a more flat profile. At about 3,000 nodes, there is a turning point in which the proposed method becomes faster than NR. This turning point is dependent of many parameters (architecture, implementation, topology, ...) and it is expected to evolve favorably for the proposed method as the number of embeded cores per chip is increasing.

Note that the parallelization of the NR method on GPU is out of the scope of this paper. Indeed, parallelizing the LU matrix factorization on GPU is a very complicated task due to irregular memory access patterns and strong data dependencies [22]. For this reason, comparisons with traditional PF methods implemented on GPU are left for future works.

3) Scalability: As mentioned in Section I, performance of asynchronous methods should increase with the number of computing cores. The above is confirmed by Fig. 6, which shows the execution time of the proposed asynchronous method on two Nvidia Kepler GPU architectures as a function of the problem size. The Tesla K40c GPU, embedding 2,880 computing cores, allows for a performance gain of about 1.3 over the GTX 680, embedding 1,536 cores. This means that

the proposed method almost achieve strong scalability.

Fig. 7 represents the number of iterations needed to converge as a function of the number of blocks launched. We compared the *synchronous* version (that synchronizes threads after each iteration), with the *asynchronous* one (that does not perform any synchronization at all). We observe that only the latter benefits from increasing number of thread-blocks. Moreover, we see a turning point in which performance does not improve and rather decays as the number of blocks keeps growing.

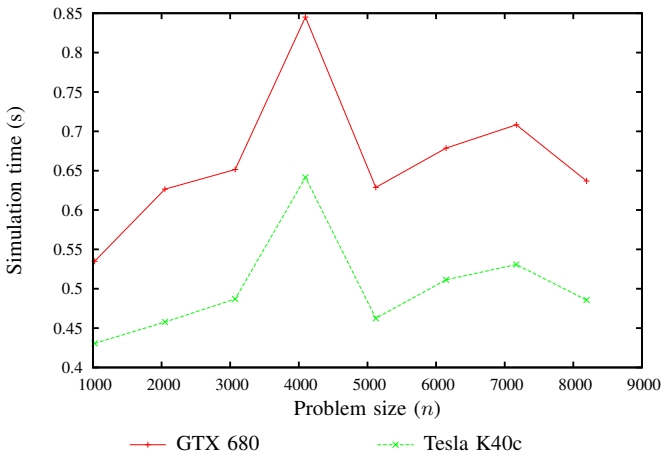

Fig. 6. Related performance of the proposed method on two GPU architectures.

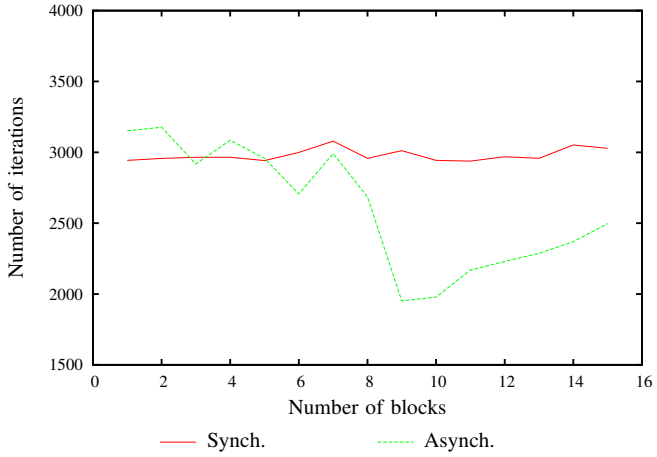

Fig. 7. Related performance of synchronous and asynchronous method.

V. Conclusions

This paper presents a partially asynchronous fixed-point method for PF analysis. The method is designed to be implemented on SIMT architectures such as the GPU. A theoretical result is provided which shows that the method is guaranted to converge, although convergence is generally sub-linear. Simulation results show that: (i) the convergence rate of the proposed method tends to be better at an early stage, which makes it a good complement of the traditional NR method; (ii) the proposed method becomes faster than the traditional NR approach for problems of considerable size; (iii) the proposed method scales well with the number of computing cores in the GPU architecture.

Future works will consider the improvement of the proposed CUDA implementation. In particular, the use of shared memory will be investigated. This addressing convergence of the method in the case where an increasing number of iterations are performed 'asynchronously', i.e., without communicating the results to all threads.

References

[1] F. Milano, *Power system modelling and scripting*. London, UK: Springer, 2010.

[2] P. Simulator, "Version 10.0 scopf," *PVQV, PowerWorld Corporation, Champaign, IL*, vol. 61820, 2005.

[3] F. Milano, "A python-based software tool for power system analysis," in *IEEE Power and Energy Society General Meeting*, 2013, pp. 1–5.

[4] M. Simulink and M. Natick, "The mathworks," 1993.

[5] I. Nagel, "Analog microelectronic emulation for dynamic power system computation," Ph.D. dissertation, École Polytechnique Fédérale de Lausanne, 2013.

[6] S. Talukdar, S. Pyo, and R. Mehrotra, *Designing algorithms and assignments for distributed processing*. Electric Power Research Institute, 1983.

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

[8] D. Bertsekas and D. El Baz, "Distributed asynchronous relaxation methods for convex network flow problems," *SIAM Journal on Control and Optimization*, vol. 25, no. 1, pp. 74–85, 1987.

[9] D. El Baz, P. Spiteri, J.-C. Miellou, and D. Gazen, "Asynchronous iterative algorithms with flexible communication for nonlinear network flow problems," *Journal of Parallel and Distributed Computing*, vol. 38, pp. 1–15, 1996.

[10] S. Xiao and W.-c. Feng, "Inter-block gpu communication via fast barrier synchronization," in *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 1–12.

[11] A. Frommer and D. Szyld, "On asynchronous iterations," *Journal of Computational and Applied Mathematics*, vol. 123, no. 1–2, pp. 201 – 216, 2000, numerical Analysis 2000. Vol. III: Linear Algebra. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S037704270000409X

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

[13] P. Tseng, D. Bertsekas, and J. 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.

[14] 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*, vol. 33, no. 1, pp. 130–150, 1986.

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

[16] M. Marin, "Gpu-enhanced power flow analysis," Ph.D. dissertation, Informatique Perpignan and University college Dublin, 2015, thèse de doctorat dirigée par Defour, David et Milano, Federico. [Online]. Available: http://www.theses.fr/2015PERP0041

[17] "Nvidia's next generation CUDA compute architecture: Kepler GK110," 2012.

[18] S. Collange, "Design challenges of GPGPU architectures: specialized arithmetic units and exploitation of regularity," Ph.D. dissertation, Univ. de Perpignan, Nov. 2010. [Online]. Available: https://tel.archives-ouvertes.fr/tel-00567267

[19] C. Cuda, "Programming guide," *Nvidia Corporation, July*, 2012.

[20] J. Miellou, D. El Baz, and P. Spiteri, "A new class of asynchronous iterative algorithms with order intervals," *Mathematics of Computation of the American Mathematical Society*, vol. 67, no. 221, pp. 237–255, 1998.

[21] T. Davis and E. Palamadai Natarajan, "Algorithm 907: KLU, a direct sparse solver for circuit simulation problems," *ACM Transactions on Mathematical Software*, vol. 37, no. 3, pp. 36:1–36:17, Sep. 2010. [Online]. Available: http://doi.acm.org/10.1145/1824801.1824814

[22] K. He, S. X.-D. Tan, H. Wang, and G. Shi, "GPU-accelerated parallel sparse lu factorization method for fast circuit analysis," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, no. 3, pp. 1140–1150, 2016.