Adéquation Algorithme Architecture pour le calcul d'un AC-Power Flow

Béatrice Thomas, Roman Le Goff Latimier, Abdelhafid Elouardi, Samir Bouaziz, H. Ben Ahmed

Adéquation Algorithme Architecture pour le calcul d'un AC-Power Flow

Béatrice THOMAS[1,2,*], Roman LE GOFF LATIMIER[2], Abdelhafid EL OUARDI[1],
Samir BOUAZIZ[1], Hamid BEN AHMED[2]
[1]Université Paris-Saclay, ENS Paris-Saclay, CNRS, SATIE, 91190, Gif-sur-Yvette, France
[2]SATIE, ENS Rennes, CNRS, 35170, Bruz, France
*beatrice.thomas@ens-rennes.fr

RESUME – Avec l'augmentation de la production décentralisée des énergies renouvelables et de la flexibilité des consommateurs, le besoin de simulations rapides pour estimer les flux dans les réseaux devient crucial, soit en tant que telles, soit couplées à des mécanismes de gestion des flexibilités. Cependant ces simulations sont confrontées à des temps de calculs prohibitifs lorsque les dimensions du problème augmentent. Cet article présente une démarche d'Adéquation Algorithme Architecture pour la résolution du problème de la complexité calculatoire de Power-Flow. Plusieurs algorithmes et implémentations sont réalisées, analysées, optimisées et comparées. Les algorithmes de Newton-Raphson, Gauss-Seidel et de Current Summation sont implémentés avec différents partitionnements sur une architecture CPU-GPU. Des évaluations et comparaisons des temps de traitement, de précision, de stabilité sont présentés dans cet article dans le cas où tous les noeuds sont de type noeud de charge.

AC-Power-Flow, Adéquation Algorithme Architecture, Partitionnement CPU-GPU, calcul parallèle

1. INTRODUCTION

La transition énergétique nécessite d'une part la multiplication des centrales d'énergies renouvelables [1], la généralisation de la flexibilité des consommations [2] et l'électrification de nos moyens de transport [3] qui pourront être agrégées [4]. Toutes ces installations entraîneront des modifications sur le réseau de transport et de distribution.

L'augmentation des productions distribuées et de l'utilisation des véhicules électrique [5] risque de provoquer un flux de puissance ascendant. Ce flux risque de détériorer la qualité de la puissance fournie et de créer des congestions, même sur les réseaux de distributions. Ainsi pour garantir le bon fonctionnement du réseau, celui-ci devra soit être renforcé ce qui consommera beaucoup de matière première, soit être plus finement simulé et contrôlé pour rester dans les limites opérationnelles.

La variabilité des moyens de productions renouvelables et la flexibilité des consommateurs encouragent à réformer la gestion du réseau vers une gestion contrôlée par la demande [6]. Ainsi cet article explorera plus particulièrement la résolution de PowerFlow (PF) dans le cas où uniquement la puissance est imposée dans les noeuds sauf à celui de référence qui fixera le déphasage des tensions.

L'estimation de l'état d'un réseau possède une littérature très riche [7]-[12]. Cependant les temps de calculs deviennent rapidement prohibitifs avec l'augmentation du nombre de bus, notamment lors de l'estimation de la sûreté par des approches *N-X* [12] (hypothèse de la perte de X équipements) qui soulèvent des enjeux combinatoires. De plus ce travail a pour objectif supplémentaire de permettre la résolution d'algorithmes de marché Pair à Pair avec prise en compte du réseau. Cette méthode étant itérative et demandant potentiellement de résoudre un Power Flow à chaque itération, la contrainte sur le temps de résolution est d'autant plus un facteur limitant à ce problème.

Ce verrou scientifique du temps de calcul peut être investigué par une démarche d'adéquation algorithme-Architecture. Cette démarche consiste à étudier et évaluer une variante d'algorithmes sur une variante d'architectures afin de définir un modèle algorithme-architecture qui répond aux exigences en termes de consistance de résultats et de temps de calcul. Dans cet article l'utilisation des *Graphic Processing Units* (GPU), sera particulièrement étudiée de part leur important degré de parallélisme matériel et logiciel. Ce type d'architecture a fait l'objet de plusieurs études dans de nombreux travaux scientifiques pour différents problèmes, comme l'optimal power flow [13], les marchés pair à pair [14] et de power flow [8]-[11].

La suite de cet article est organisée comme suit, tout d'abord la section II présentera le problème de power flow ainsi que les différents algorithmes utilisés pour sa résolution. La partie suivante sera consacrée à l'étude de complexité et à la parallélisation sur GPU. La dernière partie sera consacrée à la présentation et l'analyse des résultats obtenues. Le code ayant permis l'optention de ces résultats est disponible en open source [1].

2. MODÉLISATION

2.1. *Réseau considéré*

Soit un réseau composé de N_b Bus, de L Lignes (chaque ligne l est associée à un unique couple (i,k) de bus). Certains réseaux auront des transformateurs pour adapter la tension, représentés par un déphasage et un rapport de tension comme dans [15]. On représente la relation entre les courants et tension (*from* et *to*) via la matrice des admitances Y_{br}.

$$\begin{pmatrix} i_f \\ i_t \end{pmatrix} = Y_{br} \begin{pmatrix} v_f \\ v_t \end{pmatrix} \tag{1}$$

Avec :

$$Y_{br} = \begin{pmatrix} (y_s + j * \frac{b_c}{2}) \cdot \frac{1}{\tau^2} & -y_s \cdot \frac{1}{\tau e^{-j*\theta_{shift}}} \\ -y_s \cdot \frac{1}{\tau e^{j*\theta_{shift}}} & (y_s + j * \frac{b_c}{2}) \end{pmatrix} \tag{2}$$

A ceci on ajoute des admitances shunts directement reliées aux bus, ce qui nous permet de définir la matrice des admitances Y tel que $I = Y \cdot V = (G + j * B) \cdot V$.

On a pour chaque bus (i) la relation suivante :

$$\begin{aligned} S_i &= P_i + j * Q_i \\ &= V_i \cdot \sum_k Y_{ik}^* \cdot V_k^* \\ &= F_i(v, \phi) \end{aligned} \tag{3}$$

1. https://gitlab.com/satie.sete/adequation_algorithme_architecture_acpf

On obtient donc les puissances actives et réactives en prenant respectivement la partie réelle et la partie imaginaire de l'expression précédente :

$$P_i = V_i * \sum_k V_k(G_{ik}\cos\theta_{ik} + B_{ik}\sin\theta_{ik}) \tag{4}$$

$$Q_i = V_i * \sum_k V_k(G_{ik}\sin\theta_{ik} - B_{ik}\cos\theta_{ik}) \tag{5}$$

On considère ici un noeud de référence permettant de fixer la référence de tension v_0 et ϕ_0. En connaissant la puissance fournie et consommée par tous les autres noeuds, on cherchera à déterminer les tensions des autres noeuds.

2.2. *Algorithmes*

Les algorithmes considérés sont les méthodes de Newton Raphson (**NR**) et Gauss Seidel (**GS**). Elle ont été choisies car elles ont des comportement très différents sur une architecture basée sur un CPU [16] tout en ayant des performances comparables sur des cas de petites tailles. Dans le cas d'un réseau purement radial, un algorithme de *load Flow*, le *backward-formard algorithm* sous sa forme de *Current Sommation method* [15] (**Cur**) sera aussi utilisé ayant de meilleures propriétés de convergence dans ce type de cas. La notation **X G** sera utilisé pour la méthode X lorsqu'elle est sur GPU.

2.2.1. *Newton Raphson NR*

Cet algorithme permet en connaissant l'image d'une fonction d'en trouver l'antécédent. Ici on va donc chercher $E = (\phi, v)$ tel que $S_i - F_i(v,\phi) = 0$ pour tous les bus i.

Pour ce faire, en posant $W = (P,Q)$ le vecteur des puissances et dY la variation de la variable Y, on fait un développement limité au premier ordre autour de 0 et on obtient :

$$dW = Jac \cdot dE \tag{6}$$

$$Jac = \begin{pmatrix} Jac_{1-1} & Jac_{1-2} \\ Jac_{2-1} & Jac_{2-2} \end{pmatrix} = \begin{pmatrix} \frac{\delta P}{\delta \theta} & \frac{\delta Q}{\delta \theta} \\ \frac{\delta P}{\delta V} & \frac{\delta Q}{\delta V} \end{pmatrix} \tag{7}$$

En notant $c(d\theta) = \cos d\theta$ et $s(d\theta) = \sin d\theta$, la Jacobienne peut être définie par bloc, Tab. 1. Il faut ensuite inverser le système définie par cette matrice, (6) pour obtenir la variation de tension nous permettant de nous rapprocher de la solution. Deux méthodes directes ont été testées, une inversion de matrice via un pivot de gauss et une factorisation LU avec pivot (qui a été choisie pour la présentation des résultats). Il est intéressant de noter que pour chaque bloc de la jacobienne le terme Jac_{ij} est non nul si il existe une ligne entre les bus i et j, Tab. 1. Ainsi moins le réseau est maillé, plus la jacobienne sera creuse.

TABLEAU 1. Différents termes de la Jacobienne

Indices	diagonal	non diagonal
1-1	$-Q_i - B_{ii}V_iV_i$	$V_iV_j(G_{ij}s(d\theta) - B_{ij}c(d\theta))$
1-2	$P_i/V_i + G_{ii}V_i$	$V_i(G_{ij}c(d\theta) + B_{ij}s(d\theta))$
2-1	$P_i - G_{ii}V_iV_i$	$-V_iV_j(G_{ij}c(d\theta) + B_{ij}s(d\theta))$
2-2	$Q_i/V_i - B_{ii}V_i$	$V_i(G_{ij}s(d\theta) - B_{ij}c(d\theta))$

2.2.2. *Gauss Seidel GS*

Pour la résolution du problème en utilisant la méthode de Gauss-Seidel il faut remarquer que :

$$\begin{aligned} S_i^* &= P_i - j * Q_i \\ &= V_i^* \cdot \sum_j Y_{ij} \cdot V_j \end{aligned} \tag{8}$$

On aboutit alors à l'équation suivante :

$$\frac{P_i - j * Q_i}{V_i^*} = \sum_j Y_{ij} \cdot V_j \tag{9}$$

Ainsi, en fixant la tension du terme de gauche à partir de celle obtenue à l'itération précédente, on peut résoudre le système obtenu grâce à la méthode de Gauss-Seidel :

$$V_i^{k+1} = \frac{1}{Y_{ii}} \cdot \left(\frac{P_i - jQ_i}{V_i^*} - \sum_{l=1}^{i-1} Y_{il}V_l^{k+1} - \sum_{l=i+1}^{B} Y_{il}V_l^k \right) \tag{10}$$

Pour cet algorithme, la tension sera représentée en coordonnées cartésiennes ce qui permettra d'éviter l'utilisation de fonction trigonométrique. Cependant cet algorithme, contrairement à la méthode de Jacobi, dépend du calcul de l'itération en cours pour être réalisé ce qui n'est pas aisément parallélisable. Ainsi, à chaque itération, on initialise les tensions ainsi pour chaque bus :

$$V_i^0 = \frac{P_i - jQ_i}{Y_{ii}V_i^*} - \sum_{l=i+1}^{B} \frac{Y_{il}}{Y_{ii}} V_l^k \quad \forall i \in B \tag{11}$$

Ici, les calculs sont indépendants et peuvent donc être parallélisés sur les bus. Ensuite, il faut séquentiellement calculer les tensions ainsi (avec p l'itération jusqu'à p = B) :

$$V_i^p = V_i^{p-1} - Y_{ip-1}V_{p-1}^{p-1} \quad \forall i > p \in B \tag{12}$$

Ce calcul peut lui aussi être parallélisé sur les différents bus, par contre le calcul de l'itération suivante ne peut avoir lieux que lorsque celui-ci sera terminé.

2.2.3. *Current Sommation Method Cur*

Cet algorithme n'est utilisable que dans les réseaux de distribution non maillés. Certaines extensions permettent de prendre en compte quelques boucles, mais elles ne seront pas implémentées ici. Dans cette méthode il y a autant de bus que de ligne, ce qui permet de faire en sorte que la ligne k va vers le noeud k. Cette méthode consiste à répéter les 4 étapes suivantes :

1 On calcule le courant pour chaque ligne k :

$$j_b^k = (\frac{s_d^k}{v_k})^* + y_d^k v_k \tag{13}$$

2 *Backward sweep* : on parcourt les lignes k dans l'ordre décroissant avec i l'indice de la ligne précédente ;

$$j_{b,new}^i = j_b^i + j_b^k \tag{14}$$

3 *Forward sweep* : on parcours les lignes $k > 1$ dans l'ordre croissant pour mettre à jour les tensions :

$$v_k = v_i - z_s^k * j_b^k \tag{15}$$

4 on calcule l'erreur sur la tension :

$$\epsilon = v^{iter+1} - v^{iter} \tag{16}$$

Lorsque les tensions ne varient presque plus, l'algorithme est considéré comme ayant convergé. Comme précédemment le calcul est réalisé avec des nombres complexes, la tension sera ici aussi représentée en coordonnées cartésiennes. La parallélisation de cet algorithme a déjà été présentée dans [20] en parallélisant en plus sur plusieurs réseaux en même temps, mais ne sera pas réalisée dans ce travail.

3. ADÉQUATION ALGORITHME-ARCHITECTURE

Le principe de cette approche consiste à traduire un algorithme au niveau comportemental par un premier graphe de données avec un parallélisme maximal. Ce graphe étant progressivement modifié pour explorer différentes options d'implantation de l'algorithme permettant ainsi de guider les choix en termes d'allocation de ressources, de définition des chemins de données, de séquencement des opérateurs et de format de données. Cela passe par une modélisation et une étude qui va explorer l'espace des partitionnements et placements des calculs sur une architecture parallèle. L'objectif est de trouver la meilleure instanciation architecturale. La définition d'un modèle est basée sur l'utilisation d'un ensemble de vecteurs de test logiciels et algorithmiques et une étude analytique des performances.

3.1. Présentation des blocs fonctionnels

La première étape de la démarche Adéquation Algorithme-Architecture est de séparer chaque algorithme en plusieurs blocs fonctionnels. Cette opération permet de remplir plusieurs objectifs :

- analyser les dépendances entre les différentes parties de l'algorithme en terme de séquentialité et de transfert de données ;
- déterminer la complexité en série ou parallèle en calcul et en mémoire et la charge de calcul de chaque bloc par le calcul ou la mesure, en déduire l'intensité algorithmique ;
- identifier les goulots d'étranglement.

La séparation des différents blocs fonctionnels est données dansTab. 2.

TABLEAU 2. Bloc fonctionnel

Bloc	NR	GS	Current
FB Init	✓	✓	✓
FB tension	✓	✓	✓
FB courant			✓
FB puissance	✓	✓	(✓)
FB erreur	✓	✓	✓

L'initialisation est globalement la même entre les différents algorithmes. Les seules différences notables sont le besoin ou non de réaliser les transferts vers le GPU, et pour **GS** on peut pré-calculer les constantes $\frac{1}{Y_{ii}}$ et $\frac{Y_{ij}}{Y_{ii}}$ pour chaque ligne donc une complexité en $O(L)$. L'optimisation de l'étape pour passer de la puissance des agents (P_n) à celle des bus (W_0) a été présentée dans [10]. L'idée étant de stocker dans des vecteurs les correspondances plutôt qu'avec une matrice. Dans notre cas les agents ne sont pas supposés triés, mais le principe reste le même pour le calcul. La complexité est en $O(N)$.

Le calcul de la puissance repose sur les équations (4), (5) qui peuvent être adaptées pour être réalisées avec la tension sous sa forme cartésienne. L'expression est une somme pour chaque bus, sur tous les bus. La complexité est donc en $O(B^2)$. Cependant on peut remarquer que le terme de la somme est non nul si G_{ik} ou B_{ik} est non nul. Ce qui signifie que plutôt que de sommer sur tous les bus, il suffit de sommer sur tous les voisins de chaque bus (et sur lui même). L'équation devient donc :

$$P_i = V_i \sum_{l_i} V_{k(l_i)}(G_{l_i} \cos\theta_{ik(l_i)} + B_{l_i} \sin\theta_{ik(l_i)}) \qquad (17)$$

Pour réaliser cela on utilise un vecteur de taille $B + L$, qui indique pour chaque couple (i, l_i) le voisin $k(l_i)$ correspondant. La complexité devient donc en $O(L)$. La méthode **Cur** ne réalise ce calcul qu'une fois la convergence atteinte pour déterminer les puissances au noeud de référence.

Le calcul de la tension, dans le cadre de l'algorithme de **Cur**, correspond à l'équation (15) et a une complexité en $O(B)$. Pour l'algorithme de **GS** cela correspond à chaque itération à un calcul de (11) et à $B - 1$ calculs de (12). Si on calcule tous les termes la complexité est en $O(B^2)$. Si comme pour le calcul des puissances on ne procède qu'au calcul pour les voisins (lorsque Y_{il} est non nul), la complexité peut être réduite à $O(L)$. Pour l'algorithme de **NR**, le calcul de tension nécessite dans un premier temps le calcul de la jacobienne (7). Ce calcul est en $O(B^2)$ mais peut être réduit en $O(L)$. Ensuite il faut réaliser la factorisation LU de la Jacobienne. Il est important de noter que pour la version CPU, celle-ci est réalisée par la bibliothèque Eigen, la complexité est en $O(B^3)$. Dans les 2 versions (CPU ou GPU), la sparcité n'a pas été prise en compte. Enfin il faut obtenir le déplacement de la tension en résolvant le système, qui est un calcul en $O(B^2)$.

Le calcul du courant de **Cur** correspond aux équations (13) et (14). Ces deux calculs sont en $O(B)$.

Le calcul de l'erreur apparaît sous deux formes dans les algorithmes. Que l'erreur soit sur les puissances $err = ||dW||_\infty$ pour les algorithmes de **NR** et **GS** ou sur la tensions $err = ||dV||_\infty$, c'est la norme infinie qui a été choisie car elle limite l'accumulation d'erreur numérique. Ce calcul a une complexité en $O(B)$.

3.2. Implémentation sur GPU

Nous avons donc vu dans la partie précédente que la complexité des différents algorithmes étaient en $O(L + N)$ sauf la mise à jour de la tension qui est pour **NR** en $O(B^3)$ et celle de **GS** en $O(B^2)$. Cette partie détaillera la parallélisation choisie pour le passage sur GPU, ses avantages et inconvénients théoriques. Pour cela on se basera sur le théorème de Brent [17] pour calculer la complexité théorique en parallèle :

$$C_{para} = O(\frac{o}{p} + t) \qquad (18)$$

avec o le nombre total d'opérations (donc la complexité en série, sauf en cas de redondance des calculs), p le nombre de processeur sur lequel on parallélise, et t le nombre d'étape. Pour la suite on notera $T = N_b * N_{t/b}$ avec T le nombre total de *thread*, N_b le nombre de bloc, et $N_{t/b}$ le nombre de *thread* par bloc. Comme l'algorithme de **Cur** est très peu parallélisable et est déjà efficace il sera gardé sur CPU dans ce travail, le passage sur GPU a déjà été présenté dans [18].

Dans l'initialisation, la parallélisation des différents calculs est assez directe, mais une discussion est possible sur la manière de paralléliser le calcul de W_0. En effet dans [10], chaque thread calcule un bus (donc $p = T = B$), ce qui permet d'utiliser moins de ressources GPU. Cependant on observe que l'on

sérialise la somme des agents sur ce bus, et que les threads sont très divergents entre eux (selon le nombre d'agent par bus). La complexité parallèle est donc de $C_{para} = O(\frac{N}{B} + max(n_b)) = O(max(n_b))$, avec n_b le nombre d'agent sur le bus b. Dans le présent travail, un bloc de thread s'occupe d'un bus ($N_b = B$), ce qui permet d'éviter la divergence entre les threads et de réaliser une réduction parallèle pour la somme. La complexité parallèle est donc de $C_{para} = O(\frac{N}{B*N_{t/b}} + log(N_{t/b}))$. La première méthode est meilleure lorsqu'il y a peu d'agents par bus et que ceux-ci sont répartis de manière égale. Avec l'augmentation du nombre d'agent par bus, la méthode développée devient meilleure , l'optimum étant lorsque $max(n_b) = N_{t/b}$ et que donc $C_{para} = O(log(max(n_b))$.

Les calculs de la puissance (4) et (5) peuvent se réaliser en deux étapes. Dans un premier temps on peut calculer chaque terme de la somme dans un thread $C_{para} = O(1)$, puis on peut faire une réduction $C_{para} = O(log(max(l_i)))$. Dans le calcul intermédiaire sparce, l'accès aux admittances est coalescent, mais ce n'est pas le cas de l'accès aux tensions et déphasages. Pour limiter la séquentialisation des accès mémoires, il a été décidé de faire en sorte que chaque bloc i calcule tous les termes de la somme associée aux puissances P_i et Q_i. Chaque bloc peut charger la totalité du vecteur des tensions E et ainsi l'accès à la mémoire globale est coalescent, et devrait permettre de faire gagner du temps dans le cas où le réseau est très maillé (on n'augmente pas trop le nombre d'accès mémoire global).

Le calcul de la tension pour **GS**, (11), est une réduction. En faisant en sorte d'avoir un bloc par terme i à calculer, la complexité est en $C_{para} = O(log(max(l_i)))$. Ensuite il faut faire $B-1$ calculs (12) qui est juste une opération sur un vecteur, donc une complexité en $O(B)$. Pour **NR**, le calcul de la Jacobienne peut être parallélisé, avec un thread qui calcule les 4 termes (i, j), donc la complexité est en $O(4)$. Tout comme le calcul des puissances, chaque bloc charge tout le vecteur des tensions pour avoir des accès coalescents en mémoire globale. Ensuite il faut réaliser la factorisation LU de la Jacobienne. Celle-ci réalise aussi une recherche de pivot pour améliorer la stabilité numérique de l'opération. Pour chaque colonne, on recherche le pivot $O(log(B))$, on échange les lignes $O(1)$, et ensuite on met à jour la sous-matrice en parallèle avec un bloc qui gère une ligne, et chaque thread qui calcule un terme. La complexité totale est donc en $O(B \cdot log(B))$, on peut paralléliser sur les colonnes et utiliser la sparcité pour encore réduire la complexité, comme dans [21]. Enfin il faut obtenir le déplacement de la tension en résolvant le système, qui est un calcul en $O(B)$.

Bilan : on peut donc conclure que la parallélisation sur GPU devrait permettre de passer d'une complexité en $O(B^3)$ à une complexité en $O(B \cdot log(B))$ pour **NR** et d'une complexité en $O(B^2)$ à une complexité en $O(B)$ pour **GS**. Ce résultat pourrait être amélioré en parallélisant plus la factorisation de **NR**, et en utilisant la méthode de Jacobie (qui n'utilise que l'itération précédente) plutôt que la méthode de **GS**.

4. RÉSULTATS

4.1. Cas d'étude

Tous les cas qui seront résolus dans ce travail suivront les règles suivantes (certains cas ont été modifiés pour cela).

— Les puissances au noeud de référence sont modifiées pour que les puissances soient à l'équilibre (le noeud de référence cherchera juste à compenser les puissances dans les lignes).

— Tous les autres noeuds sont des noeuds PQ.

Ces adaptations nous permettront de nous placer dans le cas d'un réseau principalement composé d'énergie renouvelable et où un mécanisme de marché pour les échanges de puissance a déjà été réalisé et ainsi qu'étudier ce que cela implique. Dans tous les cas, la précision demandée est $||dW||_\infty < 5e^{-4}$.

4.2. Réseau maillé

Dans cette partie, les cas seront ceux définis dans MatPower [19] adaptés et partant de la solution initiale. En utilisant ce logiciel pour résoudre ces cas, le temps de calcul mesuré est constant à environ 1.3s en utilisant la méthode de Newton Raphson. La comparaison avec l'état de l'art nous permet de vérifier qu'il n'y a pas des temps aberrants et d'estimer l'influence du fait de n'avoir que des noeuds PQ sur la convergence. Les résultats sont regroupés dans le tableau suivant, Tab. 3.

TABLEAU 3. Comparaison à l'état de l'art, temps en seconde et S facteur d'accélaration

	NR			GS		
Cas	CPU	GPU	S	CPU	GPU	S
9 [10]	$2e^{-4}$	$2e^{-3}$	0.1	$3e^{-4}$	0.02	0.01
9 [23]	$1e^{-3}$	$1e^{-3}$	0.16	$1e^{-4}$	0.3	$3e^{-4}$
9	$6e^{-5}$	0.01	$7e^{-3}$	$2e^{-4}$	$8e^{-3}$	0.03
30 [10]	$1e^{-3}$	$3e^{-3}$	0.6	$6e^{-4}$	0.02	0.03
30 [23]	$1e^{-3}$	$1e^{-3}$	1	$2e^{-3}$	0.7	$3e^{-3}$
30 [24]	$1e^{-3}$	$3e^{-3}$	*0.3*	$8e^{-3}$	0.15	*0.06*
30	$2e^{-4}$	0.02	$8e^{-3}$	$4e^{-3}$	0.03	0.1
118 [10]	0.02	0.02	1.4	$6e^{-3}$	0.05	0.13
118 [22]			2.4			
118 [23]	0.3	0.2	1.6	0.3	7	0.05
118 [24]	$2e^{-3}$	$6e^{-3}$	*0.33*	0.09	0.40	*0.21*
118	$3e^{-3}$	0.06	0.05	*	*	< 0.6
300[10]	0.4	0.11	3.3	0.05	0.12	0.43
300[23]	4.7	2.7	1.7	0.34	7.3	0.06
300 [24]	0.01	0.02	*0.52*	2.5	5.6	*0.45*
300	0.1	0.7	0.16	*	*	1.5
2383[10]	87	10	8.5	8	0.8	9.6
2383	*	*	1.7	*	*	9

Il est important de noter que les cas indiqué par "*" ne convergent pas. Ceci est dû au fait que l'on n'ait que des noeuds PQ (cela peut se vérifier grâce à MatPower). C'est pourquoi le temps n'est pas indiqué car il dépend uniquement du nombre maximal d'itérations autorisées. Cependant on peut remarquer que si le CPU est plus rapide pour les petits cas le GPU devient plus rapide lorsque la taille augmente.

4.3. Analyse du passage à l'échelle

Dans cette section seront présentés les effets de l'augmentation de la taille du cas d'étude sur les propriétés de convergence et le temps de convergence dans des réseaux de distributions radiaux. La structure du réseau sera aussi variable pour étudier ses potentiels effets.

4.3.1. *Création du cas*

Pour générer un réseau aléatoirement, on part du bus 0, puis tant qu'il reste des bus à ajouter au réseau, on tire aléatoirement le bus auquel il sera raccordé. Pour cela on garde une liste de noeud que l'on considère comme en bout de branche et avant de choisir le bus, on tire aléatoirement si ce bus est en bout de branche (et le bout de branche devient le nouveau bus), ou si le bus est quelconque (et dans ce cas là le nouveau bus s'ajoute dans la liste). La probabilité du choix entre les deux options est déterminée par le rapport entre deux contraintes N_{branch} et N_{deep}. La première est le nombre maximal de branches du réseau. Lorsqu'elle est atteinte, elle force le fait de continuer une branche. La deuxième est la distance d'un bus au noeud d'origine exprimée en nombre de bus. Si un bus a atteint la valeur maximale fixée, on ne peut plus y relier d'autres bus. Ainsi en faisant varier ces deux contraintes on peut faire apparaître 4 topologies de réseaux de distribution type (entre parenthèse se trouve les paramètres pour la simulation) :

- un réseau avec autant de branches que de profondeur ($N_{deep} = \sqrt{2 \cdot B}$, $N_{branche} = \sqrt{2 \cdot B}$) ;
- un réseau très large et pas profond avec beaucoup de branches ($N_{deep} = 0.3 \cdot \sqrt{B}$, $N_{branche} = B$) ;
- un réseau très profond avec peu de branches ($N_{deep} = B$, $N_{branche} = 0.3 \cdot \sqrt{B}$) ;
- un réseau non contraint ($N_{deep} = B$, $N_{branche} = B$).

De manière générale, plus le réseau sera contraint plus le réseau sera "équilibré" dans le sens où toutes les branches auront la même profondeur. Chaque ligne créée est de type *94-AL1/15-ST1A* à 400V et de longueur $l_{long} = 1 \pm 0.5m$ tirée aléatoirement.

Une fois que le réseau est créé, il faut générer aléatoirement les agents. On tire donc aléatoirement leur puissance voulue P_n et le bus où l'agent est placé. Les agents ne sont pas triés par ordre de bus dans cette version. Lors de la génération du cas, on définira le nombre d'agents en fonction du nombre de bus, avec ainsi quatre possibilités : très peu d'agents ($N = 0.2 \cdot B$), moins d'agents que de bus ($N = 0.5 \cdot B$), autant d'agents que de bus ($N = B$) et plus d'agents que de bus ($N = 2 \cdot B$).

La numérotation des cas (de 1 à 16) commence par faire varier le nombre d'agent puis le type de réseau, par exemple les cas 4, 8, 12, 16 sont ceux où le nombre d'agents est maximal, et de 9 à 12 sont les cas où on a peu de branches.

4.3.2. *Étude de la convergence*

Dans notre cas, la convergence a trois états : convergence ($err < \epsilon$), divergence ($err > 100 \cdot \epsilon$) ou non-convergence. Comme la génération de cas est aléatoire et peut mener à des cas non faisables, il a été décidé pour éviter des calculs trop long, que si la méthode de référence **Cur** divergeait sur un cas, les méthodes de **GS** ne seraient pas calculées.

On peut donc étudier la faisabilité des cas en fonctions de nos paramètres, et le taux de convergence des différentes méthodes. On peut remarquer dans Fig. 1 que lorsque l'on a contraint le nombre de branche, cela a conduit à des cas non faisables. On peut aussi remarquer qu'augmenter le nombre d'agent et de bus augmente le risque d'avoir des cas divergents. Enfin en regardant uniquement les cas où toutes les méthodes ont convergé, on peut étudier les écart de puissance pour vérifier que toutes

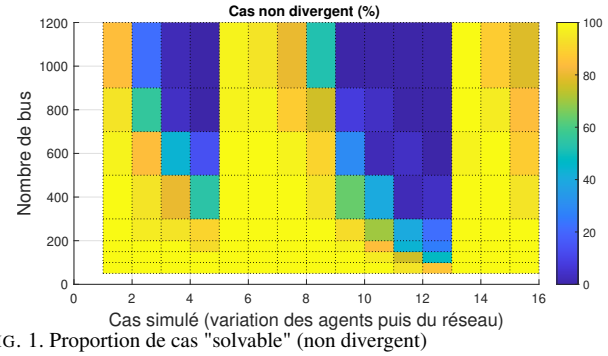

FIG. 1. Proportion de cas "solvable" (non divergent)

TABLEAU 4. Taux de convergence parmi les cas faisables, et écart max et médian de puissance sur les cas convergents

Méthode	Cur	NR	NR G	GS	GS G
Conver (%)	99.5	99.77	99.79	81.89	81.39
P^{max}_{ecart} (%)	ref	2	3	4	5
Q^{max}_{ecart} (%)	ref	16	20	104	122
P^{med}_{ecart} (%)	ref	0.03	0.03	0.21	0.22
Q^{med}_{ecart} (%)	ref	$6e^{-3}$	$7e^{-3}$	0.09	0.1

les méthodes convergent vers le même point. Les valeurs max dans Tab. 4 indiquent dans certains cas que la solution n'est pas la même. Cependant la valeur médiane montre que les résultats sont très proches dans la majorité des cas. La non convergence des méthodes **Cur** et **NR** peut s'expliquer par le fait que les calculs sont en flottant simple précision.

4.3.3. *Etude de la complexité*

Pour étudier la complexité on calcule la moyenne des temps pour tous les cas en fonction du nombre de bus et de la méthode. En posant η l'augmentation du temps par rapport au nombre de bus d'une méthode donnée, cette variable peut s'exprimer ainsi pour une variation entre B_1 et B_2 bus et en notant $O(B^{\alpha_{poly}})$ la complexité polynomiale des méthodes :

$$\eta = \frac{t_2}{t_1} = (\frac{B_2}{B_1})^{\alpha_{poly}} \tag{19}$$

Ainsi la complexité mesurée peut être calculé ainsi :

$$\alpha_{poly} = \frac{log(\eta)}{log(B_2) - log(B_1)} \tag{20}$$

L'ensemble des calculs précédents peut aussi être fait avec des temps moyens par itération pour retirer l'impact du nombre d'itérations sur le temps de calcul et donc sur la complexité. Les résultats sont regroupés dans le Tab. 5.

On peut constater sur la Fig. 2 que l'augmentation du temps de calcul liée à celle du nombre de bus est moindre pour les méthodes sur GPU que leurs équivalents sur CPU. Cependant même avec des cas de plus de 1200 bus, la méthode du current flow reste bien plus efficace (30× plus rapide), même si l'augmentation du temps est bien plus importante que ce qui est attendu. De plus le fait d'avoir un réseau de distribution radial ne permet pas de profiter au maximum des caractéristiques du GPU. En effet dans cette configuration avec autant de ligne que de bus, chaque bus a très peu de voisin ce qui a deux ef-

TABLEAU 5. temps et complexité mesurés et complexité théorique

Méthode	Cur	NR	NR G	GS	GS G
t_{50} (s)	$3e^{-4}$	0.03	$7e^{-5}$	0.01	0.04
t_{1200} (s)	0.06	2.17	2.18	103	20
η	896	6 721	77	10 747	482
α_{poly}	2.1	2.8	1.4	2.9	1.9
η_{iter}	289	3 703	42	564	25
$\alpha_{poly-iter}$	1.8	2.6	1.2	2.0	1.0
C_{the}	B	B^3	$Blog(B)$	B^2	B

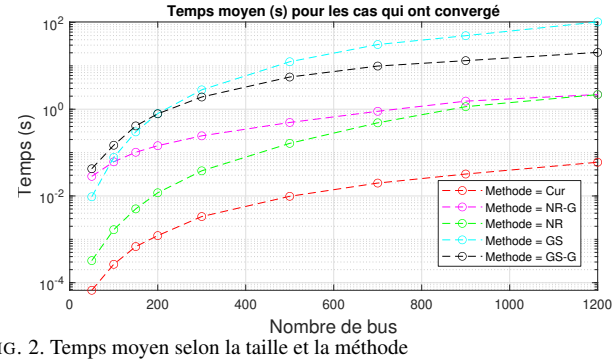

FIG. 2. Temps moyen selon la taille et la méthode

fets. Tout d'abord l'utilisation de la sparsité sur CPU est d'autant plus efficace car on a une complexité en $O(L) = O(B)$. Et dans un second temps, comme on utilise un bloc par bus et que l'on parallélise dans un bloc sur les voisins, on utilise beaucoup de blocs et ceux-ci sont peu actifs (le minimum étant un warp ou demi warp à respectivement 32 ou 16 threads). Les performances du GPU devraient donc être bien meilleures dans des réseaux maillés.

5. CONCLUSION

Cet article présente une démarche Adéquation Algorithme Architecture pour la résolution du AC PowerFlow. Les algorithmes de Newton Raphson, Gauss-Seidel et de *Current Flow* ont été étudiés pour un partitionnement sur une architecture CPU-GPU. L'ensemble a été évalué sur des jeux de données fixes et aléatoires, sur des réseaux de distribution et de transport après que la complexité théorique ait pu être étudiée.

Dans le cas de réseau non maillé, les tests réalisés mettent en évidence que la méthode du Current Flow est la meilleure même lorsqu'elle n'est pas parallélisée. De même dans le cas où le réseau est peu maillé, la méthode de Newton Raphson sur CPU reste la plus efficace jusqu'à une très grande dimension. Newton Raphson sur GPU devient intéressante à très grande dimension avec beaucoup de liens. Dans notre cas où tous les noeuds sont PQ, la méthode de Gauss-Seidel nécessite bien trop d'itérations et a de trop grande difficulté pour converger pour être une solution viable quelque soit la dimension du problème. Le passage sur GPU accélère grandement la simulation et réduit la complexité.

D'autres optimisations peuvent être réalisées pour la résolution (par exemple prendre en compte la sparcité lors de la factorisation LU). De plus amples évaluations doivent être réalisés pour déterminer la meilleure organisation et taille des blocs et thread selon la taille et le taux de maillage du réseau.

6. RÉFÉRENCES

[1] RTE. Conditions and Requirements for the Technical Feasibility of a Power System with a High Share of Renewables in France Towards 2050

[2] Bussar C et al. Large-scale integration of renewable energies and impact on storage demand in a European renewable power system of 2050—Sensitivity study, Journal of Energy Storage, Vol 6, 2016, Pages 1-10,

[3] Richardson D B, Electric vehicles and the electric grid : A review of modeling approaches, Impacts, and renewable energy integration, Renewable and Sustainable Energy Reviews, Volume 19, 2013, Pages 247-254

[4] Evangelopoulos V A, Kontopoulos T P, Georgilakis P S. Heterogeneous aggregators competing in a local flexibility market for active distribution system management : A bi-level programming approach, IJEPES, Vol 136, 2022

[5] F. G. Venegas, M. Petit, and Y. Perez, "Active integration of electric vehicles into distribution grids : Barriers and frameworks for flexibility services, "Renewable and Sustainable Energy Reviews, vol. 145,p. 111060, 2021.

[6] G. Strbac, "Demand side management : Benefits and challenges," Energy Policy, vol. 36, no. 12, pp. 4419–4426, 2008

[7] Zhang X, Flueck A J, Nguyen C P. Agent-based distributed volt/var control with distributed power flow solver in smart grid. IEEE Transactions on Smart Grid, 7(2), 600-607.2015

[8] X. Su, C. He, T. Liu and L. Wu. Full Parallel Power Flow Solution : A GPU-CPU-Based Vectorization Parallelization and Sparse Techniques for Newton–Raphson Implementation. in IEEE Transactions on Smart Grid, vol. 11

[9] Araújo I, Tadaiesky V, Cardoso D, Fukuyama Y, Santana Á . Simultaneous parallel power flow calculations using hybrid CPU-GPU approach. IJEPES, 105, 229-236. 2019

[10] Singh, J, Aruni I. (2010, December). Accelerating power flow studies on graphics processing unit. In 2010 Annual IEEE India Conference (INDICON) (pp. 1-5). IEEE.

[11] Sooknanan D J, Joshi A . GPU computing using CUDA in the deployment of smart grids. In 2016 SAI Computing Conference (SAI) IEEE. 2016, July

[12] Donnot B, Guyon I, Schoenauer M, Marot A, Panciatici P. Anticipating contingengies in power grids using fast neural net screening. In 2018 International Joint Conference on Neural Networks (IJCNN). IEEE.

[13] Roberge V, Tarbouchi M, Okou F. Optimal power flow based on parallel metaheuristics for graphics processing units. Electric Power Systems Research, Volume 140, 2016, Pages 344-353, ISSN 0378-7796,

[14] Thomas B, Le Goff Latimier R, El Ouardi A, Ben Ahmed H, Bouaziz Samir. Optimization of a peer-to-peer electricity market resolution on GPU. 2022 CISTEM

[15] Zimmerman R D, Murillo-Sanchez C E. Matpower User's Manual, Version 7.1. 2020. [Online].

[16] Abaali H, Talbi T, Skouri R. Comparison of Newton Raphson and Gauss-Seidel methods for power flow analysis. IJEPE, 12(9), 627-633 (2018)

[17] Brent R. P. 1974. The Parallel Evaluation of General Arithmetic Expressions. J. ACM 21, 2 (April 1974), 201–206.

[18] Ablakovic D, Dzafic I, Kecici S. Parallelization of radial three-phase distribution power flow using GPU. In 2012 3rd IEEE PES Innovative Smart Grid Technologies Europe (ISGT Europe) (pp. 1-7). IEEE.

[19] Zimmerman R D et al. "Matpower : Steady-State Operations, Planning and Analysis Tools for Power Systems Research and Education,". IEEE Transactions on Power Systems, vol. 26, no. 1, pp. 12–19, Feb. 2011.

[20] Ablakovic D, Dzafic I, Kecici S. Parallelization of radial three-phase distribution power flow using GPU. In 2012 3rd IEEE PES Innovative Smart Grid Technologies Europe (ISGT Europe) (pp. 1-7). IEEE.

[21] Ren L, Chen X, Wang Y, Zhang C, Yang H. Sparse LU factorization for parallel circuit simulation on GPU. In Proceedings of the 49th Annual Design Automation Conference (pp. 1125-1130).

[22] Garcia N. Parallel power flow solutions using a biconjugate gradient algorithm and a Newton method : A GPU-based approach. In IEEE Pes General Meeting (pp. 1-4). IEEE.

[23] Guo C, Jiang B, Yuan H, Yang Z, Wang L, Ren S. Performance comparisons of parallel power flow solvers on GPU system. In 2012 IEEE International Conference on Embedded and RTCSA (pp. 232-239). IEEE.

[24] Roberge V, Tarbouchi M, Okou F. Parallel power flow on graphics processing units for concurrent evaluation of many networks. IEEE Transactions on Smart Grid, 8(4), 1639-1648.