Adéquation Algorithme Architecture pour la gestion des réseaux électriques

Béatrice Thomas

To cite this version:

Béatrice Thomas. Adéquation Algorithme Architecture pour la gestion des réseaux électriques. Calcul parallèle, distribué et partagé [cs.DC]. Université Paris-Saclay, 2024. Français. NNT : 2024UPASG104 . tel-04958343

HAL Id: tel-04958343
https://theses.hal.science/tel-04958343v1

Submitted on 20 Feb 2025

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

universite
PARIS-SACLAY

Adéquation Algorithme Architecture pour la gestion des réseaux électriques

Hardware-Software codesign for the electrical grid management

Thèse de doctorat de l'université Paris-Saclay

École doctorale n°580 : sciences et technologies de l'information et de la communication (STIC)
Spécialité de doctorat : Informatique
Graduate School : Informatique et sciences du numérique (ISN)
Référent : Faculté des Sciences d'Orsay

Thèse préparée dans l'unité de recherche **SATIE** (Université Paris Saclay, ENS Paris Saclay CNRS),
sous la direction de **Abdelhafid EL OUARDI**, Maitre de Conférence HDR,
le co-encadrement de **Hamid BEN AHMED**, Professeur des Universités
et de **Roman LE GOFF LATIMIER**, Maître de Conférence

Thèse soutenue à Rennes, le 18 décembre 2024, par

Béatrice THOMAS

Composition du Jury

Membres du jury avec voix délibérative

M. Steven DERRIEN Professeur des universités, LabSTIC, Université de Bretagne Occidentale	Président
M. Abdessamad AIT EL CADI Professeur des universités, LAMIH, INSA Haut de France.	Rapporteur & Examinateur
M. Zacharie DE GREVE Associate Professor (équiv HDR), Université de Mons	Rapporteur & Examinateur
Mme. Zita VALE Full Professor, Polytechnic Institute of Porto	Examinatrice

THESE DE DOCTORAT

NNT : 2024UPASG104

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

université PARIS-SACLAY

Titre : Adéquation Algorithme Architecture pour la gestion des réseaux électriques

Mots clés : marché P2P endogène, partitionnement CPU-GPU, calcul parallèle, évaluation de performance

Résumé : L'augmentation de la production renouvelable décentralisée nécessaire à la transition énergétique complexifiera la gestion du réseau électrique.

Une riche littérature propose de décentraliser la gestion pour éviter la surcharge de l'opérateur central lors de la gestion en temps réel. Cependant la décentralisation exacerbe les problèmes de passage à l'échelle lors des simulations préliminaires permettant de valider les performances, la robustesse de la gestion ou le dimensionnement du futur réseau. Une démarche Adéquation Algorithme Architecture a été suivie dans cette thèse pour un marché pair à pair pour résoudre le problème de passage à l'échelle lors de la simulation sur une architecture matérielle de calcul unique.

L'influence des agents sur le réseau en grande dimension ne pouvant plus être négligée, l'étude a porté sur un marché endogène pair à pair.

Nous avons étudié la complexité calculatoire de différents algorithmes. Des méthodes d'optimisation de temps de traitements sur des architectures type GPU ont été développées.

L'évaluation des performances, en termes de temps de traitement et de convergence, a été réalisée.

Ainsi, un modèle de calcul parallèle sur une architecture GPU a apporté une accélération substantielle lorsque la précision n'est pas critique. Une implémentation optimisée sur une architecture GPU a permis de réduire de plus de 98% les temps de simulation d'un marché sans contraintes réseau. Comparé à un modèle de calcul sur une architecture conventionnelle type PC, la démarche d'adéquation algorithme-architecture a permis de définir un modèle de calcul sur GPU 1000 fois plus rapide lors de la simulation d'un DC-marché endogène et 10 fois plus rapide sur un marché AC-endogène sur un réseau radial.

Les résultats de cette thèse ont permis de consolider l'étude menée sur les aspects algorithmiques comme sur les aspects d'architectures matérielles pour l'accélération des simulations de réseaux électriques sur des architectures parallèles.

Title : Hardware-Software codesign for the electrical grid management

Keywords : Endogenous P2P market, CPU-GPU Partitioning, Parallel computing, Performance evaluation

Abstract : The growth of distributed energy resources raises the challenge of scaling up network management algorithms. This difficulty may be overcome in operating conditions with the help of a rich literature that frequently calls upon the distribution of computations. However, this issue persists during preliminary simulations validating the performances, the operation's safety, and the infrastructure's sizing. A hardware-software co-design approach is conducted here for a Peer-to-Peer market to address this scaling issue while computing simulations on a single machine. With the increasing number of distributed agents, the impact on the grid cannot be neglected anymore. Thus, this work will focus on an endogenous market.

The mapping between several algorithms and different partitioning models on Central and Graphic Processing Units (CPU-GPU) has been conducted. The complexity and performance of these algorithms have been analysed on CPU and GPU. The implementations have shown that the GPU is more numerically unstable than the CPU. Nevertheless, when precision is not critical, GPU gives substantial speedup. Thus, markets without grid constraints are 98% faster on GPU. Even with the grid constraints, the GPU is 1000 times faster with the DC hypothesis and ten times faster on the AC radial grid. This dimension-dependent acceleration increases with the grid size and the agent's count.

Remerciements

À tout ce qui considère que la thèse est un travail de solitaire, je répondrai ceci. En effet, j'étais souvent seule devant mon ordinateur avec pour seule discussion celle avec le compilateur. Cependant, je ne me suis jamais sentie seule. Je savais que je pouvais toujours compter sur mes encadrants pour répondre à mes questions. Je tiendrais donc à remercier Abdelhafid El Ouardi de m'avoir fait découvrir la méthode Adéquation Algorithme Architecture qui a permis de fusionner deux de mes centres d'intérêt (les réseaux électriques et la programmation) en un sujet de recherche. Merci aussi à Hamid Ben Ahmed, malgré ton emploi du temps surchargé, tu as su apporter tes remarques, toujours extrêmement pertinentes. Et enfin, je tiens à remercier Roman Le Goff Latimier, toujours présent, que ce soit au travail à l'ENS ou lors des conférences. Toujours disponible pour répondre à mes questions quand je débarquais à l'improviste dans ton bureau. Merci à ta patience, pour comprendre ce que je racontais alors que mon esprit voulait aller plus vite que ma bouche.

Je tiens aussi à remercier les deux stagiaires Lemine et Eva qui ont travaillé à l'implémentation de respectivement le Power Flow et l'optimal power flow sur FPGA. Même si leur travail n'a pas pu apparaitre dans ce manuscrit, ils ont pu défricher une partie conséquente de la démarche Adéquation Algorithme Architecture. Un merci aussi à tous les doctorants et chercheur du site, dont les conversations très intéressantes m'ont permis d'étoffer mes journées. Ils sont bien trop nombreux pour en faire une liste, mais vous êtes tous dans mon esprit.

Je remercie aussi les membres du jury, M. Steven DERRIEN et Mme. Zita VALE d'avoir pris sur le temps pour venir à ma soutenance de thèse et juger mon travail. Je remercie aussi les rapporteurs, M. Abdessamad AIT EL CADI et M. Zacharie DE GREVE qui ont dû lire cette (longue) thèse .

Un grand merci aussi à toute ma famille. Même si mon sujet était complètement obscur, ils restaient à l'écoute. C'est grâce à eux si je suis celle que je suis aujourd'hui et que j'ai été capable d'accomplir ce travail. À mes amis, Arthur, Axel, Colin, Simon^2, Lucas, même si vous êtes à distance, on a pu s'amuser, discuter et rager ensemble. Et finalement, merci à mon compagnon Youen. Je ne compte pas le nombre de fois où je l'ai interrompu pour lui poser des questions sur quelque chose qui me bloquait. Il n'avait pas toujours la réponse, mais à chaque fois, cette discussion a pu m'aider.

Et merci à toi, lecteur du futur, de t'intéresser à ce sujet et de continuer à faire vivre mon travail.

Bref, cette thèse c'est la mienne, mais c'est surtout la nôtre. Merci à tous.

Introduction

La transition énergétique pour atteindre la neutralité carbone en 2050 nécessite d'une part la multiplication des centrales d'énergies renouvelables et d'autre part la généralisation de la flexibilité des consommations. Ainsi, les énergies renouvelables, dont la production sera majoritairement distribuée, intermittente et imparfaitement prévisible, vont impacter de manière significative le réseau et sa gestion en devant constamment ajuster les flexibilités disponibles pour équilibrer le réseau. De plus, la généralisation de l'électrification du transport, des moyens de stockage et de l'automatisation de la gestion des particuliers va permettre la multiplication des agents dits "actifs". La coordination de tous ces agents nécessite une modification des mécanismes de gestions du réseau. Cette résolution risque d'être rendue encore plus difficile par la diversité des agents distribués (consommateur ou producteur) qui pourraient être agrégés pour avoir un effet significatif. Enfin, ces différentes modifications risquent de provoquer un flux de puissance ascendant du réseau de distribution vers le réseau de transport. Ce flux risque de détériorer la qualité de la puissance fournie et de créer des congestions, même sur les réseaux de distribution. 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. Le réseau électrique fera donc face à de multiples challenges : il sera utilisé de manière plus intensive par de nombreux acteurs et d'une manière plus complexe pour gérer la flexibilité et les intermittences.

Jusqu'à maintenant, les marchés de l'électricité et les différents mécanismes ont pu prouver leur aptitude à gérer des agents variés en garantissant l'équilibre entre la production et la consommation. Cet équilibre est absolument nécessaire pour éviter tout effondrement du réseau. Ainsi, plusieurs marchés sont tenus de plus en plus proches de l'heure réelle pour ajuster au fur et à mesure l'équilibre consommation-production avec la diminution des incertitudes sur les prévisions. Actuellement, un opérateur centralisé rassemble les offres et demande pour réaliser le marché et la simulation du réseau pour déterminer les congestions. Cependant, avec l'augmentation du nombre d'agents et de la complexité de leur gestion, cet agent central risque de faire face à un verrou algorithmique, en matière de complexité de calcul et de communication.

Cette situation permet l'émergence d'une riche littérature cherchant à concevoir de nouvelles règles de gestion pour opérer de manière sûre et efficace le réseau.

En effet, le mécanisme de gestion actuel faisant face à des difficultés de passage à l'échelle devant l'augmentation du nombre d'agents participant, de nombreux efforts sont dédiés à la distribution du contrôle. Le problème à grande échelle est divisé en une multitude de problèmes locaux qui sont coordonnés entre eux. Ainsi, chaque agent réaliserait une partie du calcul jusqu'à la convergence vers la solution globale en étant coordonné par des échanges d'informations. De plus, la décentralisation des calculs devrait permettre de contrôler au mieux la flexibilité des particuliers/agents tout en préservant leur vie privée. En effet, la décentralisation ou distribution des calculs permet à chaque agent de gérer sa propre flexibilité. Ainsi, des données sensibles, telles que les utilisations des différents appareils électroménagers ou la consommation en temps réel, ne sont pas transmises à une entité à laquelle l'agent pourrait ne pas faire confiance.

Parmi toutes les architectures des résolutions distribuées, un marché Pair à Pair (P2P) est complètement décentralisé du moment où il n'y a plus aucun agent central, mais seulement des communications entre les pairs. Il permet l'introduction de fonctionnalités spécifiques, comme les préférences hétérogènes entre les agents (selon différents critères, comme le type de production, le niveau de richesse, la distance...). L'un de ses principaux atouts est qu'il peut être considéré comme une généralisation formelle de tous les autres types d'architectures réseau : centralisée, hiérarchique, communautaire, avec gestion de l'équilibre... Ainsi n'importe quel type d'organisation réseau peut être représenté par un marché P2P en changeant les liens entre les agents et leur comportement.

Ces approches ouvrent de nombreuses voies pour résoudre opérationnellement et en temps réel les problèmes de gestion dont la complexité sera sans commune mesure avec l'actuelle. Cependant, avant tout déploiement réel, de nombreuses simulations sont nécessaires pour répondre entre autres aux défis du marché P2P. Ainsi des simulations préliminaires - pour les chercheurs, les gestionnaires de réseau ou les autorités normatives - sont nécessaires pour de nombreux objectifs. De manière non exhaustive, ce sera nécessaire pour régler de manière judicieuse des paramètres pour les mécanismes de gestion décentralisée et ainsi concevoir les nouvelles règles de ce marché, vérifier les performances, anticiper les nouvelles infrastructures à construire et garantir la robustesse des solutions proposées en cas de faute dans le réseau de communication ou de puissance. Ces simulations devront représenter le réseau réel et ainsi être réalisées dans des cas d'études à dimension réaliste, i.e. de grande dimension. Il faudra aussi étudier les effets du passage à l'échelle sur les algorithmes de gestion. De longues périodes devront être simulées pour conserver la cohérence temporelle des données (typiquement le stockage) tout en vérifiant le comportement durant des évènements extrêmes. Toutes ces simulations ayant lieu avant le déploiement, elles ne peuvent pas encore se reposer sur la future puissance de calcul distribuée de tous les agents. Ces simulations étant

faites de manière centralisée sur une unique cible matérielle ou cluster, des temps de calcul prohibitifs réapparaissent avec l'augmentation du nombre d'agents. Des études sur des réseaux de dimension réelle deviennent impossibles.

L'objectif de ce travail est donc de résoudre le verrou scientifique du temps de calcul pour la simulation de gestion de réseau décentralisé à grande échelle. Plus particulièrement, le problème traité sera celui d'un marché pair à pair endogène qui permet de déterminer les échanges d'énergies optimaux des agents distribués tout en considérant les contraintes réseau. Mais pour résoudre ce problème, on étudiera aussi le marché pair à pair sans contraintes réseau, le problème de Power Flow et d'Optimal Power Flow. Pour accélérer les calculs, des architectures matérielles accélératrices seront utilisées afin de pouvoir exploiter le parallélisme inhérent des réseaux électriques et des algorithmes décentralisés. Cependant, les spécificités matérielles de ces architectures doivent être prises en compte pour l'optimisation des traitements et l'augmentation des performances en manière de calcul et de convergence. Il est donc essentiel d'étudier l'interaction entre l'algorithme et l'architecture matérielle de calcul. Le choix d'une architecture (type GPU, CPU ou FPGA) est justifié par son adaptation à des problèmes parallèles et des simulations à grande échelle nécessitant des calculs intensifs (vectoriels et matriciels) et permettant des optimisations mémoires (d'autres contraintes peuvent être considérées, notamment celle relative à la consommation d'énergie). Les gains en performance peuvent être linéaires ou même exponentiels en fonction des algorithmes étudiés et des méthodes d'optimisations déployées. Pour cela, une démarche d'Adéquation Algorithme Architecture sera suivie,

Ainsi, dans ce manuscrit de thèse, le réseau sera, dans le chapitre 1, modélisé et un état de l'art sur les différents mécanismes de gestion et de simulation et de leur accélération sera présenté. Ensuite, dans le chapitre 2, la démarche suivie d'Adéquation Algorithme Architecture sera détaillée avec la présentation des différents algorithmes et des architectures matérielles dédiées pour l'accélération des calculs. Les algorithmes appliqués aux problèmes seront ensuite présentés et validés, (chapitre 3, leur complexité sera analysée et leurs traitements seront optimisés dans le chapitre 4. Enfin, les résultats comportant les effets de la méthode suivie, les optimisations réalisées et les évaluations de performance seront présentés et discutés dans le chapitre 5 avant de pouvoir conclure et présenter les perspectives de ces travaux de thèse.

Table des matières

1 État de l'art et problématique scientifique 27
1 Réseau Électrique : la gestion en pratique 27
1.1 Fonctionnement actuel d'un réseau électrique 27
1.2 Pistes de recherche de la littérature 38
1.3 Périmètre de l'étude 46
2 Les puissances de calculs mises en jeu 54
2.1 L'accélération matérielle 54
2.2 L'accélération des calculs dans le domaine du réseau électrique 58
3 Conclusion sur l'état de l'art et contribution 61
3.1 Temps de calcul de la gestion du réseau électrique 61
3.2 Contributions 65
3.3 Organisation du manuscrit 66

2 Méthodologie, métriques et critères d'évaluation 69
1 Introduction 69
2 Méthodologie A3 70
2.1 La Roofline 73
3 Choix des métriques 74
3.1 Métriques algorithmiques 75
3.2 Métriques architecturales 77
4 Justification des choix des algorithmes 79
4.1 Algorithmes de décentralisation 79
4.2 Problème non convexe 86
4.3 Optimisation quadratique 90
4.4 Conclusion 94
5 Choix des architectures et outils de programmation 95
5.1 Architectures de calcul parallèle multicœurs 95
5.2 Architectures pour l'accélération de calcul 98
5.3 Architecture sélectionnée 100
5.4 Conclusion 103
6 Choix des jeux de données 103
6.1 Cas pour vérifier la convergence 105

6.2 Cas pour comparaisons 107
6.3 Cas pour évaluer la complexité 110
6.4 Synthèse des cas d'études 112
7 Conclusion 113

3 Validation fonctionnelle et passage à l'échelle 115
1 Introduction 116
1.1 Problème d'optimisation 116
1.2 Méthodes de résolutions 117
2 Power Flow 118
2.1 Mise en forme du problème 118
2.2 Approximation DC 119
2.3 Newton-Raphson NR 120
2.4 Gauss Seidel GS 121
2.5 Méthode Backward-forward 122
2.6 Validation fonctionnelle 123
3 Marché pair à pair 126
3.1 Mise en forme du problème 126
3.2 Résolution centralisée via OSQP 127
3.3 Décentralisation par ADMM 128
3.4 Décentralisation par PAC 132
3.5 Validation fonctionnelle 137
4 Optimal Power Flow 141
4.1 Mise en forme du problème 141
4.2 Résolution de référence 143
4.3 Déplacement de la contrainte des puissances injectées 146
4.4 Validation fonctionnelle 148
5 Marché endogène 149
5.1 Résolution avec relaxation d'un PF 150
5.2 Résolution par consensus avec un OPF 157
5.3 Résolution directe 159
5.4 Validation fonctionnelle 161
6 Conclusion 164

4 A3 sur une architecture de calcul CPU-GPU 167
1 Introduction 168
2 Partitionnement en bloc fonctionnel 168
2.1 Marché pair à pair 169
2.2 Power Flow 173
2.3 Optimal Power Flow 176
2.4 Marché endogène 178

3 Étude de complexité . . . 182
3.1 Vision globale . . . 182
3.2 Marché pair à pair . . . 185
3.3 Power Flow . . . 187
3.4 Optimal Power Flow . . . 191
3.5 Marché endogène . . . 195
4 Réécriture algorithmique pour partitionnement sur CPU-GPU . . . 197
4.1 Lien entre le langage de programmation et l'architecture de calcul . . . 197
4.2 Optimisation générale . . . 200
4.3 Prise en compte de la sparsité . . . 201
4.4 Problème de partage . . . 203
4.5 Stockage de la sensibilité . . . 208
4.6 Fonction avec dépendance . . . 215
4.7 Parcours des bus . . . 216
4.8 Calcul de la tension . . . 217
5 Conclusion . . . 219

5 Performances atteintes et sensibilité **221**
1 Introduction . . . 221
2 Évaluation de la complexité . . . 222
2.1 Marché Pair à Pair . . . 223
2.2 Marché endogène DC . . . 225
2.3 Marché endogène AC . . . 227
2.4 Conclusion . . . 230
3 Performance sur cas tests . . . 231
3.1 Marché Pair à Pair . . . 231
3.2 Marché endogène DC . . . 233
3.3 Marché endogène AC . . . 235
3.4 Conclusion et état de l'art . . . 237
4 Limites et études paramétriques . . . 238
4.1 Paramètres du cas d'études . . . 239
4.2 Paramètres algorithmiques et architecturaux . . . 242
5 Conclusion . . . 249

6 Conclusion et perspectives **251**

Annexe **255**
1 Démonstration formulation ADMM pour marché . . . 255
1.1 Méthode 1 . . . 255
1.2 Méthode 2 . . . 258

2 Démonstration de la résolution directe du marché Endogène 259
2.1 Problème global 260
2.2 Problème dual 261
2.3 Problème direct 262
3 Recherche des racines 268
3.1 Polynôme du 3 ème degré 268
3.2 Polynôme du 4 ème degré 271
4 Sur la non convergence de l'OPFADMMGPU 273

Table des figures

1.1 Évolution des moyens de production d'électricité [1] 28
1.2 Tension et fréquences dans le monde [3] 29
1.3 Réglages de la fréquence [2] 30
1.4 Mécanismes de marché décrits par RTE [5] 33
1.5 Résolution d'un merit Order 34
1.6 Fonctionnement du marché de capacité [6] 35
1.7 Différents modes de gestion 39
1.8 Relaxation et approximation d'un problème non convexe 45
1.9 Organisation du marché Simulé 46
1.10 Différentes manières de résoudre le marché endogène 48
1.11 Modélisation d'une ligne par un modèle en Π 49
1.12 Ancêtre et enfants d'un bus b 51
1.13 Application de la loi d'Amdahl 55

2.1 Méthode Adéquation Algorithme appliqué à notre problème 72
2.2 Séparation d'une application en blocs fonctionnels 72
2.3 Représentation d'une application sous forme de graphe acyclique . . 73
2.4 Exemple d'une Roofline 74
2.5 Exemple d'une architecture d'un CPU dual core avec 4 unités de calcul par cœur 97
2.6 Architecture d'un GPU 99
2.7 Architecture Cuda 101
2.8 Roofline du GPU utilisé, mesuré par [103] 102
2.9 Modélisation du cas 2 bus dans le cas DC (gauche) et AC (droite) . 106
2.10 Modélisation du cas 3 bus dans le cas DC (gauche) et AC (droite) . 107

3.1 Différentes manières de résoudre le marché endogène 117
3.2 Taux de convergence des différents algorithmes 125
3.3 Temps de calcul mesuré des différents algorithmes 125
3.4 Valeurs propres pour la méthode PAC 137
3.5 Temps de calcul mesuré (noir) et courbe de tendance selon la dimension du problème 139

3.6 Temps de calcul mesuré des différents algorithmes (C++ sur CPU, AMD RYZEN 5 5600H) 140
3.7 Temps de calcul mesuré des différents algorithmes 149
3.8 Nombre d'itérations mesuré des différents algorithmes 150
3.9 Fonctionnement du marché P2P DC-endogène 155
3.10 Taux de convergence des différents algorithmes 162
3.11 Résidus des différentes méthodes sur les cas (par taille croissante en nombre de bus, avec 50 simulations par nombre de bus) 163
3.12 Temps de calcul mesuré des différents algorithmes 163

4.1 Partitionnement en bloc fonctionnel pour le marché pair à pair . . . 171
4.2 Répartition des temps de calcul pour les marchés P2P 172
4.3 Partitionnement en bloc fonctionnel pour le Power Flow 175
4.4 Répartition des temps selon la méthode, cas Matpower 85 175
4.5 Partitionnement en bloc fonctionnel pour l'Optimal Power Flow . . 177
4.6 Répartition des temps selon la méthode, cas TestFeeder 178
4.7 Partitionnement en bloc fonctionnel pour le marché endogène . . . 180
4.8 Répartition des temps de simulations selon les méthodes et le cas simulé 181
4.9 Évolution de la répartition des temps sur le cas 154 agents 187
4.10 Évolution de la répartition des temps sur le cas Test Feeder 190
4.11 Évolution de la répartition des temps sur le cas Test Feeder 194
4.12 Évolution de la répartition des temps sur le cas Test Feeder 197
4.13 Fonctionnement synchrone (haut) et asynchrone (bas) par rapport au CPU 200
4.14 Exemple de vecteurs créés à partir de la matrice des échanges pour trois agents 202
4.15 Parallélisation sur les agents (gauche), sur les échanges (milieu), ou sur les échanges par block (droite). Zoom sur deux agents, un avec 2 pairs et l'autre avec un unique pair. Les contours gris représentent les appels kernels 205
4.16 Comparaison des différentes résolutions du problème local 207
4.17 Position sur la Roofline des méthodes (sauf Q^{part}) 214
4.18 Position sur la Roofline du calcul de Q^{part} 214
4.19 Temps du cumul des meilleurs méthodes pour la version avec ou sans transposée 215
4.20 Temps moyen minimal et maximal pour chaque méthode de calcul 219

5.1 Évolution des temps de calcul avec le nombre d'agent, marché P2P parallélisé 223

5.2 Évolution des temps de calcul avec le nombre d'agent, marché P2P en série . . . 223
5.3 Évolution des temps de calcul avec le nombre d'agent, marché P2P résolu par ADMM . . . 225
5.4 Ratio entre les temps de calcul du SO et du marché P2P . . . 227
5.5 Temps de calcul (s) divisé par le nombre d'itérations . . . 228
5.6 Ratio entre les temps de calcul sur CPU et sur GPU . . . 228
5.7 Évolution des temps de calcul avec le nombre de bus (ou agents), marché AC-Endogène . . . 229
5.8 Temps de calcul (s) divisé par le nombre d'itérations, marché AC-Endogène . . . 230
5.9 Temps de calcul selon la méthode et l'heure simulée, cas Européen (2463 agents) . . . 232
5.10 Temps de calcul d'un marché P2P pour chaque heure simulée sur GPU, début du jeu de test le 01/09/2013 . . . 234
5.11 Temps de calcul selon l'heure simulée pour le **DC-EndoPF** sur GPU 235
5.12 Nombre d'itération selon l'heure simulée pour le **DC-EndoPF** sur GPU . . . 236
5.13 Temps de calcul (s) selon la méthode et la minute simulée . . . 237
5.14 Temps par itération (s) selon la minute simulée et la méthode . . . 238
5.15 Type de réseau généré . . . 239
5.16 Taux de convergence (**EndoDirect** en haut puis **EndoConsensus** en bas), cas de la forme "type de réseau - ration N/B" . . . 240
5.17 Temps de simulation selon le cas et la taille (**EndoDirect** en haut puis **EndoConsensus** en bas) . . . 241
5.18 Répartition et moyenne des temps de calculs et nombre d'itération selon le facteur de pénalité, grande plage de variation de a . . . 241
5.19 Répartition et moyenne des temps de calculs et nombre d'itération selon le facteur de pénalité, petite plage de variation de a . . . 242
5.20 Influence du facteur de pénalité pour le marché P2P sur CPU . . . 243
5.21 Influence du facteur de pénalité pour le marché P2P sur GPU . . . 243
5.22 Influence du facteur de pénalité pour le marché endogène DC sur GPU . . . 244
5.23 Simulation d'un DC marché endogène avec des cas non convergeant 245
5.24 Flux dans les lignes . . . 246
5.25 Analyse des flux dans les lignes selon le décalage . . . 246
5.26 Temps de simulation et nombre d'itérations selon le pas et la minute simulée . . . 248
5.27 Temps par itération selon l'heure simulée pour le **DC-EndoPF** sur GPU avec refroidissement . . . 248

5.28 Temps par itération selon l'heure simulée pour le **DC-EndoPF** sur GPU sans refroidissement 249

6.1 Résidus dans le cas MatPower 69 bus (avec racines carrés en simple précision) 274
6.2 Résidus dans le cas MatPower 69 bus (avec racines carrés en simple précision) 275

Liste des tableaux

1.1 Référence de l'état de l'art pour les problèmes de marché 61
1.2 Référence de l'état de l'art pour les problèmes de Power Flow . . . 63
1.3 Référence de l'état de l'art pour les problèmes de Optimal Power Flow 64

2.1 Synthèse des algorithmes 94
2.2 Caractéristique matériel du GPU utilisé 102
2.3 Définition des paramètres pour le cas 2 nœuds 105
2.4 Définition des paramètres des agents pour le cas 3 bus 107
2.5 Définition des paramètres des lignes pour le cas 3 bus 108
2.6 Définition des paramètres, lorsque non fournis par matPower 109
2.7 Définition des paramètres Européen 109
2.8 Caractéristique d'un cas généré aléatoirement, modélisation DC . . 111
2.9 Ensemble des cas d'études utilisés 113
2.10 Description des cas Matpower utilisés 113

3.1 Différents termes de la Jacobienne 121
3.2 Temps en ms et (nombre d'itérations) pour converger pour différents algorithmes et cas 124
3.3 Valeur des matrices et vecteur par blocs 128
3.4 Nombre d'itération et temps (ms) pour converger pour les différents algorithmes et cas 138
3.5 Paramètres pour la simulation 138
3.6 Temps moyen (s) (nombre d'itération) selon la méthode et la taille 140
3.7 Nombre d'itération et temps (ms) pour converger pour les différents algorithmes et cas 148
3.8 Nombre d'itération et temps (ms) pour converger pour les différents algorithmes et cas 161
3.9 Résumé de l'ensemble des algorithmes et problème 164

4.1 Correspondance entre blocs fonctionnels et équations 170
4.2 Blocs fonctionnels 174
4.3 Blocs fonctionnels et complexité en série et en parallèle des algorithmes pour le marché pair à pair 186

4.4 Blocs fonctionnels et complexité en série et en parallèle des algorithmes pour le Power Flow 190
4.5 Blocs fonctionnels et complexité en série et en parallèle des algorithmes pour l'optimal power Flow 194
4.6 Blocs fonctionnels et complexité en série et en parallèle des algorithmes pour le marché endogène sur CPU 196
4.7 Blocs fonctionnels et complexité en série et en parallèle des algorithmes pour le marché endogène sur GPU 196
4.8 Temps total (s) de résolution, cas TestFeeder 197
4.9 étude Rooofline (asymptotique) 213
4.10 Speed-up (%) d'utiliser la transposée 215

5.1 Étude du passage à l'échelle des différentes implémentations 224
5.2 Augmentation du temps et complexité théorique dans le cas d'un marché selon s'il est mono-énergie ou poly-énergies 225
5.3 Caractéristique de la génération aléatoire pour le cas d'étude 226
5.4 Paramètres de simulation 226
5.5 Paramètres de simulation 229
5.6 Augmentation du temps et complexité théorique (N=B) 230
5.7 Augmentation du temps et complexité théorique 231
5.8 Paramètres de simulation pour le marché pair à pair 232
5.9 Moyenne, médiane, minimum, et maximum pour le temps de calcul (s) selon les méthodes 233
5.10 Speedup de la parallélisation par rapport à la version en série . . . 233
5.11 Paramètres de simulation pour le marché endogène DC 234
5.12 Paramètres de simulation pour le marché endogène AC 235
5.13 Moyenne, médiane, minimum, et maximum pour le temps de calcul (s) (nombre d'itération) selon les méthodes 237

Nomenclature

Variable d'optimisation

$\mathbf{\Delta} = (\delta_l)$ Prix nodal pour la congestion de la ligne l (variable duale)

$\mathbf{\Lambda} = (\lambda_{nm})$ Matrice des prix d'échange entre l'agent n et m (variable duale de l'antisymétrie)

$\mu = (\mu_n)$ Vecteur des variables duales pour la relation entre la puissance totale et les échanges de l'agent n (problème local)

ρ facteur de pénalité pour l'ADMM sous forme de consensus (problème global)

ρ_l facteur de pénalité pour l'ADMM sous forme de partage (problème local)

ρ_x facteur de pénalité pour l'ADMM relaxant les contraintes réseaux

$g_n(p_n) = a_n \cdot p_n^2 + b_n \cdot p_n$ fonction coût de l'agent n, cas quadratique

Dimension du problème

$N = |\Omega|$ Nombre total d'agent

$B = |\mathcal{B}|$ Nombre de bus sur le réseau

$C_b = |\mathcal{C}_b|$ Nombre d'enfant du bus b

$L = |\mathcal{L}|$ Nombre de ligne sur le réseau

$L_b = |\mathcal{L}_b|$ Nombre de ligne lié au bus b

$M = \sum_{n\in\Omega} M_n$ Nombre total de voisin (donc nombre total d'échange possible)

$M_n = |\omega_n|$ nombre de voisin de l'agent n

$N_b = |\mathcal{N}_b|$ Nombre d'agent sur le bus b

Marché Pair à Pair

$\mathbf{P}^{agent} = (p_n)$ Vecteur des puissances totales échangées par l'agent n

$\mathbf{Q}^{agent} = (Q_n)$ Vecteur des autres puissances totales échangées par l'agent n dans le cas d'un marché multi-énergies

$\mathbf{T} = (t_{nm})$ Matrice des échanges (ou trades) entre les agents n et m

Ω ensemble des agents

Ω_c Ensemble des consommateurs

Ω_g Ensemble des générateurs

ω_n Ensemble des voisins de l'agent n

Ω_p Ensemble des consomacteurs (prosumers en anglais)

$\underline{p_n}/\overline{p_n}$ limite haute et basse de la puissance totale échangée par l'agent n

lb_n/ub_n limite haute et basse de chaque échange de l'agent n

$P0$ Puissance objectif d'un agent

$\beta = (\beta_{nm})$ Matrice des charges du réseaux entre l'agent n et l'agent m (terme exogène ou tarif différencié)

Réseau électrique physique

$\mathbf{\Phi} = (\phi_{ij}) = (\phi_l)$ Flux de puissance dans la ligne l entre les bus i et j

$\mathbf{\Phi}^{part} = (\phi_{nl}^{part})$ Flux de puissance partiel induit dans la ligne l par tous les agents $j > n$

$\mathbf{B}_{diag} = B_{ll}$ Matrice diagonale de la suceptance des lignes

$\mathbf{B}_{grid} = B_{ij}$ Matrice de la suceptance de la ligne entre les bus i et j

$\mathbf{C}$ Matrice de correspondance entre les bus et les lignes

$\mathbf{E} = (\theta, \mathbf{V})$ Vecteur de la tension dans chaque bus (d'abord les déphasages puis les amplitudes)

$\mathbf{G}_{grid}$ Matrice de la réactance de la ligne entre les bus i et j

$\mathbf{P}^{bus} = (P_b)$ Vecteur des puissances injectée échangées sur le bus b

$\mathbf{W} = (\mathbf{P}, \mathbf{Q})$ Vecteur de la puissance dans chaque bus (active puis réactive)

$\mathcal{B}$ Ensemble des bus du réseau

$\mathcal{C}_b$ Ensemble des "enfants" du bus b dans le cas d'un réseau radial

$\mathcal{L}$ Ensemble des ligne du réseau

$\mathcal{L}_b$ Ensemble des ligne relié au bus b

$\mathcal{N}_b$ Ensemble des agents sur le bus b

$\overline{\phi}_{ij} = \overline{\phi}_l$ Limite thermique de la ligne l entre les bus i et j (en puissance)

A_b Nœud "ancêtre" du bus b dans le cas d'un réseau radial

$E_b = V_b e^{i\theta_b} = e_b + i \cdot f_b$ Tension complexe dans le bus b sous la forme polaire ou cartésienne

l_{ij} Norme au carré du flux de courant dans la ligne entre les bus i et j

$R_l + i \cdot X_l$ Impédance en série de la ligne l

$Y_l = G_l + i \cdot B_l$ Admittance de la ligne l

Acronymes

AAA *ou* $A3$ Adéquation Algorithme Architecture

AC Alternative current (Modélisation qui prend en compte la puissance réactive)

AVX Advanced Vector Extensions

CPU Central Processing Unit (processeur)

CRE Commission de la régulation de l'électricité

DC Direct current (Approximation où l'on néglige la résistance des lignes et les chutes de tension)

DSP Digital Signal Processor (processeur de traitement de signaux numérique)

$FPGA$ Field Programmable Gate Array (carte re-programmable)

$GP - GPU$ General purpose - Graphic Processing Unit (processeur graphique)

GRD Gestionnaire du réseau de distribution

GRT Gestionnaire du réseau de transport (appelé RTE en France)

IA Intelligence Artificielle

KKT Condition de Karush-Kuhn-Tucker

$MIMD$ Multiple Instruction Multiple Data

$MPPA$ Massively Parallel Processor Array (Réseau de processeurs massivement parallèles)

NPU Neural Processing Unit (Puce d'accélération de réseaux de neurones)

OPF Problème d'Optimal Power Flow

$P2P$ (marché) Pair à Pair

PF Problème de Power Flow

$PME - PMI$ Petites et moyennes entreprises/industries

REF, PQ, PV Représente les types de noeuds (référence, puissances connues, puissance active et amplitude de tension connues)

$SCADA$ Supervisory Control And Data Acquisition (équipement de téléconduite)

$SIMD$ Single Instruction Multiple Data (paradigme où on applique la même instruction su plusieurs données)

SoC System on Chip (système sur puces ou puces intégrées)

SSE Streaming SIMD Extensions

Algorithme

$ADMM$ Alternating direction method of multipliers

$BackPQ$ Power Summationn Method (méthode backward forward basée sur les flux de puissance)

CI Consensus + innovation

Cur Current Sommation Method (méthode backward forward basée sur le courant dans les lignes)

GS Gauss Seidel

NR Newton Raphson

$OSQP$ Operator Splitting solver for Quadratic Programs

PAC Proximal Atomic Coordination

$SLAM$ Simultaneous Localization And Mapping

Chapitre 1

État de l'art et problématique scientifique

Sommaire

1 Réseau Électrique : la gestion en pratique 27
1.1 Fonctionnement actuel d'un réseau électrique 27
1.2 Pistes de recherche de la littérature 38
1.3 Périmètre de l'étude 46
2 Les puissances de calculs mises en jeu 54
2.1 L'accélération matérielle 54
2.2 L'accélération des calculs dans le domaine du réseau électrique 58
3 Conclusion sur l'état de l'art et contribution 61
3.1 Temps de calcul de la gestion du réseau électrique . . . 61
3.2 Contributions 65
3.3 Organisation du manuscrit 66

1 Réseau Électrique : la gestion en pratique

1.1 Fonctionnement actuel d'un réseau électrique

L'objectif de cette partie sera d'expliquer le fonctionnement du réseau électrique (plus particulièrement du réseau français) et les limites relevées sur ce mode de fonctionnement. La première partie restera assez générale afin de permettre de proposer un point de vue global sur le réseau électrique.

1.1.1 Constitution physique

Le développement de la production d'électricité au milieu du 20e siècle a débouché sur un système de production d'électricité centralisé. C'est-à-dire ce sont quelques importants moyens de production (principalement l'hydraulique, et le thermique qui a été remplacé par le nucléaire) qui produisent l'électricité pour tout

le pays. Le graphique Fig. 1.1 montre la répartition de la production d'énergie par année. On peut y voir la spécificité française où la production repose beaucoup sur la production nucléaire. On peut aussi remarquer l'arrivée et l'augmentation de la production éolienne et solaire.

Le réseau électrique a donc pour objectif de transporter l'électricité produite dans les centrales et autres moyens de production vers les consommateurs en limitant les pertes pendant le transport. On distingue principalement trois types de réseau (entre parenthèses, les chiffres pour le réseau français) [2] : le réseau de grand transport et d'interconnexion (45 000km), les réseaux régionaux de répartition (51 000km) et les réseaux de distribution (1 200 000km).

Le premier type de réseau permet le transport sur de longues distances de grandes quantités d'énergie avec des pertes minimales grâce aux hauts niveaux de tension utilisés (400 kV ou 225 kV). En effet, les pertes par effet Joule étant liées au courant, pour une même quantité de puissance, une plus grande tension implique une plus petite valeur de courant et donc d'échauffement. Ce réseau est maillé, fortement instrumenté et possède de la redondance pour continuer à fonctionner malgré la perte d'un équipement.

La tension est ensuite abaissée à 225 kV, 90 kV ou 63 kV dans le réseau de répartition grâce à des transformateurs. Ce réseau permet d'alimenter de gros clients industriels et les réseaux de distribution dans les différentes régions.

Ces derniers réseaux à 20kV ou 400V desservent les consommateurs finaux, PME-PMI, client domestique, tertiaire... L'ensemble du réseau est en triphasé, le raccordement des particuliers est principalement en monophasé pour une tension de 230 V. Contrairement au réseau de transport, le réseau de distribution est faiblement instrumenté et de topologie radiale (en arbre). Le réseau européen a une fréquence de 50Hz, et ce type d'électricité est répandu dans le monde. Cependant,

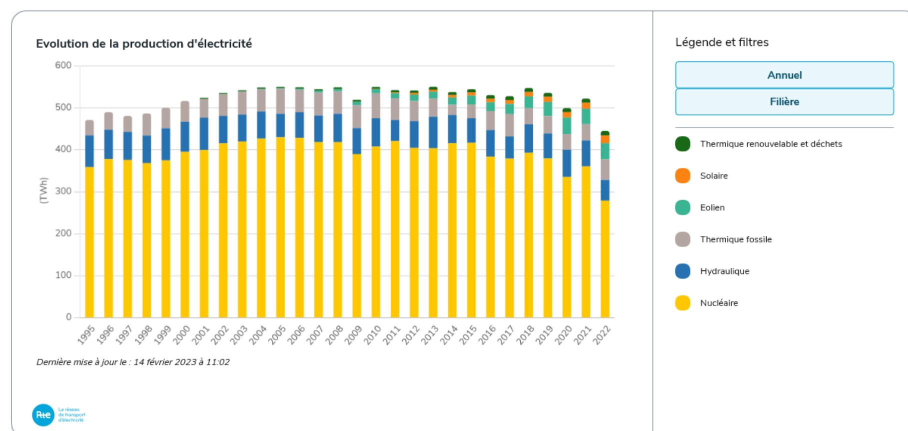


FIGURE 1.1 – Évolution des moyens de production d'électricité [1]

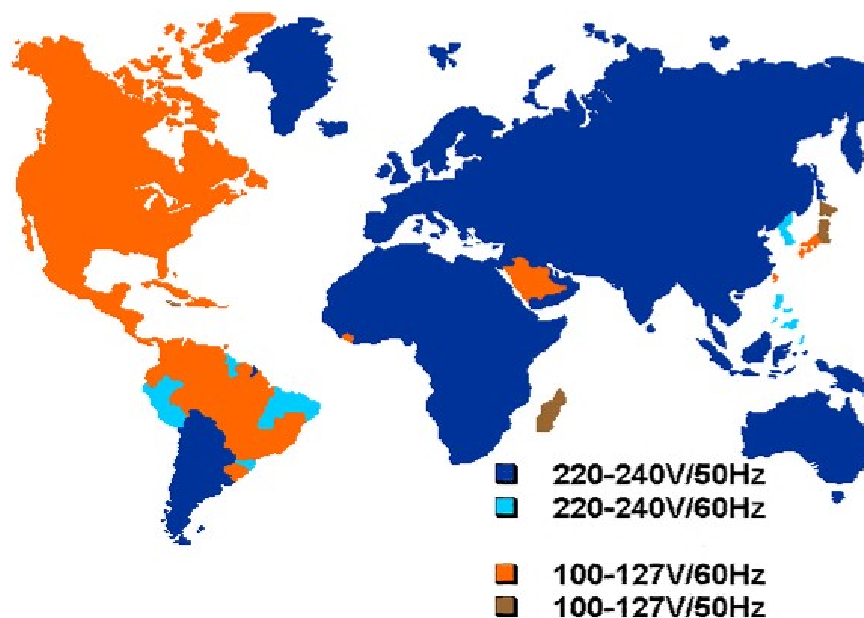


FIGURE 1.2 – Tension et fréquences dans le monde [3]

les tensions et fréquences ne sont pas les mêmes partout : la carte Fig. 1.2 montre la répartition de l'utilisation des différents niveaux de tension et de fréquence dans le monde.

Afin de permettre le transport sur des distances extrêmement longues ou sur des liaisons souterraines ou sous-marines, avec moins de pertes ou d'encombrement, des liaisons en courant continu peuvent aussi être utilisées.

Afin de permettre la surveillance et le pilotage du réseau électrique en temps réel, des équipements de téléconduite aussi appelés SCADA (*Supervisory Control And Data Acquisition*) sont utilisés. Ces outils sont composés de différents modules servant à contrôler les disjoncteurs, mais aussi à l'acquisition, le traitement, la visualisation et la transmission de l'état du réseau électrique. Cet état correspond aux différents niveaux de tensions des bus ou des flux de puissance dans les lignes.

1.1.2 Nécessité d'un équilibre

Contrairement à d'autres domaines, la gestion de l'électricité sur le réseau électrique a la particularité suivante : à tout moment, tout déséquilibre (même minime) entre la production et la consommation d'électricité entraîne une variation de la fréquence. Ainsi, si la puissance consommée est plus grande que la puissance produite, la conservation de l'énergie fera diminuer la fréquence (ou inversement). Tant que l'équilibre n'est pas retrouvé, la fréquence continuera de diminuer. L'équilibre

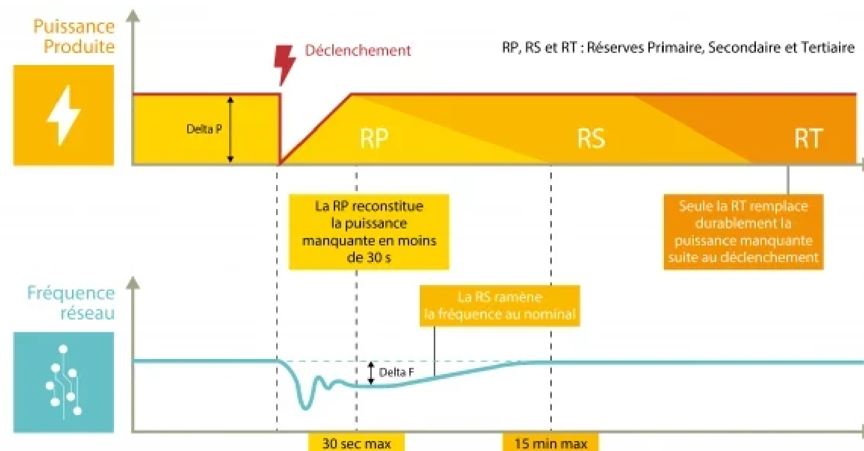


FIGURE 1.3 – Réglages de la fréquence [2]

ne faisant qu'arrêter la variation, il faut inverser momentanément le déséquilibre pour permettre à la fréquence de retourner à sa valeur nominale de $50Hz$. Ainsi, dans la pratique, la fréquence oscille autour de $50Hz$ avec une précision d'au plus $0,050Hz$ [4]. Pour garantir cette fréquence, différents réglages permettent de réagir en allant du plus rapide (réglage primaire) où les machines synchrones sont directement contrôlées par la fréquence au plus lent (réglage tertiaire) où l'on change la répartition de la production par zone pour équilibrer localement, Fig. 1.3. Cet équilibre est critique sur le réseau, car une fréquence qui s'éloigne trop de la valeur de 50 Hz risque d'endommager les différents systèmes reliés au réseau. Ainsi, par précaution, ceux-ci peuvent se déconnecter pour se protéger. Cette déconnexion peut, dans certains cas atténuer le déséquilibre, mais, dans d'autres cas, peut entraîner un vrai effet en cascade où, par exemple l'ensemble des productions renouvelables doivent se déconnecter à cause d'une trop grosse chute de fréquence entraînant un encore plus grand déficit de production.

1.1.3 Acteurs du réseau électrique

Les acteurs gérant le réseau physique sont les suivants.

Le gestionnaire du réseau public de transport (GRT) d'électricité en France est **RTE**. Il est responsable de l'exploitation du réseau de transport. Il doit donc garantir l'équilibre offre demande, réaliser la gestion des services système et des interconnexions avec le réseau européen. Il est donc le garant de la sécurité, de la fiabilité et de l'efficacité du système en minimisant les pertes. Il est responsable de la planification en prévoyant différents futurs possibles pour la gestion et la production de l'électricité [9], tout en étant responsable des investissements et de l'entretien. Ils ont aussi une obligation de garantir un accès au réseau non

discriminatoire et le respect de la confidentialité de par son statut de monopole naturel réglementé.

Le réseau de distribution appartient aux communes, celles-ci peuvent se regrouper et gérer ce réseau via une administration publique appelée syndicat d'énergie. *"Si elles n'assurent pas elles-mêmes, par le biais de régies, la gestion de leurs réseaux de distribution, les autorités concédantes la confient, par contrats, à un gestionnaire de réseau de distribution (GRD)"* [2]. Le gestionnaire du réseau de distribution est pour 95% du territoire **Enedis**, mais il existe aussi SER (Strasbourg), réséda (Metz) ; Gérédis (Deux-Sèvres) ; SRD (Vienne) ; GEG (Grenoble). La gestion des îles étant particulière, celles-ci étant isolées, un seul acteur remplit l'ensemble des missions comme cela était fait historiquement à l'échelle de la France. Ainsi, la direction des systèmes électriques insulaires d'EDF (EDF SEI) dessert la Corse et la plupart des départements et collectivités d'outre-mer, et l'électricité de Mayotte. Ce gestionnaire a exactement les mêmes missions que RTE, sauf qu'elles concernent les réseaux de distribution (exploitation, entretien, planification...). Historiquement, le gestionnaire du réseau de distribution n'avait besoin de ne gérer que des flux de consommations. Ainsi, le GRD n'était pas chargé de respecter l'équilibre de puissance, contrairement à RTE. Cependant, avec l'essor des énergies renouvelables placées sur le réseau de distribution, il peut arriver que des flux de puissance remontent sur le réseau de transport.

Ensuite, les acteurs qui vont intervenir sur les marchés de l'électricité sont les producteurs : principalement **EDF** et **ENGIE** ou d'autres, plus récents, ayant investi dans les énergies renouvelables, comme **Enercoop**. Ce sont des acteurs qui possèdent et donc contrôlent des moyens de production.

Les fournisseurs d'électricité sont les acteurs qui vendent et achètent l'électricité pour fournir l'électricité nécessaire aux consommateurs. Ils sont principalement des producteurs (donc **EDF** et **ENGIE**).

Les consommateurs sont soit petits (consommation domestique, petite entreprise) et donc n'interviennent qu'à travers leur fournisseur ; soit assez grand pour directement intervenir sur le marché.

Il existe aussi des acteurs dont le rôle n'est ni de consommer ni de produire, mais d'intervenir financièrement ou administrativement sur le marché. Cela peut concerner aussi bien des agrégateurs, les communautés, des acteurs d'effacement. Les agrégateurs et les communautés sont des entités juridiques qui permettent le regroupement de consommateurs et de producteurs. Ils permettent ainsi à ces derniers d'intervenir comme un ensemble de petits acteurs ou de réaliser de l'autoconsommation collective en étant rémunérés sans passer par des fournisseurs. Les acteurs d'effacements proposent d'instrumenter et d'ajouter du contrôle sur différents équipements des particuliers (chauffage, chauffe-eau). L'idée étant de décaler la consommation en dehors des pics de consommations et de consommer moins

grâce à un contrôle plus fin. Ces acteurs vendent sur le marché l'électricité non consommée par leurs clients et donnent une compensation financière aux fournisseurs de ces clients. En effet, l'énergie non consommée ne pouvant pas être prévue par le fournisseur, celui-ci aura acheté de l'électricité aux producteurs sans que celle-ci soit consommée par leurs clients.

Les gestionnaires d'équilibres sont des entités qui doivent garantir à RTE qu'à tout instant ils sont à l'équilibre. C'est-à-dire que leur production ajoutée à leurs achats est égale à leurs ventes et consommations. On y retrouve les gros producteurs-fournisseurs, mais aussi des banques. S'ils ont trop acheté ou produit, RTE rachète leurs surplus, mais à un prix inférieur au marché. Dans l'autre sens, l'acteur doit acheter au prix fort ce qui lui manque. Ainsi, les responsables d'équilibre sont encouragés à être le plus proches de l'équilibre possible pour que d'un point de vue global le système soit à l'équilibre.

Les pouvoirs publics, tels que le gouvernement ou l'autorité de la concurrence peuvent aussi avoir une influence sur la gestion du réseau. En effet, des mécanismes de taxes, d'obligation d'achat, de tarifs sociaux ou autres lois peuvent être utilisés pour agir sur la gestion du réseau. Enfin, la Commission de régulation de l'électricité (la **CRE**), quant à elle, est une autorité administrative indépendante qui "*veille au bon fonctionnement des marchés de l'électricité et du gaz en France, au bénéfice des consommateurs finaux et en cohérence avec les objectifs de la politique énergétique*" [2].

1.1.4 Marchés de l'électricité

Pour permettre l'achat et la vente d'électricité entre tous les acteurs, trois types différents de mécanismes existent. Le schéma Fig. 1.4 de RTE [5] résume les différents mécanismes de marché présentés dans cette partie.

Les premiers types de transactions sont les contrats bilatéraux ou de gré à gré. Ces contrats de vente et d'achat sont conclus directement entre deux parties à des prix fixés au moment de la transaction ou indexés sur une référence publique.

Le deuxième type d'échange est sous la forme de bourse. Le concept consiste à rassembler toutes les propositions et exigences, puis à les comparer à l'aide d'un « merit order » (Fig. 1.5) pour établir simultanément le prix de transaction et les échanges qui seront réalisés. De manière simplifiée, les moyens de production sont triés par ordre croissant de leur coût de fonctionnement marginal (qu'est ce que cela coûterait en plus de les faire produire par rapport à ne rien faire). Tandis que les demandes sont triées par ordre décroissant de prix. À l'intersection des quantités-prix, on obtient l'ensemble des transactions à réaliser. En Europe, le choix a été fait d'avoir un seul prix unique, la marge réalisée dépend donc du prix marginal du dernier moyen de production appelé.

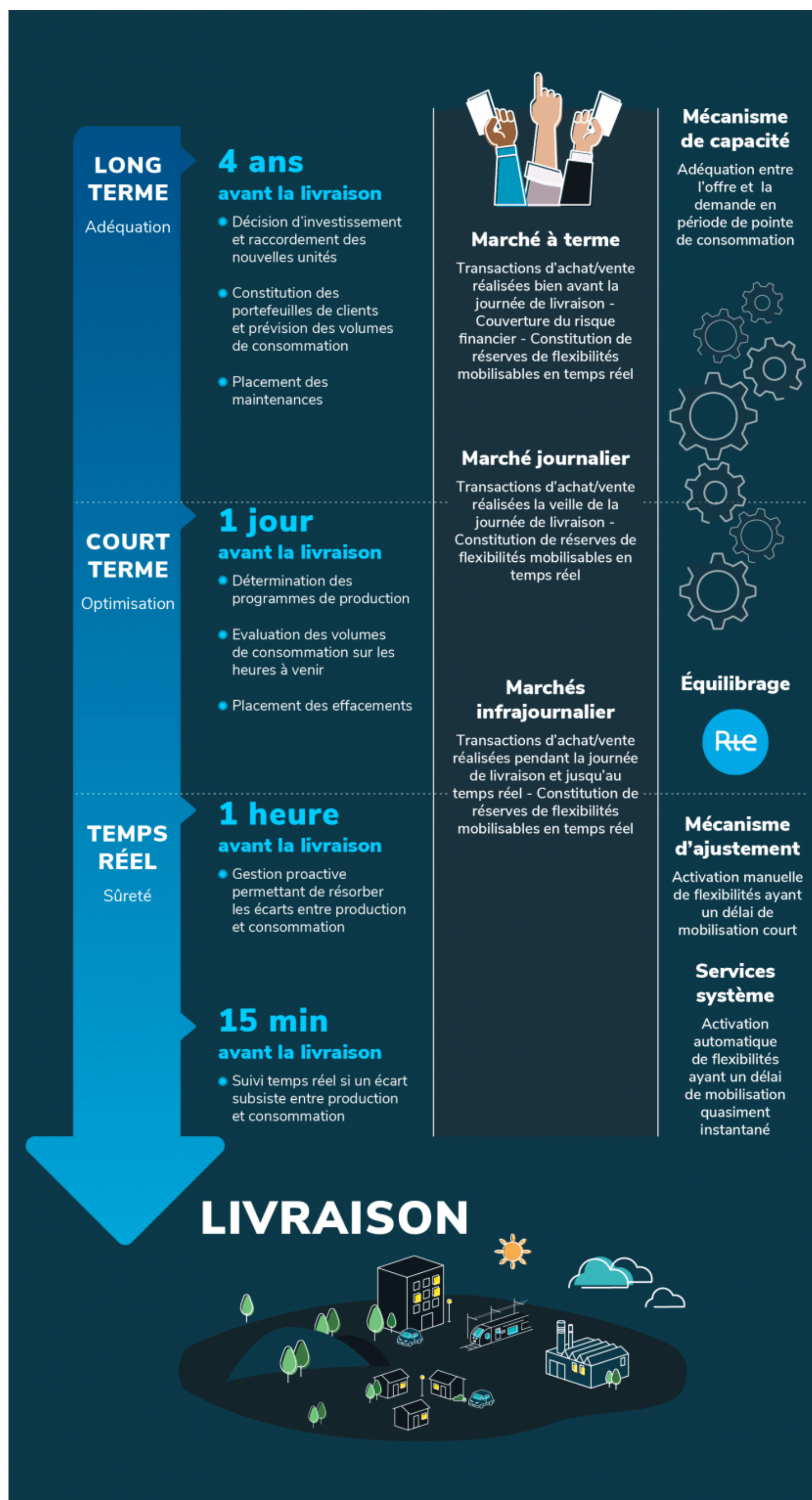


FIGURE 1.4 – Mécanismes de marché décrits par RTE [5]

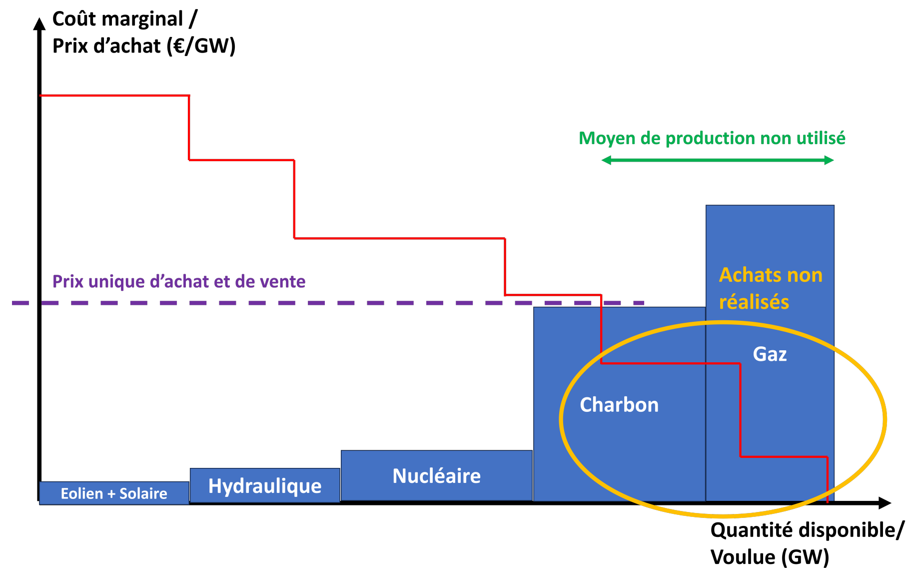


FIGURE 1.5 – Résolution d'un merit Order

Il existe différents types de transactions selon l'échelle de temps considérée. La plus connue est le marché spot qui concerne l'échéance journalière (livraison le lendemain) ou infrajournalière (produit demi horaires, horaires ou multihoraire pour le jour même). C'est le marché le plus important, car c'est par rapport aux estimations du prix qu'atteindra ce marché que les acteurs arbitrent sur les prix à proposer sur les autres marchés à plus long terme. Ce prix est aussi utilisé en tant que prix de référence pour le marché de l'électricité français. Les achats d'électricités sur de plus longues durées avant le marché spot sont appelés marché à terme. Ils permettent d'avoir des prix moins volatils pour le gros de la consommation/production qui peut être prévue bien en avance.

La disponibilité de l'électricité à tout moment étant un enjeu stratégique majeur, différents mécanismes existent pour essayer de la garantir tout en permettant une concurrence. Ainsi, il existe par exemple des marchés de capacités, Fig. 1.6. Depuis sa création, les fournisseurs doivent être capables de garantir de pouvoir remplir les besoins de leurs clients grâce à des moyens de production ou de l'effacement. Pour prouver ce fait, ils doivent obtenir des contrats avec des producteurs. Ainsi, le fait qu'une centrale puisse être disponible en cas de besoin peut être rémunéré. Ce marché est particulièrement utile pour rémunérer (et ainsi payer pour les coûts d'investissement) les moyens de production qui ne sont appelés que lors des pics de consommation.

Enfin, afin de permettre la transition énergétique, d'autres mécanismes peuvent s'ajouter. On peut citer par exemple l'obligation d'achat des énergies renouvelables

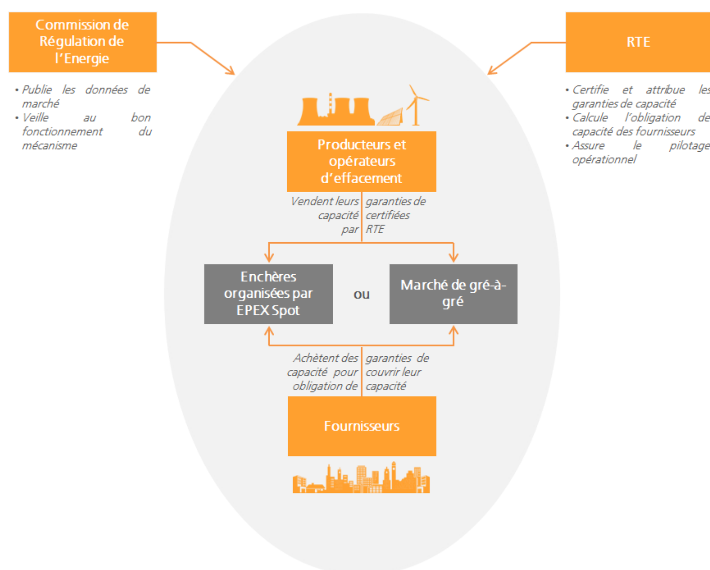


FIGURE 1.6 – Fonctionnement du marché de capacité [6]

à un prix fixe. Il est important de noter que les productions d'énergies renouvelables ne sont pas obligées d'utiliser ce mécanisme. Ainsi, ils peuvent aussi vendre au prix du marché spot si celui-ci est plus avantageux. De même, un surcoût à l'achat est ajouté en fonction de la quantité de CO2 émise par quantité d'électricité produite.

La multitude des marchés et le fait qu'un acteur puisse participer dans plusieurs d'entre eux en même temps complexifient la prise de décision. En effet, les incertitudes et les multiples possibilités demandent aux acteurs de réaliser de nombreuses simulations afin de maximiser leurs bénéfices. La bonne gestion du réseau électrique sur le long terme repose ainsi sur la capacité des gestionnaires à être capable de simuler avec précision les différents mécanismes pour connaître l'utilisation du réseau.

1.1.5 Marchés locaux

Afin de permettre de faciliter l'intégration des moyens de productions d'énergies renouvelables ; des marchés locaux de l'énergie sont proposés [7]. Ces marchés ont pour objectif d'inciter et de permettre à des petits consommateurs ou producteur à échanger directement de l'énergie entre eux. Ces échanges devraient permettre d'équilibrer localement la consommation et la production sans nécessiter l'intervention des gestionnaires de réseau. On peut distinguer trois types de mécanismes distincts qui correspondent à des marchés locaux de l'énergie. Tout d'abord, le marché pair à pair est un marché où les participants peuvent échanger de l'énergie

sans aucun intermédiaire. L'objectif de ce marché est principalement de permettre aux agents d'utiliser leur flexibilité pour faire des gains économiques (en achetant lorsque les prix sont bas ou en vendant lorsqu'ils sont élevés). Les marchés transactifs de l'énergie (*transactive energy markets*) équilibrent l'offre et la demande dans les systèmes électriques au moyen d'une coordination décentralisée. Leur objectif est de gérer les ressources décentralisées de manière autonome en utilisant des signaux de prix pour assurer la stabilité du système plutôt qu'un gain économique. Le dernier type de marché local repose sur une législation européenne et s'appelle les communautés d'autoconsommation collective. Cette structure permet à différents agents proches géographiquement de se rassembler en une entité appelée communauté. Cette communauté permet de répartir les coûts d'investissements pour l'installation d'infrastructure. De plus, ses membres peuvent s'échanger de l'électricité prioritairement entre eux. Ainsi, mensuellement, les consommations et les productions sont comparées a posteriori. Si ces événements se produisent au cours de la même demi-heure, la communauté est considérée comme ayant auto-consommé cette énergie et l'ayant échangée en interne. Ainsi, cette énergie ne sera ni achetée ni vendue au fournisseur. Une détaxe peut être appliquée sur cet échange, résultant en un gain économique à la fois pour les producteurs et les consommateurs de la communauté. L'énergie échangée est répartis par la Personne Morale Organisatrice avec des clés de répartitions qui peuvent permettre de viser des objectifs sociaux en plus des gains économiques.

1.1.6 Limites et défis du réseau électrique

La gestion du réseau électrique est un défi qui se complexifiera de plus en plus.

Tout d'abord, en l'état, les nombreuses imprécisions lors de la gestion provoquent de grands manques à gagner. En effet, les quantités échangées sur le marché ne représentent qu'une prévision de ce qui sera consommé ou produit. Les multiples marchés permettent d'être de plus en plus précis en s'approchant de l'échéance. Mais il restera quand même une erreur à la fin du marché le plus proche de l'échéance. En outre, les systèmes en « merit order » ne tiennent que partiellement compte des non-linéarités des centrales, telles que la rampe d'activation, ou de leur positionnement dans un réseau idéal. Ainsi, si, après la sélection des centrales, la simulation démontre des congestions dans les lignes, il faut changer l'appel. De plus, cette simulation est elle-même une approximation pour des raisons de temps de calcul ou de robustesse [8]. Pour faire en sorte que le tout fonctionne, les gestionnaires suivent des ensembles de règles pour adapter la production et le réseau et ainsi garantir son bon fonctionnement. Ces règles permettent d'avoir un réseau fonctionnel, aucun black-out n'ayant eu lieu depuis longtemps. Cependant, elles ne permettent pas d'atteindre le point de fonctionnement optimal du réseau.

L'amélioration des techniques d'instrumentation et de simulation actuelles est

donc toujours un sujet de recherche à fort enjeu énergétique et économique. De plus à partir de la planification de l'évolution du réseau fournie par RTE [9] on peut identifier de nouveaux défis.

En effet, la neutralité carbone sur notre territoire en 2050 va provoquer de nombreux changements. Tout d'abord, la multiplication de l'implantation d'unités de production renouvelables va bouleverser les moyens de gestion du réseau [10]. Ces modes de production imparfaitement prévisibles et peu contrôlables vont remplacer les moyens de production prévisibles et contrôlables à énergie fossile. Les simulations devront être réalisées à des pas plus fins pour prendre en compte la variabilité de la météo. De plus, un autre avantage des moyens de production non renouvelables était leur utilisation de machines synchrones dont l'existence permettait de physiquement augmenter l'inertie du réseau. En effet, leurs présences permettaient de réduire la vitesse de changement de la valeur de la fréquence lors de déséquilibre. Enfin, ces énergies renouvelables seront placées sur tout le territoire. De par leur faible puissance, elles seront plus logiquement reliées directement au réseau de distribution. Ce réseau est comparativement au réseau de transport très peu instrumenté, et non dimensionné pour gérer des congestions. Les réseaux ne sont pas non plus conçus pour gérer la possibilité d'avoir des flux de puissances, montants du réseau de distribution vers le réseau de transport, ce qui complexifie d'autant plus la gestion des **GRT** et des **GRD**.

La nécessité de maintenir l'équilibre entre la consommation et la production est toujours critique. Comme la production via des énergies renouvelables est peu flexible, RTE prévoit qu'une partie de la flexibilité viendra des consommateurs. Ces agents dits "actifs" se serviront de leurs moyens de stockage, de l'électrification de leur moyen de transport [11] et de l'automatisation de la gestion de leur domicile pour agir sur le réseau [12]. La coordination de tous ces agents n'est pas possible à grande échelle avec le mode de gestion actuel du réseau. Cette résolution risque d'être rendue encore plus difficile par la diversité des agents distribués (consommateur ou producteur) qui pourraient être ou non agrégés pour avoir un effet significatif [13]. Enfin, ces différentes modifications risquent de provoquer un flux de puissance ascendant du réseau de distribution vers le réseau de transport. Ce flux risque de détériorer la qualité de la puissance fournie et de créer des congestions, même sur les réseaux de distribution. Ainsi, pour garantir le bon fonctionnement du réseau, celui-ci devra soit être renforcé, ce qui consommera beaucoup de matières premières, soit être plus finement simulé et contrôlé pour rester dans les limites opérationnelles. Le réseau électrique fera donc face à de multiples challenges : il sera utilisé de manière plus intensive par de nombreux acteurs et d'une manière plus complexe.

1.2 Pistes de recherche de la littérature

La partie précédente a montré l'importance et la difficulté de la gestion du réseau électrique. L'organisation actuelle de la production par quelques centrales importantes permet à la gestion centralisée d'être adaptée. Bien que le résultat des différents marchés dépende d'un "*merit order*", il est possible de les modéliser par des problèmes d'optimisation [25]. Ainsi, quel que soit le marché considéré, la gestion demande l'optimisation d'un problème de la forme :

$$\min_{P} \quad \sum_{n\in\Omega} g_n(p_n) \tag{1.1a}$$

$$\text{t.q.} \quad \sum_{n} p_n = 0 \tag{1.1b}$$

$$\underline{p_n} \leq p_n \leq \overline{p_n} \qquad n \in \Omega \tag{1.1c}$$

La gestion cherche, pour tous les agents n, les puissances $\mathbf{P} = (p_n)_{n\in Omega}$ que ceux-ci doivent produire ou consommer pour minimiser le coût de fonctionnement total. La production ou la consommation a un coût noté $g_n(p_n)$. Chaque agent a une limite sur sa flexibilité notée $\underline{p_n}$ et $\overline{p_n}$. Un agent tel que $\underline{p_n} = \overline{p_n}$ est dit non flexible et produira ou consommera la puissance voulue, quel que soit le prix du marché. L'équilibre entre la production et la consommation est représenté par la contrainte $\sum_n p_n = 0$.

Afin de répondre aux nouveaux défis du réseau électrique, une transformation du réseau est nécessaire [14]. La première voie de transformation serait de continuer à renforcer le réseau pour avoir suffisamment de marge sur les contraintes pour les ignorer. Cette solution a pour principaux défauts le coût important d'investissement nécessaire et la difficile acceptabilité sociale des nouvelles infrastructures. La mise en place d'un réseau plus actif et intelligent, le *SmartGrid* offre une alternative intéressante. D'après la CRE, la définition d'un SmartGrid est la suivante :

On désigne par Smart grid un réseau d'énergie qui intègre des technologies de l'information et de la communication, ce qui concourt à une amélioration de son exploitation et au développement de nouveaux usages tels que l'autoconsommation, le véhicule électrique ou le stockage. Désormais, à la couche physique pour le transit d'énergie des réseaux vient se superposer une couche numérique qui joue un rôle de plus en plus important pour son pilotage. De nombreux points d'interface (capteurs, automates etc.) relient ces deux couches. Les compteurs évolués de type Linky pour l'électricité et Gazpar pour le gaz naturel sont une brique essentielle de cette nouvelle architecture des réseaux en France.

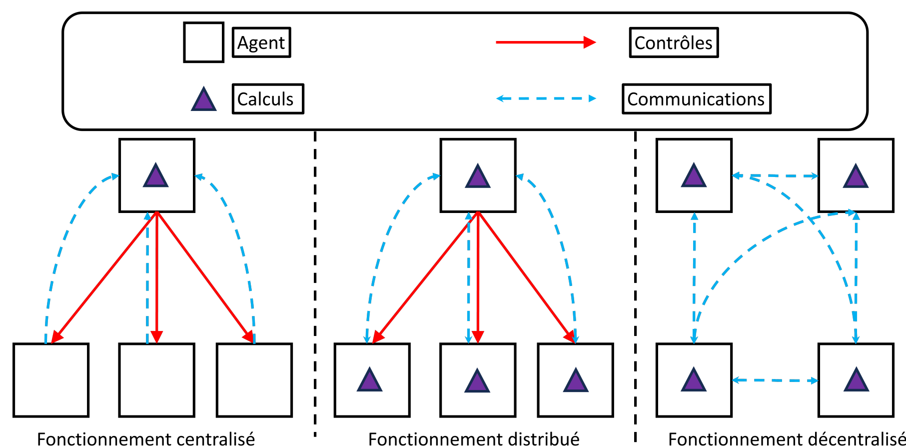


FIGURE 1.7 – Différents modes de gestion

1.2.1 Décentralisation de la gestion

La couche numérique du SmartGrid permet d'envisager d'autres manières de gérer le fonctionnement du réseau. La décentralisation (ou distribution) de la gestion consiste à diviser la résolution du problème global en différents sous-problèmes. On parle de distribution lorsque les calculs sont réalisés à différents endroits, mais qu'il reste une entité centralisatrice pour l'organisation de la résolution. Ce type de gestion existe déjà, par exemple pour la gestion du réseau européen. En effet, même s'il existe un marché de l'électricité européen, certains calculs sont distribués chez les **GRT** de chaque pays. On parlera de décentralisation lorsqu'il n'y aura plus de nœud central pour coordonner la résolution. L'utilisation de contrat de gré à gré est une manière de gestion décentralisée. L'absence d'entité coordinatrice centralisatrice permet à chaque agent d'être relié à n'importe quel autre agent. Lorsque le marché est complètement décentralisé, on parlera de **marché pair à pair** (**P2P**). Le schema Fig.1.7 récapitule les trois paradigmes. Il est évidemment possible de trouver des intermédiaires entre ces paradigmes en utilisant des structures hiérarchiques.

La décentralisation peut se faire à différentes étapes de la gestion du réseau électrique, comme lors de la résolution du marché [15] ou de la détermination de l'état du réseau [16]. Dans les deux cas, les avantages sont les suivants [17].

— Permettre d'éviter la surcharge en temps de calcul ou en mémoire nécessaire d'une unique unité de calcul.

— Permettre d'éviter la présence de la centralisation en un unique endroit de toutes les données sensibles, un *"single point of failure"*.

— Permettre que, dans l'hypothèse où l'on utiliserait la flexibilité de chaque particulier, celui-ci garde ses informations personnelles pour lui (si c'est lui

qui réalise l'optimisation) ou qu'il les fournisse à un tiers intermédiaire de confiance : une association citoyenne, une communauté, un agrégateur, etc.
- Permettre d'y ajouter des mécanismes de gestion tels que des tarifs préférentiels pour permettre de représenter d'autres volontés (écologique, philanthropique) que des volontés économiques.

Cependant, la décentralisation apporte aussi des inconvénients [18].
- Elle augmente la quantité totale de calculs et le nombre de communications.
- La convergence dans des problèmes non convexes ou dans le cas de perte d'information peut être plus dure à garantir et à vérifier [17], [19] ;
- la vérification d'abus de la part de certains acteurs ou de manipulation du marché peut être plus ardue à repérer.
- La décentralisation nécessite l'utilisation d'unité de calcul à faible consommation, à forte puissance de calcul et capable d'interagir avec beaucoup de données.

Chaque inconvénient a un ou plusieurs axes de recherche qui s'attellent à le résoudre. Ainsi, pour les deux premiers inconvénients, on peut s'intéresser à la recherche de meilleurs algorithmes de décentralisation [20] ou à l'utilisation de l'asynchronisme [21] pour limiter le temps perdu à attendre les messages ou converger même en cas de perte de messages. L'étude de l'application de la théorie des jeux et de l'équilibre de Nash permet d'étudier l'équilibre même en cas de présence d'agents stratégiques cherchant à optimiser leur gain au détriment des autres [22]. Enfin, la recherche s'attelle à trouver de nouvelles technologies pour les SCADA [23] ou pour la communication et la gestion des données [24].

Différents algorithmes sont utilisés dans la littérature pour réaliser la décentralisation. On citera les principaux qui sont la méthode des multiplieurs à direction alternée **ADMM** [25], [26], le *Proximal Atomic Coordination* (**PAC**) [27]- [28] et le Consensus + Innovation **CI** [20], [29].

Dans les parties suivantes, on se concentrera sur la modélisation dans la littérature des différents problèmes que nous allons traiter.

1.2.2 Problème du marché

Le problème du marché a pour objectif de déterminer les échanges économiques optimaux pour l'ensemble des agents. Ainsi, chaque agent souhaite acheter ou vendre une certaine quantité. Cette volonté peut dépendre ou non du prix d'achat. De plus, la dépendance de la volonté envers le prix peut être continue ou se présenter sous forme de paliers. Comme cela a été mentionné plus tôt, le marché peut prendre deux formes. Soit une forme centralisée *pool market* où l'ensemble des offres et demandes est regroupé pour pouvoir en déduire un prix unique et la quantité totale échangée. Soit une forme décentralisée avec des contrats de gré à gré où le prix et la quantité sont déterminés directement par les deux intervenants.

Le premier type de marché n'est pas très adapté aux énergies renouvelables en raison de leur petite taille et de leur incertitude [32]. Différents paradigmes existent pour représenter ce type de problème [33].

— La théorie des jeux permettant de représenter les comportements coopératifs ou compétitifs des agents [34], [35], [36].
— La théorie des enchères où l'on représente vraiment des échanges discrets entre les agents [37]
— L'optimisation sous contrainte où le problème de marché est modélisé par un problème mathématique d'optimisation [21] [25], [26], [38].

> La recherche sur la blockchain vise à déterminer comment, après que l'optimisation a été effectuée, réaliser des contrats de confiance sans entité centralisatrice.

Dans le cadre de cette thèse, on se concentrera sur ce dernier paradigme pour représenter le marché. De plus, on choisira de résoudre un marché pair à pair. Il a été montré que cette formulation était une généralisation de toutes les autres [39]. En effet, dans un marché pair à pair, n'importe quels liens peuvent exister entre n'importe quels agents. Ainsi, en imposant certains liens et en enlevant les autres, on peut retrouver une structure centralisée ou hiérarchisée. Le problème d'optimisation du marché pair à pair peut donc s'écrire ainsi [21] [25], [26], [38] :

$$\min_{T,P} \quad \sum_{n\in\Omega}\Big(g_n(p_n) + \sum_{m\in\omega_n}\beta_{nm}t_{nm}\Big) \tag{1.2a}$$

$$\text{t.q.} \quad \mathbf{T} = -{}^t\mathbf{T} \qquad (\Lambda) \tag{1.2b}$$

$$p_n = \sum_{m\in\omega_n} t_{nm} \qquad (\mu) \qquad n\in\Omega \tag{1.2c}$$

$$\underline{p_n} \leq p_n \leq \overline{p_n} \qquad n\in\Omega \tag{1.2d}$$

$$t_{nm} \leq 0 \qquad n\in\Omega_c \tag{1.2e}$$

$$t_{nm} \geq 0 \qquad n\in\Omega_g \tag{1.2f}$$

$$\underline{p_n} \leq t_{nm} \leq \overline{p_n} \qquad n\in\Omega_p \tag{1.2g}$$

L'objectif de cette optimisation est de trouver les échanges $\mathbf{T} = (t_{nm})_{n,m\in\Omega^2}$ optimaux entre les agents minimisant les fonctions objectifs $g_n(p_n)$ dépendant de la puissance totale $\mathbf{P} = (p_n)_{n\in\Omega}$ de chaque agent n. Ces fonctions peuvent inclure des préférences hétérogènes pour les échanges $\beta_{nm}t_{nm}$. Ce terme peut aussi représenter un terme exogène déterminé par les gestionnaires de réseau [25]. La contrainte (1.2b) est l'antisymétrie des échanges car ce qui est acheté doit être vendu pour être à l'équilibre. La puissance de chaque agent est la somme de ses échanges

(1.2c). Enfin la puissance et les échanges sont bornés (1.2e)-(1.2g) selon le type des agents avec respectivement Ω_c, Ω_g et Ω_p les consommateurs, les producteurs et les consomm-acteurs ou *prosumers*.

Le type de problème que l'on doit résoudre pour le marché dépend de la forme des fonctions coûts. En effet, selon qu'elles sont discrètes [40] ou définies par morceau [8], convexes ou non, les algorithmes utilisables diffèrent.

1.2.3 Gestion sous contraintes opérationnelles

Que la résolution soit centralisée ou décentralisée, la prise en compte des contraintes opérationnelles reste un défi à résoudre.

Le problème du Power Flow permet, à partir de la connaissance de la moitié des grandeurs caractéristiques (tension en amplitude et angle V, θ et puissances actives et réactives P, Q) sur les bus, de déterminer l'autre moitié [41]– [50]. Parmi les nœuds, un est choisi pour servir de référence à la tension ($V = V_0$) et à l'angle ($\theta = \theta_0$). Les autres nœuds sont des nœuds PQ si l'on connaît les puissances ou PV si l'on connaît l'amplitude de la tension et la puissance active. En notant $\underline{Y} = G + iB$, l'admittance des lignes, les équations de Kirchhoff nous donnent la relation suivante :

$$\underline{S}_i = \underline{E}_i \cdot \sum_k \underline{Y}_{ik}^* \cdot \underline{E}_k^* = F(\theta, V) \tag{1.3}$$

Ce qui donne l'expression des puissances actives et réactives en utilisant la tension sous sa forme angulaire :

$$P_i = V_i \cdot \sum_k V_k \cdot (G_{ik} \cos\theta_{ik} + B_{ik} \cdot \sin\theta_{ik}) \tag{1.4}$$

$$Q_i = V_i \cdot \sum_k V_k (G_{ik} \sin\theta_{ik} - B_{ik} \cos\theta_{ik}) \tag{1.5}$$

Différents algorithmes existent pour résoudre ce type de problème, on pourra citer les principaux qui sont Newton Raphson (**NR**) et Gauss-Seidel (**GS**). D'autres formulations existent, notamment lorsque le réseau est radial. Enfin, pour faciliter la résolution, le problème peut être découplé en séparant le problème des puissances actives et réactives. Le problème peut aussi être approximé en linéarisant les expressions. Plus de détails sur les algorithmes et seront présentés dans le chapitre suivant.

Historiquement, l'opérateur lance la simulation d'un Power-Flow (**PF**) après la résolution du marché pour vérifier si les contraintes sont respectées. Si elles ne le sont pas, les gestionnaires de réseaux interviennent pour déplacer les points de fonctionnement des agents du marché pour ensuite relancer la simulation. Ceci

est réalisé tant que la solution du marché ne respecte pas toutes les contraintes. L'ensemble marché et **PF**, avec l'intervention indirecte du gestionnaire de réseau, est ce que l'on appellera un marché exogène. Dans la littérature, on peut retrouver ce type de marché dans [25] où une pénalité est ajoutée dans le marché pour déplacer le point de fonctionnement.

1.2.4 Optimal Power Flow et Marché endogène

Comme vu précédemment, il est possible de prendre en compte les contraintes du réseau par des interactions entre le gestionnaire de réseau et le marché. Cependant, ce type de fonctionnement demande de réaliser plusieurs fois les différentes optimisations pour atteindre un optimum qui respecte les contraintes.

Ainsi, cette section montrera les problèmes qui permettent de prendre en compte les contraintes du réseau directement dans la résolution du marché. Dans le cas où l'on considère un marché centralisé, un problème d'Optimal Power Flow **OPF** peut être posé. Dans ce problème ni les échanges entre les agents, ni les préférences hétérogènes, ne sont pris en compte.

De manière générale, un Optimal Power Flow est un problème cherchant à minimiser une fonction coût à l'échelle du réseau tout en respectant les contraintes dans les lignes ou le plan de tension. C'est un problème hautement non convexe, qui possède donc de nombreuses relaxations ou approximations et algorithmes pour résoudre les différentes fonctions coût existantes avec différents choix de variable [55]- [70].

Pour la suite, on considérera que la fonction coût à minimiser sera la somme des fonctions coûts des agents. Cependant, cela pourrait être de manière non exhaustive, la minimisation des pertes dans les lignes, la minimisation des échanges de puissances réactive... Les vraies variables de décision sont les puissances des agents s_n. Pour faciliter la résolution (par exemple, la prise en compte des contraintes), le vecteur d'optimisation, noté X pourra aussi contenir les tensions complexes et les puissances dans les lignes en plus des puissances des agents. Le problème d'optimisation peut s'écrire ainsi :

$$\min_X \sum_{n \in \Omega} g_n(\underline{s_n}) \tag{1.6a}$$

$$\text{t.q. } \textbf{contraintes physiques} \tag{1.6b}$$

$$\textbf{contraintes operationnelles} \tag{1.6c}$$

> Contrairement à ce qui est souvent fait dans la littérature, on considérera les consommateurs flexibles. Ainsi, pour prendre en compte leur flexibilité, chaque consommateur aura sa propre fonction coût. C'est donc bien les fonctions coût de l'ensemble des agents qui est considéré et pas uniquement celle des générateurs.

Les contraintes physiques correspondent à l'application des lois de Kirchhoff (et donc à l'utilisation d'un **PF**). Tandis que les contraintes opérationnelles correspondent aux bornes sur les grandeurs physiques que l'on doit respecter pour garantir le bon fonctionnement du réseau. Cela correspond ainsi à des bornes sur les puissances admissibles des agents, des bornes sur la tension à chaque nœud à respecter et sur les flux de puissances dans les lignes pour éviter les congestions. La méthode de référence pour résoudre un OPF dans le cas général est l'utilisation d'un point intérieur (IPM) [65].

Problème décentralisé Il y a plusieurs manières de décentraliser le problème [17]. En effet, pour un même problème, on peut complètement décentraliser sur chaque bus [59] ou bien décentraliser sur des zones plus ou moins grandes [60], [61], [62]. Par souci de concision, on se concentrera dans notre cas à un problème totalement décentralisé avec un problème par bus. Mais la plupart des algorithmes peuvent aussi s'appliquer dans le cas où il y a plusieurs bus par zone.

Problème relaxé Une relaxation convexe inclut, dans un espace convexe, l'espace des solutions faisables non convexes des équations de l'optimal Power Flow. Elle donne une solution meilleure que celle du problème original et permet de certifier l'infaisabilité d'un problème. En effet, si aucune solution n'est trouvée dans le problème relaxé, c'est qu'il n'y en a pas dans le problème original. Sous certaines conditions, la relaxation peut être exacte. Cela signifie que le résultat du problème relaxé est aussi l'optimum du problème original. Cependant, même dans ce cas-là, les variables de décision peuvent être différentes. Sur le schéma Fig. 1.8, la forme verte représente les solutions respectant les contraintes du problème non convexe. L'espace en bleue présente une relaxation possible qui inclut l'espace du problème original. Par exemple, ici, la solution du problème relaxé est exacte lorsque l'optimum de la zone bleue est aussi dans la zone verte. Il y a différents niveaux et manières de relaxer ; en voici plusieurs [63](sans vouloir être exhaustifs).

- Semi défini :*Shor relaxation*, *Moment/Sum-of-Squares Relaxation Hierarchies...*
- Second Cone Order : *Jabr's Relaxation*, *QC Relaxation ...*
- Linéaire/Quadratique : *Network Flow*, *Copper Plate*, *The Taylor-Hoover*, *McCormick...*

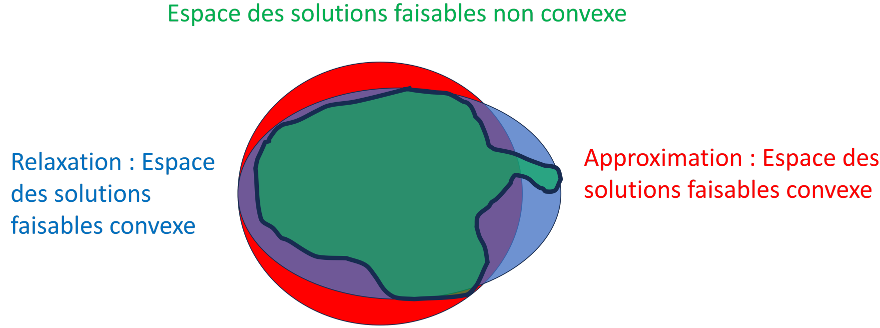


FIGURE 1.8 – Relaxation et approximation d'un problème non convexe

Problème approximé L'approximation d'un problème permet de le rendre convexe et de le simplifier grandement. Ceci est principalement fait sur l'OPF, mais peut aussi être fait sur le problème de PF. Lorsque l'on approxime le problème, on obtient un nouvel espace de solution convexe. Mais cet espace n'inclut pas l'espace original et n'est pas inclus dans celui-ci. La figure Fig. 1.8 montre qu'il existe des solutions dans l'espace vert qui ne sont pas dans l'espace rouge et inversement. Ainsi, la solution trouvée n'est pas une borne de la solution originale (on pourrait peut-être trouver mieux) et l'on ne peut rien conclure quant à l'infaisabilité d'une solution. Tout comme pour les relaxations, il existe plusieurs niveaux d'approximation et de manière de les réaliser [63].

— Second cone Order : *Jabr's Approximation*, *QPAC Approximation*, *The Baradar–Hesamzadeh Approximation*...
— Linéaire/Quadratique : autour d'un point, DC-PF [29], découplage P-Q ...

Marché endogène Dans l'hypothèse où un marché P2P atteint de grande dimension (en nombre d'agents et en quantité de puissance échangée), il devient important de prendre en compte les contraintes du réseau. Lorsque l'on considère le problème d'OPF avec un marché décentralisé, on appelle cela un marché **endogène**. Ce problème peut s'écrire ainsi (1.7). On cherchera à résoudre ce problème sous une forme totalement décentralisée ou avec juste la partie de marché décentralisée.

$$\min_{T,P} \quad \sum_{n\in\Omega} \left(g_n(p_n) + \sum_{m\in\omega_n} \beta_{nm} t_{nm} \right) \tag{1.7a}$$

$$\text{t.q. } (1.2b) - (1.2g) \tag{1.7b}$$

$$\textbf{contraintes physiques} \tag{1.7c}$$

$$\textbf{contraintes operationnelles} \tag{1.7d}$$

Ainsi, le problème que nous allons considérer est le suivant, Fig. 1.9. Le mar-

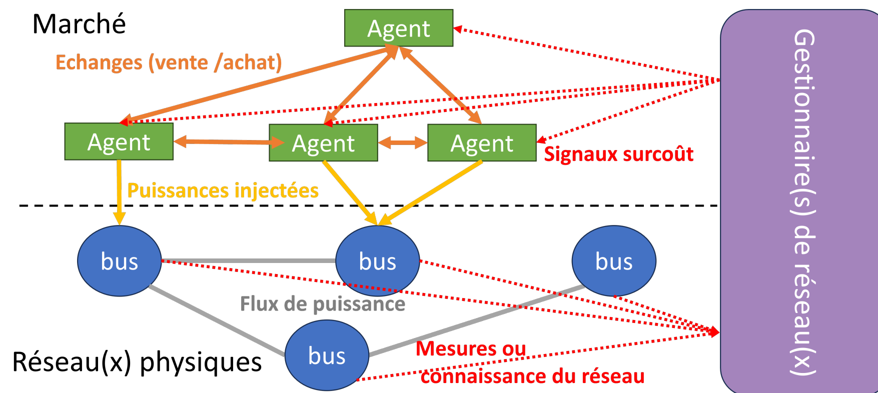


FIGURE 1.9 – Organisation du marché Simulé

ché est constitué d'agents qui s'échangent des messages. Ces agents injectent leur puissance sur le réseau. Ce réseau est monitoré par un ou plusieurs gestionnaires de réseau qui vont intervenir sur le marché pour faire en sorte que la solution du marché respecte les contraintes. Le gestionnaire peut aussi simuler l'état actuel du marché pour voir ses potentiels effets sur le réseau et donc intervenir avant que la puissance ne soit réellement injectée. Le fait que les contraintes soient prises en compte pendant la résolution du marché en fait ce que l'on appelle un **marché endogène**. Ce type de marché est à opposer au marché exogène où le gestionnaire impose des pénalités ou contraintes aux agents a priori pour espérer respecter les contraintes.

On peut remarquer que cette modélisation permet de prendre en compte l'ensemble des problèmes précédents. En effet, en enlevant les contraintes de réseau, on retrouve un marché. De plus, la représentation P2P du marché permettant de représenter n'importe quelle configuration, cette modélisation permet de garantir la généralisation de ce travail.

1.3 Périmètre de l'étude

1.3.1 Problématique de recherche

La simulation est un pilier indispensable pour la gestion du réseau, pour l'optimisation des stratégies des agents, ou pour la vérification de nouvelles réglementations ou gestions avant exercice. Ces simulations sont soumises à de grandes contraintes de temps. En effet, lors du fonctionnement, il est important que les simulations soient assez rapides pour permettre de réagir en temps réel. D'un autre côté, les simulations à long terme peuvent être réalisées indépendamment du fonctionnement. Cependant, la logique combinatoire lors des dimensionnements ou la

plage de temps à regarder pour prévoir le futur rendent difficile le fait d'empêcher que les temps de calcul explosent et que la simulation devienne irréalisable.

Dans le cadre de cette thèse, on se concentrera sur la résolution du verrou du temps de calcul d'application de simulations dites *off-line*. C'est-à-dire que cette application sera indépendante de tout effet temps réel. Ainsi, la récupération de données, l'acquisition, l'envoi de messages et traitement à la volée sont en dehors du périmètre de cette thèse. De manière non exhaustive, cette application permettra de vérifier le comportement du réseau face à des évènements extrêmes [72] tout en garantissant une cohérence temporelle des données. Elle permettra aussi de vérifier la robustesse du réseau face à des accidents matériels et des problèmes de communications [21]. Simuler de vastes réseaux sur des intervalles étendus devrait permettre de valider la pertinence de nouvelles approches de gestion [7] ou de diverses combinaisons de productions, en tenant compte de diverses architectures du réseau. Cela devrait aider les gestionnaires de réseau à entretenir, à investir et à réaliser la planification de leur réseau. Résoudre rapidement permettra de faire des simulations combinatoires, telles que la sécurité à N-1 ou N-k (garantie de fonctionnement lorsqu'il manque k équipements), ou de régler différents paramètres. L'objectif étant d'être plus rapide que du temps réel, ces simulations pourront aussi être utilisées dans une certaine mesure pour du temps réel. Enfin, le framework développé pourrait permettre d'entraîner plus rapidement des intelligences artificielles.

Quels que soient les travaux de recherche dans ce domaine, la décentralisation augmentant le nombre de calculs et ralentissant la convergence, les cas d'études sont soit petits, soit soumis à des temps de calcul augmentant rapidement avec la taille. En effet, les applications ne peuvent pas s'appuyer sur la puissance de calcul distribuée et sont donc réalisées de manière centralisée sur une seule machine de calcul ou cluster. L'objectif dans cette thèse est donc de mener une démarche détaillée tout le long du manuscrit appelée Adéquation Algorithme Architecture dans un but d'accélération des calculs pour la simulation de ce type de problème. Cette accélération devrait permettre de faire de la simulation de grands cas d'étude sur de longues périodes de temps.

Comme démontré précédemment, la modélisation d'un marché endogène permet de déterminer le plan de puissance optimal de manière décentralisée en prenant en compte les contraintes du réseau. De plus, cette formulation inclut les autres problèmes majeurs de la gestion des réseaux électriques. Le marché pair à pair étant une généralisation des autres organisations, tous les paradigmes peuvent être résolus par cette formulation. Ainsi, appliquer cette démarche d'Adéquation Algorithme Architecture sur un marché endogène pair à pair permet d'aussi optimiser les autres types de problèmes.

Comme l'accélération des problèmes du réseau électrique est un domaine de re-

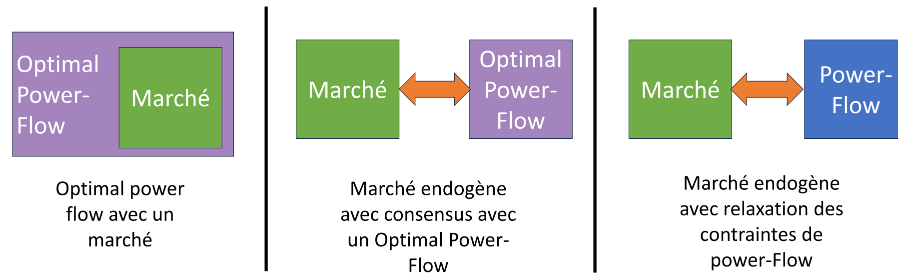


FIGURE 1.10 – Différentes manières de résoudre le marché endogène

cherche très récent et que des travaux sont toujours en cours, nos solutions seront organisées par blocs, Fig. 1.10. En effet, on peut remarquer qu'il existe plusieurs manières de résoudre un marché Endogène. Mathématiquement, un marché endogène est un OPF où l'on considère les échanges entre agents. Le fonctionnement actuel ressemble plus à un consensus entre un marché qui prend en compte les échanges et un OPF qui considère les contraintes réseau. On pourrait aussi, comme dans [26] relaxer les contraintes réseau dans les fonctions coûts des agents. Cette organisation permet d'appliquer la démarche d'Adéquation Algorithme Architecture (détaillé plus tard) sur l'ensemble des problèmes indépendamment et utiliser les algorithmes de la littérature. Ainsi, si une nouvelle recherche découvre un nouvel algorithme ou bibliothèque pour résoudre un des problèmes, celui-ci peut être inclus sans modifier le reste de l'application.

L'organisation par bloc a aussi l'avantage dans un contexte d'application d'open source de permettre plus facilement la collaboration entre les utilisateurs ou alors de permettre aux utilisateurs d'inclure leur propre implication lors de leur utilisation.

De plus au-delà de l'accélération des blocs, une étude sur les interactions et propriétés de l'ensemble pourra être réalisée.

Ce type de problème considère deux plans. Le premier est le plan matériel et considère le réseau électrique (sa topologie et les grandeurs physiques). Le second plan est économique et considère les acteurs (aussi appelés agents) du marché électrique et leur échange économique.

1.3.2 Modélisation du réseau

Dans le cadre de ce travail, on supposera un réseau équilibré sans harmoniques. Ceci nous permettra de ne considérer qu'une seule phase pour le réseau. Le réseau est composé d'un ensemble $\mathcal{B}$ de bus et d'un ensemble $\mathcal{L}$ de lignes. À chaque ligne l est associé un unique couple de bus (i, j). De même, il y a au plus une ligne pour chaque couple de bus. Ainsi, on notera la correspondance $l = (i, j)$ pour la ligne

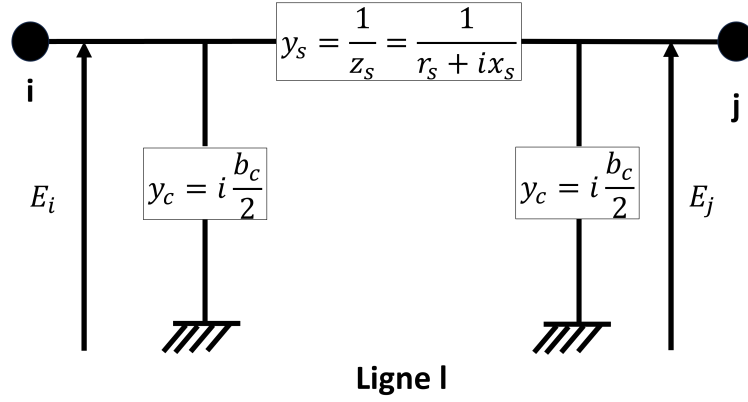


FIGURE 1.11 – Modélisation d'une ligne par un modèle en Π

l allant du bus i vers le bus j. Soit Y une grandeur caractéristique de la ligne $l = (i, j)$, les notations Y_{ij} et Y_l sont équivalentes.

On notera $\mathcal{L}_b$ l'ensemble des lignes qui partent ou arrivent au bus b. On peut définir une matrice C représentant la correspondance entre les bus et les lignes, tel que (ssi signifiant si et seulement si) :

$$C_{il} = 1 \qquad \text{ssi } l = (i, j) \tag{1.8a}$$
$$= -1 \qquad \text{ssi } l = (j, i) \tag{1.8b}$$
$$= 0 \qquad \text{sinon} \tag{1.8c}$$

Ces mêmes informations sont aussi stockées sous la forme d'une matrice $CoresLB$ de taille $L*2$, où la ligne de la matrice correspond aux 2 bus de la ligne représentée ($CoresLB_l = [i,\ j]$ si $l = (i, j)$).

Chaque ligne est modélisée par un modèle en Π [73], représenté sur la Fig. 1.11. Il est important de noter que, lorsque i n'est pas utilisé comme indice, il représente le nombre imaginaire tel que $i^2 = -1$. Chaque ligne possède donc une impédance en série notée $\underline{z}_s = r_s + i \cdot x_s = \frac{1}{y_s}$. Chaque ligne possède aussi une admittance en parallèle $\underline{y}_c = i \cdot \frac{b_c}{2}$ à chaque bus. Pour chaque bus, une admittance shunt peut être ajoutée $\underline{y}_{sh}$.

Ainsi, en notant respectivement g et b la partie réelle (réactance) et imaginaire (susceptance) des admittances, l'admittance en série est notée $\underline{y}_{s,l} = g_l + i \cdot b_l$ pour chaque ligne l. On définit l'admittance parallèle de chaque bus b comme étant égale à $y_{p,b} = y_{sh,b} + \sum_{l \in \mathcal{L}_b} y_{c,l}$.

Ainsi, on définit respectivement deux vecteurs de taille $L = |\mathcal{L}|$ et deux de tailles $B = |\mathcal{B}|$ pour stocker respectivement les parties réelles et imaginaires des impédances en série des lignes et admittances en parallèle des bus.

On définit aussi deux matrices de taille $B{\cdot}B$ notées $\mathbf{G}_{grid}$ et $\mathbf{B}_{grid}$ pour représenter respectivement les réactances et les susceptances des lignes et des bus. Ces matrices permettent de passer des tensions aux courants avec $\underline{I} = (G_{grid} + i \cdot B_{grid})\underline{E}$. Ainsi, pour ces matrices, on a respectivement des termes non diagonaux non nuls *ssi* une ligne existe entre les bus i et j, et valant $G_{ij} = -g_{s,l}$ et $B_{ij} = -b_{s,l}$. Les termes diagonaux sont égaux à :

$$\begin{aligned} G_{ii} &= g_{sh,i} + \sum_{l \in \mathcal{L}_i} g_{s,l} \\ B_{ii} &= b_{sh,i} + \sum_{l \in \mathcal{L}_i} b_{s,l} + \frac{b_{c,l}}{2} \end{aligned} \tag{1.9}$$

Le fait d'utiliser ces deux manières de représenter les impédances nous permettra de nous adapter aux différents algorithmes existants.

Chaque ligne possède également une limite thermique, celle-ci sera exprimée en courant (noté $\overline{I}$) ou en puissance (noté $\overline{\phi}$). Sur chaque bus, les inconnues sont la tension et la puissance. On notera la puissance $\underline{s}^{bus} = p^{bus} + iq^{bus}$ injectée dans chaque bus (négative si soutirée). On notera $\mathbf{W} = (p^{bus}, q^{bus})$ le vecteur des puissances (actives puis réactives). Pour la tension, on pourra utiliser la notation angulaire $\underline{E}_b = V_b e^{i\theta_b}$ ou une notation rectangulaire avec $\underline{E}_b = e_b + if_b$. Quelle que soit la notation choisie, on utilisera la notation $\mathbf{E}$ pour représenter le vecteur stockant la tension de tous les bus. La puissance sur la ligne $l = (i, j)$ sera notée $\underline{S}_l = S_{ij} = P_l + i \cdot Q_l$. Selon les équipements se trouvant sur le bus, 3 types de bus sont possibles.

- Le bus de référence (nœud **REF**), impose la référence de tension du réseau, ainsi E_b est connu, p et q sont à déterminer.
- Les bus avec des charges ou des productions d'énergies renouvelables sont des nœuds **PQ**, *i.e.* les puissances p et q sont connues et $\underline{E}_b$ est à déterminer.
- Les bus avec des producteurs contrôlables sont des nœuds **PV**, c'est-à-dire que p et V_b sont connus et il reste à déterminer q et θ_b.

Dans le cadre de cette thèse, on considéra une intégration massive des énergies renouvelables. Ainsi, on considèrera que l'ensemble des nœuds, sauf celui de référence, sera des nœuds de type **PQ**.

Dans le cadre d'un réseau de distribution, on supposera le réseau radial. Ainsi, dans ce cas-là, on définit A_b l'unique ancêtre du bus b (qui existe pour tous les bus sauf celui de référence) et C_b l'ensemble des "enfants" du bus b (possiblement vide). On pourra aussi noter respectivement $\underline{S}_b$, l_b et v_b le flux de puissance arrivant au bus b, la norme du courant au carré arrivant au bus b et la norme au carré de la tension au bus b ($v_b = V_b^2$). Un schéma peut être trouvé dans la Fig. 1.12.

Il existe de nombreuses relaxations et approximations du réseau pour simplifier les résolutions du PF ou de l'OPF en rendant le problème convexe. Dans cette

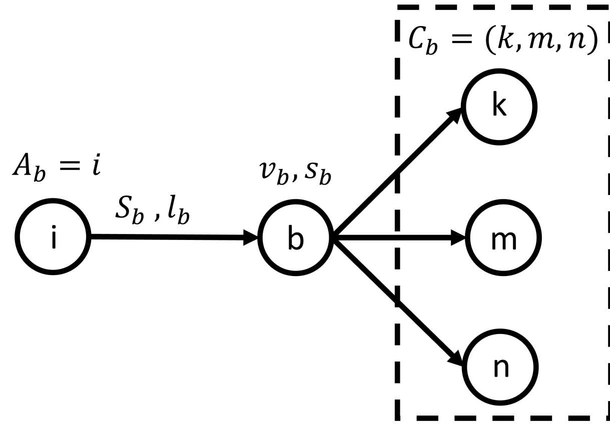


FIGURE 1.12 – Ancêtre et enfants d'un bus b

thèse, on se contentera de deux possibilités.

— Dans le cas d'un réseau radial, on utilisera la relaxation du DistFlow, celle-ci étant exacte sous certaines conditions [59]. Cette relaxation correspond au fait de transformer l'égalité physique $v_b \cdot l_b = S_b^2$ en inégalité.
— Dans le cas d'un réseau de transport, on utilisera une approximation DC, étant une des approximations les plus courantes et pouvant bien se justifier sur les réseaux de transport.

Dans la suite de cette thèse, on parlera de modélisation du réseau **AC** dans le premier cas, et **DC** dans le dernier cas. Les hypothèses induites par cette dernière approximation sont les suivantes :

— la résistance r de chaque ligne est négligeable devant la réactance x : donc $z_l \approx i \cdot x_l$ et $Y_l = -i/x_l = B_l \cdot i$ pour toutes les lignes ;
— il n'y a pas de chute de tension entre les bus, chaque tension a donc pour amplitude la tension de base $V_b = V_0$;
— la différence d'angle entre chaque ligne est suffisamment faible pour que $\cos(\delta_{km}) = 1$ et $\sin(\delta_{km}) = \delta_{km}$.

1.3.3 Modélisation des agents

On considère Ω l'ensemble des N agents répartis sur les bus. Soit b un bus fixé, on note $\mathcal{N}_b$ l'ensemble des agents situés sur ce bus. En supposant qu'un agent ne peut être situé que sur un bus, on peut générer des vecteurs de correspondances qui permettent de passer des bus vers les agents ou inversement. On notera aussi une matrice $\mathbf{I}$ telle que $I_{b,n} = 1$ si l'agent n est sur le bus b et 0 sinon. Cette notation permet de passer de la puissance des agents $\mathbf{P}^{agent}$ à la puissance injectée dans les bus $\mathbf{P}^{bus}$ avec $\mathbf{P}^{bus} = \mathbf{I} \cdot \mathbf{P}^{agent}$. Cette modélisation permet aussi d'envisager des cas où certains agents du marché gèrent plusieurs agents physiques et ont donc

une influence sur plusieurs bus du réseau. Cependant, cette possibilité ne sera pas explorée dans cette thèse.

Les agents peuvent être des consommateurs Ω_c, des producteurs Ω_g ou des consommacteurs Ω_p. L'organisation de ce réseau est consommateur centrique [74]. Cela signifie que, comme pour le réseau actuel, ce sont les consommateurs qui, par leur consommation, imposent le niveau de puissance à produire. Ainsi, la seule volonté des producteurs flexibles est de produire suffisamment pour les consommateurs tant que le prix de vente est plus grand que le coût marginal de fonctionnement. Les producteurs renouvelables ayant un coût marginal nul chercheront à produire le plus possible. Ils seront donc modélisés comme des consommateurs avec des puissances positives. Dans cette thèse, on supposera que les consommateurs sont flexibles. Afin de pouvoir supposer que cette flexibilité est continue (et ne pas devoir considérer l'activation d'un équipement). On supposera que chaque agent flexible a accès à un moyen de stockage. Ce moyen de stockage ne sera pas modélisé. Ainsi, le surplus ou le manque de consommation par rapport à leur puissance visée $P0$, sera absorbée par le moyen de stockage.

Ainsi, chaque agent a une fonction coût représentant ce qu'il souhaite consommer ou produire ($P0$) et le coût de s'éloigner de cet optimum. Pour un producteur flexible, la fonction représente le coût de production d'une certaine quantité d'énergie. Pour un consommateur ou un producteur renouvelable, le minimum correspond à la puissance souhaitée par l'agent. La pente autour du minimum représente la variation de prix nécessaire pour que ces agents acceptent d'utiliser leur flexibilité. Ainsi, lorsque les prix sont bas (déficit de consommation), le producteur renouvelable accepte de moins injecter et le consommateur pourra consommer plus. L'inverse arrive si les prix augmentent. Cette fonction coût est a priori connue uniquement de l'agent et rien n'impose de forme particulière lors du fonctionnement réel. Cependant, pour réaliser la simulation de la gestion, il faut déterminer les fonctions coûts des agents pour être capable de simuler le comportement des agents. Cette connaissance sera utilisée pour le choix de l'algorithme afin d'être le plus rapide possible. Cependant, on simulera un problème décentralisé où seul l'agent connaitra la valeur de sa fonction coût.

En se basant sur [25], [26], [21], on supposera pendant toute la thèse une fonction coût quadratique. Lors d'une optimisation, les termes constants n'influent pas sur la position de l'optimum. Ainsi, les deux expressions suivantes pourront être utilisées pour représenter la fonction coût des agents :

$$g(p_n) = 0.5a_n \cdot p_n^2 + b_n \cdot p_n = 0.5a_n(p_n - P_0)^2 \tag{1.10}$$

On peut remarquer qu'en multipliant la fonction coût par une constante, la puissance pour laquelle la fonction est minimale ne change pas. Ainsi, la valeur de a_n n'a pas en soi d'impact sur la minimisation atteinte pour un agent seul, mais

ce qui importera sera les rapports entre les différents a_n des agents. Le coefficient a_n représente la "flexibilité" des agents. En effet, plus il est faible, plus les agents sont flexibles, car la valeur de la fonction coût change moins vite en s'éloignant de l'optimum. La valeur de a_n sera souvent choisie arbitrairement en fonction du type de l'agent (consommateur, producteur contrôlable ou producteur renouvelable).

De manière générale pour un consommateur, la valeur de a_n représente sa flexibilité, *i.e.* à quel point il accepte de s'éloigner de la puissance voulue ($P0$). La valeur de b_n est déterminée par l'emplacement du minimum : $b_n = -P0 \cdot a_n$. La fonction coût d'un producteur renouvelable suit le même principe que celle des consommateurs. Pour un producteur contrôlable, b_n représente le coût variable de production et a_n est faible pour à la fois permettre d'être strictement convexe tout en ayant une fonction presque linéaire. On pourrait imaginer des expressions plus complexes pour déterminer les valeurs de a_n et b_n des agents. Par exemple, choisir les coefficients pour prendre en compte leur stratégie selon leur prévision, leur moyen de stockage... Chaque agent possède aussi des limites sur les puissances totales qu'il peut échanger, notées $\underline{p}_n$ et $\overline{p}_n$ qui permettent aussi de représenter leur flexibilité. En effet, que les valeurs de a_n et b_n soient élevées ou basses, si $\underline{p}_n = \overline{p}_n$, l'agent n'est pas flexible.

Il est intéressant de noter que cette fonction coût peut être étendue dans le cadre d'un réseau multiénergie. On peut donc aussi prendre en compte la volonté de consommation/production d'autres puissances, comme le thermique ou le gaz, par exemple.

Le marché peut aussi être utilisé comme un artefact d'optimisation pour atteindre un point de fonctionnement respectant les contraintes. On peut ainsi rajouter un marché de "puissance réactive" tel que l'optimum de ce marché respecte les contraintes physiques du réseau. Ce marché n'a pas de but économique. En réalité, l'idée d'acheter ou de vendre de la puissance réactive n'a pas de sens, car il ne s'agit pas réellement d'une consommation. Cet artefact permettra par exemple, de prendre en compte de manière décentralisée les consignes du gestionnaire de réseau pour respecter les contraintes de tension. Le fait que les énergies renouvelables peuvent contrôler à la fois leur puissance active et réactive rend ce type de résolution possible.

Quelle que soit la puissance considérée, on notera celle-ci avec la lettre q. La nouvelle fonction coût des agents peut s'écrire :

$$g_n(p_n, q_n) = \frac{1}{2}a_n^p p_n^2 + b_n^p p_n + \frac{1}{2}a_n^q q_n^2 + b_n^q q_n \tag{1.11}$$

Dans le cadre de la thèse, on supposera que chaque agent sera relié avec tous les autres pour pouvoir échanger la puissance q. Le fait qu'un agent cherchera à acheter ou vendre cette deuxième puissance (*i.e* le signe de la puissance visée $Q0$) sera indépendant du type de l'agent. Ces échanges ne seront pas soumis à des

préférences hétérogènes $\beta^{q}_{nm} = 0$. En lien avec les modélisations rencontrées dans le domaine des **PF**, on désignera sous les noms de "marché DC" ou de "marché AC" un marché P2P sur lequel s'échangent uniquement de la puissance active ou bien de la puissance active et réactive.

Lorsque l'on devra prendre en compte les pertes dans les lignes dans le cadre d'un marché pur, un agent devra être ajouté. Cet agent devra échanger sur le marché pour "racheter les pertes" dans les lignes. Ceci permettra de forcer l'ensemble des agents à produire plus tout en ayant un marché à l'équilibre. Cet agent n'est pas physiquement présent sur le réseau.

Pour rappel, la modélisation choisie pour le problème est un marché pair à pair, car il a été prouvé que ce type de marché était une généralisation de tous les autres types de marchés (communauté, centralisée, hétérogène). Chaque agent peut ainsi échanger avec n'importe quel autre agent. Cependant, le lien entre 2 consommateurs ou 2 producteurs étant inutiles (ils ne peuvent pas échanger), ils ne seront pas considérés.

2 Les puissances de calculs mises en jeu

2.1 L'accélération matérielle

Durant les débuts de l'informatique, un grand effort était réalisé pour implémenter les applications voulues sous de grandes contraintes de temps de calcul et de mémoire disponible. En effet, la puissance de calcul du matériel disponible était très limitée. La loi de Moore a prévu un doublement des transistors dans les puces puis par processeur. Cette loi s'étant vérifiée, cela a permis de grandement relaxer ces contraintes de développement.

En effet, lorsqu'une application avait des contraintes physiques (de temps de calcul ou de mémoire) à respecter. Il était plus rapide de faire des applications sans effort d'optimisation et d'attendre la nouvelle génération de processeur ; que de passer du temps à optimiser une application sur un processeur particulier. En effet, le gain d'optimisation de l'application sur un vieil équipement est plus faible que celui apporté par l'amélioration de l'équipement.

La difficulté de dissiper l'énergie des pertes dans les transistors de plus en plus petits couplée avec les latences d'utilisation d'autres composants (tel que les mémoires) a mis un arrêt à cette évolution rapide des performances. D'un autre côté, l'augmentation des dimensions, de la complexité et des précisions demandées aux différents modèles utilisés (quel que soit le domaine étudié) appelle une demande pour une puissance de calcul de plus en plus grande.

Pour répondre à cette demande, deux voies concurrentes ont été explorées. La première est le développement et l'amélioration de *super-computer* (ou supercal-

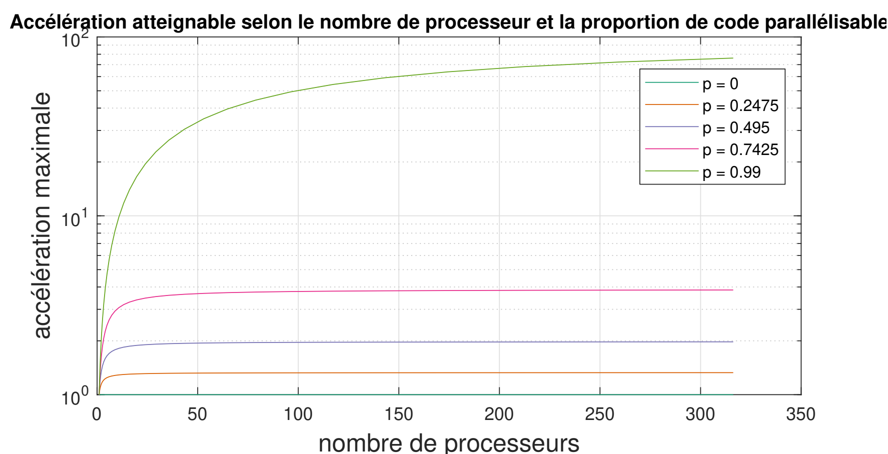


FIGURE 1.13 – Application de la loi d'Amdahl

culateur). Cette voie est particulièrement coûteuse à cause de l'équipement nécessaire. Ce genre de système est parfaitement adapté pour des entités centralisatrices, comme pour notre cas la gestion actuelle du réseau électrique. En effet, ce sont les calculs réalisés dans les super calculateurs situé dans le Centre National d'exploitation du Système (CNES) de RTE qui permettent d'ajuster à tout moment la consommation et la production. Cependant, l'utilisation de la parallélisation dans un supercalculateur est soumise aux mêmes limitations que dans un processeur classique : les dissipations thermiques limitent le nombre de nœuds dans un volume donné, la communication entre les nœuds (accès mémoire lointain) limite les performances. Dans tous les cas, l'application est limitée par sa partie séquentielle. En effet, soit p la proportion du code pouvant être parallélisé et soit s le nombre d'unités permettant la parallélisation (cœurs, processeurs...), la loi Amdahl nous donne l'accélération maximale atteignable :

$$SpeedUp = \frac{1}{1 - p + \frac{p}{s}} \tag{1.12}$$

Cette loi, représentée dans la Fig. 1.13 nous montre que, quel que soit le nombre de processeurs que l'on peut utiliser, il y a une accélération maximale atteignable dépendant de la partie qui ne peut pas être parallélisée. Ainsi si 3/4 du code peut être parallélisé, l'accélération maximale est de 5 même avec des milliers de processeurs.

Dans le cas où un supercalculateur ne peut pas être utilisé, d'autres moyens sont explorés. En effet, de nombreuses autres technologies existent. Ainsi afin d'atteindre les performances voulues sans avoir besoin de supercalculateurs, concevoir l'unité de calcul est une solution. En effet, le fait de concevoir un système sur puce (**SoC** : system on Chip) pour une application spécifique permet de n'en garder

que l'essentiel et ainsi être plus efficaces.

Sans vouloir être exhaustifs, trois systèmes différents nécessitant une puissance de calcul embarquée seront présentés.

Dans le domaine de la téléphonie, des besoins de plus en plus importants en traitement d'image se font ressentir. En essaie ainsi d'ajouter dans le téléphone des options de filtre, reconnaissance d'image, retouche, etc. L'utilisation de cartes hétérogènes ou SoC, composées directement des différents composants spécialisés, permet d'améliorer grandement les performances pour une application donnée. Ainsi on retrouve sur cette carte par exemple une unité DSP pour le traitement du signal, une unité pour l'analyse d'image via l'IA. Cependant, la création de telle puce n'est pas accessible pour tous, ainsi elles ne sont pas adaptées à du prototypage.

L'utilisation de carte FPGA permet de concevoir une carte adaptée à l'utilisation à des coûts plus faibles. Ces cartes sont ainsi utilisées lors de la gestion décentralisée d'un SmartGrid [100]. Pour que cette gestion soit possible, il faut que le réseau possède des milliers d'unités permettant de mesurer l'état du réseau, recevoir les messages, calculer les actions à réaliser et contrôler les équipements alentour. Le design de ces unités de calculs est critique. En effet, les tâches à réaliser sont multiples. La grande quantité d'unité à déployer nécessite que le matériel utilisé soit peu cher (coût monétaire ou en matériau) et énergiquement performant pour que le SmartGrid permette d'effectivement économiser de l'électricité.

Le troisième exemple [82] est la conception de véhicule autonome. Un grand défi de ce système est la capacité de se localiser et d'analyser l'environnement. La méthode du **SLAM** (Simultaneous Localization And Mapping) peut être utilisée pour réaliser cette mission. Cependant, ces algorithmes sont si demandants en ressources qu'ils sont testés sur des ordinateurs hautes performances. Le contexte de la voiture autonome impose de grandes contraintes sur les performances en termes de temps de calcul, de consistance des résultats et d'efficacité énergétique. Dans ce cas-là, la démarche d'adéquation algorithme vise à la fois à déterminer l'algorithme le plus adapté tout en concevant l'architecture qui respecte les contraintes.

Pour rendre plus accessible cette grande puissance de calcul au plus grand nombre se développe en parallèle le *personal super-computer* depuis le début des années 2000. Ce système est une plate-forme d'accélération de calculs généralement constituée d'un processeur CPU généraliste considéré comme l'hôte d'une ou plusieurs cartes accélératrices telles que des Graphic Processing Unit (GPU), des Massively Parallel Processor Array (MPPA) de Kalray, des Digital Signal Processing (DSP), des puces FPGA ou enfin des Neural Processing Unit (NPU). Ces systèmes peuvent être directement vendus en tant que *personal super-computer* comme le système Nvidia DGX. Ces systèmes sont conçus pour permettre aux entreprises (ou structures telles que les hôpitaux ou université) d'utiliser des mé-

thodes très demandant en termes de puissance de calcul comme l'IA. De plus, les ordinateurs grands publics sont eux même de plus en plus performants. Ainsi l'utilisation de CPU avec des cartes accélératrices comme un GPU est possible directement sur ces ordinateurs.

Quel que soit le système utilisé, afin d'atteindre les meilleures performances, la première étape est de déterminer quelles architectures, cartes et/ou utilisations du cluster sont optimales pour l'application voulue. En parallèle de cette étape, il faut adapter l'application ou l'algorithme pour utiliser optimalement le système choisi. Le tout permettant de respecter les différentes contraintes de temps de calcul (rapide/temps réel), de mémoire (stockage ou flot de données), de consommation (batterie, thermique) ...

La détermination des contraintes, des architectures compatibles et l'adaptation des algorithmes pour réaliser l'application optimale est une voie de recherche dans de nombreux domaines scientifiques.

De nombreux travaux sur l'accélération d'applications sur les différents composants CPU, GPU et FPGA ont déjà été réalisés. Ces applications sont dans différentes thématiques très variées telles que (sans être exhaustifs) les radars [75], la génomique, la biologie moléculaire [76], l'analyse d'option financière [77], médical [78], le traitement d'image [79]- [80] et la cartographie [81]...

Cependant, ces applications nécessitent systématiquement un important travail spécifique sur l'algorithme à implémenter afin d'exploiter les spécificités de l'architecture cible. S'il est possible de récupérer des règles et certains résultats démontrés, il reste difficile de transposer ces résultats dans un autre domaine.

Les différences de caractéristiques entre les cartes accélératrices par rapport au CPU font en sorte qu'uniquement certaines opérations sont efficaces sur ces cartes. De plus l'utilisation de carte ajoute un surcoût dû au transfert mémoire d'autant plus important que l'on accélère lorsque l'on est à de très grande dimension. Ainsi, un travail d'implémentation sans une bonne méthode risque de donner des accélérations qui auraient pu être obtenues directement sur CPU en optimisant l'implémentation [84]. De plus, que ce soit en comparant des langages de programmation ([85]), des algorithmes ([41]) ou des architectures différentes ([84]), on constate que l'influence du programmeur et de la plateforme de test est loin d'être négligeable. Par exemple, l'article [41] contient à la fin de celui-ci une discussion entre les reviewers qui posent des questions et les auteurs qui y réponde. Dans cette discussion, les reviewers s'interroge sur le fait que leur implémentation d'un des algorithmes présentés est bien plus rapide chez eux.

Ainsi des temps de calcul seront présentés dans cette thèse afin de permettre d'avoir une évaluation de la vitesse du simulateur proposé. Ceci permettant au lecteur de savoir l'ordre de grandeur de temps de la simulation de chaque problème, et de se comparer avec l'état de l'art. Cette comparaison permettra de pointer les

différences entre les implémentations, les langages ou le matériel utilisé (notamment selon l'ancienneté de l'article). Les vrais points d'études de cette thèse seront les suivants.

- La présentation et le suivi de la méthode Adéquation Algorithme Architecture afin de permettre la réplication et l'analyse du travail réalisé. En effet, celle-ci étant dans les grandes lignes indépendantes de la technologie de la carte CPU/GPU/FPGA utilisée ou des performances de programmation. L'ensemble des étapes de cette démarche sera présenté dans le chapitre suivant.
- L'accélération entre les différentes versions pour démontrer l'effet d'optimisation spécifique et la comparaison des comportements des algorithmes.
- L'évolution du temps en fonction des dimensions du problème pour vérifier si, en effet l'implémentation permet ou non de résoudre le verrou scientifique du passage à l'échelle.

2.2 L'accélération des calculs dans le domaine du réseau électrique

Les architectures utilisées pour accélérer les calculs dans le domaine du génie électrique dépendent fortement du problème considéré. Ainsi, dans cette partie, les manières d'accélérer seront séparées par problème.

2.2.1 Marché

Dans le cas de la résolution d'un marché centralisé ou décentralisé, l'architecture essentiellement utilisée est le CPU. En effet, dans le cadre d'un marché décentralisé, le passage à l'échelle, en matière de temps de calcul, du problème de l'implémentation réel est garanti par sa décentralisation. Ce domaine étant encore nouveau, les articles se concentrent sur l'implémentation et non pas sur le fait de réaliser des simulateurs. Ainsi, les dimensions des cas d'études restant assez petites dans ces articles, les temps de calcul restent assez faibles pour ne pas nécessiter d'accélération matérielle. Par exemple, les articles [25] et [26] considèrent un cas d'une trentaine d'agents sur un seul pas. L'article [38] considère une simulation sur un an, mais le cas ne contient que 12 agents et 2 bus. Dans la majorité des cas, le temps n'est même pas indiqué, car, lors de l'implémentation réelle, les calculs seront parallélisés sur les différents agents et le temps dominant risque d'être les communications [21].

Cependant, afin de déterminer les propriétés algorithmiques des algorithmes de décentralisation de marché l'étude de grands cas est nécessaire. Ainsi, [20] utilise des *super-Computer* pour paralléliser les calculs sur différents cœurs. Sur le cluster

du DTU, 912 cœurs divisés en 38 nœuds ont été utilisés sur les 2500 cœurs dont il est composé.

Le *framework* MASCEM [30] permet de simuler le fonctionnement d'un marché en se concentrant sur les stratégies des différents agents se basant sur des données historiques de la prévision ou de l'apprentissage. Chaque agent est créé sur un thread Java permettant de paralléliser les calculs. Cet article ne fournit aucune indication sur la durée nécessaire, mais, d'après [31], la version initiale, qui ne tenait pas compte des stratégies les plus évoluées, mettait 1 minute pour traiter un cas avec 24 agents. Ce temps étant beaucoup trop long pour respecter les contraintes de temps inhérentes à de la simulation pour de la prise de décision. L'architecture du code a été modifiée et des heuristiques ont été ajoutées pour pouvoir réduire le temps de calcul et être plus de 10 fois plus rapides. La nouvelle version met 1 min pour environ 1500 agents, ce qui peut être toujours trop lent pour certaines applications. Ainsi, un temps compris entre 4 et 20 s est atteignable via l'utilisation d'heuristique et en diminuant le nombre d'agents.

2.2.2 Power Flow

Le problème de Power Flow est une résolution d'un système d'équations non linéaires. Ce problème était historiquement résolu sur des *super-Computer* [42] qui fonctionnaient sur des milliers de nœuds. Pour que des chercheurs puissent résoudre ce problème sur leurs ordinateurs personnels, de nombreuses recherches algorithmiques ont été menées [41], [43]. Ces recherches visaient à trouver les meilleurs algorithmes pour résoudre ce problème ou améliorer ceux existant, par exemple en changeant l'utilisation de la mémoire. Les temps nécessaires pour résoudre des cas de même envergure peuvent varier considérablement en fonction du matériel utilisé et du programmeur, allant de quelques secondes à quelques minutes (voir les commentaires des relecteurs dans la discussion de [41]).

La recherche plus récente utilise les avancées dans les cartes accélératrices de type GPU et FPGA pour paralléliser les calculs. L'utilisation du GPU permet d'accélérer les calculs à partir d'une certaine dimension du problème [44]. Celle-ci peut aussi se faire grâce à l'utilisation de texture en ayant linéarisé le problème pour faciliter la résolution [45]. Le GPU permet aussi de faire la résolution de multiple Power-Flow en simultané [46]. Une accélération de 2 pour une résolution de 200 cas avec 9241 bus en simultané est atteinte par rapport à une version sur CPU sérialisé. Cette faible accélération montre l'importance d'optimiser le partitionnement sur CPU-GPU : paralléliser ne suffit pas. Dans [47], l'évaluation de milliers de *Probalistic Power Flow* est rendu 165 fois plus rapide avec l'utilisation d'un GPU plutôt qu'une bibliothèque KLU sur CPU. En ce qui concerne l'utilisation de carte FPGA, l'article [48] estime qu'un speed-up d'au moins 10 est atteignable avec leur implémentation sur FPGA de l'algorithme de recherche de racine, Newton-

Rapshon. L'article [50] montre que la méthode Jacobi tombe en désuétude pour les applications sur CPU, mais qu'elle est extrêmement parallélisable et pourrait devenir plus efficace sur FPGA que les algorithmes plus usuels.

2.2.3 Optimal Power Flow

L'Optimal Power Flow est vraiment l'un des problèmes majeurs de la gestion du réseau électrique et l'un des problèmes où le verrou calculatoire est le plus visible. Pour rappel, même avec la puissance disponible de RTE et avec le réseau actuel, l'OPF est trop lent pour être calculé sans approximation [8]. En effet, le problème d'OPF sur le réseau couple les contraintes de calculs en grande dimension et le fait que le problème soit non convexe.

Tout comme pour le Power-Flow, la première étape d'accélération a été l'étude des différents algorithmes et différentes formulations permettant de résoudre le problème. De par les reviews très fournies, on peut voir les innombrables algorithmes existant pour résoudre [55]- [58]. De cette riche littérature, il en ressort que les algorithmes déterministes les plus efficaces (c'est-à-dire hors Intelligence artificielle et méthodes stochastiques) sont les points intérieurs (et leur variation) dans la plupart des cas. Cependant, les relaxations possibles, notamment dans les réseaux de distribution, peuvent être exactes et permettent d'utiliser des algorithmes plus efficaces [63]. La linéarisation DC ou le découplage P-Q sont des approximations qui continuent d'être utilisées en raison de leur grande simplicité et de leurs résultats robustes. À ces différents algorithmes se rajoute la décentralisation permettant d'augmenter le parallélisme pour éventuellement réaliser une implémentation distribuée [17].

L'utilisation de GPU pour accélérer les calculs de l'Optimal Power Flow est un domaine très récent. Ainsi, on pourra remarquer que certains articles ont tout juste un an. Selon l'article [64], l'utilisation d'un GPU permet d'obtenir un facteur d'accélération de 4 lors d'un démarrage à froid (*cold-start*) et d'environ 100 lors de la simulation de plusieurs pas successifs par rapport au solveur Ipopt sur CPU. Les mêmes auteurs proposent une version en programmation mixte non linéaire et entière sur GPU dans [40]. Dans ce type de problème, l'accélération entre GPU et CPU est d'un facteur 4.7 sur un cas à 300 bus. Les auteurs de [65] proposent un solveur de type point intérieur pour résoudre un AC-OPF sur Julia. Ce solveur permet la réduction du temps de calcul des cas à partir de 10^4 variables. Le facteur d'accélération atteint 5 pour le passage sur GPU de leurs méthodes. Ce facteur atteint plus de 10 si leurs méthodes sont comparées avec les méthodes de l'état de l'art (Ipopt, JuMP.jl, and Ma27).

Dans [66], le calcul d'un *Stability Constrained* OPF est accéléré jusqu'à 25 fois pour un cas d'environ 13000 bus par rapport à une version CPU en séquentiel. L'article montre que l'utilisation d'un CPU à plusieurs cœurs permet de diviser

par deux ou trois le temps de calcul (le GPU ne devenant qu'un peu plus de 6 fois plus rapide).

L'utilisation d'une carte FPGA pour résoudre un DC-OPF est décrite en détail dans [70]. Ils prévoient grâce à leur implémentation un speed-up d'au moins 3.

3 Conclusion sur l'état de l'art et contribution

3.1 Temps de calcul de la gestion du réseau électrique

L'objectif de cette partie est de synthétiser les principaux articles de l'état de l'art sur lesquels se basera la thèse. On regroupe ainsi dans les tableaux séparés par problème les différents travaux sur les algorithmes et les architectures en indiquant les dimensions du problème et les matériels utilisés. Pour rappel, B indique le nombre de bus considéré et N le nombre d'agents. Comme il est possible qu'il n'y ait aucun agent sur un bus, ou que plusieurs soient présents sur un autre, ces deux nombres sont indépendants.

TABLE 1.1 – Référence de l'état de l'art pour les problèmes de marché

Article	Formulation	cible matérielle	taille (N / B)	temps
[25]	ADMM	CPU	31 /39	9.5s
[26]	ADMM	CPU I5-7200U	31 /39	59s
[38]	CI + gradient	CPU I7-6500U	12 / 2	0.1s
[20]	CI	2 Intel Xeon 2650v4	25 /0 300 / 0	0.1s 28s
[30]	MASCEM	CPU	24 /0	58s
[31]	MASCEM	CPU	24 /0 1446 / 0	4s 65s

Le Tab. 1.1 résume les temps de calcul pour différentes résolutions de Marché. Il est important de noter que les formulations MASCEM sont centralisées là où toutes les autres sont des marchés décentralisés.

L'article [26] simule un marché sur le même cas que [25]. Cependant le premier est un marché endogène, ce qui explique des temps de calcul plus long. Pour [38], le temps indiqué est un temps moyen sur un an avec un pas horaire. Enfin pour [20] les temps indiqués dont dû à la communication entre les nœuds de calculs.

On peut voir la différence entre [30] et [31] qui montre l'importance d'optimiser la simulation pour gagner du temps. L'optimisation de MASCEM nécessitant de changer l'ensemble de l'architecture, ceci montre l'importance d'optimiser avant l'implémentation et de privilégier une implémentation qui permet une optimisation sans devoir tout changer. L'article [20] montre qu'en parallélisant les calculs il est possible de réduire drastiquement le temps de calcul de telle sorte que les temps mesurés sont les temps de transfert de données entre les nœuds.

Cependant, il ressort de ce tableau qu'actuellement la simulation à grande échelle de réseau pair à pair n'est pas possible. En effet, dans le cas où l'on considère un marché complètement connecté, le nombre d'échange a une variation quadratique par rapport au nombre d'agents. Ainsi, une multiplication par 10 du nombre d'agents implique une multiplication par au moins 100 des temps de calcul. Ainsi, même en utilisant un supercalculateur, le calcul d'un pas de temps d'un marché de plusieurs milliers d'agents, nécessite presque une heure. Il devient donc impensable de simuler plusieurs dimensionnements sur plusieurs pas de temps.

Dans le tableau Tab. 1.2 peuvent être trouvés des exemples de temps de calcul de différents PF sur des architectures CPU-GPU. Aucun temps n'est indiqué pour des FPGA, car les articles se concentrent sur le speed-up atteignable de leur implémentation, mais n'ont pas mesuré de temps. On peut voir avec ces exemples que l'augmentation des dimensions du problème peut impliquer une augmentation drastique des temps de calcul. Par exemple dans [41] où une multiplication par 2 de la taille donne un temps multiplié par 10. La différence entre les temps de calcul montre que l'utilisation d'une carte accélératrice de type GPU permet d'effectivement réduire les temps de calcul.

On peut cependant remarquer une grande disparité entre les temps qui ne représentent pas la même opération pour tous les articles. On peut remarquer que [49] met plus de temps pour calculer une fois la Jacobienne d'un cas de taille 2383 que [46] pour résoudre 200 cas de taille 9241. Les temps de [44] ne considèrent que le calcul du pré-conditionneur de Chebyshev. On remarque aussi que [41] et [42] ont des temps très élevés, mais cela est sûrement dû à l'ancienneté des articles et donc aux faibles performances du matériel.

Le Tab. 1.3 donne des ordres de grandeur des temps pour la résolution d'OPF. On peut voir que l'OPF peut exister sous différentes formes avec des variables discrètes comme [40], ou non. Il peut aussi être résolu de manière décentralisée, avec des fonctions coût quadratiques ou linéaires, etc. Quel que soit le problème considéré, il apparait de manière assez claire que le temps de calcul augmente plus que linéairement avec les dimensions du problème. La référence [29] nous montre l'importance de bien choisir la formulation et l'algorithme pour réduire les temps de calcul. En effet les différents temps de cet articles sont obtenus en utilisant différent algorithmes.

Table 1.2 – Référence de l'état de l'art pour les problèmes de Power Flow

Article	Formulation	cible	taille	temps
[41]	NR NR GS GS	CPU VAX-ll /785	30 57 30 118	102s 1019s 40s 800s
[42]	NR	IBM 7040	487	51.57s
[44]	DC-PF via NR et Gradient Conjugué	8-core Intel Xeon E5607	188 2736	0.9ms 200ms
		GPU Nvidia Tesla M2070	188 2736	1.5ms 4.2ms
[45]	DC - PF avec Gauss Jacobi	dual core Pentium 4	118 1000	1.1s 390s
		GPU Nvidia 7800 GTX	118 1000	0.48s 93s
[46]	PF- NR	Intel Xeon E5-2620	14 9241	0.1s 197s
		GPU Nvidia Tesla K20c	14 9241	1.5s 86s
[49]	PF - NR	CPU I7-6500U	14 2383	20ms 865s
		GPU NVIDIA GT745m	14 2383	0,07s 20.47s

On peut remarquer que des implémentations efficaces et open sources ont pu être présentées pour la résolution d'AC-OPF sur GPU. Toutes les citations [64]-[67] se concentrent sur la résolution d'un OPF sur réseau maillé. Ainsi pour cette thèse on implémentera uniquement des OPF sur réseau radiaux.

TABLE 1.3 – Référence de l'état de l'art pour les problèmes de Optimal Power Flow

Article	Formulation	cible	taille	temps
[59]	décentralisé via ADMM	Intel Core i5	108 2065	16s 1153s
[29]	décentralisé DC-OPF	CPU	48	$> 0,3s$ $< 483s$
[68]	OPF par Essaim de particules	CPU E5-2650	30 300	212.9s 37 000s
		GPU Nvidia Tesla K20c	30 300	13.1s 2 300s
[66]	Stability Contraint OPF IPM	Intel Xeon E5-2650	678 12 951	15s 1063s
		GPU Nvidia Tesla K20c	678 12951	7.2s 163s
[40]	OPF MINLP	Intel Xeon 6140	9 300	4.1s 653s
		GPU Nvidia Quadro	9 300	9.9s 140s
[64]	décentralisé OPF ADMM + ExaTron [71]	Intel Xeon 6140	1354 70k	2.4s 469s
		GPU Nvidia Quadro GV100	1354 70k	2s 99s
[65]	centralisé OPF IPM	CPU	1354 30k	0.7s 16.79s
		GPU Nvidia	1354 30k	0.73s 94.62s
[8]	MatPower (MINOS)	Intel 3.3Gz	30 300	0.06s 6.3s
	C PDIPM		30 2383	0.05s 12s
[67]	MatPower PDIPM	Intel E5649 Westmere	30 2383	1s 149s
		GPU Nvidia Tesla M2070	30 2383	4.3s 46s

3.2 Contributions

L'analyse de l'état de l'art nous permet de remarquer qu'aucun travail d'accélération matériel n'a été proposé pour le problème de marché décentralisé. En effet, l'objectif étant la démonstration de l'efficacité de mécanisme de gestion et qu'une hypothèse est le fait que le marché P2P a une influence négligeable sur le réseau, très peu de simulations considèrent des cas à grande dimension. L'article [20] augmente les dimensions du marché simulé, et l'utilisation d'un supercalculateur permet de maintenir des temps de calcul faibles. Cependant, ce type de matériel n'est pas accessible à tous, et 300 agents restent une valeur assez faible devant les dimensions sur réseau réel. Un marché endogène a été proposé dans [26], mais les temps de calcul obligent à se limiter à des simulations à faible dimension. De plus, les contraintes sont prises en compte sous la forme d'un DC Power-Flow, ce qui peut ne pas être suffisant. Les contributions de cette thèse sont :

L'adéquation algorithme architecture d'un marché Pair à pair Thomas, Beatrice, et al. "Hardware–software codesign for peer-to-peer energy market resolution." Sustainable Energy, Grids and Networks 35 (2023) : 101122.

L'optimisation d'un partitionnement GPU d'appels kernels pour le marché Pair à Pair Thomas, Beatrice, et al. "Optimization of a peer-to-peer electricity market resolution on GPU." 2022 IEEE International Conference on Electrical Sciences and Technologies in Maghreb (CISTEM). Vol. 4. IEEE, 2022

Analyse de l'algorithme PAC et de la parallélisation sur CPU avec OpenMP pour le marché pair à pair Thomas, Béatrice. "Propriétés algorithmiques des simulations de Smartgrid sur CPU-GPU." Journées couplées du GDR SEEDS et de la conférence JCGE-Jeunes Chercheurs en Génie Electrique. 2024

Optimisation sur CPU-GPU et analyse en grande dimension d'un DC maché endogène Thomas, B., et al. "GPU Optimisation of an Endogenous Peer-to-Peer Market with Product Differentiation." 2023 IEEE Belgrade PowerTech. IEEE, 2023.

L'adéquation algorithme architecture d'un power Flow Thomas B, El Ouardi A, Bouaziz Samir, Le Goff Latimier R, Ben Ahmed H. "Adéquation Algorithme Architecture pour le calcul d'un AC-Power Flow." Symposium de Génie Électrique. 2023

Implémentation sur GPU d'un optimal power flow Thomas, Beatrice, et al. " Acceleration of a decentralized radial AC-OPF on GPU." 2024 IEEE Power & Energy Society Innovative Smart Grid Technologies Conference (ISGT). IEEE, 2024

Ainsi une des premières contributions de cette thèse sera l'accélération via la méthode d'Adéquation Algorithme Architecture d'un marché pair à pair [110]. Un accent a été mis sur la parallélisation afin de tirer le maximum des caractéristiques matérielles du GPU [111]. Pour compléter cette étude, l'algorithme PAC et la parallélisation avec openMP ont été testés [114]. Le partitionnement d'un DC marché endogène avec une étude du comportement à grande échelle avec une solution pour contrer les problèmes de précision a été proposé dans [112]. L'Adéquation Algorithme Architecture a ensuite été appliqué au problème de Power-Flow [113]. L'implémentation GPU d'une résolution d'un OPF sur réseau radial a été faite dans [115]. Le tout a permis de réaliser l'**Adéquation Algorithme Architecture sur le marché Endogène**. L'étude d'un des algorithmes du marché endogène a été proposé à PowerTech et un article présentant l'application de la démarche sur le marché Endogène est en cours de rédaction.

Chaque code utilisé pour un article possède son propre répertoire sur gitlab (dont le line est disponible dans l'article). Le lien `https://github.com/beatho/EndogenousP2PMarket` permet d'accéder à un répertoire contenant l'ensemble des codes de cette thèse.

3.3 Organisation du manuscrit

Même si les différents problèmes de marché P2P, PF, OPF sont indépendants, l'objectif central est de réaliser un marché endogène en suivant la démarche Adéquation Algorithme Architecture (A3). Cette méthode ne dépendant pas du problème considéré, c'est le suivi de cette méthode qui rythmera ce manuscrit. Mais pour les lecteurs cherchant un problème en particulier, les parties se rapportant à chaque problème sont les suivantes :

- Marché : application des algorithmes 3.3, partitionnement 4.2.1, complexité 4.3.2, optimisation 4.4.4, résultats 5.2.1 et 5.3.1
- PF : application des algorithmes 3.2, partitionnement 4.2.2, complexité 4.3.3, optimisation 4.4.6
- OPF : application des algorithmes 3.4, partitionnement 4.2.3, complexité 4.3.4, remarques sur la convergence en Annexe 4
- Marché endogène : application des algorithmes 3.5, partitionnement 4.2.4, complexité 4.3.5, optimisation 4.4.5, résultats 5.2.2 et 5.2.3 et 5.3.2 et 5.3.3

L'organisation du manuscrit est donc la suivante. La partie II porte sur la présentation de la méthode Adéquation Algorithme Architecture, le choix des métriques, algorithmes, architectures et cas d'étude. La partie III présentera l'ap-

plication des algorithmes précédents sur les différents problèmes et leur validation fonctionnelle sur CPU en indiquant des premiers temps de calcul. La partie IV présentera le partitionnement sur GPU des différentes méthodes avec l'analyse par blocs fonctionnels de la complexité et des dépendances afin de réaliser la réécriture algorithmique des problèmes afin de les adapter au GPU. La partie V présentera les résultats pour les métriques considérées avec une étude de sensibilité. On pourra enfin conclure sur le travail réalisé et les perspectives pour des travaux futurs.

Chapitre 2

Méthodologie, métriques et critères d'évaluation

Sommaire

1 Introduction . . . 69
2 Méthodologie A3 . . . 70
2.1 La Roofline . . . 73
3 Choix des métriques . . . 74
3.1 Métriques algorithmiques . . . 75
3.2 Métriques architecturales . . . 77
4 Justification des choix des algorithmes . . . 79
4.1 Algorithmes de décentralisation . . . 79
4.2 Problème non convexe . . . 86
4.3 Optimisation quadratique . . . 90
4.4 Conclusion . . . 94
5 Choix des architectures et outils de programmation . . 95
5.1 Architectures de calcul parallèle multicœurs . . . 95
5.2 Architectures pour l'accélération de calcul . . . 98
5.3 Architecture sélectionnée . . . 100
5.4 Conclusion . . . 103
6 Choix des jeux de données . . . 103
6.1 Cas pour vérifier la convergence . . . 105
6.2 Cas pour comparaisons . . . 107
6.3 Cas pour évaluer la complexité . . . 110
6.4 Synthèse des cas d'études . . . 112
7 Conclusion . . . 113

1 Introduction

De l'état de l'art présenté précédemment, on peut tirer différentes informations. Tout d'abord, on peut remarquer qu'il y a deux types de but pour les articles :

soit l'objectif est de faire une démonstration de la validité fonctionnelle de l'algorithme proposé, soit l'objectif est de permettre l'utilisation d'un algorithme à grande échelle. Dans le premier cas, la démonstration est réalisée sur un cas de petite taille, le temps n'est donc pas une contrainte et celui-ci peut ne pas être indiqué. Souvent, ce genre de cas repose sur le fait que l'algorithme est décentralisé et que l'implémentation parallèle restera aussi rapide qu'à petite échelle. Dans le second cas, un travail d'accélération des algorithmes est proposé, soit en parallélisant sur CPU, GPU ou FPGA, soit en utilisant des clusters de calcul. Cependant, ces travaux d'accélération se concentrent sur la détermination de l'état du réseau, via principalement des PF [44]- [48] et des OPF [64]- [67].

Le problème de marché endogène que nous allons traiter dans le cadre de la thèse fait partie de la première catégorie. C'est-à-dire qu'il y a déjà eu des articles traitant de ce problème, mais qu'aucun travail d'accélération n'a eu lieu. Comme ce qui a été montré, les temps de calcul posent un verrou scientifique à la simulation à grande échelle de la gestion du réseau électrique. L'objectif de cette thèse est donc de suivre une démarche d'Adéquation Algorithme Architecture pour résoudre ce verrou dans le cas d'un marché endogène.

L'Adéquation Algorithme Architecture a été théorisée pendant les années 90, notamment dans le domaine du traitement du signal et est devenue un thème complet du groupe de recherche (GDR) ISIS (anciennement TdSI). Cette méthode consiste "*à étudier en même temps les aspects algorithmiques et architecturaux en prenant en compte leurs interactions, en vue d'effectuer une implantation optimisée de l'algorithme (minimisation des composants logiciels et matériels) tout en réduisant les temps de développement et les coûts finaux de l'application étudiée*" [83]. Ceci permet de viser deux objectifs distincts : l'amélioration des techniques de "prototypage rapide" (par exemple en prenant en compte en amont les contraintes énergétiques d'une application embarquée) et la réalisation d'une démarche de conception conjointe logicielle matérielle afin d'améliorer les performances finales pour un moindre coût.

Cette démarche est composée de plusieurs étapes à suivre. Ce chapitre présentera ces différentes étapes. Ensuite, la présentation et le choix préliminaire des métriques, algorithmes, architectures et cas d'études seront réalisés.

2 Méthodologie A3

La méthode Adéquation Algorithme Architecture comme indiqué par son nom est une méthode consistant en l'analyse simultanée de l'algorithme et de l'architecture afin de réaliser un prototypage rapide de l'application et obtenir les meilleures performances. L'analyse de l'algorithme comprend celle de la méthode de résolution du problème et celle de la manière de gérer la mémoire avec les variables

utilisées. L'architecture est définie par le type du matériel utilisé (par exemple, quel type de carte accélératrice est utilisé) et la technologie utilisée. En effet, selon que la technologie est plus ou moins récente, le matériel aura accès à plus ou moins de ressources physiques de calcul et de mémoire. Ces différences pourront impacter les temps de calcul.

Selon l'axe de recherche du GDR ISIS sur l'$\mathbf{A}^3$ cette méthode repose sur 4 axes principaux :

- la modélisation générique de l'algorithme et de l'architecture,
- la caractérisation de l'algorithme et de l'architecture pour l'exploration de l'espace de conception,
- l'évaluation de performances et des optimisations de l'ensemble des implémentations considérées,
- la génération de code et le test sur les architectures matérielles retenues.

Un schéma représentant l'ensemble de la démarche peut être vu Fig. 2.1. La première étape (en bleue) est de choisir l'application qui est ciblée et les contraintes qu'elle devra respecter. On identifiera aussi les verrous devant être résolus par la démarche. Comme présenté dans le chapitre précédent, notre application est un marché endogène. Les contraintes associées sont que l'expression du problème doit être décentralisée avec des contraintes de respect de la vie privée des agents (c'est-à-dire ne pas utiliser trop d'informations). De plus, l'application doit être réalisée pour de la simulation, notamment dans le cadre de la recherche. Ainsi, les performances doivent être atteintes sur des ordinateurs classiques, c'est-à-dire utilisant des processeurs modernes, mais sans être des super-ordinateurs. L'optimisation visée est principalement la réduction du temps de calcul pour être le plus rapide possible.

La seconde étape se base sur l'état de l'art pour identifier les métriques pertinentes pour déterminer les critères de sélection des algorithmes permettant de réaliser l'application cible en respectant les contraintes. Ces métriques nous permettront aussi de réaliser une évaluation et des comparaisons entre les différentes applications réalisées. Une sélection des architectures utilisées et des jeux de données qui seront utilisés pour évaluer les performances finales est aussi nécessaire. L'ensemble de ces points sera vu dans ce chapitre.

La troisième étape consiste en l'étude de l'utilisation des algorithmes sur notre application. Une fois la validation fonctionnelle réalisée, l'algorithme sera séparé en différents blocs fonctionnels, donnant des schémas sous la forme Fig. 2.2. L'étude de ces blocs en matière de charge de travail, de complexité, de dépendance des données et de temps de calcul donnera des informations sur les goulots d'étranglement de l'algorithme et sur la parallélisation possible des étapes. Cette dépendance pourra être représentée sous la forme de graphe acyclique, Fig. 2.3.

Ces informations nous permettront de déterminer les voies d'optimisation de

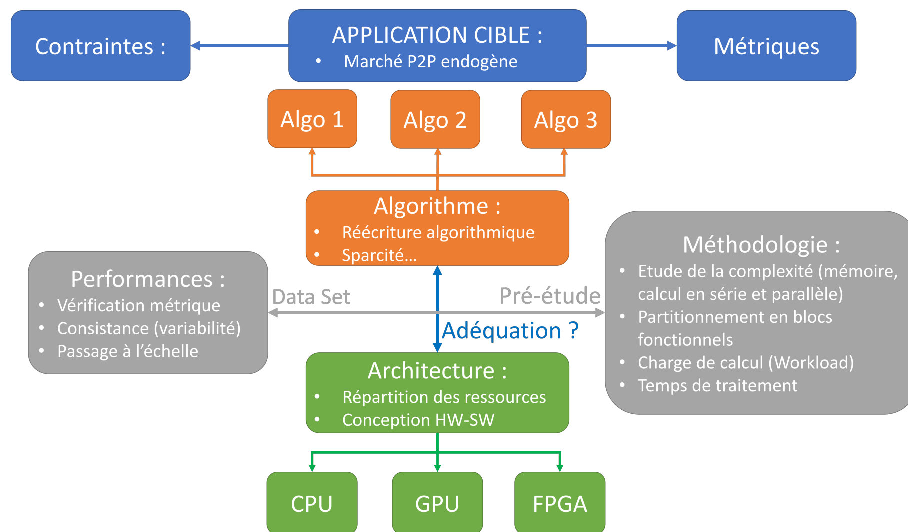


FIGURE 2.1 – Méthode Adéquation Algorithme appliqué à notre problème

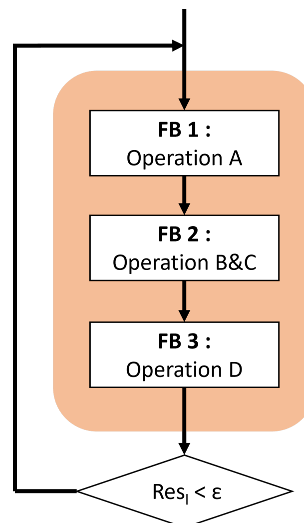


FIGURE 2.2 – Séparation d'une application en blocs fonctionnels

l'application. Ainsi, on pourra optimiser l'ensemble en effectuant une ré-écriture algorithmique, et en optimisant l'implémentation pour prendre en compte les caractéristiques physiques d'un partitionnement sur CPU-GPU. Pour chaque appel kernel ayant plusieurs voies d'optimisations contradictoires, toutes les versions seront implémentées et la sélection sera réalisée par un benchmark permettant de positionner les différentes versions sur la Roofline. Plusieurs implémentations d'un même algorithme pourront être réalisées pour permettre la comparaison des comportements de l'ensemble de l'algorithme et non pas des appels kernels individuels.

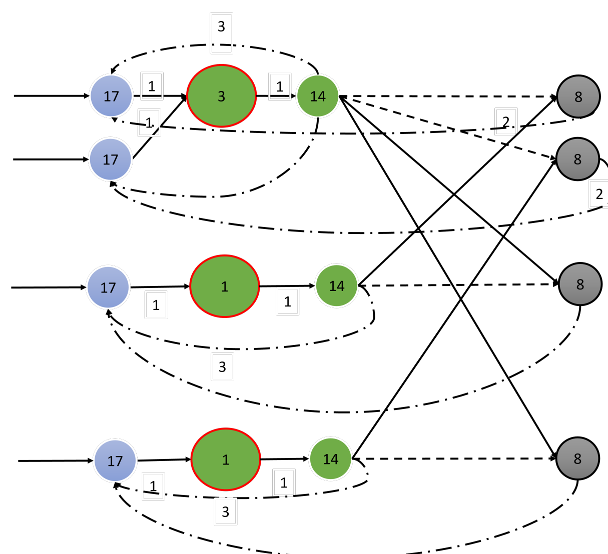


FIGURE 2.3 – Représentation d'une application sous forme de graphe acyclique

Une fois l'ensemble des étapes de l'optimisation réalisé, une étude des résultats obtenus sur les différents cas tests sélectionnés permettra de réaliser des comparaisons avec l'état de l'art et entre les différents couples algorithme et architecture. La sensibilité de ces résultats pourra être évaluée en fonction des paramètres des cas d'études, de paramètre de l'algorithme ou de l'architecture pour un même type de matériel.

2.1 La Roofline

Pour analyser les performances d'un algorithme donné sur une certaine architecture, le modèle de la Roofline peut être utilisé. Pour cela, ce modèle s'intéresse aux limitations matérielles et à leurs interactions avec l'application. Dans la version la plus simplifiée, on définit une architecture par deux paramètres. Le premier est le débit pour l'accès à la mémoire (β) en Mbytes/s. Le deuxième est le pic de performance de l'architecture, soit le nombre maximum d'opérations par seconde pouvant être effectuées, noté π.

Deux éléments sont nécessaires pour définir une application : le travail W, qui correspond au nombre d'opérations que l'algorithme doit exécuter, et le trafic mémoire Q, qui correspond à la quantité de mémoire qui doit être déplacée. On définit donc l'intensité algorithmique $I = \frac{W}{Q}$ comme le nombre d'opérations par octet de mémoire déplacé.

Par conséquent, pour illustrer les attributs des applications et du matériel, on représente la puissance de calcul P en $GFlop/s$ (en milliard d'opérations par

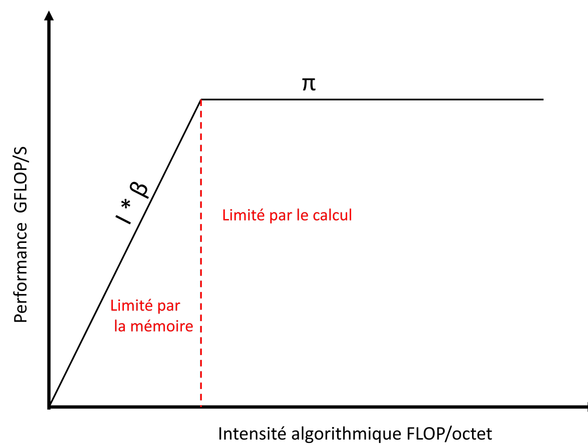


FIGURE 2.4 – Exemple d'une Roofline

seconde) en fonction de l'intensité algorithmique. Pour un matériel donné, on peut approximer la performance par $P = min(\beta * I, \pi)$. Ceci représente la performance maximale atteignable par les applications sur ce matériel en fonction de leurs intensités. En posant $I_{lim} = \frac{\pi}{\beta}$, on peut identifier deux limites pour une application donnée. Si $I \leq I_{lim}$, cela signifie que l'application sera limitée par la mémoire (*memory bound*). C'est-à-dire que les unités de calculs devront attendre les accès mémoires pour calculer. À l'opposé, si $I \geq I_{lim}$ l'application sera limitée par les calculs (*compute bound*). Le graphe en résultant est sur la Fig. 2.4. Ainsi, pour optimiser le temps de calcul dans le premier cas, il vaut mieux faire plus de calculs (recalculer une valeur que l'on stocke autre part, par exemple) si cela permet d'économiser des transferts mémoires.

3 Choix des métriques

L'objectif de cette partie est de présenter les différentes métriques qui seront utilisées pour caractériser tous les triptyques algorithme, architecture et cas d'étude. On commencera par présenter des métriques utilisées dans l'état de l'art. Ensuite, on justifiera les métriques conservées dans la suite de cette thèse. Afin de différencier plus facilement les leviers d'action agissant sur les métriques, celles-ci seront séparées en deux catégories. On les appellera métriques algorithmiques et métriques architecturales.

Cependant, il est important de préciser que, même si une métrique est présentée comme algorithmique, cela ne signifie pas qu'elle dépendra uniquement de l'algorithme. Dans tous les cas, c'est bien l'ensemble du couple algorithme-architecture sur un cas donné qui va être évalué. On étudiera aussi l'évolution de ces métriques

avec les dimensions du problème (on parlera du **passage à l'échelle** de l'implémentation et de l'algorithme).

3.1 Métriques algorithmiques

Dans cette partie seront évoquées les métriques dites algorithmiques, *i.e.* dont la valeur dépend essentiellement de l'algorithme choisi. Leurs valeurs peuvent dépendre du type d'architecture choisie, puisque les implémentations d'un algorithme sur deux architectures différentes seront nécessairement différentes. Cependant, sauf effet de bord, ces métriques ne devraient pas dépendre du matériel choisi. C'est-à-dire que les valeurs des métriques devraient être semblables si l'on réalise les tests sur 2 CPU différents, par exemple.

Pour rappel, l'objectif de cette thèse est de lever le verrou scientifique du temps de calcul. Afin de réaliser l'optimisation, la méthode de gestion et de simulation est fixée. Ainsi, on n'utilisera pas toutes les métriques qui servent à caractériser les effets des différentes méthodes de gestion (équité, coût, pertes finales, énergie échangée, etc. [88]). En effet, quels que soient les algorithmes implémentés, le problème résolu étant le même, ces métriques seront identiques pour toutes les implémentations (hors convergence vers un minimum local sur un problème non convexe). La valeur de la fonction coût sera utilisée uniquement pour vérifier que les différents algorithmes convergent bien vers le même optimum.

3.1.1 Convergence

Pour la première métrique choisie, on se concentrera sur la robustesse et la fiabilité de l'implémentation : est-ce que l'algorithme converge pour tous les cas [41]. On réalisera des tests sur des cas qui sont mal posés [42] ou des cas plus classiques. Dans le cas où les algorithmes ne convergent pas, on s'intéressera au fait que la solution diverge ou non. Pour la suite, pour une implémentation donnée et un nombre d'itérations maximal donné arbitrairement, on considérera trois états pour l'implémentation. Soit l'algorithme a un résidu suffisamment faible et s'arrête avant la fin. Dans ce cas-là, on considère qu'il a **convergé**. On considérera qu'il a **divergé** lorsque le résidu ne diminue pas et que le résultat s'éloigne de la solution. Si l'implémentation s'arrête parce que le nombre maximum d'itérations est atteint et que les résidus ne sont pas encore sous le seuil, mais que l'ajout d'itérations n'éloigne pas le résultat de l'optimum, alors on dira que l'algorithme **n'a pas convergé**. On considéra aussi que l'algorithme n'a pas convergé lorsque le résultat retourné n'a pas la valeur de l'optimum attendu (par exemple à cause d'un minimum local). À partir de ces trois états, on pourra déterminer le taux de convergence sur des cas aléatoires et les états sur des cas tests fixes.

Cette métrique est utilisée, car, même si l'objectif principal est l'accélération des calculs, cela n'a de sens que si le résultat obtenu est utile. En effet, obtenir rapidement un résultat faux ou pas assez précis par rapport à l'application que l'on vise est inutile. Elle est principalement intéressante lorsque le problème n'est pas convexe et que la convergence n'est donc pas garantie. Elle est donc plus particulièrement utilisée lors de l'étude des **PF**, des **OPF** [8] ou des marchés endogènes [21] en AC. Dans notre cas, on l'utilisera aussi sur des problèmes convexes lorsqu'on limitera le nombre d'itérations maximal autorisé. Cela permettra de repérer une sensibilité aux cas d'étude ou à certains paramètres sur la vitesse de convergence.

Enfin, cette métrique aura une importance lorsque les différentes implémentations seront limitées par la précision de leurs calculs. En effet, selon l'algorithme choisi, la précision maximale ou celle qui est atteignable en un nombre d'itérations donné sera différente à cause d'instabilité numérique ou de propagation d'erreurs numériques.

3.1.2 Précision

La précision est une métrique intrinsèquement liée à la convergence. Théoriquement, dans le cas convexe, les différents algorithmes qui seront présentés convergent tous vers l'optimum unique. Donc, pour atteindre la précision demandée, il suffit de permettre une infinité d'itérations. En pratique, la précision atteignable va dépendre des erreurs numériques dues à la précision des nombres manipulés (flottant, double, entier...) et du cas d'étude choisi [41]- [42]. Ainsi, il sera possible de déterminer, selon les algorithmes, si ceux-ci convergent avec des nombres à virgule flottante sur quatre octets, ou bien s'ils ont besoin de nombres en double précision (de même, mais sur huit octets) pour une précision demandée ou dans le cas général. Enfin, les performances en matière de nombre d'itérations ou de temps de calcul peuvent être très variables selon la précision demandée.

Cette métrique sera importante puisqu'elle déterminera les architectures et le matériel utilisable selon les algorithmes. Par exemple, un algorithme ne convergeant qu'en double précision pourra difficilement utiliser une carte accélératrice utilisant des unités de calcul en simple précision.

3.1.3 Nombre d'itérations

Cette métrique est très proche de la métrique précédente. En effet, elle représente le nombre d'itérations nécessaires pour converger vers la solution [88], [19], [41], [42]. Plus la précision demandée est importante, plus ce nombre sera important. L'idée étant que plus l'ordre de convergence de l'algorithme est élevé, plus le nombre d'itérations sera faible.

De plus, même si l'on peut déterminer un ordre de convergence, celle-ci peut

être irrégulière. Certains algorithmes peuvent être plus rapides pour se rapprocher de la solution lorsqu'ils sont loin et d'autres ne sont rapides qu'une fois qu'ils sont assez proches. Ainsi, on regardera cette valeur pour différents cas d'étude et pour différentes précisions demandées.

Enfin, dans le cadre d'une simulation d'un problème décentralisé, le nombre de messages échangés avant convergence sera aussi calculé [19]. En effet, même si cette métrique a peu d'impact sur une application sur un CPU ou un GPU lorsque les données sont déjà sur place, elle sera cruciale au moment de l'implémentation réelle, ou si la simulation a lieu sur un cluster de calcul.

3.1.4 Complexités

Dans le cadre de cette thèse, plusieurs types de complexité seront étudiés. Ces complexités pourront être en mémoire (l'espace mémoire nécessaire pour stocker les différentes variables) et en calcul (nombre d'opérations nécessaires afin de finir l'algorithme) [41]. Pour cette dernière, on s'intéressera aussi à la complexité en série et celle en parallèle. Le théorème de Brent [89] fournit une méthode pour mesurer la complexité théorique d'une application fonctionnant sur plusieurs processeurs (ou unités de calcul). Soit o le nombre total de calculs à réaliser, t les opérations ou étapes non parallélisables et p le nombre de processeurs, le théorème de Brent définit la complexité parallèle comme :

$$C_{para} = O(\frac{o}{p} + t) \tag{2.1}$$

Un algorithme sera considéré comme parallélisable si sa complexité théorique en parallèle est plus faible que celle en série.

L'étude des différents comportements asymptotiques des complexités en fonction des dimensions du problème nous permettra d'évaluer le passage à l'échelle théorique des différents couples algorithmes-architecture.

Ces métriques seront à la fois déterminées analytiquement et mesurées en appliquant les algorithmes sur des cas d'étude aléatoires de différentes tailles.

3.2 Métriques architecturales

Dans cette partie, on présentera les métriques qui vont quant à elles fortement dépendre de l'architecture et de la qualité du matériel choisi. L'implémentation et l'algorithme pourront aussi influencer ces métriques dans une moindre mesure. Dans le cadre de cette thèse, une même machine a été utilisée pour l'ensemble des tests pour permettre une comparaison des résultats. Une des métriques architecturales qui aurait pu être pertinente est la consommation électrique nécessaire pour faire une simulation. Cependant, la simulation n'étant ni sur un matériel embarqué

ni dans un cluster de calcul, la consommation électrique (et donc l'échauffement) n'est pas contraignante. Ainsi, cette métrique n'a pas été mesurée.

3.2.1 Ressources Mémoire

L'utilisation de mémoire est un aspect important, puisqu'elle conditionne le matériel utilisable et peut limiter les performances. Dans notre cas, l'utilisation de mémoire concerne à la fois l'espace de mémoire nécessaire, le nombre d'allocations et les transferts entre différents matériels. Dans cette thèse, on considèrera que le CPU et la carte accélératrice possèdent une grande quantité de stockage qui sera suffisante pour tous les cas utilisés. Ainsi, pour cette métrique, on utilisera la complexité en mémoire pour étudier le passage à l'échelle. En effet, la valeur exacte de mémoire utilisée n'est pas utile en soi.

On se concentrera dans le cas de l'utilisation d'une carte pour d'accélération des calculs à la quantité de mémoire transférée entre le CPU et cette carte. On s'intéressera plus précisément à l'évolution de cette valeur lors de l'augmentation de la taille des problèmes. En effet, quelle que soit cette carte, les transferts mémoires depuis ou vers cette carte peuvent représenter un coût temporel non négligeable selon le cas d'étude.

3.2.2 Temps de calcul

La métrique la plus contraignante, et donc celle sur laquelle se concentrera cette thèse, est le temps de calcul. Cela s'explique par le fait que l'objectif de ce travail est de permettre la simulation à grande dimension et que le verrou associé est justement le temps de calcul. Dans notre cas, le temps de calcul commencera au moment où le cas d'étude est fourni à l'algorithme de résolution et s'achèvera à la récupération des résultats. On ne considérera pas le temps de créer le cas d'étude, puisqu'il est identique pour tous les algorithmes et implémentations. Le **temps de calcul** (aussi appelé **temps total**) [8], [88] est essentiel, puisque c'est ce qui va être limitant pour des tests de grande envergure. Cependant, on s'intéressera aussi à d'autres temps. Par exemple, le **temps par itération** pourra indiquer si le temps pourra être réduit plutôt en accélérant les calculs ou s'il vaut mieux réduire le nombre d'itérations [41], [42]. On étudiera aussi le **temps par bloc** de l'algorithme afin d'indiquer où le temps peut être gagné.

Enfin, on pourra étudier l'accélération ou le **speed-up** entre deux implémentations. La première sera définie par le rapport entre les temps de calcul entre les implémentations. Ce qu'on nommera **speed-up** est une accélération relative définie par :

$$S = \frac{t_{reference} - t_{nouvelle}}{t_{reference}} \tag{2.2}$$

4 Justification des choix des algorithmes

Dans cette partie, on présentera les différents algorithmes qui sont utilisés dans la littérature pour résoudre les problèmes présentés dans le chapitre précédent. Pour mémoire, les problèmes de marché endogène, d'Optimal Power Flow (OPF) et de marché peuvent tous s'écrire sous la forme suivante :

$$\begin{aligned} &\min \ f(x) \\ &\text{t.q. } h(x) = 0 \\ &\quad \underline{g} \leq g(x) \leq \overline{g} \end{aligned} \tag{2.3}$$

Le problème de Power Flow (PF), quant à lui, consiste en la détermination d'inconnus du réseau. Le problème peut donc s'écrire sous la forme d'un système d'équations potentiellement non linéaires :

$$F(x) = 0 \tag{2.4}$$

Selon les hypothèses réalisées sur le réseau ou les agents, ce problème peut être convexe ou non avec des contraintes linaires ou non. Cette section présentera donc les différents algorithmes qu'il vaut mieux utiliser selon les hypothèses, en se basant sur ce qui a déjà été fait dans la littérature.

Comme ce qui a déjà été montré dans le premier chapitre, avec l'augmentation des dimensions du problème, un risque de surcharge de l'opérateur central apparait. Ainsi, la première partie de cette section présentera comment il est possible de séparer le problème initial en plusieurs sous-problèmes plus petits et potentiellement plus simples à résoudre.

Dans un second temps, on présentera des méthodes permettant de résoudre les systèmes et optimisations même lorsque ceux-ci ne sont pas convexes. Enfin, on présentera des méthodes de résolution sous des hypothèses plus fortes, comme le fait d'avoir un problème quadratique ou des contraintes linéaires.

4.1 Algorithmes de décentralisation

4.1.1 Méthode des multiplieurs à direction alternée

Le principe de la Méthode des Multiplieurs à Direction Alternée (ADMM) [90] pour décentraliser est le suivant : on suppose que la variable x et la fonction coût $f(x)$ est séparable en plusieurs parties que l'on notera x et z, et $f(x)$ et $v(z)$. On écrira le problème d'optimisation ainsi :

$$\begin{aligned} &\min \ f(x) + v(z) \\ &\text{t.q. } Ax + Bz = c \end{aligned} \tag{2.5}$$

Le Lagrangien augmenté associé vaut :

$$L_\rho(x, z, y) = f(x) + v(z) + y^T(Ax + Bz - c) + \frac{\rho}{2}||Ax + Bz - c||_2^2 \tag{2.6}$$

L'algorithme d'ADMM consiste dans les itérations suivantes :

$$x^{k+1} = \underset{x}{\operatorname{argmin}}\, L_\rho(x, z^k, y^k) \tag{2.7a}$$

$$z^{k+1} = \underset{z}{\operatorname{argmin}}\, L_\rho(x^{k+1}, z, y^k) \tag{2.7b}$$

$$y^{k+1} = y^j + \rho \cdot (Ax + Bz - c) \tag{2.7c}$$

L'algorithme ainsi présenté relaxe des contraintes linaires représentant le lien entre les variables séparées. Un cas particulier de l'application de cet algorithme est de faire en sorte que z soit une copie des variables x. Dans ce cas là $A = I_{N \cdot N}$, $B = -I_{N \cdot N}$ et $c = 0_N$ (avec $I_{N \cdot N}$ la matrice identité de taille $N \cdot N$ et N le nombre de variables).

Dans le cas où la minimisation est sous contrainte, celles-ci sont reprises dans les sous-problèmes. Selon la forme des contraintes et sur quelles variables elles s'appliquent, celles-ci devront être reprises dans la minimisation de x ou de z.

On peut remarquer que la résolution des minimisations précédentes dépend de la forme de la fonction coût et des contraintes restantes. Les sections suivantes montreront comment résoudre ces minimisations dans le cas général (Section. 4.2) ou dans le cas quadratique (Section. 4.3).

Cet algorithme devient intéressant lorsqu'il permet de séparer un problème complexe sous contraintes en deux sous problèmes avec moins de contraintes. Faire ainsi permet d'utiliser des algorithmes ne pouvant être utilisés que sous certaines hypothèses sur les contraintes (linéaire, uniquement égalité, inégalité ou borne, etc...). Cet algorithme est notamment utilisé dans l'article [60]. Pour différencier avec les autres formulations de l'ADMM, celle-ci sera appelée ADMM sous sa forme de base.

4.1.2 ADMM sous forme de consensus

L'algorithme ADMM [90] a la particularité de pouvoir se mettre sous plusieurs formes et ainsi grandement se simplifier selon les problèmes traités. Dans cette partie, on supposera que la fonction coût peut se séparer en N termes et s'écrire :

$$f(x) = \sum_i f_i(x) \tag{2.8}$$

Ensuite, on considérera que chaque partie de la fonction coût $f_i(x)$ sera, en réalité, effectuée sur une copie de x. On note chaque copie locale de la variable x_i. La

contrainte que chaque copie locale soit égale à la même valeur, notée z, est ajoutée. Le problème global peut donc s'écrire sous la forme suivante :

$$\begin{aligned} &\min \sum_i f_i(x_i) \\ &\text{t.q. } x_i = z \ \forall i \end{aligned} \tag{2.9}$$

Le Lagragien augmenté s'écrit :

$$L_\rho(x,z,y) = \sum_i f_i(x_i) + y^T(x_i - z) + \frac{\rho}{2}||x_i - z||_2^2 \tag{2.10}$$

On en déduit les étapes suivantes pour la résolution de l'optimisation :

$$x_i^{k+1} = \underset{x_i}{\text{argmin}}\, f_i(x_i) + y_i^{kT}(x_i - z^k) + \frac{\rho}{2}||x_i - z^k||_2^2 \tag{2.11a}$$

$$z^{k+1} = \frac{1}{N}\sum_i (x_i^{k+1} + \frac{1}{\rho} y_i^k) \tag{2.11b}$$

$$y_i^{k+1} = y_i^k + \rho \cdot (x_i^{k+1} - z^{k+1}) \tag{2.11c}$$

En notant $\overline{y}$ la moyenne d'un vecteur y, on peut remarquer les faits suivants :

$$\begin{aligned} z^{k+1} &= \overline{x^{k+1}} + \frac{1}{\rho}\overline{y^k} && = \overline{x} \\ \overline{y}^{k+1} &= \overline{y}^k + \rho(\overline{x}^{k+1} - z^{k+1}) && = 0 \end{aligned} \tag{2.12}$$

Ces relations permettent de simplifier la résolution et ainsi on obtient les mises à jour suivantes :

$$x_i^{k+1} = \underset{x_i}{\text{argmin}}\, f_i(x_i) + y_i^{kT}(x_i - \overline{x}^k) + \frac{\rho}{2}||x_i - \overline{x}^k||_2^2 \tag{2.13a}$$

$$z^{k+1} = \overline{x}^{k+1} \tag{2.13b}$$

$$y_i^{k+1} = y_i^k + \rho \cdot (x_i^{k+1} - \overline{x}^{k+1}) \tag{2.13c}$$

Pour toute la suite de la thèse, on appellera la résolution de l'optimisation des x_i le "problème local". En effet, l'implémentation réelle de ce genre d'algorithme permet de faire en sorte que chaque agent intervenant dans le problème résolve son propre problème localement avant d'en communiquer le résultat. Et les résidus peuvent être calculés ainsi :

$$||r^k||_2^2 = \sum_i ||x_i^{k+1} - \overline{x}^k||_2^2 \tag{2.14a}$$

$$||s^k||_2^2 = N\rho^2||\overline{x}^k - \overline{x}^{k-1}||_2^2 \tag{2.14b}$$

Tout comme l'algorithme précédemment, si le problème initial possède des contraintes, celles-ci seront conservées dans la minimisation de x. Pour que cela soit possible, il faut que celles-ci soient séparables. Ainsi, chaque problème "local" sur les x_i devra respecter sa partie des contraintes. Une autre possibilité sera de relaxer les contraintes dans la fonction coût grâce au Lagrangien. Si cette relaxation est séparable en N termes, comme la fonction coût initial, l'algorithme peut être utilisé.

L'intérêt de cette méthode c'est qu'elle peut être adaptée dans le cas où chaque x_i ne contient qu'une partie des variables stockées dans z. En écrivant $\tilde{z}_i$ pour les variables qui sont copiées dans x_i. Dans ce cas-là, le problème devient (après simplification) :

$$x_i^{k+1} = \underset{x_i}{\operatorname{argmin}}\, f_i(x_i) + y_i^{kT} x_i + \frac{\rho}{2} ||x_i - \tilde{z}_i{}^k||_2^2 \tag{2.15a}$$

$$y_i^{k+1} = y_i^k + \rho \cdot (x_i^{k+1} - \tilde{z}_i{}^{k+1}) \tag{2.15b}$$

avec z_g qui vaut la moyenne de toutes ses copies dans les x_i pour toute variable globale g. Plus de détails sur la forme de l'algorithme et la démonstration sont disponibles dans [90]. Cet algorithme est utilisé pour distribuer la résolution des marchés pair à pair ou de communautés [20], [25], [26].

4.1.3 ADMM sous forme de partage

Dans le cas d'une ADMM sous la forme de partage, [90], on suppose que le problème d'optimisation peut se mettre sous la forme :

$$\underset{x}{\operatorname{argmin}} \sum_i f_i(x_i) + v(\sum_i x_i) \tag{2.16}$$

Ce genre d'expression est assez courante lorsque l'on travaille avec les puissances dans le réseau électrique. En effet, la puissance totale d'un agent est la somme des puissances (*trades*) qu'il échange avec ses pairs et la puissance à un bus est la somme des puissances des agents sur ce bus. Ce type de problème permet de représenter les cas où chaque agent doit minimiser sa propre fonction coût ($f_i(x_i)$) et un terme dépendant de l'ensemble des agents (par exemple dû à l'utilisation de ressources commune comme le réseau électrique).

On va donc réécrire le problème d'optimisation sous la forme suivante :

$$\begin{aligned} \min \ & \sum_i f_i(x_i) + v(\sum_i z_i) \\ \text{t.q. } & x_i = z_i \qquad \forall i \end{aligned} \tag{2.17}$$

L'utilisation de l'algorithme ADMM sous sa forme réduite ($u = \frac{y}{\rho}$) nous donne les trois étapes suivantes pour la résolution du problème d'optimisation :

$$x_i^{k+1} = \underset{x_i}{\text{argmin}}\, f_i(x_i) + \frac{\rho}{2}||x_i - z_i + u_i^k||_2^2 \quad (2.18a)$$

$$z^{k+1} = \underset{z}{\text{argmin}}\, v\big(\sum_i z_i\big) + \frac{\rho}{2}\sum_i ||z_i - u_i^k - x_i^{k+1}||_2^2 \quad (2.18b)$$

$$u_i^{k+1} = u_i^k + x_i^{k+1} - \overline{x}^{k+1} \quad (2.18c)$$

On remarque que la première et la troisième mise à jour peuvent être réalisées en parallèle sur les différents problèmes locaux i. En ajoutant la moyenne des z, notée $\overline{z}$, les différentes étapes peuvent être simplifiées :

$$x_i^{k+1} = \underset{x_i}{\text{argmin}}\, f_i(x_i) + \frac{\rho}{2}||x_i - x_i^k + \overline{x}^k - \overline{z}^k + u_i^k||_2^2 \quad (2.19a)$$

$$z^{k+1} = \underset{z}{\text{argmin}}\, v(N\overline{z}) + \frac{N\rho}{2}||\overline{z} - u^k - \overline{x}^{k+1}||_2^2 \quad (2.19b)$$

$$u^{k+1} = u^k + \overline{x}^{k+1} - \overline{z}^{k+1} \quad (2.19c)$$

Ce type d'algorithme ressemble à une résolution distribuée où l'agent central récupère les valeurs de x_i afin de calculer la valeur de z et u_i et le transmettre à tous les agents i. Cependant ceci peut aussi être fait de manière totalement décentralisée avec l'ensemble des agents recevant toutes les variables pour que tous calculent z. Cette résolution peut aussi être réalisée de manière centralisée en parallélisant.

4.1.4 Proximal Atomic Coordination

L'algorithme *Proximal Atomic Coordination*, connu sous l'acronyme **PAC** [27], [28], est une méthode alternative de décentralisation qui serait plus efficace sur le plan algorithmique que l'ADMM et qui offrirait une meilleure protection de la vie privée et des données personnelles des participants au système. En effet, plutôt que de transmettre directement les optimums et variables duales de chaque itération, ce sont des valeurs modifiées qui sont transmises. Ces modifications doivent permettre d'éviter que des agents malveillants sur le réseau puissent déterminer des informations sur les autres intervenants grâce à la réception des communications et influencer la convergence pour en tirer un profil.

Pour appliquer l'algorithme, on suppose que le problème peut se mettre sous la forme suivante avec entre parenthèses les variables duales associées aux contraintes et H, G des matrices :

$$\begin{aligned} x = &\text{argmin}\, f(x) \\ \text{t.q.} \quad & Gx = b \qquad (\mu) \\ & Hx \leq d \qquad (y) \end{aligned} \quad (2.20)$$

Pour pouvoir relaxer la contrainte d'inégalité, on introduit la variable *slack* $u \geq 0$ tel que la contrainte devient $Hx - d + u = 0$.

Tout comme ceux vus précédemment, l'objectif de l'algorithme est de séparer l'ensemble de l'optimisation (fonction coût et contrainte) en différents sous-problèmes indépendants. Lorsqu'une variable est nécessaire dans plusieurs sous-problèmes, un des sous-problèmes "possède" vraiment cette variable et les autres ont une "copie" de cette variable. Supposons que $x \in \mathrm{R}^N$ soient les variables d'optimisations, et que l'on souhaite séparer le problème d'optimisation en M groupes. On a donc :

- L_m l'ensemble des variables x que le groupe m possède
- O_m l'ensemble des copies de x dont le groupe m a besoin
- T_m l'ensemble des variables possédées et copiées d'optimisation du groupe m, $T_m = L_m \bigcup O_m$

Pour la suite, on notera B la matrice d'incidence du graphe orienté reliant les variables avec leur copie, et T le vecteur composé de l'ensemble des variables possédées et copiées. On notera aussi B_m les lignes de B correspondant aux copies du groupe m (et on notera B^m les colonnes correspondant aux variables du groupe m), on rajoute donc la contrainte suivante pour chaque groupe m :

$$B_m \cdot T = 0 \quad (\nu_m) \qquad\qquad \forall m \tag{2.21}$$

Chaque ligne de la matrice B_m correspond à la contrainte pour la copie. Ainsi, chaque ligne i a deux coefficients non nuls tels que $b_{ij} = 1$ et $b_{ik} = -1$ si la variable T_i et T_k sont respectivement une valeur copiée ou possédée. En notant ν la variable duale associée à cette dernière contrainte, le Lagrangien de l'ensemble du problème s'écrit :

$$\begin{aligned} L(T, \mu, y, \nu) &= \sum_{m \in M} f_m(t_m) + \nu_m^T B_m T \\ &+ \mu_k^T (G_m t_m - b_m) + y_m^T (H_m t_m + u_m - d_m) \\ &= \sum_{m \in M} f_m(t_m) + \nu^T B^m t_m \\ &+ \mu_k^T (G_m t_m - b_m) + y_m^T (H_m t_m + u_m - d_m) \\ &= \sum_{m \in M} L_m(t_m, \mu_m, y_m, \nu) \end{aligned} \tag{2.22}$$

Le Lagrangien peut être séparé sur les différents groupes pour décentraliser le calcul. Pour accélérer la convergence, on peut augmenter le Lagrangien et y ajouter les termes suivants :

$$\begin{aligned} L_m^{augm} &= \frac{\rho_m \gamma_m}{2} (||G_m t_m - b_m||_2^2 + ||H_m t_m + u_m - d_m||_2^2 + ||B_m t_m||_2^2) \\ &+ \frac{1}{2\rho_m} ||t_m - t_m^k||_2^2 \end{aligned} \tag{2.23}$$

Les différentes étapes de l'optimisation sont donc les suivantes :

$$T_m^{k+1} = \underset{T_m}{\operatorname{argmin}}\, L_m(T_m, \hat{\mu}^k, \hat{y}^k, \nu^k) + L_m^{augm} \tag{2.24a}$$
$$\hat{T}_m^{k+1} = T_m^{k+1} + \alpha_m^{k+1}(T_m^{k+1} - T_m^k) \tag{2.24b}$$
$$\mu_m^{k+1} = \hat{\mu}_m^k + \rho_m \gamma_m (G_m \hat{T}_m^{k+1} - b_m) \tag{2.24c}$$
$$\hat{\mu}_m^{k+1} = \mu_m^{k+1} + \phi_m^{k+1}(\mu_m^{k+1} - \mu_m^k) \tag{2.24d}$$
$$y_m^{k+1} = \hat{y}_m^k + \rho_m \gamma_m (H_m \hat{T}_m^{k+1} + u_m - d_m) \tag{2.24e}$$
$$\hat{y}_m^{k+1} = y_m^{k+1} + \beta_m^{k+1}(y_m^{k+1} - y_m^k) \tag{2.24f}$$
$$u_m^{k+1} = max(0, H_m \hat{t}_m^{k+1} - d_m - y_m^k) \tag{2.24g}$$
$$\nu_m^{k+1} = \hat{\nu}_m^k + \rho_m \gamma_m B_m \hat{T}_m^{k+1} \tag{2.24h}$$
$$\hat{\nu}_m^{k+1} = \nu_m^{k+1} + \theta_m^{k+1}(\nu_m^{k+1} - \nu_m^k) \tag{2.24i}$$

Les coefficients α, ϕ, β et θ sont des suites dépendant de l'itération et du groupe m. Ils permettent de brouiller les communications pour protéger la confidentialité du dit groupe.

Si le problème que l'on cherche à résoudre possède des contraintes non linéaires, celles-ci devront être respectées lors de la minimisation des t_m à chaque itération. Selon la forme des contraintes restantes, différents algorithmes pourront être utilisés.

4.1.5 Performance des algorithmes et autres possibilités

On peut remarquer que tous ces algorithmes introduisent une résolution itérative. On se demandera dans un premier temps la valeur de la complexité de chaque itération, pour ensuite se concentrer la vitesse de convergence théorique des algorithmes. Dans tous les cas, la complexité sera calculée en fonction du nombre de variables.

Tout d'abord, les trois méthodes de l'ADMM se font en 4 étapes qui sont les calculs de x, z, y et des résidus. La complexité des calculs de x et z dépendra du problème et de la méthode utilisée pour le résoudre. La complexité pour calculer y est quadratique (multiplication matrice-vecteur) dans le cas général et linéaire dans le cas du consensus et du partage.

Tout comme pour l'ADMM, la complexité de l'algorithme de décentralisation PAC dépend de la complexité de résoudre la minimisation du problème local t_m. L'ensemble des autres étapes est des opérations linéaires sur des vecteurs ou des multiplications matrices-vecteurs. Ainsi, la complexité restante est, dans le cas général, au plus quadratique.

Le taux de convergence de l'ADMM est au plus linéaire, mais peut être accéléré jusqu'à une convergence quadratique [88] en modifiant l'algorithme. Cependant,

sans cette modification l'ADMM a une convergence sous-linéaire dans le cas général. L'algorithme de PAC a une vitesse de convergence linéaire si le facteur de pénalité ρ vérifie une certaine relation [27].

D'autres algorithmes de décentralisation existent. On pourra citer Consensus-Innovation (CI) [20] ou ALADIN [91]. Cependant, le premier nécessite le réglage d'une suite de paramètres rendant trop dépendante la vitesse de convergence à ce réglage. Tandis que le deuxième est une amélioration de l'ADMM pour mieux garantir la convergence dans un cas non convexe. Dans le cadre de cette thèse, l'ADMM ne sera utilisé que sur des cas convexes, c'est pourquoi cet algorithme n'a pas été utilisé.

4.2 Problème non convexe

Dans le cadre de cette thèse, la résolution d'un OPF ne sera réalisée que sur des réseaux radiaux. On utilisera donc une relaxation pour que le problème (2.3) soit toujours convexe. Ainsi, cette partie se concentra sur la relaxation des contraintes et sur la résolution de système non linéaire (2.4). Les algorithmes qui sont présentés ci-après sont très utilisés dans la littérature dans le domaine de la simulation des réseaux électriques. En effet, les lois physiques régissant les puissances et tensions dans le réseau ne sont pas linéaires. La principale difficulté des problèmes non convexes provient du fait que la présence d'une solution unique n'est pas garantie. Ainsi, quel que soit l'algorithme utilisé, la convergence n'est pas garantie dans le cas général.

4.2.1 Relaxation des contraintes convexes

Dans cette partie, on s'intéressera sur la manière de relaxer les contraintes $i.e$ les enlever en rajoutant des termes dans la fonction coût. On considère qu'une partie des contraintes $h(x) = 0$ et $\underline{g} \leq g(x) \leq \overline{g}$ peuvent s'écrire sous la forme $x \in \mathcal{C}$ avec $\mathcal{C}$ convexe. On notera u la variable duale mise à l'échelle (c'est-à-dire divisée par le facteur de pénalité) associée aux contraintes. Le problème peut se ré-écrire sous la forme suivante [90] avec g la fonction indicatrice de l'espace $\mathcal{C}$:

$$\begin{aligned} &\min \; f(x) + g(z) \\ &\text{t.q. } x + z = 0 \end{aligned} \tag{2.25}$$

Le Lagrangien augmenté associé est :

$$L_\rho(x, z, u) = f(x) + g(z) + \frac{\rho}{2}||x - z + u||_2^2 \tag{2.26}$$

La résolution du problème (2.25) peut se faire par les résolutions successives du problème primal et du problème dual. Cet algorithme est encore une autre forme

de l'**ADMM**. Les étapes de l'algorithme de l'ADMM sont les suivantes avec $\Pi_{\mathcal{C}}$ représentant une projection dans $\mathcal{C}$:

$$x^{k+1} = \underset{x}{\operatorname{argmin}}\, f(x) + (\frac{\rho}{2}||x - z^k + u^k||_2^2) \tag{2.27a}$$

$$z^{k+1} = \Pi_{\mathcal{C}}(x^{k+1} + u^k) \tag{2.27b}$$

$$u^{k+1} = u^k + x^{k+1} - z^{k+1} \tag{2.27c}$$

Les résidus permettant d'étudier la convergence sont les suivants :

$$r^k = x^k - z^k \tag{2.28a}$$

$$s^k = \rho * (z^k - z^{k-1}) \tag{2.28b}$$

Les diverses sous-parties du problème de mise à jour de x^{k+1} ou de projection de z^{k+1} peuvent être résolues par différents algorithmes, en fonction des spécificités de la fonction coût et des contraintes qui n'ont pas été relaxées. Pour la suite, lorsque l'on utilisera cette méthode, on dira que l'on a relaxé les contraintes.

4.2.2 Newton Rapshon

Dans cette partie, on s'intéressera à l'algorithme de Newton-Rapshon. Cet algorithme permet de trouver une racine réelle d'une fonction, c'est-à-dire de trouver un x réel tel que $F(x) = 0$. Ceci permet de résoudre trois problèmes distincts :

- si cet algorithme est utilisé sur les équations de KKT, il permet de trouver itérativement l'optimum d'un problème (c'est le principe du point intérieur [69]) ;
- l'algorithme peut être directement utilisé pour résoudre un problème de Power Flow ;
- En ayant $F(x) = \frac{dG}{dx}$, l'algorithme permet de trouver un extremum de la fonction G. Toutefois, si $G(x)$ n'est pas convexe, il n'y a aucune garantie de trouver le minimum global.

Quelle que soit son utilisation, l'algorithme repose sur le développement limité autour d'un point supposé proche de la racine notée x^*. On a donc :

$$f(x) = f(x^*) + f'(x) \cdot (x - x^*) = f'(x) \cdot (x - x^*) \tag{2.29}$$

On peut noter que cette relation est valable que x soit un réel ou un vecteur. La seule distinction est sur $f'(x)$, qui devient la Jacobienne de $f(x)$ dans le cas où x est un vecteur. En résolvant le système défini par la Jacobienne, on peut trouver itérativement la valeur de x^*. C'est-à-dire qu'en posant $\Delta x = (x - x^*)$, on obtient la relation suivante :

$$\Delta x = f'(x)^{-1} \cdot f(x) \tag{2.30}$$

entrée : F, ϵ and k_{max}
sortie : x_{min}, variable duales, Residuals
1 Initialisation x_0 ;
2 **while** $Res > \epsilon$ and $k < kmax$ **do**
3 calcul de $F(x)$;
4 calcul de $f'(x)$;
5 $\Delta x \leftarrow f'(x)^{-1} \cdot F(x)$;
6 $x \leftarrow x + \Delta x$;
7 $Res \leftarrow ||F(x)||$
8 **end**

Algorithme 1 : Algorithme de Newton Raphson

Il est important de noter que le signe $^{-1}$ signifie qu'il faut résoudre le système, mais qu'aucune méthode spécifique n'est imposée. On choisit le x suivant tel que :

$$x^{k+1} = x^k + \Delta x \tag{2.31}$$

L'algorithme qui en découle est Alg. 1.

Le calcul de la Jacobienne et la résolution du système sont des opérations pouvant être coûteuses sur le plan de la complexité. Ainsi, certaines versions ne les calculent pas toutes les itérations, mais cela ne sera pas exploré dans cette thèse.

Cet algorithme est beaucoup utilisé dans la gestion du réseau, soit en support d'un autre algorithme pour résoudre un système, soit en tant que tel, comme vu précédemment, pour le calcul d'un Power Flow [44], [42].

4.2.3 Gauss Seidel

De manière analogue à la méthode précédente, un système d'équations non linéaires peut être résolu avec la méthode de Gauss Seidel. Ceci permettant de rechercher l'extremum d'une fonction lorsque la méthode est appliquée sur la dérivée. Le principe de cette méthode est d'itérativement résoudre le système en fixant toutes les variables sauf une. Dans le cas où f est de classe C^1, coercive et strictement convexe, le problème d'optimisation admet une unique solution x^*. De plus, la suite des solutions x^k converge vers celle-ci quelque soit le point initial choisi. Dans le cas où le système à résoudre est sous la forme $Ax = b$, la condition de convergence est A symétrique définie positive ou à diagonale strictement dominante [92]. En l'absence de convexité, il n'y a aucune garantie de convergence. L'algorithme en résultant est Alg. 2.

La résolution de chaque étape peut être faite par différentes méthodes. Dans

```
entrée : F, ϵ and k_max
sortie : x_min, variable duales, Residuals
1 Initialisation x_0 ;
2 while Res > ϵ and k < kmax do
3     for i=1,...M do                                  // all equations
4         x_i^{k+1} ← résoudre F_i(x_i, x_{j<i}^{k+1}, x_{j>i}^k) = 0 ;
5     end
6     Res ← ||F(x)||;
7 end
```

Algorithme 2 : Algorithme de Gauss Seidel

le cas d'un système linéaire, la résolution est même sous une forme close :

$$x_i^{k+1} = \frac{1}{a_{ii}}\Big(b_i - \sum_{j=1}^{i-1} a_{ij}x_i^{k+1} - \sum_{j=i+1}^{N} a_{ij}x_i^k\Big) \tag{2.32}$$

4.2.4 Performance des algorithmes et autres possibilités

De même que précédemment, tous ces algorithmes introduisent une résolution itérative. On se demandera dans un premier temps la valeur de la complexité de chaque itération, pour ensuite se concentrer la vitesse de convergence théorique des algorithmes.

La méthode de Newton nécessite l'évaluation de $F(x)$, de la Jacobienne et enfin la résolution d'un système. Dans le cas général, la résolution d'un système a une complexité au moins cubique. Cependant, en utilisant des matrices creuses, il est possible de diminuer la complexité jusqu'à une moyenne en $O(N^{1.4})$ [86]. Si l'algorithme converge (c'est-à-dire que le point initial est assez proche de la racine) la convergence est quadratique. C'est-à-dire que l'erreur à une itération k peut être majorée par un terme quadratique $\epsilon^k = |x^k - x^*| \leq K|x^0 - x^*|^{2k}$.

La méthode de résolution de chaque étape de Gauss Seidel va fortement dépendre de la forme de l'équation. Cependant, une seule dimension est utilisée par minimisation, réduisant les dimensions de chaque minimisation. Dans un cas non convexe, puisqu'il n'y a aucune garantie de convergence, on ne peut déterminer de vitesse de convergence. Dans tous les cas, la méthode de Gauss Seidel ne reposant que sur peu d'hypothèses n'a pas une bonne vitesse de convergence et nécessite en général un grand nombre d'itérations pour converger.

D'autres algorithmes existent, tels que l'algorithme de Jacobi [92] ou une méthode de gradient conjugué. Le premier algorithme présente des propriétés de convergence moins favorables que GS, tout en restant similaire. C'est pourquoi on a choisi ce dernier.

4.3 Optimisation quadratique

Un problème d'optimisation quadratique est un problème où l'on minimise une fonction coût quadratique sur un espace de contrainte de la forme d'un polyèdre convexe. Ainsi, on peut définir les matrices P, A_I, A_E et les vecteurs l, u, b, q tel que le problème (2.3) peut s'écrire sous la forme suivante :

$$\begin{aligned} \min \ & 0.5x^T P x + q^T x \\ \text{t.q. } & A_E x - b = 0 \\ & l \leq A_I x \leq u \end{aligned} \tag{2.33}$$

On notera que la fonction est convexe si la matrice P est semi-définie positive, et strictement convexe si P est définie positive.

Dans le cadre de cette thèse, le problème (2.3) sera quadratique pour le problème de marché lorsque les fonctions coût sont quadratiques. Certaines versions de l'Optimal Power Flow sont relaxées ou approximées pour être quadratiques. Les algorithmes qui seront présentés ne sont efficaces que dans le cas où les fonctions coûts sont convexes. Ainsi P sera dans la suite toujours définie positive. Il est important de noter que, puisque P est définie positive, on peut en déduire que P est inversible, ce qui signifie que P^{-1} existe et peut être calculé. De plus, on considérera que la matrice P est symétrique. Effectivement, lorsqu'on aborde ce type de problème, il est toujours possible d'obtenir une matrice symétrique P en la transformant en $P' = 0,5(P + P^T)$.

On regardera dans un premier temps comment résoudre un problème d'optimisation quadratique dans le cas général. Ensuite, on présentera des manières de résoudre les cas où les contraintes et fonctions coûts ont des formes particulières. On ne s'intéressera qu'aux formes particulières qui seront utilisées dans cette thèse.

4.3.1 OSQP

L'algorithme *Operator Splitting solver for Quadratic Programs* **OSQP** [93], est un algorithme de l'état de l'art considéré comme une référence pour la résolution de problème quadratique dans le cas général. Une implémentation est disponible en ligne [1] et peut donc être considérée comme le solveur par défaut pour ce type de problème. Leur notation ne prend en compte que des inégalités entre deux bornes, mais on peut faire en sorte que ces deux bornes soient égales à b pour que cela devienne une contrainte d'égalité. Cet algorithme est notamment utilisé pour résoudre les problèmes de marché [19], [21].

Pour résoudre le problème, une variable de décision notée z est ajoutée pour

1. `https://github.com/oxfordcontrol/osqp`

prendre en compte les contraintes. L'équation (2.33) devient :

$$\begin{aligned} x = & \operatorname{argmin} \frac{1}{2} x^T P x + q^T x \\ \text{t.q.} \quad & Ax = z \\ & l \leq z \leq u \end{aligned} \tag{2.34}$$

Ensuite, la résolution nécessite les étapes suivantes :

$$\begin{aligned} (\tilde{x}^{j+1}, \tilde{z}^{j+1}) = & \underset{(\tilde{x},\tilde{z})\ A\tilde{x}=\tilde{z}}{\operatorname{argmin}} \frac{1}{2} \tilde{x}^T P \tilde{x} + q^T \tilde{x} \\ & + \frac{\sigma}{2} \left\| \tilde{x} - x^j + \sigma^{-1} \cdot w^j \right\|_2^2 \\ & + \frac{\rho_l}{2} \left\| \tilde{z} - z^j + \rho_l^{-1} \cdot y^j \right\|_2^2 \end{aligned} \tag{2.35}$$

$$x^{j+1} = \alpha \cdot \tilde{x}^{j+1} + (1-\alpha) x^j + \sigma^{-1} w^j \tag{2.36a}$$

$$z^{j+1} = \Pi(\alpha \tilde{z}^{j+1} + (1-\alpha) z^j + \rho_l^{-1} y^j) \tag{2.36b}$$

$$w^{j+1} = w^j + \sigma(\alpha \tilde{x}^{j+1} + (1-\alpha) x^j - x^{j+1}) \tag{2.36c}$$

$$y^{j+1} = y^j + \rho_l(\alpha \tilde{z}^{j+1} + (1-\alpha) z^j - z^{j+1}) \tag{2.36d}$$

avec Π étant la projection euclidienne dans l'espace respectant les contraintes, c'est-à-dire la projection entre les bornes de chaque composant de z^{j+1}. Il est intéressant de noter que la variable w s'annule dès la première étape et que son calcul peut donc être enlevé du système. Soit v le multiplicateur de Lagrange associé à la contrainte $Ax = z$. Il peut être prouvé, [90], [93] que résoudre le problème (2.35) est équivalent à résoudre le système suivant :

$$\begin{pmatrix} P + \sigma I_{M_n} & A^T \\ A & -\rho_l^{-1} I_{M_n+1} \end{pmatrix} \begin{pmatrix} \tilde{x}^{j+1} \\ v^{j+1} \end{pmatrix} = \begin{pmatrix} \sigma x^j - q \\ z^j - \rho_l^{-1} y^j \end{pmatrix} \tag{2.37}$$

$$\tilde{z}^{j+1} = \tilde{z}^j + \rho_l^{-1}(v^{j+1} - y^j) \tag{2.38}$$

La matrice de KKT (Karush Kuhn Truker) est celle qui définit le système, (2.37), à résoudre pour obtenir l'optimum. L'algorithme de base pour OSQP est présenté dans l'algorithme 3. La version disponible utilise d'autres opérations pour que la convergence soit plus rapide, comme la variation du facteur de pénalité ρ_l selon la valeur des différents résidus [93].

La résolution du système (2.37), peut être faite au choix par une méthode directe avec une factorisation LDL^T ou indirecte avec la méthode du gradient conjugué. Ainsi, la résolution par inverse de la matrice n'est pas implémentée et n'est donc pas utilisable, même dans le cas où cette matrice serait constante.

entrée : H, q, A, l, u, ϵ and k_{max}
sortie : x_{min}

1 $\tilde{z} \leftarrow 0$;
2 $\tilde{x} \leftarrow 0$;
3 $x \leftarrow x0$;
4 $z \leftarrow Ax0$;
5 **while** $Res > \epsilon$ and $k < kmax$ **do**
6 $(\tilde{x}^{j+1} v^{j+1}) \leftarrow$ solve system ;
7 $\tilde{z}^{j+1} \leftarrow \tilde{z}^{j} + \rho^{-1}(v^{j+1} - y^{j})$;
8 $x^{j+1} \leftarrow \alpha \cdot \tilde{x}^{j+1} + (1-\alpha)x^{j}$;
9 $z^{j+1} \leftarrow \Pi(\alpha\tilde{z}^{j+1} + (1-\alpha)z^{j} + \rho^{-1}y^{j})$;
10 $y^{j+1} \leftarrow y^{j} + \rho(\alpha\tilde{z}^{j+1} + (1-\alpha)z^{j} - z^{j+1})$;
11 $r1 \leftarrow \|Ax - z\|$;
12 $r2 \leftarrow \|Px + q + A^{T}y\|$;
13 $err \leftarrow max(r1, r2)$;
14 **end**

Algorithme 3 : Algorithme d'OSQP

Dans la suite, on regardera des cas particuliers qui peuvent se résoudre analytiquement avec une forme fermée (*closed* en anglais). C'est-à-dire qu'un nombre fini d'opérations connues a priori permet d'obtenir la solution exacte (à opposer aux méthodes itératives).

4.3.2 Cas P diagonale et sans inégalités

Si P est une matrice diagonale et positive, et A une matrice de rang maximal, et que toutes les grandeurs sont réelles, alors le minimum peut être trouvé analytiquement via la relation suivante :

$$x_{min} = (P^{-1}A^{T}(AP^{-1}A^{T})^{-1}AP^{-1} - P^{-1})q \tag{2.39}$$

Pour simplifier la notation, on définit une matrice M tel que $x_{min} = Mq$. On peut remarquer que le calcul de la matrice M peut être assez coûteux en termes de complexité. Cependant, la matrice P étant diagonale, son inverse peut être calculé aisément. De même, la matrice $AP^{-1}A^{T}$ qu'il faut aussi inverser a pour dimension le nombre de contraintes d'égalité. Si ce nombre ne dépend pas de la dimension du problème, la complexité est amoindrie. Ce type de forme est notamment utilisé après la décentralisation, par exemple dans [59].

4.3.3 Cas sans égalité et sans inégalité affine

Dans le cas où il n'y a pas de contraintes d'égalité ou d'égalités affines, mais seulement des bornes sur les variables d'optimisation ($l \leq x \leq u$), on peut résoudre le problème en deux étapes. La première est la résolution d'un système :

$$x_{min} = -P^{-1}q \tag{2.40}$$

Il est important de noter qu'ici on utilise la notation P^{-1}, mais cela ne signifie pas que l'on doive inverser la matrice. Ainsi, toutes méthodes de résolution de système peuvent être utilisées ici. Ensuite, on doit projeter x_{min} dans l'espace de définition de x. Étant donné que la fonction est convexe et que l'intervalle est un ensemble fermé, on est assuré de trouver l'optimum global de manière certaine.

$$x_{min} = min(max(x_{min}, l), u) \tag{2.41}$$

Dans le cas où P est diagonale on peut écrire l'optimisation comme la minimisation pour chaque coordonnée x_i indépendamment. On peut donc écrire la minimisation ainsi (avec $a_j > 0 \ \forall j$) :

$$x_i = \text{argmin} \sum_j \frac{1}{2} a_j (x_i - b_j)^2 + \sum_j c_j x_i \qquad \forall i \tag{2.42}$$

On peut aisément enlever les termes b_j pour les mettre dans la deuxième somme. Cependant, une minimisation apparaîtra sous cette forme dans la suite c'est pourquoi on écrit la minimisation ainsi.

Une dérivée pour trouver l'extremum nous permet de trouver le minimum pour chaque coordonnée i après une projection dans l'espace de définition de x :

$$x_i = \max(\min(\frac{\sum_j b_j a_j - \sum_j c_j}{\sum_j a_j}, u), l) \tag{2.43}$$

Ce genre d'expression peut se trouver dans les problèmes de marché après avoir décentralisé et relaxé toutes les autres contraintes.

4.3.4 Performance des algorithmes et autres possibilités

Les méthodes en forme fermée n'étant pas itératives, elles sont, sauf dans des cas extrêmes, nécessairement meilleures que des méthodes de résolutions classiques.

Les trois méthodes nécessitent la résolution d'un système. Ainsi, la complexité est cubique dans le cas général, et d'environ $O(N^{1.4})$ dans le cas des matrices

TABLE 2.1 – Synthèse des algorithmes

Algorithme	utilité	Complexité	Convergence
ADMM	relaxation et/ou décentralisation	selon $f_n(x_n)$	sub-lineaire
PAC	décentralisation	selon $f(x)$	linéaire
NR	$F(x) = 0$	$O(N^3) \mid O(N^{1.4})$	quadratique
GS	$F(x) = 0$	selon $F(x_n)$	sub-lineaire
OSQP	$\text{argmin}\, 0.5x^t Hx + q^t$	$O(N^3) \mid O(N^{1.4})$	sub-lineaire
forme closed	cas particuliers	$O(N^3) \mid O(N^{1.4})$	instantanée

creuses. Il est important de noter que, pour la forme fermée, la taille considérée est le nombre de contraintes d'égalité.

Définir la vitesse de convergence des méthodes *closed* n'a pas de sens, puisque ces méthodes atteignent une erreur de 0 dès la première itération.

La méthode d'OSQP se basant sur une ADMM, la convergence est dans le cas général la même que pour une ADMM.

4.4 Conclusion

Dans les parties précédentes, différents algorithmes pour différents types de problèmes ont été présentés. Tous ces algorithmes ont des complexités en ce qui concerne la mémoire utilisée, du nombre de calculs et des types de calcul très différent. De plus, ces algorithmes peuvent être plus ou moins parallélisables. En fonction de ces particularités, leurs comportements (précision, convergence ...) et performances (rapidité ...) peuvent dépendre de l'architecture utilisée. C'est pourquoi il est important de déterminer les différentes architectures et langages de programmation qui seront utilisés pour implémenter ces algorithmes. De plus, ces algorithmes ont des paramètres devant être imposés (par exemple, le facteur de pénalité pour toutes les méthodes qui utilisent le Lagrangien) et qui influe plus ou moins sur la convergence en fonction du cas d'étude. Il est donc important de choisir des cas d'études variés afin d'évaluer l'impact de ces paramètres et les phénomènes de passage à l'échelle en fonction des algorithmes et de leur implémentation.

5 Choix des architectures et outils de programmation

La partie précédente nous a permis d'expliciter les principaux algorithmes utilisés dans le domaine du génie électrique. Dans cette partie, les différentes architectures, utilisées pour l'accélération des calculs, seront présentées. La première partie sera consacrée aux architectures mono et multi-CPU. Ensuite seront présentées les architectures utilisées dans des *personal super-computers*. Ce dernier système sera composé d'un séquenceur CPU qui sera l'hôte et d'une ou plusieurs cartes accélératrices pour les calculs. Dans notre cas, on s'intéressera à l'utilisation de **GPU** et de cartes **FPGA** pour accélérer les calculs.

Les architectures seront caractérisées et comparées pour enfin pouvoir montrer l'adéquation entre ces architectures et l'utilisation que l'on vise dans le cadre de cette thèse.

Pour chacune des architectures, les différents langages pouvant être utilisés seront présentés afin d'expliciter les choix ayant été faits.

5.1 Architectures de calcul parallèle multicœurs

5.1.1 Présentation

L'unité centrale de calcul (ou plus communément appelé en anglais *central processing unit* CPU) est connue pour son efficacité à traiter séquentiellement des opérations complexes. Le CPU a, en général, un faible nombre d'unités de calcul, permettant à une très grande partie de l'architecture d'être dédiée au contrôle et à la mémoire cache. Grâce à ses nombreux caches, la latence pour l'accès à la mémoire est très faible. Pour ce faire, les caches utilisent la cohérence spatiale et temporelle des programmes. C'est-à-dire que le processeur utilise le fait que dans la majorité des cas lorsqu'une variable est calculée, c'est pour être réutilisée plus tard (cohérence temporelle), et il faut donc la garder dans les caches. De même en utilisant le fait que les données utiles d'un programme sont stockées physiquement dans la même zone mémoire (cohérence spatiale), le processeur peut gagner du temps en mettant dans le cache toute la zone mémoire autour de la variable qu'il utilise. Le faible nombre d'unités de calcul permet d'avoir pour chacun d'entre eux différents matériels dédiés aux calculs. Ceci permet au CPU d'effectuer des opérations complexes rapidement. Ainsi certains CPU, peuvent être, par exemple, aussi rapides pour des calculs sur des flottants simple ou double précisions.

Le CPU est l'architecture de référence dans le domaine du génie électrique. Il est utilisé pour permettre la démonstration de la validité des méthodes de gestion proposée [26], [25]. En effet, le CPU peut être assez rapide, même sans parallélisation, avec des temps de calcul de l'ordre de la seconde sur des petits cas.

L'amélioration continue des bibliothèques de calcul (par exemple les *solvers*) couplée à la multitude de langages disponibles permet la validation fonctionnelle des algorithmes en un temps d'implémentation faible.

5.1.2 Accélération des calculs sur CPU

Pour accélérer les calculs sur CPU, il existe plusieurs leviers d'action : augmenter la vitesse de chaque cycle ou permettre plusieurs opérations en un cycle [94].

Si le premier point dépend plutôt de la fréquence de fonctionnement et donc de la technologie de CPU utilisée. En réalité, les CPU modernes permettent de *booster* la fréquence et ainsi dépasser la valeur maximale. Sur certains CPU, ceci n'est utilisable que lorsque l'utilisation est sur un seul cœur et avec un seul thread.

Le deuxième point consiste à exploiter le parallélisme du CPU et de l'application considérée. Il existe différents niveaux de parallélisme sur CPU. Le premier est le parallélisme au niveau des instructions. En effet, les processeurs modernes sont capables de réaliser des exécutions superscalaires. C'est-à-dire qu'ils sont capables de repérer les instructions indépendantes et les exécuter en parallèle. Le CPU peut donc, par exemple, exécuter plusieurs additions pendant l'exécution d'une multiplication, la dernière opération étant bien plus lente. Le compilateur peut intervenir pour changer l'ordre des instructions ou pour les rendre indépendantes. Cette parallélisation se réalisant au niveau du processeur, elle est difficilement exploitable en amont de l'implémentation. Un moyen d'agir indirectement sur cette parallélisation est d'éviter les embranchements (*branchless programming*) afin de permettre au processeur d'anticiper les instructions à exécuter et ainsi être plus efficace. Une autre manière d'agir est d'essayer d'avoir un code demandant le moins de mémoire possible pour que l'ensemble tienne dans les caches du CPU. Ce dernier point est bien souvent incompatible avec d'autres optimisations.

Un autre moyen de paralléliser consiste à utiliser le matériel vectoriel que possède les CPU comme il a été fait dans [95]. Là où la parallélisation précédente effectuait plusieurs instructions différentes, ici le CPU est capable d'effectuer une seule instruction sur des données différentes (SIMD : *single instruction multiple data*). En effet les composants *Streaming SIMD Extensions* (appelé **SSE**) et *Advanced Vector Extensions* (**AVX**) permettent au CPU d'appliquer la même instruction sur plusieurs données en même temps. S'il existe des mots clés pour indiquer quelles parties du code sont vectorisables, le compilateur peut de lui-même repérer ces zones. Ainsi il peut être intéressant de coder en faisant ressortir les zones où la vectorisation peut être utilisée. Cela peut être fait, par exemple, en séparant plusieurs calculs sur un vecteur en plusieurs boucles *for* successives (une par opération).

Le dernier moyen de paralléliser consiste à exploiter la structure multicœurs des CPU en parallélisant sur plusieurs threads comme dans [96]. Il est important

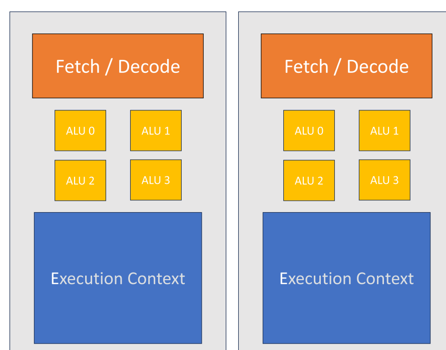


FIGURE 2.5 – Exemple d'une architecture d'un CPU dual core avec 4 unités de calcul par cœur

de noter que les threads peuvent s'exécuter sur un ou plusieurs cœurs. Dans le cas où plusieurs threads sont sur un même cœur, ils sont sérialisés, mais leur entrelacement permet de cacher la latence des accès mémoires.

La Fig. 2.5 montre un schéma d'un CPU multicœur possédant plusieurs unités de calcul par cœur. On peut y voir que chaque cœur possède sa propre unité de décodage d'instruction permettant donc de faire différentes opérations sur les différents cœurs (MIMD : *multiple instruction multiple data*). Tandis que la vectorisation se faisant sur un seul cœur ne peut se faire que sur la même instruction. Quelle que soit la manière de paralléliser, il est important de faire la distinction entre la mémoire locale à chaque thread et la mémoire partagée accessible par tous les threads. En effet, tout conflit en mémoire, par exemple si plusieurs threads doivent écrire une valeur dans une même variable, doit être évité pour maximiser la parallélisation possible. Dans le cas où ce conflit est inévitable, il faut protéger la variable et sérialiser les accès.

La gestion des accès mémoires et l'utilisation ou non des caches sont les points majeurs limitant les performances des CPU.

5.1.3 Langage utilisé

De très nombreux langages existent et sont utilisés pour réaliser des simulations du réseau électrique sur CPU. Ainsi on peut retrouver dans la littérature du Python avec Pandapower [97], du Matlab avec MatPower [73]. On y trouve aussi du Julia avec PowerModels [98] ou [21]. Python et Matlab sont des langages plus accessibles, cependant ce sont des langages très lents et avec peu de marge de manœuvre pour s'adapter au matériel utilisé. Un des langages permettant d'être le plus rapide est **C++**. Le langage Julia semble réussir à atteindre les mêmes performances que **C++** en termes de rapidité [99]. Ces deux langages permettent de s'interfacer

de manière très précise avec des cartes accélératrices. Dans l'optique de réaliser l'adéquation entre l'architecture et l'algorithme, le langage tant qu'il permet de réaliser des optimisations permettant de s'adapter à l'architecture ne devrait pas avoir d'impact sur la méthodologie et les résultats. Les performances en termes de temps de calcul ou de mémoire utilisée seront évidemment dépendantes du langage, mais devraient garder les mêmes ordres de grandeur. Ainsi dans le cadre de cette thèse, c'est le langage C++ qui a été arbitrairement choisi.

Pour la programmation parallèle sur CPU, on pourra citer les langages d'OpenMP pour une programmation en mémoire partagée (tous les threads ont accès à la mémoire globale) et MPI pour une programmation en mémoire distribuée.

5.2 Architectures pour l'accélération de calcul

5.2.1 Présentation

Afin d'accélérer les calculs, il est possible d'utiliser une carte extérieure au CPU pour que celui-ci y distribue les calculs. De nombreux types de cartes sont possibles, mais les plus communes sont les cartes GPU et les cartes FPGA.

GPU : L'unité de traitement graphique ou plus communément appelée en anglais *Graphics Processing Unit* (GPU) a une fréquence de calcul plus faible et une plus grande latence que le CPU. Cependant, cela est compensé par son grand nombre d'unités de calcul, qui en fait une architecture hautement parallélisable. Dans le cadre de cette thèse, on se concentrera sur les cartes graphiques Nvidia, celles-ci étant les pionnières en tant que GP-GPU (*General purpose GPU*). Les principes sont les mêmes pour les autres cartes. L'architecture générale est représentée en Fig. 2.6, les variations entre les cartes physiques proviendront de la quantité de chaque élément, des types d'unité de calcul contenu dans les processeurs et des modes de gestions de l'exécution.

Le GPU est donc composé de plusieurs multiprocesseurs capables d'exécuter en parallèle des instructions différentes et d'une mémoire globale. Chaque multiprocesseur est composé de plusieurs processeurs (aussi appelé cœur cuda), de différentes mémoires accessibles en lecture seule (mémoire cache et texture), ou en lecture-écriture (la mémoire partagée). Chaque processeur a une mémoire accessible par lui seul : des registres. Les processeurs sont contrôlés pour effectuer les instructions demandées en fonction des données disponibles. Ils sont contrôlés par groupe de 32 processeurs appelé *warp*. Les processeurs dans un warp exécutent la même instruction (sur des données différentes), et accèdent à la mémoire en même temps. Si cela n'est pas possible (instructions différentes ou mémoires à des emplacements différents) les accès et exécutions sont sérialisés.

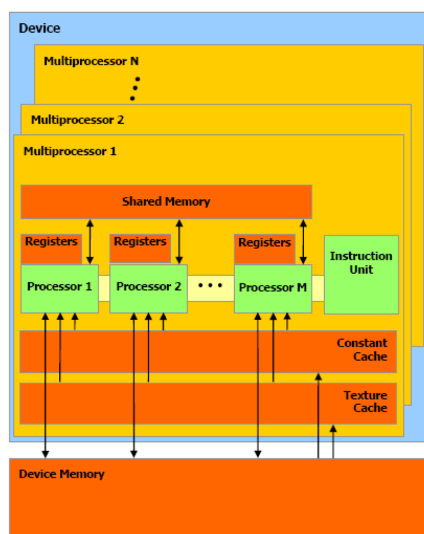


FIGURE 2.6 – Architecture d'un GPU

FPGA Les circuits intégrés reprogrammables aussi appelé *Field Programmable Gate Array* (FPGA) permettent, comme le GPU, d'accélérer les calculs grâce à leur architecture hautement parallèle. Les FPGA ont l'avantage majeur de consommer beaucoup moins d'énergie. Ceci permet de faire du multi-FPGA pour faire du calcul intensif en ayant moins de contraintes thermiques. De plus là où le CPU et le GPU ont une structure rigide, les FPGA permettent plus de souplesse grâce à leur architecture qui peut être adaptée à l'utilisation réalisée. Ainsi les FPGA peuvent être utilisées pour la résolution de problèmes, tel que l'Optimal Power Flow [70].

La fréquence de fonctionnement est largement inférieure à celle d'un GPU, rendant les calculs plus "lent" par essence cependant les FPGA sont capables de faire du traitement de données à la volée permettant de travailler avec des flux de données. Elles sont ainsi utilisées en tant que capteur dans le réseau électrique [100]. Les FPGA sont aussi très utilisées dans les domaines nécessitants des traitements indépendants sur un grand nombre de données (traitement d'image, de signal) ou lorsqu'il faut s'interfacer avec un système physique pouvant transférer des données en temps réel. Par exemple, les FPGA peuvent servir pour implémenter un algorithme de machine learning supervisé de classification (SVM) afin d'analyser les communications pour chaque agent dans un SmartGrid et éviter des attaques [101]. Le FPGA permettant de traiter les données reçues tout en faisant une simulation (en *modele predictive control*) avec une actualisation des données pendant la simulation. Ceci permettant par exemple de contrôler la tension dans le réseau [102]

5.2.2 Le langage

Originellement, les cartes FPGA étaient paramétrées via un langage de description tel que VHDL ou verilog. Les constructeurs proposent actuellement un moyen de paramétrer les cartes via un outil de prototypage rapide permettant de "coder" l'application en faisant abstraction de la carte utilisée. Des exemples de ce type de langage sont OpenCL, OneApi et LegUp. Si la première manière permet de concevoir finement l'architecture, celle-ci est très chronophage et dépend de la carte à utiliser. La deuxième méthode permet de gagner du temps, mais rend plus difficiles les réglages fins.

Plusieurs langages existent pour la programmation sur GPU. Même s'il est possible de les comparer [85], leurs performances restent dépendantes de la carte utilisée, de l'application considérée et du programmeur. Dans notre cas, les cartes utilisées étant exclusivement des cartes Nvidia, ce sera le langage Cuda qui sera conservé.

Cuda est un langage de programmation de haut niveau fourni par Nvidia qui permet d'utiliser leur carte graphique pour réaliser des opérations parallélisées tout en faisant abstraction de l'architecture de la carte. Le principe de fonctionnement est représenté sur la Fig. 2.7. Le code *Host* est le code qui est exécuté sur le CPU et qui est ici du C++. Dans ce code-ci, on peut utiliser des fonctions nommées *kernel* qui vont en réalité s'exécuter sur le GPU. Lorsque l'on utilise ces fonctions, on doit indiquer la taille de la *grid* (grille), en indiquant le nombre de blocs et le nombre de *thread* par bloc. Les threads d'un même bloc peuvent se synchroniser et partager une mémoire commune : la mémoire partagée (*shared memory*). Sinon tous les threads ont accès à la même mémoire globale et possèdent une mémoire locale.

5.3 Architecture sélectionnée

5.3.1 Justification

Pour rappel, l'objectif de ce travail est de réaliser une application de simulation de gestion et fonctionnement du réseau. On se considère en amont du contrôle, en *off-line*. On travaille donc avec des données que l'on supposera entièrement connues au moment du lancement de la simulation. On considèrera de grands cas avec plus que des milliers d'agents ou bus. Cependant, dans cette thèse, on considèrera que la mémoire nécessaire pour simuler ne dépassera pas la mémoire disponible du CPU ou des cartes. Les données peuvent donc être transférées une fois pour toutes dans les cartes. Il n'y a pas besoin, par exemple, de réaliser de traitement de données à la volée ou de *swaper* la mémoire des cartes pour traiter en plusieurs fois.

Le CPU est l'architecture de référence lorsque l'on programme une application. Dans le cadre de cette thèse, la combinaison CPU et C++ seront utilisés en tant

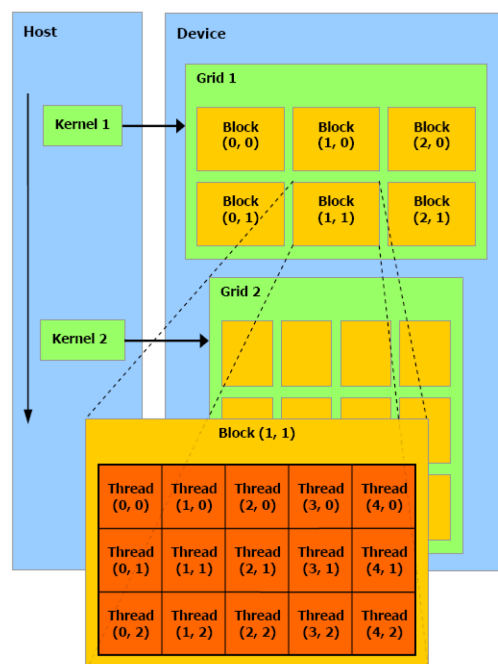


FIGURE 2.7 – Architecture Cuda

que base pour comparer avec les autres applications. Cette base représentera la vitesse atteignable avant l'application de la démarche **A3**. Elle permettra aussi de réaliser la validation fonctionnelle de notre application. La comparaison avec l'état de l'art pourra montrer l'influence conjuguée d'un changement de langage, de matériel. Tandis que la comparaison de cette base avec l'application optimisée montrera l'impact de la démarche. De plus, il est possible que plusieurs algorithmes soient dans certains cas bien plus rapides sur CPU que sur d'autres architectures. Il est donc intéressant d'étudier aussi l'implémentation sur cette architecture. Le CPU sera aussi utilisé en tant qu'hôte pour l'utilisation des cartes d'accélération des calculs.

La parallélisation sur CPU via OpenMP est très simple à mettre en place. Cependant, cette simplicité d'utilisation provient du fait que le programmeur n'a que très peu de marge de manœuvre pour optimiser cette parallélisation. Ainsi des méthodes utilisant openMP seront utilisées pour permettre des comparaisons, mais l'accent ne sera pas mis dessus. D'autres langages comme Matlab et Python seront utilisés pour faire du pré et post-traitement de données, *i.e.* pour mettre en forme les fichiers de cas d'études ou pour analyser et afficher les résultats.

Même si le GPU fonctionne à plus faible fréquence que le CPU et est donc plus lent par essence, son très grand nombre de cœurs permet de paralléliser son fonctionnement. Or la simulation des problèmes du réseau électrique est intrin-

Cuda caractéristique		GPU caractéristique	
Cuda driver version	11.4	Global Memory	6.144GB
Cuda capability version	8.6	Multiprocesseur	30
Nombre registre par block	65536	Coeur cuda total	3840
Taille d'un warp	32	Fréquence horloge	1.42GHz
Nombre de thread par MP	1536	Fréquence de la mémoire	7GHz
Nombre de thread par bloc	1023	Taille cache L2	3.14 MB

TABLE 2.2 – Caractéristique matériel du GPU utilisé

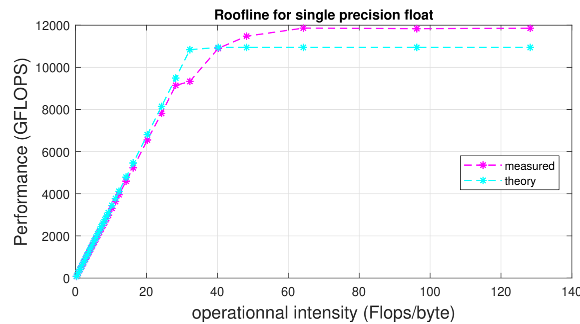


FIGURE 2.8 – Roofline du GPU utilisé, mesuré par [103]

sèquement parallèle [87] justifiant l'utilisation d'architecture hautement parallèle. L'objectif étant une simulation sur une seule machine, ces communications resteront à l'intérieur du GPU, et correspondront à une écriture et une lecture dans la mémoire globale du GPU.

Le fait que le traitement des données est en place (et non pas à la volée) et que l'on n'utilisera qu'une seule carte (donc pas de contrainte thermique), nous permet de ne pas conserver la carte FPGA dans la suite de cette thèse.

5.3.2 Caractéristique matérielle

La carte GPU qui a été utilisée dans le cadre de ce travail de thèse (sauf mention explicite) est une carte Nvidia GPU GeForce RTX 3060. Cette carte a une architecture appelée Ampere (noté GA 106) avec 30 **MP** et 3840 cœurs Cuda. Les caractéristiques indiquées par le script de test téléchargé avec le logiciel cuda sont regroupées dans le Tab. 2.2. Le graphe suivant, Fig. 2.8 représente la Roofline théorique [104] et celle mesurée par le benchmark de [103].

5.4 Conclusion

Dans cette partie, nous avons pu voir une présentation des différentes architectures qui peuvent être utilisées dans le cadre de ce travail. Il est important de noter que ces architectures sont vraiment physiquement différentes des unes des autres. Ainsi elles ont chacune des contraintes à respecter pour être utilisée efficacement. Ces contraintes vont conditionner leurs performances en fonction des algorithmes et des optimisations qui seront accessibles. La démarche d'Adéquation Algorithme Architecture va nous permettre d'optimiser les algorithmes et leur implémentation en tenant en compte a priori des contraintes du matériel pour ainsi obtenir les meilleures performances possibles.

6 Choix des jeux de données

L'objectif de cette partie sera de présenter les différents cas d'étude qui ont été sélectionnés dans cette démarche d'Adéquation Algorithme Architecture. Pour rappel, les notations sont définies dans le premier chapitre. De manière très résumée, nos cas d'études sont définis par trois types de données qui sont la description du réseau, la description des agents et enfin le positionnement des agents sur le réseau. Les informations sur le réseau contiennent la topologie du réseau et les caractéristiques physiques des lignes. Les informations sur les agents sont leurs contraintes et leur fonction coût qui est sous la forme :

$$\begin{aligned} g_n(p_n) &= 0.5a_n^p p_n^2 + b_n^p p_n + 0.5a_n^q q_n^2 + b_n^q q_n \\ &= 0.5a_n^p (p_n - P_0)^2 + 0.5a_n^q (Q_n - P_0)^2 \end{aligned} \tag{2.44}$$

Pour rappel, il y a bien équivalence entre les deux écritures de la fonction coût, car l'objectif est de trouver le minimum. Or le terme constant ne change pas l'emplacement du minimum, il ne change que la valeur minimale de la fonction objectif.

Lorsque l'on ne souhaite que travailler avec la puissance active, on ne conserve que les termes en p. En connaissant la puissance voulue (P_0) par chaque agent, on peut déterminer la valeur de b_n^p ainsi :

$$b_n^p = -a_n^p \cdot P0 \tag{2.45}$$

On peut déterminer de la même manière b_n^q à partir de a_n^q et $Q0$.

Enfin, on rajoute lorsque l'on simule le marché un terme de préférence hétérogène, noté β. Ce terme permet de représenter différentes taxes ou préférences

des agents par une pénalité sur chaque échange. Le nouveau terme qui en découle pour chaque agent est :

$$f_n(\mathbf{T}_n) = \sum_{m\in\omega_m} \beta_{nm} t_{nm} \tag{2.46}$$

avec $\mathbf{T}_n$ l'ensemble des échanges t_{nm} de l'agent n, ω_m l'ensemble des partenaires économiques de l'agent n et β_{nm} la pénalité associé à l'échange t_{nm} telle que $\beta_{nm} t_{nm} \geq 0$.

Les consommateurs ne peuvent que consommer/acheter tandis que les producteurs ne peuvent que produire/vendre. Ainsi, les consommateurs ne sont pas reliés entre eux ; de même pour les producteurs. Les consommacteurs peuvent tout faire. Ils sont donc reliés à tous les autres acteurs.

Dans le cas "AC", on prend en compte les pertes dans les lignes. Pour que l'équilibre soit fait même dans le cadre d'un marché, on ajoute au cas d'étude un agent des pertes. Cet agent est un consommateur qui à l'instar de RTE, achète la puissance nécessaire pour compenser les pertes dans le réseau. Par convention, cet agent sera toujours l'agent 0. De même, puisqu'à chaque agent est associé un bus, il sera placé sur le bus 0 dans le cas d'une modélisation AC (et n'existera pas dans le cas "DC"). Cependant, on considérera qu'il n'injectera ou ne soutirera jamais de puissance sur le réseau. En effet, lorsque l'on simule le réseau, les pertes sont déjà comptées dans les lignes, l'agent 0 a une existence exclusivement économique. Ainsi, les caractéristiques de cet agent sont $a_0 = 0$, $b_0 = 0$ pour la puissance active et réactive, $\overline{p_n} = \underline{p_n} = P_{perte}$. Lorsque les pertes sont évaluées itérativement, les limites de l'agent varient à chaque itération. Pour ne pas devoir faire aussi varier les limites économiques des échanges, ceux-ci ne sont pas contraints ($l_b = -\inf$). La même chose s'applique à la puissance q.

Il est important de noter que l'ensemble des cas est défini en *per unit*.

Dans le cadre de cette thèse, les cas d'études seront séparés en trois catégories selon leur rôle dans la démarche. Les premiers cas sont des cas d'étude fixes de petite taille. Ces cas serviront pour la validation fonctionnelle des différentes implémentations. Ensuite, des études de cas fixes à plus grande échelle nous permettront de comparer les différents couples algorithmes-architectures en grande dimension, ou encore de nous comparer à l'état de l'art. Enfin, les derniers cas sont des cas aléatoires dans lesquels on peut faire varier les dimensions ou certains paramètres. Ces indicateurs nous permettront d'évaluer la complexité mesurée (autrement dit le passage à l'échelle) de différentes implémentations des algorithmes. Ils nous permettront aussi de faire des études paramétriques pour les différentes implémentations.

TABLE 2.3 – Définition des paramètres pour le cas 2 nœuds

Agent	indice	bus	$\underline{p_n}$	$\overline{p_n}$	a_P	b_P	β	Q_0
Consommateur	1	0	-30	0	1	8	-1	-1
Producteur	2	1	0	60	1	4	1	0

6.1 Cas pour vérifier la convergence

6.1.1 Cas 2 bus

Ce cas simple a pour particularité d'être solvable analytiquement. Ainsi, il s'agit d'un cas constitué de deux agents, chacun sur un bus, et reliés l'un à l'autre par une ligne. Le cas est représenté sur la Fig. 2.9. La ligne a une admittance $b = 100$ (puisqu'il n'y en a qu'une seule, cette valeur n'a pas d'influence) et une limite thermique en puissance de $\overline{\phi} = 0,8$. Les paramètres des agents sont regroupés dans Tab. 2.3, en se basant sur les fonctions coûts de (2.44). La fonction à minimiser se note donc ainsi :

$$f(x) = 0.5a_1p_1^2 + b_1p_1 + \beta_{12}t_{12} + 0.5a_2p_2^2 + b_2p_2 + \beta_{21}t_{21} \tag{2.47}$$

On remarquera que de par les contraintes d'équilibre des puissances ((1.2c) et (1.2b) du chapitre suivant) on a $p_1 = t_{12} = -p_2 = -t_{21} = \pm\phi$ ce qui nous donne après simplification :

$$f(x) = p_1^2 + 2p_1 \tag{2.48}$$

Le minimum est donc en $p_1 = -1$. Cependant si on prend en compte les contraintes dans les lignes on a $p_1 = -\overline{\phi} = -0.8$. Dans le cas sans préférences hétérogènes, l'optimum sans contraintes de ligne est à 2.

cas dit "AC" : Dans le cas où l'on considère aussi la puissance réactive, chaque agent a une puissance réactive objectif notée Q_0. Dans ce cas là on considère que les impédances valent $r = 0.003$ et $x = 0.01$ pour la ligne. L'agent des pertes étant un consommateur, il n'échange qu'avec le producteur 2.

cas dit "AC" : Dans le cas où l'on considère aussi la puissance réactive, chaque agent a une puissance réactive objective notée Q_0. Dans ce cas-là, on considère que les impédances valent $r = 0.003$ et $x = 0.01$ pour la ligne. L'agent des pertes étant un consommateur, il n'échange qu'avec le producteur 2.

Les avantages de ce cas sont les suivants.

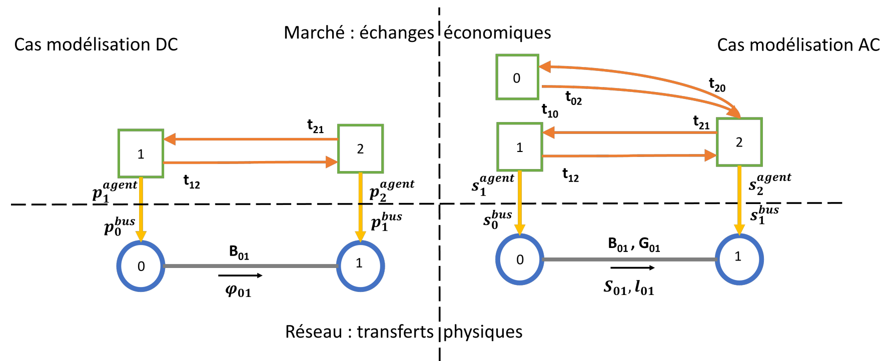


FIGURE 2.9 – Modélisation du cas 2 bus dans le cas DC (gauche) et AC (droite)

— Le cas est petit, donc il est rapide à calculer. Dans le cas de l'étude d'un marché, la solution est calculable analytiquement, ce qui permet de vérifier la convergence.

— C'est un cas où les bornes des puissances minimales et maximales ne sont pas reliées à la puissance voulue des agents. Se placer au milieu des bornes, donne donc une très mauvaise initialisation. De même, les puissances pour le minimum du marché ne respectent pas du tout les contraintes du réseau. Par conséquent, cet exemple sert à tester la convergence lorsque les contraintes sont actives et que la solution initiale est éloignée de l'optimum.

— On peut également noter que c'est un réseau radial avec de la puissance qui "remonte" vers le bus 0.

6.1.2 Cas 3 bus

Ce cas simple est constitué de 3 bus et de 3 lignes. Le cas est représenté sur la Fig. 2.10. Pour rappel, les agents ne font pas d'échange avec des pairs du même type qu'eux. Contrairement au cas précédent, ce n'est pas un cas radial. Dans ce cas-ci, on a posé $S_{base} = 100MVA$ et $V_{base} = 230kW$ pour normaliser le cas d'étude. On ne considère pas de préférence hétérogène, $\beta = 0$ pour tous les échanges. Les paramètres des agents sont regroupés dans Tab. 2.4, et les paramètres pour les lignes sont dans Tab. 2.5.

Lorsque les limites thermiques des lignes sont notées ∞, cela signifie que les lignes ne sont pas contraintes. Dans le cas où l'implémentation permet de ne considérer que certaines contraintes, celles-ci ne seront pas considérées. Dans le cas contraire, les limites seront fixées à une valeur arbitrairement grande (typiquement $1e6$) pour que ces contraintes ne soient jamais actives.

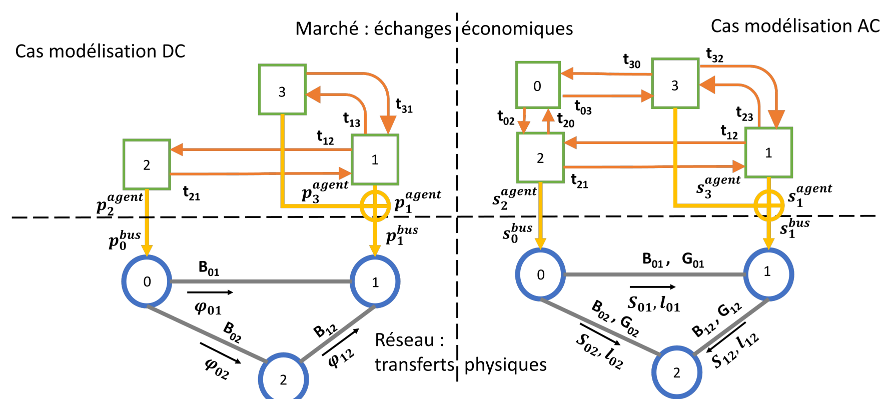


FIGURE 2.10 – Modélisation du cas 3 bus dans le cas DC (gauche) et AC (droite)

Les avantages de ce cas sont les suivants.
- Le cas est petit et donc il est rapide.
- C'est un cas avec un bus qui n'a pas d'agent et un bus avec 2 agents ;
- C'est un cas où la puissance voulue est au milieu des bornes des puissances minimales et maximale. Ainsi, une initialisation au milieu des bornes est très efficace.

6.2 Cas pour comparaisons

6.2.1 Cas Matpower

L'installation de MatPower [73] sur Matlab permet d'avoir accès à de nombreux cas d'études. Ces cas d'études sont construits pour réaliser des Power Flow et des Optimal Power Flow avec uniquement les producteurs comme entités flexibles. Les cas d'études ont donc été adaptés pour pouvoir être résolus par nos différents algorithmes. En effet, dans notre cas, on se concentre sur la résolution d'un marché et son impact sur le réseau. Ainsi, on supposera que tous les moyens de production et toutes les charges contrôlables (ou flexibles autour d'un point) imposent les

TABLE 2.4 – Définition des paramètres des agents pour le cas 3 bus

indice	type	bus	$\underline{p_n}$	$\overline{p_n}$	P_0	β	Q_0
0	Pertes	0	0	0	0	0	0
1	Consommateur	1	$1.05P_0$	$0.95P_0$	-2	0	-1.2
2	Producteur	0	0	$2P_0$	1.3	0	0
3	Producteur	1	0	$2P_0$	1	0.7	0

puissances actives et réactives. On ne considère donc que des nœuds dit **PQ** dans notre problème. Pour atteindre ce point tout en restant proches du cas choisi, les étapes suivantes ont été réalisées (le code Matlab réalisant ces opérations est accessible avec les codes open sources).

- La numérotation est changée pour que le nœud de référence soit toujours le numéro 0 (1 dans Matlab) .
- Le cas Matpower est chargé et un Power Flow est lancé tel quel.
- La valeur du déséquilibre (hors perte) est récupérée, et un agent est crée sur le nœud de référence avec pour valeur de consommation/production l'opposée de cette valeur (ainsi le bus de référence ne doit que compenser les pertes comme si une étape du marché avait déjà été réalisée).
- La valeur des puissances finales des nœuds **PV** est récupérée pour initialiser ces nœuds qui sont transformés en nœud **PQ**.
- Les consommateurs (puissance active négative) et les producteurs peu flexibles (puissance active positive ou nulle) sont créés à partir des puissances initiales indiquées sur l'ensemble des nœuds.
- Les producteurs sont définis à partir de la puissance finale simulée et de leur fonction coût lorsqu'elle est renseignée (plus de détails dans le Tab. 2.6).

Parmi les cas tests, le cas IEEE à 39 bus est particulier. Pour ce cas, les fonctions coûts des agents ont été fixées, comme dans [25] et [26]. Ainsi, les agents sont bien plus flexibles que dans les autres cas. Cependant, cela nous permet d'avoir un cas de référence pour nous comparer avec l'état de l'art. De plus, ce cas ne considère que la puissance active des agents.

6.2.2 Cas Européen

Ce cas réaliste représente le marché européen. Pour cela, on utilise les données en open source *DTU-ELMA/European Dataset* [105]. Ce jeu de données inclut un fichier de générateurs flexibles avec leur nom, position, type (charbon, hydraulique, gaz, nucléaire, géothermique), capacités et coût de production. Il inclut aussi un fichier pour la description de chaque bus (tension, position et charge en puissance) à chaque instant avec un pas de temps de 1h pendant 3 ans.

TABLE 2.5 – Définition des paramètres des lignes pour le cas 3 bus

indice	bus avant	bus après	G	B	Y_d	$\overline{\phi}$
0	0	1	1.69	-10.19	0.09205	∞
1	0	2	2.64	-15.93	0.0587	∞
2	1	2	2.11	-12.74	0.07355	∞

TABLE 2.6 – Définition des paramètres, lorsque non fournis par matPower

Agent	flexibilité en P	a_P	b_P	Flexibilité en Q
Consommateur	20%	0.1	$-0.1 * P$	5 %
Producteur peu flexible	20%	0.1	$-0.1 * P$	5 %
Producteur sans coût		0.1	1	

Ce cas contient donc au moins 969 générateurs que l'on supposera totalement contrôlables, et 1494 bus auxquels on associera un agent consommateur par bus. La puissance d'un générateur est limitée par sa capacité. La puissance soutirée de chaque bus sera considérée comme étant celle souhaitée par les agents P_0. Les consommateurs seront supposés comme flexibles à 10% autour de cette valeur.

Pour rappel, la fonction coût des agents a pour expression $g(P_n) = a \cdot P_n^2 + b \cdot P_n$. La fonction coût des consommateurs a son minimum à la puissance voulue ($-P_0$) et a un coefficient directeur de 1. Pour les producteurs, le terme linéaire est égal au coût de production linéaire et le coefficient directeur est de 0.1. Ainsi, il est plus important de satisfaire les consommateurs que les producteurs. Les choix des coefficients sont rappelés dans le tableau, Tab. 2.7.

Le cas possède aussi un fichier permettant d'avoir les informations sur les lignes entre les bus (les bus de liaisons, la longueur, la limite et la susceptance). Le cas contient 2156 lignes, mais 2014 d'entre elles ne sont pas contraintes. Ce qui donne en réalité un cas à 142 lignes contraintes. N'ayant accès qu'à la susceptance des lignes et pas à leur résistance, ce cas n'est pas utilisable pour les problèmes non linéarisés, tels que le **PF** ou l'**OPF** en AC.

Cas Européen réduit Le cas européen contenant les 2500 agents peut être un trop grand cas pour des implémentations non optimisées. Il peut donc être intéressant d'avoir des cas intermédiaires possédant quelques centaines d'agents.

TABLE 2.7 – Définition des paramètres Européen

Agent	Producteur	Consommateur
nombre	969	1494
$\underline{p_n}$ (MW)	0	$-1.1 \cdot P0$
$\overline{p_n}$ (MW)	capacité	$-0.9 \cdot P0$
a (MW^{-2})	0.1	1
b (MW^{-1})	coût de production	P0

Pour cela, on peut réduire le cas européen en ne sélectionnant qu'un seul pays. Selon le pays, le nombre de nœuds et le rapport production - consommation ne sera pas le même. Il peut être important de noter que les lignes avec des contraintes de flux de puissance étant principalement entre les pays, ce genre de cas n'a que quelques lignes contraintes.

6.2.3 Cas Test Feeder

Ce jeu de donnée est en open source, et est fourni par le groupe de travail IEEE PES Test Feeder [106]. Ce système est plutôt basé sur un système du style Nord-Américain. Cependant, il est dans sa configuration de base radiale et à faible tension (416V phase à phase) ce qui est compatible avec un réseau de distribution européen.

En plus de la description du réseau par la description des types de chaque ligne, ce cas d'étude possède des séries temporelles sur une plage de 24h pour les consommations avec un pas par minute. Si le cas possède une centaine de séries, seuls 50 agents sont placés sur le réseau.

Le réseau possède 905 bus et donc 904 lignes. Une analyse des données montre que la plupart des lignes sont extrêmement petites (de l'ordre du décimètre). Cela, couplé au fait qu'il n'y ait que 55 agents, rend le cas très adaptable.

En effet, il est possible d'augmenter le nombre d'agents du cas d'étude pour passer à un cas d'étude plus citadin. Il est aussi possible au contraire de "fusionner" des lignes en ajoutant leur impédance pour réduire la taille du problème tout en simulant le même cas physique.

Ce cas pourra être utilisé pour tous les problèmes comme on a accès à toutes les informations nécessaires du réseau.

6.3 Cas pour évaluer la complexité

Les cas générés aléatoirement n'ont pas pour objectif principal de représenter un cas d'étude réaliste, mais plutôt d'évaluer les caractéristiques algorithmiques des différentes implémentations. Ainsi, on pourra se servir de ces cas pour évaluer la complexité réelle des algorithmes en mesurant le temps en fonction de la taille du cas d'étude. On pourra aussi mesurer la sensibilité des performances à différents paramètres, comme les fonctions coûts, le déséquilibre entre les consommateurs et producteurs ou le fait d'avoir un marché hétérogène.

Dans les cas aléatoires, il y a plusieurs méthodes de générations possibles. Soit tout le cas est généré, soit on part d'un réseau fixe existant et on le modifie. Les parties suivantes présenteront les différentes méthodes de génération des agents, du réseau et du positionnement des agents sur le réseau.

6.3.1 Génération des agents

Dans le cas de génération d'un marché P2P, les différents paramètres sont tirés aléatoirement avec une loi uniforme entre deux bornes pour l'ensemble des agents. On considérera que tous les agents sont économiquement reliés entre eux (ils peuvent tous échanger avec tous les autres). Lors de cette étape, on ne choisit pas sur quel bus se situe l'agent. De nombreux paramétrages sont possibles, dans le tableau Tab. 2.8 sont regroupées les données du paramétrage qui sera utilisé.

Table 2.8 – Caractéristique d'un cas généré aléatoirement, modélisation DC

Caractéristique	moyenne	variation
Power (MW)	1000	400
a (MW^{-2})	0.07	0.02
b (MW^{-1})	50	20
Consomateur (%)	50	0
Consom-acteur (%)	12,5	0

On peut remarquer que le marché qui sera généré avec la méthode ci-dessus sera très homogène. Le comportement des différents types d'agents est identique tout comme leurs niveaux de puissance. Dans un objectif de pouvoir représenter différents types de configuration du réseau et ainsi pouvoir évaluer leur impact potentiel sur les implémentations, d'autres configurations doivent être testées.

6.3.2 Génération du réseau physique

Afin d'étudier l'effet du nombre de lignes contraintes (c'est à dire qui peuvent être congestionnées) sur les performances, un cas d'étude a été généré à partir du cas européen. Pour cela, le réseau du cas européen est créé et à partir de ce réseau, on peut soit enlever les contraintes sur les lignes, soit prendre une ligne non contrainte existante et y ajouter une contrainte thermique. En faisant ainsi, on ne change pas la topologie du réseau, cependant, selon les limites que l'on rajoute, l'optimum peut changer. À partir de ce réseau, on peut conserver les agents du réseau européen original ou utiliser un autre générateur d'agent.

Il est possible de générer un réseau de distribution radial aléatoire. Pour cela, on fixe une profondeur (N_p) et une largeur maximale (N_l) et un nombre de bus (B). On part du bus initial (0). Pour chaque bus que l'on génère, on décide aléatoirement avec une probabilité de $\frac{N_p}{B}$ si l'on continue une branche existante. Sinon, on crée une autre branche. Le choix de créer une nouvelle branche n'est possible que si l'on n'a pas dépassé la largeur visée, et l'on ne peut pas se rajouter sur une branche si celle-ci a atteint la profondeur maximale. Le type de ligne est choisi dans le code

selon une référence accessible sur le site de PandaPower [97], *94-AL1/15-ST1A* ($r = 0.306\Omega/km$, $x = 0.29\Omega/km$, $b = 13.2nF/km$ et $I_{max} = 0.35A$) et la longueur est tirée aléatoirement uniformément entre deux bornes pour chaque ligne.

6.3.3 Génération d'un lien entre réseau et agents

Afin de finir de générer le cas d'étude, il faut déterminer le positionnement des agents. Dans notre cas, on supposera que chaque agent est sur au plus un bus. Pour réaliser la correspondance entre les agents et le réseau, plusieurs choix sont possibles.

- Soit on positionne aléatoirement les agents sur les bus (en tirant pour chaque agent aléatoirement son bus).
- Soit on fait comme précédemment, mais en ne tirant pas uniformément, avec certains bus ou certaines zones plus probables ou forcer pour avoir une répartition équilibrée ou déséquilibrée.
- On peut essayer de positionner de manière "réaliste" les agents, avec, par exemple, les agents d'une communauté sont tous sur le même bus si l'on ne considère que le réseau de transport, ou sont sur le réseau de distribution associé au gestionnaire de communauté si l'on considère le réseau de transport et celui de distribution.

Par défaut, c'est la répartition uniforme qui sera choisie pour construire les cas aléatoires. Les autres répartitions seront utilisées lors des optimisations et des études paramétriques pour les différentes implémentations.

6.4 Synthèse des cas d'études

Comme ce qui a été présenté précédemment, les différents cas d'étude poursuivent différents objectifs. Des cas petits permettent de faire une validation fonctionnelle, tandis que des cas plus grands nous permettent de nous comparer à la littérature. Enfin, les cas aléatoires nous permettent de faire une étude de passage à l'échelle. Certains cas ou générateurs étant basés sur la littérature, ils ne sont pas applicables à tous les types de problèmes. Ainsi, le tableau Tab. 2.9 permet de résumer l'ensemble des cas qui seront utilisés dans l'ensemble de ce manuscrit. Il est important de noter que, la modélisation **DC** étant une approximation de la modélisation **AC** du réseau, tous les cas pouvant être utilisés en modélisation **AC** peuvent aussi être utilisés en **DC**. La génération de cas aléatoire d'agent ne permet en soit que de simuler un marché. Cependant, en la couplant à un réseau (généré aléatoirement ou fixe), la génération peut servir pour étudier l'ensemble des problèmes. Le tableau indique aussi si le jeu de donnée peut être simulé sur plusieurs pas de temps consécutif (dans la colonne donnée temporelle).

Table 2.9 – Ensemble des cas d'études utilisés

Nom	modèle	réseau	Taille	données temporelles
Cas 2 bus	AC	radial	2 | 2	non
Cas 3 bus	AC	maillé	3 | 3	non
Cas 39 nœuds	DC	maillé	39 | 31	non
MatPower casB	AC	dépend	B | dépend	non
Européen	DC	maillé	1494 | 2463	oui
Test Feeder	AC	radial	905 | 50	oui
cas aléatoire	AC		0 | N	non

Table 2.10 – Description des cas Matpower utilisés

Nom	réseau	B	L	N	N_{cons}	$N_{gen,Nflex}$	S_{base}
cas9	maillé	9	9	6	3	0	100
cas10ba	radial	10	9	11	9	1	10
cas30	maillé	30	41	26	20	0	100
cas69	radial	69	68	49	48	0	10
cas85	radial	85	84	60	58	1	1
cas118	maillé	118	186	153	99	0	100

Le tableau 2.10 détaille les cas Matpower qui ont été utilisés dans cette thèse. Pour rappel, B, L et N représentent respectivement le nombre de bus, de lignes et d'agents. Les quantités N_{cons} et $N_{gen,Nflex}$ correspondent aux nombres de consommateurs et de générateurs non flexibles.

Dans le tableau est aussi indiqué S_{base}, qui est la valeur de la puissance qui est utilisée pour normaliser les puissances. Cela est important, notamment, car plus cette valeur est élevée, plus les coefficients des fonctions coûts le seront.

7 Conclusion

Dans cette partie la méthodologie Adéquation Algorithme Architecture a pu être présentée. Les métriques nous permettant d'évaluer les différents algorithmes et implémentations réalisées. Le verrou scientifique étant le temps de calcul, les principales métriques seront le temps de calcul et la complexité, celle-ci indiquant le comportement lors du passage à l'échelle. L'ensemble des algorithmes qui sera utilisé a été présenté. Parmi les architectures, le choix a été le CPU en tant qu'hôte

et le GPU en tant que carte accélératrice. Le code sera écrit en C++ et OpenMP sera employé pour paralléliser sur les CPU, tandis que Cuda sera utilisé pour mettre en œuvre sur les GPU. Enfin, l'ensemble des cas d'études qui seront utilisés pour le test des implémentations ont été détaillés.

Chapitre 3

Validation fonctionnelle et passage à l'échelle

Sommaire

1 Introduction . . . 116
1.1 Problème d'optimisation . . . 116
1.2 Méthodes de résolutions . . . 117
2 Power Flow . . . 118
2.1 Mise en forme du problème . . . 118
2.2 Approximation DC . . . 119
2.3 Newton-Raphson NR . . . 120
2.4 Gauss Seidel GS . . . 121
2.5 Méthode Backward-forward . . . 122
2.6 Validation fonctionnelle . . . 123
3 Marché pair à pair . . . 126
3.1 Mise en forme du problème . . . 126
3.2 Résolution centralisée via OSQP . . . 127
3.3 Décentralisation par ADMM . . . 128
3.4 Décentralisation par PAC . . . 132
3.5 Validation fonctionnelle . . . 137
4 Optimal Power Flow . . . 141
4.1 Mise en forme du problème . . . 141
4.2 Résolution de référence . . . 143
4.3 Déplacement de la contrainte des puissances injectées . . 146
4.4 Validation fonctionnelle . . . 148
5 Marché endogène . . . 149
5.1 Résolution avec relaxation d'un PF . . . 150
5.2 Résolution par consensus avec un OPF . . . 157
5.3 Résolution directe . . . 159
5.4 Validation fonctionnelle . . . 161
6 Conclusion . . . 164

1 Introduction

Le chapitre précédent a pu présenter la méthodologie de l'Adéquation Algorithme Architecture. Les métriques, algorithmes, architectures et jeux de données ont été définis. Ce chapitre vise à appliquer les algorithmes précédents pour résoudre le problème du **marché endogène**. L'implémentation des différents algorithmes en série sur CPU visera différents objectifs. Tout d'abord, cela nous permettra de réaliser la validation fonctionnelle des différents algorithmes. Pour cela, on étudiera leur convergence sur des cas fixes et des cas aléatoires. Ensuite, la simulation sur des cas aléatoires permettra de montrer l'explosion des temps de calcul avec l'augmentation des dimensions des problèmes.

Toutes les implémentations ont été réalisées en C++ et exécutées sur un CPU, AMD RYZEN 5 5600H à une fréquence de 3.3 GHz.

1.1 Problème d'optimisation

Pour rappel, l'objectif de l'optimisation d'un marché endogène est de trouver les échanges t_{nm} optimums entre les agents minimisant les fonctions objectifs notées $g_n = \frac{1}{2}a_n p_n^2 + b_n p_n$ tout en respectant les contraintes de réseau. Ces fonctions peuvent être enrichies par des préférences hétérogènes sur les échanges $\beta_{nm}t_{nm}$, qui peuvent représenter un surcoût imposé par le réseau [25], une pénalité sur la distance entre les agents ou une pénalité carbone par exemple [26]. Dans le cas d'un réseau multiénergie, on pourra aussi rajouter une ou plusieurs autres grandeurs échangées, qui seront notées q. Cette grandeur pourra représenter, par exemple, la puissance réactive ou la puissance thermique. L'équation résultante est la suivante :

$$\min_{\mathbf{T},\mathbf{P}} \quad \sum_{n\in\Omega} \Big(g_n(p_n, q_n) + \sum_{m\in\omega_n} \beta_{nm} t_{nm} \Big) \tag{3.1a}$$

$$\text{t.q. } \mathbf{T} = -\mathbf{T}^t \tag{3.1b}$$

$$\textbf{Lois de la physique} \tag{3.1c}$$

$$\textbf{contraintes operationnelles} \tag{3.1d}$$

La contrainte d'antisymétrie ($\mathbf{T} = -{}^t\mathbf{T}$) représente le fait que le marché doit être à l'équilibre (ce qui est acheté doit être vendu et inversement). Les contraintes opérationnelles concernent à la fois le marché (limites physiques des agents, Section 3.3) et le réseau (plan de tension et congestion, Section 3.4). Les lois de la physique consistent dans le respect des lois de Kirchhoff (lois des mailles et lois des nœuds 3.2).

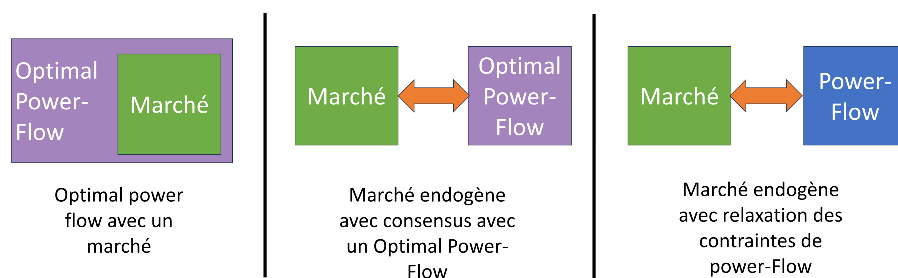


FIGURE 3.1 – Différentes manières de résoudre le marché endogène

Un marché endogène centralisé sans préférences hétérogènes (sans les variables d'échange) est un Optimal Power Flow.

1.2 Méthodes de résolutions

Le problème de marché décentralisé endogène est un problème complexe, mais qui peut être résolu directement sous la forme (3.1). En effet, sa formulation est similaire à celle d'un OPF avec des contraintes supplémentaires pour le marché et potentiellement des termes en plus dans la fonction coût. Ainsi, la première manière de résoudre ce problème serait donc de résoudre un OPF dans lequel on ajoute un marché interne. Cette manière est la plus directe pour résoudre ce problème, mais il est difficile de savoir à priori comment s'organiserait l'implémentation réelle de ce type de gestion. Notamment selon l'algorithme utilisé, les échanges d'informations nécessaires ne seront pas les mêmes. Cette résolution est représentée dans la partie gauche de la Fig. 3.1.

Une autre manière de faire qui serait plus proche du réel est de séparer la gestion du réseau et la résolution du marché. La séparation en deux blocs de la résolution permet aussi de séparer les informations nécessaires à chaque bloc et ainsi d'éviter d'avoir besoin que tous les agents du marché aient connaissance du réseau. On peut donc soit relaxer les contraintes du réseau via une ADMM (Section 2.4.2.1). Ainsi, le marché doit communiquer avec un Power Flow pour respecter les contraintes (partie droite de la Fig. 3.1). Cette méthode a l'avantage de limiter la puissance de calcul nécessaire pour le gestionnaire de réseau (SO).

L'autre possibilité est de séparer le problème en deux sous problèmes avec une autre version de l'ADMM (Section 2.4.1.1). Dans ce dernier cas, un marché et un OPF doivent communiquer pour arriver à un consensus (partie centrale de la Fig. 3.1).

Un des enjeux de cette thèse sera de déterminer lequel de ces trois types de résolution a les meilleures performances selon nos métriques. De plus, il sera possible de se demander comment faire interagir les blocs entre eux. En effet, de nombreux choix sont possibles, par exemple :

— chaque bloc fait son optimisation complète à tour de rôle ;
— chaque bloc fait un petit nombre d'itérations (éventuellement une seule) à tour de rôle ;
— un bloc fait une optimisation complète et l'autre ne fait que quelques itérations ;
— n'importe quelle combinaison de nombres d'itérations peut aussi dépendre de l'état de la convergence...

Cette question sera discutée dans la section 5.4.2.3.

Comme de nombreuses solutions sont possibles, il est important de faire une sélection des algorithmes pour chaque bloc (sous problème). Pour ce faire, les différents blocs seront implémentés séparément et on suivra la démarche d'Adéquation Algorithme Architecture pour chacun de ces problèmes. Ainsi, les parties suivantes présenteront les différentes implémentations des différents problèmes sur CPU pour réaliser la validation fonctionnelle.

2 Power Flow

2.1 Mise en forme du problème

Les contraintes du réseau que l'on considère dans le marché endogène sont des bornes sur les variables physiques du réseau (tension et puissance) et le respect des lois de la physique (lois de Kirchhoff). Le problème du Power-Flow permet de déterminer la moitié des grandeurs caractéristiques (tension V, θ et puissance P, Q) sur les bus à partir de la connaissance de l'autre moitié [42]- [50]. Parmi les nœuds, un est choisi pour être la référence de la tension $V = V_0$ et $\theta = \theta_0$, et sera dans notre cas toujours le premier nœud. Les autres nœuds sont des nœuds PQ si l'on connait les puissances ou PV si l'on connait l'amplitude de la tension et la puissance active. Les équations de Kirchhoff nous donnent la relation suivante :

$$\underline{S}_i^{bus} = \underline{E}_i \cdot \sum_k \underline{Y}_{ik}^* \cdot \underline{E}_k^* = F(\theta, V) \tag{3.2}$$

avec $\underline{E}_i$ la tension complexe du bus i. Ce qui donne en termes de puissances actives et réactives en exprimant la tension sous sa forme angulaire :

$$P_i^{bus} = V_i \cdot \sum_k V_k \cdot (G_{ik} \cos\theta_{ik} + B_{ik} \cdot \sin\theta_{ik}) \tag{3.3}$$

$$Q_i^{bus} = V_i \cdot \sum_k V_k (G_{ik} \sin\theta_{ik} - B_{ik} \cos\theta_{ik}) \tag{3.4}$$

Les algorithmes considérés sont les méthodes de Newton Raphson (**NR**) et Gauss Seidel (**GS**). Ils ont été choisis, car ils ont des comportements très diffé-

rents sur une architecture basée sur un CPU [43] tout en ayant des performances comparables sur des cas dans la littérature. Pour rappel, Newton Raphson a des opérations très coûteuses à cause du calcul de la Jacobienne et de la résolution du système, mais n'a besoin que de très peu d'itération (section 4.2.2). L'algorithme de Gauss Seidel (section 4.2.3) a besoin de beaucoup d'itérations (et ce nombre pourrait augmenter avec la taille du problème), mais ses opérations sont peu coûteuses, ce qui le rend aussi rapide que l'autre algorithme. Pour un réseau entièrement radial, l'algorithme de résolution des flux, le *backward-forward algorithm* [73], présente de meilleures propriétés de convergence et sera donc utilisé.

Comme le rôle du Power-Flow développé sera de faire partie d'un marché endogène, on se placera pour la suite dans un cas où un marché est utilisé en parallèle de cette résolution. C'est-à-dire que :

- l'ensemble des nœuds est de type PQ (puisque c'est ce qui est contrôlé dans le marché) ;
- les problèmes traités seront avec des puissances soutirées ou injectées déjà proche de l'équilibre avec le nœud de référence cherchant juste à compenser les pertes dans les lignes.

2.2 Approximation DC

Pour rappel (section 1.3.2), l'approximation DC [45] est une version simplifiée de ce problème qui peut donner des résultats assez proches du problème complet lorsqu'elle est appliquée sur certains réseaux de transport.

Cette approximation négligeant les chutes de tension et considérant un faible déphasage, on remarque que cela signifie que, sur les contraintes de départ, seule la contrainte sur les flux de puissance peut encore être activée. On va donc chercher à déterminer les flux de puissance dans les lignes (ϕ) à partir des puissances injectées dans les bus.

De par ces approximations les équations de départ, (3.3) et (3.4) deviennent :

$$P_i^{bus} = V_0^2 \cdot \sum_k B_{ik} \cdot \theta_{ik} = \sum_{l \in \mathcal{L}_i} \phi_l \tag{3.5}$$

$$Q_i^{bus} = -V_0^2 * \sum_k B_{ik} \tag{3.6}$$

Ainsi en notant $\mathbf{B}_{diag}$ une matrice diagonale contenant la susceptance de chaque ligne, $B_{diag,l} = 1/x_l$, et C la matrice d'incidences entre les bus et les lignes, on peut remarquer que l'on a les relations suivantes :

$$\mathbf{P}^{bus} = \mathbf{C}\mathbf{B}_{diag}\mathbf{C}^T\theta \tag{3.7}$$

$$\phi = \mathbf{B}_{diag}\mathbf{C}^T\theta \tag{3.8}$$

Ces deux relations (qui sont en réalité deux systèmes d'équations linéaires) nous permettent de chercher une relation directe entre la puissance dans les lignes et les puissances injectées dans les bus. L'idée de base est de résoudre le premier système pour obtenir les valeurs de θ pour ensuite l'injecter dans le deuxième système pour obtenir ϕ. Cependant, avant de pouvoir faire cela, il est important de noter qu'en cas de Power Flow, l'angle θ est imposé sur le nœud de référence. De plus, en approximation DC, il n'y a pas de perte dans les lignes. Ainsi, la puissance inconnue dans le nœud de référence est exactement égale au déséquilibre. On a donc une équation de plus que le nombre de variables.

$$\theta = (\mathbf{C}\mathbf{B}_{diag}\mathbf{C}^T)^{-1}\mathbf{P}^{bus} \tag{3.9}$$

il faut enlever la ligne correspondant au nœud de référence (la première ligne dans notre cas) dans la matrice $CB_{diag}C^T$. Cela aura pour conséquence que lorsque l'on écrit la relation suivante :

$$\phi = \mathbf{B}_{diag} \cdot \mathbf{C}^T \cdot (\mathbf{C} \cdot \mathbf{B}_{diag} \cdot \mathbf{C}^T)^{-1} \cdot \mathbf{P}^{bus} = \mathbf{G}^{bus}_{sensi}\mathbf{P}^{bus} \tag{3.10}$$

Quel que soit le terme de ϕ choisi, celui-ci sera indépendant de la puissance au nœud de référence. On appellera la matrice $\mathbf{G}^{bus}_{sensi}$ la matrice de sensibilité du réseau à l'injection de puissance. Le coefficient g_{lb} de cette matrice représente la variation de ϕ_l due à une variation de P_b.

L'ensemble des nœuds étant de type PQ, sauf le nœud de référence, les inconnues sont donc la puissance sur le nœud de référence et la tension sur tous les autres nœuds. Dans le cas **DC**, la puissance au nœud de référence est égale à la somme de toutes les puissances injectées/soutirées aux nœuds et la tension vaut $1pu$ sur chaque nœud. Ainsi, l'application de l'équation (3.10) permet de résoudre complètement le DC-PF.

2.3 Newton-Raphson NR

Cet algorithme permet en connaissant l'image d'une fonction d'en trouver l'antécédent. Ici, on cherchera $\mathbf{E} = (\theta, \mathbf{V})$ tel que $S_i - F_i(\theta, \mathbf{V}) = 0$ pour tous les bus, et telle que S_i est la puissance imposée par le bus i.

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

$$\mathbf{dW} = \mathbf{Jac} \cdot \mathbf{dE} \tag{3.11}$$

$$\mathbf{Jac} = \begin{pmatrix} \mathbf{Jac}_{1-1} & \mathbf{Jac}_{1-2} \\ \mathbf{Jac}_{2-1} & \mathbf{Jac}_{2-2} \end{pmatrix} = \begin{pmatrix} \frac{\delta \mathbf{P}}{\delta \theta} & \frac{\delta \mathbf{Q}}{\delta \theta} \\ \frac{\delta \mathbf{P}}{\delta \mathbf{V}} & \frac{\delta \mathbf{Q}}{\delta \mathbf{V}} \end{pmatrix} \tag{3.12}$$

La Jacobienne (3.12) peut être définie par bloc. Chaque ligne d'un bloc correspond à un bus. Les termes diagonaux correspondent donc à la dérivée de l'équation d'un bus par ses variables. Tandis que les termes non diagonaux correspondent à la dérivée par les variables des autres bus. L'ensemble des termes est regroupé dans le tableau Tab. 3.1. Il faut ensuite inverser le système défini par cette matrice, (3.11) pour obtenir la variation de tension nous permettant de nous rapprocher de la solution. Deux méthodes directes ont été testées, une inversion de matrice par un pivot de Gauss et une factorisation LU avec pivot. Il est intéressant de noter que, pour chaque bloc de la Jacobienne, le terme Jac_{ij} est non nul s'il existe une ligne entre les bus i et j, Tab. 3.1. Ainsi, moins le réseau est maillé, plus la Jacobienne sera creuse.

TABLE 3.1 – Différents termes de la Jacobienne

Indices	diagonaux	non diagonaux
1-1	$-Q_i - B_{ii} \cdot V_i \cdot V_i$	$V_i \cdot V_j \cdot (G_{ij} \cdot sin(d\theta) - B_{ij} \cdot cos(d\theta))$
1-2	$P_i/V_i + G_{ii} \cdot V_i$	$V_i \cdot (G_{ij} \cdot cos(d\theta) + B_{ij} \cdot sin(d\theta))$
2-1	$P_i - G_{ii} \cdot V_i \cdot V_i$	$-V_i \cdot V_j \cdot (G_{ij} \cdot cos(d\theta) + B_{ij} \cdot sin(d\theta))$
2-2	$Q_i/V_i - B_{ii} \cdot V_i$	$V_i \cdot (G_{ij} \cdot sin(d\theta) - B_{ij} \cdot cos(d\theta))$

La résolution est itérative. Afin de déterminer si l'algorithme a convergé, l'erreur peut être calculée :

$$err = ||W0 - W|| \tag{3.13}$$

L'erreur est la différence entre les puissances injectées connues notées **W0** (actives des nœuds **PV** ou puissances actives et réactives des nœuds **PQ**) avec les puissances obtenues à partir des tensions calculées.

2.4 Gauss Seidel GS

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

$$\begin{aligned} \underline{S}_i^* &= P_i - i * Q_i \\ &= \underline{E}_i^* \cdot \sum_j \underline{Y}_{ij} \cdot \underline{E}_j \end{aligned} \tag{3.14}$$

On aboutit alors à l'équation suivante :

$$\frac{P_i - i * Q_i}{\underline{E}_i^*} = \sum_j \underline{Y}_{ij} \cdot \underline{E}_j \tag{3.15}$$

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 :

$$\underline{E}_i^{k+1} = \frac{1}{\underline{Y}_{ii}} \cdot \Big(\frac{P_i - iQ_i}{\underline{E}_i^*} - \sum_{l=1}^{i-1} \underline{Y}_{il}\underline{E}_l^{k+1} - \sum_{l=i+1}^{B} \underline{Y}_{il}\underline{E}_l^{k}\Big) \tag{3.16}$$

Pour cet algorithme, la tension sera représentée en coordonnées cartésiennes, ce qui permettra d'éviter l'utilisation de fonction trigonométrique. Ainsi en séparant la partie réelle et imaginaire de cette dernière équation, la tension peut être itérativement déterminée. Le calcul de l'erreur est identique à celui de la méthode de Newton-Rapshon (3.13).

2.5 Méthode Backward-forward

Les algorithmes précédents ont des limites connues. En effet **NR** converge difficilement dans les cas mal initialisés ou très peu maillés. L'algorithme **GS** a des problèmes pour passer à l'échelle. Pour remplacer ces algorithmes dans des réseaux de distribution où ils sont peu efficaces, d'autres algorithmes ont été proposés.

Ces algorithmes ne sont utilisables que dans les réseaux de distribution non maillés. Certaines extensions permettent de prendre en compte quelques boucles, mais elles ne seront pas implémentées ici. Ces méthodes sont basées sur celles présentées dans le document de Matpower [73]. L'algorithme backward-forward existe sous plusieurs formes, selon qu'il utilise le courant, le flux de puissance ou des impédances équivalentes. Dans notre cas, on considérera la version utilisant le courant (Current Sommation Method **Cur**) et celle sur les puissances (Power Summation Method **BackPQ**). Ces méthodes s'appliquant sur un réseau radial, il y a un bus de plus qu'il y a de ligne. La numérotation entre ce document et Matlab sera différente. En effet, si, dans Matlab, la ligne k va vers le bus k, dans notre cas, la ligne k arrive au bus $k+1$. Ceci nous permet d'éviter d'avoir besoin d'ajouter une ligne fictive arrivant au premier bus. On stocke aussi pour chaque bus k l'antécédent du bus dans un vecteur **F** tel que $F_k = i = A_k$.

Ces méthodes consistent à répéter les 4 étapes suivantes :

1 On calcule le courant/ la puissance pour chaque ligne k en fonction de la puissance soutirée s_d :

$$\underline{j}_b^k = (\frac{\underline{s}_d^{k+1}}{\underline{E}_{k+1}})^* + \underline{y}_d^k \underline{E}_{k+1} \tag{3.17}$$

$$\underline{s}_t^k = \underline{s}_d^{k+1} + (\underline{y}_d^k)^* \cdot v_{k+1}^2 \tag{3.18}$$

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

$$\underline{j}^i_{b,new} = \underline{j}^i_b + \underline{j}^k_b \tag{3.19}$$

$$\begin{aligned} \underline{s}^k_f &= \underline{s}^k_t + \underline{z}^k_s |\frac{\underline{s}^k_t}{\underline{E}_k}|^2 \\ \underline{s}^i_{t,new} &= \underline{s}^i_t + \underline{s}^k_f \end{aligned} \tag{3.20}$$

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

$$\underline{E}_k = \underline{E}_i - \underline{z}^k_s \cdot \underline{j}^k_b \tag{3.21}$$

$$\underline{E}_k = \underline{E}_i - \underline{z}^k_s \cdot (\frac{\underline{s}^k_f}{\underline{v}_i})^* \tag{3.22}$$

4 on calcule l'erreur sur la tension :

$$\epsilon = ||\mathbf{E}^{iter+1} - \mathbf{E}^{iter}|| \tag{3.23}$$

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.

2.6 Validation fonctionnelle

Dans cette partie, on testera nos algorithmes sur les cas de Matpower qui sont utilisés dans la littérature.

Pour rappel, les cas ont été modifiés pour n'avoir que des nœuds PQ. Cela peut provoquer des différences sur les propriétés de convergences des méthodes sur certains cas par rapport à la littérature.

La précision demandée est de $5 \cdot 10^{-4}$ pour les méthodes **NR** et **GS**. Pour les méthodes **Cur** et **BackPQ** la précision est de $5 \cdot 10^{-5}$. En effet comme les dernières méthodes calculent l'erreur sur la tension, il faut demander sur certains cas plus de précision pour obtenir les mêmes puissances.

Par défaut, les cas sont initialisés avec les valeurs contenues dans les fichiers des cas MatPower. Dans le tableau Tab. 3.2 sont regroupés l'ensemble des résultats. Le symbole $*$ signifie que le cas diverge avec ce type d'initialisation. Le temps n'est dans ce cas là pas indiqué puisque dépendant du nombre d'itérations maximales autorisées. Le symbole X indique que la méthode n'a pas été utilisée puisque

TABLE 3.2 – Temps en ms et (nombre d'itérations) pour converger pour différents algorithmes et cas

Cas (précision)	NR simple	NR double	GS simple	GS double	Cur simple	BackPQ simple
Cas 2 bus	0.06 (2)	0.03 (2)	0.04 (7)	0.4 (7)	0.04 (3)	0.03 (2)
Cas 3 bus	0.09 (2)	0.03 (2)	0.06 (7)	0.5 (7)	X	X
Matpower 9	0.09 (3)	0.08 (3)	0.2 (74)	0.9 (74)	X	X
Matpower 10ba	0.1 (3)	0.05 (3)	0.3 (94)	0.6 (94)	0.09 (6)	0.04 (4)
Matpower 30	0.3 (2)	0.2 (2)	1.5 (197)	2 (197)	X	X
Matpower 69	*	10 (2)	*	790 (12015)	0.2 (5)	0.2 (3)
Matpower 85	38 (3)	38 (3)	100 (4487)	88 (4742)	0.3 (6)	0.5 (4)
Matpower 118	75 (2)	72 (2)	*	*	X	X
Matpower 300	830 (9)	740 (8)	*	*	X	X

le réseau n'est pas radial. Puisque le résultat peut dépendre de la précision des calculs, les méthodes **NR** et **GS** ont aussi été testées avec tous les calculs en double précision.

On peut remarquer qu'il existe des cas où les méthodes ne convergent pas. Pour le cas Matpower 69, ceci est dû à un problème de précision dans les calculs. En effet, les mêmes calculs en flottant double précision convergent.

Le problème étant non convexe le choix de l'initialisation est très important. Par exemple pour le cas 118 nœuds si l'initialisation est proche de la solution ou avec des tensions plates, la méthode **GS** converge vers une autre solution (avec plus de puissance active et réactive sur le nœud de référence) que la méthode de **NR** (que cela soit résolu via notre solveur ou celui de MatPower).

2.6.1 Passage à l'échelle

L'objectif de cette partie est d'étudier le comportement des différents algorithmes avec l'augmentation des dimensions des problèmes. Afin de pouvoir comparer l'ensemble des algorithmes, des réseaux radiaux seront générés. Les lignes sont de type *94-AL1/15-ST1A 0.4*. Les agents sont répartis aléatoirement sur l'ensemble des bus. Dans notre cas, il y a autant de consommateurs que de producteurs et leurs puissances sont égales en moyennes. Le réseau est généré de telle sorte qu'il soit globalement deux fois plus profond que large (*i.e.* lors d'un placement d'un bus, il a deux fois plus de chance d'être mis en bout de branche que de créer une

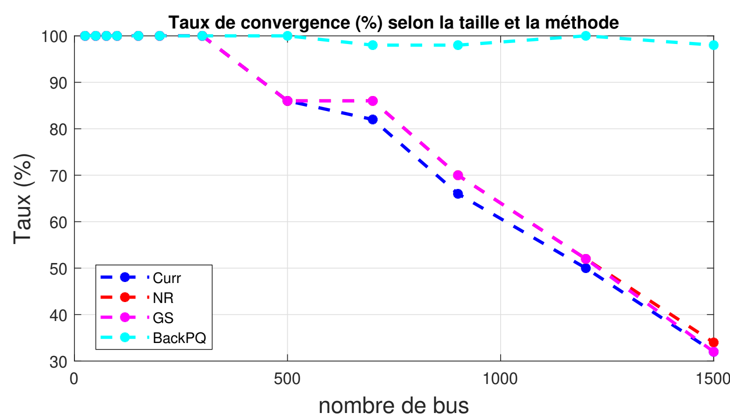


FIGURE 3.2 – Taux de convergence des différents algorithmes

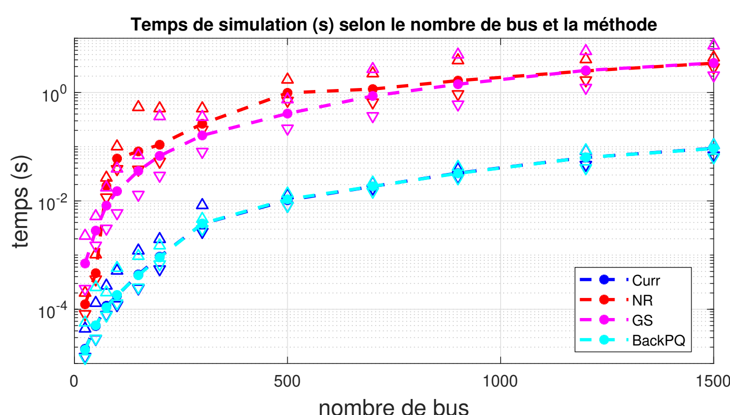


FIGURE 3.3 – Temps de calcul mesuré des différents algorithmes

nouvelle branche). L'influence du réseau sera étudiée en section 4.1.1 du Chapitre 5.

Le taux de convergence des différents algorithmes en fonction des dimensions du problème est représenté sur la Fig. 3.2. On peut voir que celui-ci chute avec l'augmentation des dimensions du problème pour toutes les méthodes, sauf pour **BackPQ**. Cela montre la supériorité de cette méthode dans le cas de réseau radial dans ce type de configuration. Lorsque l'algorithme ne converge pas, son temps de calcul dépend en grande partie du nombre maximal d'itérations permises. Ainsi, la Fig. 3.3 représente le temps de simulation selon la méthode et les dimensions pour les cas ayant convergé.

On peut y voir une augmentation rapide des temps de calcul avec l'augmentation des dimensions du problème. Les temps équivalents entre **NR** et **GS** cachent une différence majeure entre ces méthodes. En effet, le nombre moyen d'itérations

du **NR** varie de 2 à 4 avec l'augmentation des dimensions, alors que la méthode de **GS** en nécessite en moyenne de 140 à 10 000. Même si les temps de calcul augmentent rapidement pour toutes les méthodes, les méthodes **Curr** et **BackPQ** restent bien plus rapides.

3 Marché pair à pair

3.1 Mise en forme du problème

On appelle le marché pair à pair la résolution du problème suivant :

$$\min_{T,P} \quad \sum_{n\in\Omega}\Big(g_n(p_n) + \sum_{m\in\omega_n}\beta_{nm}t_{nm}\Big) \tag{3.24a}$$

$$\text{t.q.} \quad \mathbf{T} = -{}^t\mathbf{T} \qquad (\Lambda) \tag{3.24b}$$

$$p_n = \sum_{m\in\omega_n} t_{nm} \qquad (\mu) \qquad n\in\Omega \tag{3.24c}$$

$$\underline{p_n} \leq p_n \leq \overline{p_n} \qquad n\in\Omega \tag{3.24d}$$

$$t_{nm} \leq 0 \qquad n\in\Omega_c \tag{3.24e}$$

$$t_{nm} \geq 0 \qquad n\in\Omega_g \tag{3.24f}$$

$$\underline{p_n} \leq t_{nm} \leq \overline{p_n} \qquad n\in\Omega_p \tag{3.24g}$$

L'objectif de cette simulation est de trouver les échanges t_{nm} optimaux entre les agents qui minimisent les fonctions objectifs. Dans le cadre de cette thèse, on supposera que les fonctions coûts sont quadratiques $g_n(p_n) = \frac{1}{2}a_n p_n^2 + b_n p_n$. À ces fonctions peuvent être ajoutées des préférences hétérogènes sur les échanges $\beta_{nm}t_{nm}$. Les contraintes sont l'antisymétrie des échanges (3.24b), car ce qui est acheté doit être vendu pour être à l'équilibre, la puissance de chaque agent est la somme de ses échanges (3.24c), la puissance et les échanges sont bornés (3.24e)-(3.24g) selon le type des agents.

Ce type de problème est une optimisation quadratique sous contraintes. Ainsi, il peut être résolu de manière centralisée via OSQP (section 2.4.3.1). Cette méthode sera désignée par l'acronyme **OSQPCen**. Comme il sera montré dans cette partie, une résolution avec un algorithme centralisée d'un marché pair à pair ne passe pas à l'échelle. On utilisera donc deux algorithmes de décentralisation, **ADMM** (section 2.4.1.2), et **PAC** (section 2.4.1.4) afin de pouvoir les comparer. Lors de l'utilisation de l'ADMM, il faut encore résoudre un problème de minimisation pour chaque agent. Il y a plusieurs méthodes pour le résoudre. Les algorithmes décentralisés avec une ADMM sont désignés sous les termes **OSQP** et **ADMM**, selon qu'ils sont résolus localement par une autre ADMM ou par OSQP. La méthode décentralisée, avec l'algorithme PAC, s'appellera **PAC**.

3.2 Résolution centralisée via OSQP

L'équation (3.24) lorsque les fonctions coûts sont quadratiques est une minimisation d'une fonction quadratique, donc strictement convexe pour les puissances $\mathbf{P}$. Pour les variables d'échange $\mathbf{T}$, la fonction coût est linéaire donc convexe. Le problème peut s'écrire ainsi :

$$\begin{aligned} \mathbf{x} = \operatorname{argmin} \frac{1}{2}\mathbf{x}^T\mathbf{H}\mathbf{x} + \mathbf{q}^T\mathbf{x} \\ \text{t.q.} \quad \mathbf{l} \leq \mathbf{A}\mathbf{x} \leq \mathbf{u} \end{aligned} \tag{3.25}$$

C'est que l'on appellera une résolution centralisée. Deux choix sont possibles ici, la variable p_n et la contrainte (3.24c) sont conservées, soit on applique cette contrainte pour substituer les puissances par les échanges dans les fonctions coûts.

Dans le premier cas, on pose $\mathbf{x} = (\mathbf{P}, \mathbf{T}_{lin})$ de taille $N_v = N^2 + N$ avec $\mathbf{T}_{lin}$ la linéarisation ligne par ligne de la matrice des échanges $\mathbf{T}$. La matrice $\mathbf{H}$ est nulle sauf sur les N premiers termes de la diagonale correspondante aux termes de la puissance :

$$H_{nn} = a_n \qquad ssi\ n \leq N \tag{3.26a}$$
$$H_{ij} = 0 \qquad \text{sinon} \tag{3.26b}$$

De même, on peut définir le vecteur $\mathbf{q}^T$ ainsi (en considérant que l'indice i est défini telle que l'on ait $x_i = t_{nm}$) :

$$q_n = b_n \qquad ssi\ n \leq N \tag{3.27a}$$
$$q_i = \beta_{nm} \qquad \text{sinon} \tag{3.27b}$$

Enfin on peut définir les $N_c = N^2 + 2 \cdot N = N_v + N$ contraintes ainsi :

$$l_n = \underline{p_n} \qquad ssi\ n \leq N \tag{3.28a}$$
$$l_n = 0 \qquad ssi\ N_v \leq n < N_c \tag{3.28b}$$
$$l_i = lb_n \qquad \text{sinon} \tag{3.28c}$$
$$u_n = \overline{p_n} \qquad ssi\ n \leq N \tag{3.28d}$$
$$u_n = 0 \qquad ssi\ N_v \leq n < N_c \tag{3.28e}$$
$$u_i = ub_n \qquad \text{sinon} \tag{3.28f}$$

Et on peut définir la matrice des contraintes par blocs avec respectivement $\mathbf{I}_N$ et $\mathbf{1}_N$ les matrices identités et les matrices ne contenant que des 1 de taille $N \cdot N$:

$$A_{ij} = I_{N_v} \qquad ssi\ (i,j) \leq N_v \tag{3.29a}$$
$$A_{ij} = I_N \qquad ssi\ i \geq N_v\ j \leq N \tag{3.29b}$$
$$A_{ij} = 1_{N \cdot N^2} \qquad ssi\ (i,j) \geq (N, N_v) \tag{3.29c}$$
$$A_{ij} = 0 \qquad \text{sinon} \tag{3.29d}$$

Dans le deuxième cas on a $\mathbf{x} = \mathbf{T}_{lin}$ $(N_v = N^2)$, ce qui donne $\mathbf{q}^T = b_n + \beta_{nm}$, $H_{ij} = a_n$ tel que $n = i/N$ (ou j). Pour les contraintes on a la même forme (avec juste N_v qui ne vaut plus la même valeur), et il n'y a pas besoin d'exprimer le fait que la puissance est égale à la somme des échanges.

L'ensemble des valeurs prises par les vecteurs et matrice est rappelé dans les 2 premières lignes du tableau Tab. 3.3.

TABLE 3.3 – Valeur des matrices et vecteur par blocs

Cas	taille	x	H	q	l	A		u
cen 1	N	P_n	$a_n \cdot I_N$	b_n	$\underline{p_n}$	I_N	0	$\overline{p_n}$
	N^2	T_{lin}	0	β_{lin}	lb_n	0	I_{N^2}	ub_n
	N	$\times$	$\times$	$\times$	0	$-I_N$	$1_{N \cdot N^2}$	0
cen 2	N^2	T_{lin}	$a_{n=(i,j)/N}$	$b_n + \beta_{lin}$	lb_n	I_{N^2}	$\times$	ub_n
	N	$\times$	$\times$	$\times$	$\underline{p_n}$	1_{N^2}	$\times$	$\overline{p_n}$
décen 1	N	T_n	$\rho/2 \cdot I_N$	$\beta_{nm} + 2 \cdot (\lambda_{nm} - z_{znm})$	lb_n	I_N	0	ub_n
	1	p_n	a_n	b_n	$\underline{p_n}$	0	1	$\overline{p_n}$
	1	$\times$	$\times$	$\times$	0	$1_{1 \cdot N}$	-1	0
décen 2	N	T_n	$\rho/2 \cdot I_N$ $+a_n 1_N$	$\beta_{nm} + 2 \cdot (\lambda_{nm} - z_{znm})$ $+b_n \cdot 1_{N \cdot 1}$	lb_n	I_{N^2}	$\times$	ub_n
	1	$\times$	$\times$	$\times$	$\underline{p_n}$	$1_{1 \cdot N}$	$\times$	$\overline{p_n}$

3.3 Décentralisation par ADMM

On peut remarquer que le problème d'optimisation (3.24), en posant $f_n(\mathbf{T}, \mathbf{P}) = g_n(p_n) + \sum_{m \in \omega_n} \beta_{nm} t_{nm}$, peut s'écrire sous la forme suivante :

$$\begin{aligned} \mathbf{T}_{min} = \underset{\mathbf{T},\mathbf{P}}{\operatorname{argmin}}\, f(\mathbf{T}, \mathbf{P}) &= \underset{\mathbf{T},\mathbf{P}}{\operatorname{argmin}} \sum_n f_n(\mathbf{T}, \mathbf{P}) \\ \text{t.q.} \quad (\mathbf{T}, \mathbf{P}) &\in \mathbf{C} \end{aligned} \tag{3.30}$$

avec $\mathbf{T}$ représentant l'ensemble des échanges que peuvent réaliser tous les agents et $\mathbf{C}$ l'espace convexe regroupant l'ensemble des contraintes (3.24c)-(3.24g). Afin de permettre la décentralisation, la contrainte (3.24b) est modifiée afin de permettre la définition de la variable *slack*, z et la variable duale λ associé :

$$\mathbf{T} - \mathbf{z} = 0|_{\boldsymbol{\Lambda}} \tag{3.31}$$

Afin de pouvoir utiliser un ADMM sous forme de consensus, on va considérer $\mathbf{z}$ comme l'ensemble des vecteurs des échanges. D'un autre côté, on va considérer

que $\mathbf{T}_n$ contient une copie des échanges de l'agent n et des vis-à-vis. La fonction coût ne dépend plus que de variables locales $f_n(\mathbf{T}_n, p_n)$.

Ainsi en utilisant la méthode des multiplieurs à direction alternée (ADMM), [90] sous la forme de consensus, (3.24) et (3.24b) peuvent être ré-écrites pour chaque agent n, et pour différentes itérations k avec ρ le facteur de pénalité :

$$\begin{aligned} \mathbf{T}_n^{k+1} = \underset{\mathbf{T}_n}{\operatorname{argmin}} \quad & g_n(p_n) + \sum_{m\in\omega_n} \gamma_{nm} t_{nm} \\ & + \sum_{m\in\omega_n} \Big(\frac{\rho}{2}(t_{nm} - \frac{t_{nm}^k - t_{mn}^k}{2} + \frac{\lambda_{nm}^k}{\rho})^2 \Big) \\ \text{t.q.} \quad & t_{nm} \in C' \end{aligned} \tag{3.32}$$

$\mathbf{T}_n$ étant le vecteur contenant les offres d'échange de l'agent n et C' les mêmes contraintes excepté celle de symétrie. La démonstration peut être trouvée en annexe.

Pour toute la suite du document la minimisation de (3.32) sera appelé problème local de chaque agent.

À chaque étape, les multiplicateurs de Lagrange sont mis à jour pour assurer l'antisymétrie des échanges.

$$\lambda_{nm}^{k+1} = \lambda_{nm}^k + \frac{\rho}{2}(t_{nm}^{k+1} + t_{mn}^{k+1}) \tag{3.33}$$

Les résidus primaux (associé à la symétrie) et les résidus duaux (variation entre les itérations) sont calculés comme suit :

$$\begin{aligned} r_n^{k+1} &= \sum_{m\in\omega_n} (t_{nm}^{k+1} + t_{mn}^{k+1})^2 \\ s_n^{k+1} &= \sum_{m\in\omega_n} (t_{nm}^{k+1} - t_{nm}^k)^2 \end{aligned} \tag{3.34}$$

3.3.1 Résolution du problème local par OSQP

On peut remarquer que comme la fonction coût est quadratique, la fonction à minimiser (3.32) est une fonction quadratique. On peut donc résoudre la minimisation du problème local avec OSQP. La formulation est très proche de celle en centralisé mis à part le fait que la contrainte d'équilibre est relaxée, ce qui rajoute des termes dans H et q mais réduit le nombre de contraintes. Grâce à la relaxation de la contrainte d'anti-symétrie le problème est strictement convexe pour P et pour T. Il y a toujours le choix de laisser ou non la puissance totale en tant

que variable à optimiser, les équations lorsque l'on garde la puissance en tant que variable sont les suivantes :

$$H_{N+1,N+1} = a_n \tag{3.35a}$$
$$H_{ii} = \frac{\rho}{2} \qquad ssi\ i \leq N \tag{3.35b}$$
$$H_{ij} = 0 \qquad \text{sinon} \tag{3.35c}$$
$$q_{N+1} = b_n \tag{3.35d}$$
$$q_m = \beta_{nm} + 2 \cdot (\lambda_{nm} - z_{nm}) \qquad \text{sinon} \tag{3.35e}$$

Enfin on peut définir les $N_c = N + 2 = N_v + 1$ contraintes ainsi :

$$l_{N+1} = \underline{p_n} \tag{3.36a}$$
$$l_{N+2} = 0 \tag{3.36b}$$
$$l_m = lb_n \qquad \text{sinon} \tag{3.36c}$$
$$u_{N+1} = \overline{p_n} \tag{3.36d}$$
$$u_{N+2} = 0 \tag{3.36e}$$
$$u_m = ub_n \qquad \text{sinon} \tag{3.36f}$$

L'ensemble des valeurs des différents termes de l'équation présentés précédemment et dans le cas où p_n n'est pas une variable donne les deux lignes du tableau, Tab. 3.3. Il est important de noter que contrairement aux deux premières lignes du tableau, celles-ci ne représentent que le problème local d'un agent.

3.3.2 Résolution du problème local par ADMM de partage

Une autre manière de résoudre le problème local se concentre sur l'exploitation du parallélisme sur les échanges dans la minimisation locale (3.32) en le re-formulant sous la forme d'un problème de partage (section 2.4.1.3).

$$\begin{aligned} \mathbf{T}_n^{k+1} = \underset{t_{nm}}{\text{argmin}} \quad & g\Big(\sum_{m \in \omega_n} t_{nm}\Big) + \sum_{m \in \omega_n} f_{nm}(t_{nm}) \\ \text{t.q.} \quad & t_{nm} \in \mathbf{C}' \end{aligned} \tag{3.37}$$

avec :

$$\begin{aligned} f_{nm}(t_{nm}) &= \beta_{nm} t_{nm} + \frac{\rho}{2}\Big(t_{nm} - \frac{t_{nm}^k - t_{mn}^k}{2} + \lambda^k\Big)^2 \\ g\Big(\sum_{m \in \omega_n} t_{nm}\Big) &= g_n\Big(\sum_{m \in \omega_n} t_{nm}\Big) \end{aligned} \tag{3.38}$$

Selon [90], soit k l'itération globale et j l'itération de ce que l'on appellera le problème local, t_i le i^{ime} terme du vecteur T_n correspondant au i^{ime} échange

entrée : j_{max}, ub_n, lb_n, pairs
sortie : t_{min}

```
1  while err > ϵ and j < j_max do
2  |  for i=1,...M do                                    // Tous les pairs
3  |  |  t_i^{j+1} ← (3.39a);
4  |  |  t_i^{j+1} ← max(min(t_i^{j+1}, ub_n), lb_n) ;
5  |  end
6  |  t̄^{j+1} ← mean(t^{j+1}) ;
7  |  p̃^{j+1} ← (3.39b) ;
8  |  p̃^{j+1} ← max(min(p̃^{j+1}, p̄_n/M_n), p_n/M_n) ;
9  |  μ^{j+1} ← (3.39c) ;
10 |  (s^{j+1}, r^{j+1}) ← (3.44) ;
11 |  err ← max(s^{j+1}, r^{j+1}) ;
12 end
13 t_min ← t^j ;
```

Algorithme 4 : Algorithme ADMM pour le problème local

de l'agent considéré, $\bar{t}$ est la valeur moyenne des échanges de l'agent n, et enfin $\tilde{p} \approx \frac{p_n}{M_n}$. La solution (où $\tilde{p} = \bar{t}$) est atteinte itérativement en suivant les étapes suivantes pour chaque agent n :

$$t_i^{j+1} = \underset{lb_n<t_i<ub_n}{\text{argmin}} \left(f(t_i) + \frac{\rho_l}{2} \left\| t_i - t_i^j + \bar{t}^j - \tilde{p}^j + \mu^j \right\|_2^2 \right) \tag{3.39a}$$

$$\tilde{p}^{j+1} = \underset{\underline{p_n}<M_n\tilde{p}<\overline{p_n}}{\text{argmin}} \left(g(M_n\tilde{p}) + \frac{M_n\rho_l}{2} \left\| \tilde{p} - \mu^j - \bar{t}^{j+1} \right\|_2^2 \right) \tag{3.39b}$$

$$\mu^{j+1} = \mu^j + \bar{t}^{j+1} - \tilde{p}^{j+1} \tag{3.39c}$$

$$\begin{aligned} f_i(t_i) &= \frac{\rho}{2}(t_i - \frac{t_{ni}^k - t_{in}^k}{2} + \frac{\lambda_{ni}^k}{\rho})^2 + \beta_{ni} \cdot t_i \\ g(M_n\tilde{p}) &= 0.5 \cdot M_n^2 \cdot a_n \cdot \tilde{p}^2 + M_n b_n \tilde{p} \end{aligned} \tag{3.40}$$

On remarque que l'on doit réaliser la minimisation de deux fonctions scalaires de la forme :

$$\sum_j 0.5 \cdot a_j \cdot (y - b_j)^2 + \sum_j c_j \cdot y \tag{3.41}$$

La majorité des coefficients sont constants, en utilisant respectivement les indices t et p pour les coefficients des équations (3.39a) et (3.39b), les coefficients qui

changent selon l'itération globale (k) ou locale (j) sont les suivants :

$$b_{t1} = 0.5(t^k_{ni} - t^k_{in}) - \frac{\lambda^k_{ni}}{\rho} \tag{3.42a}$$
$$b_{t2} = t^j_i - \overline{t}^j + \tilde{p}^j - \mu^j \tag{3.42b}$$
$$b_{p1} = \mu^j + \overline{t}^{j+1} \tag{3.42c}$$

Et l'ensemble des coefficients constants sont les suivants :

$$a_{t1} = \rho \tag{3.43a}$$
$$a_{t2} = \rho_l \tag{3.43b}$$
$$c_{t1} = \beta_{ni} \tag{3.43c}$$
$$a_{p1} = M_n \cdot \rho_l \tag{3.43d}$$
$$a_{p2} = M_n^2 \cdot a_n \tag{3.43e}$$
$$c_{p1} = M_n \cdot b_n \tag{3.43f}$$

Chaque itération de la résolution du problème local est donc une forme fermée (section 4.3.3), ce qui nous permet de résoudre. Des résidus peuvent être calculés pour vérifier la convergence anticipée de l'algorithme :

$$\begin{aligned} r_l^{j+1} &= \left\|\overline{t}^{j+1} - \tilde{p}^{j+1}\right\| \\ s_l^{j+1} &= \left\|t_i^{j+1} - t_i^j\right\| \end{aligned} \tag{3.44}$$

Pour prendre en compte les contraintes (3.24d)-(3.24g), les deux solutions t_i et $\tilde{p}$ doivent être respectivement projetées dans leur intervalle admissible $[lb, ub]$ (qui dépend du type de l'agent) et $[\underline{p_n}/M_n, \overline{p_n}/M_n]$. L'algorithme est résumé dans Alg.4.

3.4 Décentralisation par PAC

L'algorithme de *Proximal Atomic Coordination* (PAC) est un autre moyen de décentraliser sensé être algorithmiquement plus rapide que l'ADMM et protégeant mieux la *privacy* des agents intervenant dans le système [27], [28]. En plus de ne pas transmettre directement les optimums de chaque itération pour chaque agent (mais une version modifiée), le PAC transmet une version modifiée également des variables duales.

> Dans le cas général, l'ADMM nécessite le partage des valeurs des variables duales. Cependant, la symétrie de la contrainte permet de ne transmettre que les optimums dans le cas d'un marché pair à pair.

Ici on parallélise encore sur chaque agent n constitué :

— de leur propre variable (p_n et t_{nm} pour tous ses voisins m), noté $\mathbf{L}$;
— de leur fonction coût ($g_n(p_n) + \sum_m \beta_{nm} t_{nm}$) ;
— de leur contraintes (1.2b), (1.2c), et les bornes sur les puissances et trades pour l'agent, noté $\mathbf{G}$;
— des copies des variables dont ils ont besoin ($a_{mn} = t_{mn}$) pour tous ses voisins m, noté $\mathbf{O}$.

L'ensemble de ces variables et des copies des variables utiles est noté $\mathbf{x}_n = (p_n, t_{nm}, a_{mn})$ pour l'agent n. Lorsque un agent n a besoin de la valeur stockée par un autre agent m, on note cela ainsi $t_{mn}^{[m]}$. Le Lagrangien associé à ce problème d'optimisation est le suivant :

$$\begin{aligned}\mathbf{L}_n &= f_n(x_n) + \mu_n^T \mathbf{G}_n \mathbf{x}_n + \nu^T \mathbf{B}^n \mathbf{x}_n \\ &= g(p_n) + \sum_m \beta_{nm} t_{nm} + \mu_1 (p_n - \sum_m t_{nm}) \\ &+ \sum_{p>1} \mu_p (t_{nm} + a_{mn}) + \sum_{m=1}^{M_n} \nu_m^n a_{mn} - \nu_n^m t_{nm}\end{aligned} \tag{3.45}$$

Ce Lagrangien peut être augmenté pour améliorer la convergence :

$$\begin{aligned}\mathbf{L}_{aug} &= \frac{\rho_n \gamma_n}{2} ||\mathbf{G}_n \mathbf{x}_n||_2^2 + \frac{\rho_n \gamma_n}{2} ||\mathbf{B}_n \mathbf{x}_n||_2^2 + \frac{1}{2\rho_n} ||\mathbf{x}_n - \mathbf{x}_n^k||_2^2 \\ &= \frac{\rho_n \gamma_n}{2} [(p_n - \sum_m t_{nm})^2 + \sum_{p>1} (t_{np} + a_{pn})^2)] \\ &+ \frac{\rho_n \gamma_n}{2} [\sum_{p=1} (a_{pn} - t_{pn}^{[p]})^2] + \frac{1}{2\rho_n} [\sum_p (x_n - x_n^k)^2]\end{aligned} \tag{3.46}$$

tel que $\mathbf{L}_\rho = \mathbf{L}_n + \mathbf{L}_{aug}$. L'équation à résoudre est une optimisation quadratique avec pour seule contrainte des bornes sur les variables. Ainsi il suffit de trouver le minimum et de projeter la solution dans ses bornes admissibles. On peut donc réécrire l'équation ainsi :

$$\begin{aligned}\mathbf{x} &= \text{argmin} \, \frac{1}{2} \mathbf{x}^T \mathbf{H} \mathbf{x} + \mathbf{q}^T \mathbf{x} \\ \text{t.q.} \quad & \mathbf{l} \leq \mathbf{x} \leq \mathbf{u}\end{aligned} \tag{3.47}$$

En posant $\mathbf{x} = (p_n, t_{nm}, a_{mn})$ de taille $N_v = N + 2M$, la matrice $\mathbf{H}$ est symétrique

et peut être définie ainsi :

$$\begin{aligned}
&H_{11} = a_n + \rho_n\gamma_n + 1/\rho_n && \\
&H_{1p} = H_{p1} = -\rho_n\gamma_n && 1 < p \leq m_n + 1 \\
&H_{pp} = 2\rho_n\gamma_n + 1/\rho_n && 1 < p \leq m_n + 1 \\
&H_{pp} = 2\rho_n\gamma_n + 1/\rho_n && m_n + 1 < p \leq 2m_n + 1 \\
&H_{ij} = H_{ji} = \rho_n\gamma_n && (i,j) \in [2, m_n+1]^2 \ i \neq j \\
&H_{ij} = H_{ji} = \rho_n\gamma_n && i \in [2, m_n+1]^2 \ j = i + m_n \\
&H_{ij} = 0 && sinon
\end{aligned} \tag{3.48}$$

On peut remarquer que la plupart des termes proviennent de la version augmentée de l'algorithme. Si la version augmentée n'était pas utilisée, il ne resterait que les termes a_n et $1/\rho_n$ dans la matrice. La matrice H serait donc diagonale. De même, on peut définir le vecteur $\mathbf{q}^T$ ainsi :

$$\begin{aligned}
q_1 &= b_n + \hat{\mu}_1 - (1/\rho_n)p_n^j && \\
q_p &= \beta_{nm} - \hat{\mu}_1 + \hat{\mu}_p - (\hat{\nu}^m)_n - (1/\rho_n)t_{np}^j && 1 < p \leq m_n + 1 \\
q_{p+m_n} &= \hat{\mu}_p + (\hat{\nu}^n)_m - (1/\rho_n)a_{pn}^j - \rho_n\gamma_n t_{pn}^{[p]} && 1 < p \leq m_n + 1
\end{aligned} \tag{3.49}$$

Enfin, on peut définir les $N_c = m_n + 1$ contraintes pour chaque agent ainsi :

$$l_1 = \underline{p_n} \tag{3.50a}$$

$$l_p = lb_n \qquad sinon \tag{3.50b}$$

$$u_1 = \overline{p_n} \tag{3.50c}$$

$$u_p = ub_n \qquad sinon \tag{3.50d}$$

On peut donc trouver la solution de manière directe, on a donc si $\mathbf{H}$ est définie positive :

$$\mathbf{x}_{min} = \mathbf{H}^{-1}\mathbf{q} \tag{3.51}$$

Puis il faut projeter dans les bornes de x pour chaque agent :

$$\mathbf{x}_{min} = max(min(\mathbf{x}_{min}, \mathbf{ub}), \mathbf{lb}) \tag{3.52}$$

On remarque que la matrice H est constante, on peut donc l'inverser une fois au début pour chaque agent. En effet tant que a_n, ρ_n, et γ_n ne changent pas de valeur l'inverse de la matrice reste valide. L'algorithme en résultant est représenté dans Alg. 5. Cependant, si γ change comme pourrait le laisser présager l'algorithme, il faudrait inverser à chaque changement les agents concernés. La résolution du système par factorisation pourrait être envisagée.

```
entrée : G, H, d, pairs
sortie : a_min
1  while err > ε and j < j_max do
2    for n=1,...N do                         // Tous les agents en parallèle
3      x_n^{j+1} ← (3.51) ;
4      x_n^{j+1} ← (3.52) ;
5      x̂_n^{j+1} ← x_n^{j+1} + α_n^{j+1}(x_n^{j+1} − x_n^j) ;
6      μ_n^{j+1} ← μ̂_n^j + ρ_n γ_n (G_n x̂_n^{j+1}) ;
7      μ̂_n^{j+1} ← μ_n^{j+1} + φ_n^{k+1}(μ_n^{j+1} − μ_n^j) ;
8    end
9    Communication x̂ ;
10   for n=1,...N do                         // Tous les agents en parallèle
11     ν_n^{j+1} ← ν̂_n^j + ρ_n γ_n B_n X̂^{j+1} ;
12     ν̂_n^{j+1} ← ν_n^{j+1} + θ_n^{j+1}(v_n^{j+1} − v_n^j) ;
13   end
14   Communication ν̂ ;
15   for n=1,...N do                         // Tous les agents en parallèle
16     q_n ← (3.49);
17     Update α_n, φ_n, θ_n and γ_n ;
18     Compute err
19   end
20 end
```

Algorithme 5 : Algorithme PAC

3.4.1 Convergence du PAC

Pour assurer la bonne convergence de l'algorithme, la fonction coût allant de $\mathcal{R}^n$ vers $\mathcal{R}$ doit être α fortement convexe et L fortement doux ce qui signifie que $\forall(x,y)$ on a :

$$f(x)+\nabla f(x)^T(y-x)+\frac{L}{2}||x-y||_2^2 \geq f(y) \geq f(x)+\nabla f(x)^T(y-x)+\frac{\alpha}{2}||x-y||_2^2 \quad (3.53)$$

Dans le cas où on a $f(x)=\sum_i 0.5a_ix_i^2+b_ix_i$, on peut montrer que cela revient à dire que $\forall(x,y)$:

$$\sum_i \frac{L-a_i}{2}(x_i-y_i)^2 \geq 0 \geq \sum_i \frac{\alpha-a_i}{2}(x_i-y_i)^2 \quad (3.54)$$

On peut montrer qu'une condition suffisante est de prendre $\alpha = min(a_i)$ et $L = max(a_i)$ pour que cela fonctionne.

Cependant, dans notre cas, les échanges font aussi partis des variables à optimiser. Ainsi, en notant p les variables sur les puissances et t celles sur les échanges,

la condition devient :

$$\begin{aligned}\sum_i \frac{L-a_i}{2}(x_i^p - y_i^p)^2 + \sum_m \frac{L}{2}(x_{im}^t - y_{im}^t)^2 \geq 0 \\ \geq \sum_i \frac{\alpha - a_i}{2}(x_i^p - y_i^p)^2 + \sum_m \frac{\alpha}{2}(x_{im}^t - y_{im}^t)^2\end{aligned} \tag{3.55}$$

Ainsi le fait que la fonction coût ne soit pas convexe sur les échanges empêche de trouver une condition suffisante sur le α à choisir. En effet, comme $x_i^p = \sum_m x_{im}^t$, on pourrait trouver des x_{im}^t et des y_{im}^t différents tels que leurs sommes sur m soient égales ($x_i^p = y_i^p$). Ce qui donne $0 \geq \sum_i \sum_m \frac{\alpha}{2}(x_{im}^t - y_{im}^t)^2$ qui n'est possible que pour $\alpha = 0$.

Pour la suite on choisira malgré tout l'heuristique suivante $\alpha = min(a_i)$ et $L = max(a_i)$ pour choisir les paramètres.

Pour optimiser le *Strong primal rate*, les coefficients sont :

$$\begin{aligned}\rho_P^* &= \left(\sqrt{\gamma_P^* \lambda_{max}(V_1)}\right)^{-1} \\ \gamma_P^* &= \frac{2\alpha L}{2\lambda_{max}(V_1) + \hat{\lambda}_{min}(V_1)}\end{aligned} \tag{3.56}$$

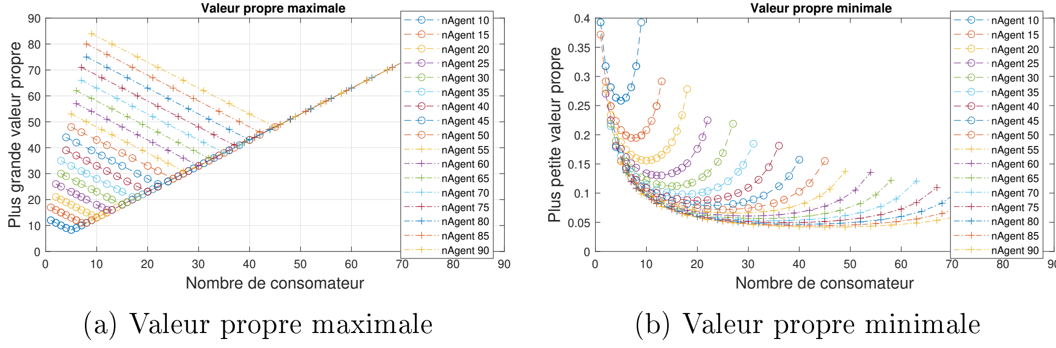
Avec $V_1 = G^T G + B^T B$ (avec G étant la concaténation en diagonale des différentes contraintes) et $\hat{\lambda}_{min}$ étant la plus petite racine non nulle de la matrice. On peut remarquer qu'avec l'augmentation de la taille du problème, la taille de la matrice V, étant en $O(2(Ncons^2 + Ngen^2)^2) \approx O(N^4)$, explose rendant de plus en plus coûteux de réaliser ces calculs.

Par exemple, le cas de la France dans le jeu de donnée Européen contient 484 agents dont 318 consommateurs, la taille de V est d'environ 210 000 · 210 000 et Matlab n'arrive pas à déterminer ses valeurs propres par manque de mémoire.

Dans le cas d'un problème de marché, les matrices G et B ne contiennent que des 1 et des -1 dont leur positionnement ne dépend que des dimensions du cas et de la répartition consommateurs, producteurs, et consommacteurs. Ainsi on peut chercher une heuristique pour déterminer les valeurs propres à grande dimension de la matrice V_1.

Le calcul des valeurs propres en fonction du nombre d'agents N et du nombre de consommateurs N_c dans le cas sans consommacteurs avec uniquement un marché de puissance active donne les Fig. 3.4.

On remarque les points suivants :



(a) Valeur propre maximale (b) Valeur propre minimale

FIGURE 3.4 – Valeurs propres pour la méthode PAC

— Il y a une symétrie par rapport à la droite verticale à $N_c = N/2$.
— On a une fonction de la forme $\lambda_{max}(N, N_c) = a(N)|N_c - \frac{N}{2}| + b(N)$.
— le terme $a(N)$ tend vers 1 avec l'augmentation du nombre d'agent.
— $b(N)$ varient linéairement avec le nombre d'agent ce qui donne environ $b(N) = 0.5 * N + 3.36$
— On a une fonction de la forme $\lambda_{min}(N, N_c) = c(N)(N_c - \frac{N}{2})^2 + d(N)$
— Comme λ_{min} tend vers 0 et qu'il n'intervient que sommé avec λ_{max} (qui augmente avec les dimensions du problème) ce terme peut être négligé à forte dimension.
— Pour information les coefficients peuvent être approximées ainsi $c(N) = \frac{c_1 N + c_2}{N^2}$ et $d(N) = \frac{log(d_1 N + d_2)}{d_3 N}$.

Dans le cas où l'on ajoute une deuxième puissance à échanger où tous les agents sont reliés entre eux, il en découle que λ_{max} et λ_{min} ne dépendent plus du nombre de consommateurs. L'heuristique proposée, dans ce cas-là, est donc $\lambda_{max} = N + 2$ et $\lambda_{min} = \frac{1.99N - 3.3}{N^2}$

3.5 Validation fonctionnelle

L'objectif de cette partie sera de vérifier la convergence des différentes méthodes sur différents cas. Des temps sont donnés pour indiquer un ordre de grandeur du temps de résolution pour chacun des algorithmes. Le tableau Tab. 3.4 rassemble l'ensemble des résultats. Les cas *TestFeeder* et Européen possèdent plusieurs pas de temps. Cependant, dans le cadre de cette étude, seule la simulation sur le premier pas de temps sera réalisée. Les paramètres pour l'ensemble de ces simulations sont dans le Tab. 3.5, sauf pour le cas européen où $\rho_g = 100$.

TABLE 3.4 – Nombre d'itération et temps (ms) pour converger pour les différents algorithmes et cas

Cas	OSQPCen		ADMM		OSQP		PAC	
Nom	temps	iter	temps	iter	temps	iter	temps	iter
cas 2 bus (DC)	2	25	0.2	16	0.2	16	0.3	71
cas 2 bus (AC)			0.3	21	0.8	21	1	261
cas 3 bus (DC)	0.2	50	0.1	21	0.3	21	0.3	166
cas 3 bus (AC)			0.3	21	0.9	21	1.1	196
Matpower 10			6	86	45	391	58	1061
Cas 39 nœuds	7	100	50	161	41	136	5.1e3	27 126
TestFeeder			58.8	61	98.7	66	1e3	3371
Matpower 85			20	16	35	16	4e3	1031
Européen			1.6e6	591	2.1e6	1046		

La méthode **OSQP** n'a été implémentée que pour gérer un marché monoénergie et n'arrive pas à créer un cas de la taille du réseau européen (memory error). C'est pourquoi il n'a été testé que sur trois des cas.

TABLE 3.5 – Paramètres pour la simulation

Features	value	Features	value
k_{max}	100 000	j_{max}	1000
$step_g$	5	$step_L$	5
ϵ_g	0.001	ϵ_l	0.0005
ρ_g	1	ρ_l	ρ_g

On peut voir dans le tableau des temps très faibles pour toutes les méthodes pour résoudre les cas les plus petits. Cependant, les temps deviennent très importants avec l'augmentation des dimensions du problème.

À titre de comparaison avec la littérature, le cas 39 nœuds pour une précision de $1e^{-4}$ converge en environ 80ms pour $\rho = 1$ avec l'**ADMM**. C'est nettement plus rapide que la version de [25] qui est résolu en 9.5s sur Matlab. Cette différence peut s'expliquer par la différence de matériel, de langage et du fait de résoudre les problèmes locaux avec une ADMM de partage plutôt qu'avec un solveur de Matlab.

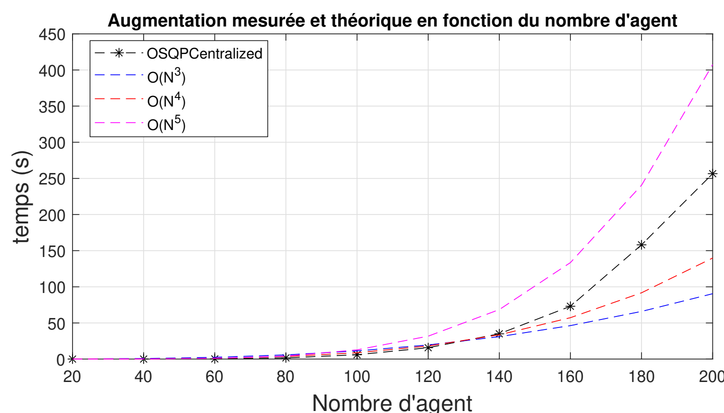


FIGURE 3.5 – Temps de calcul mesuré (noir) et courbe de tendance selon la dimension du problème

3.5.1 Passage à l'échelle

L'objectif de cette partie sera de montrer le comportement des différents algorithmes lors de l'augmentation des dimensions du problème. Pour cela, on va générer des cas d'études aléatoirement en faisant varier la taille du problème. Pour chaque taille de problème, plusieurs cas d'études sont générés et l'on regardera le temps moyen pour résoudre le cas d'étude.

Dans un premier temps, on s'intéressera à la résolution d'un marché pair à pair sous sa forme centralisée. La fig. 3.5 représente la variation du temps moyen de simulation des cas de différentes tailles. Pour représenter la complexité du problème, on tracera les courbes suivantes $C_\alpha(N) = t_0 * N^\alpha$ avec $t_0 = t_{20}/20^\alpha$ le temps mesuré à 20 agents, divisé par le nombre d'agents à la puissance de la complexité.

On y remarque un temps de plus de 250s en moyenne pour résoudre un cas de 200 agents, ce qui montre clairement les limites du passage à l'échelle de la résolution centralisée. Au-delà du temps de calcul qui peut être très dépendant de la machine, ce qu'il faut remarquer ici est l'augmentation très rapide du temps de calcul avec la dimension du problème qui présente une complexité entre $O(N^4)$ et $O(N^5)$. Ainsi on peut voir que simuler un marché pair à pair lève dans sa forme centralisée une barrière de complexité de calcul. En conséquence, la simulation de marché à grande échelle est compromise à cette étape.

En réalité, comme la complexité n'est pas linaire, le fait de réduire la taille des problèmes à optimiser en le séparant en plusieurs sous-problèmes devrait permettre de réduire le temps de calcul même sans parallélisation. Ainsi, pour la suite, on regardera le passage à l'échelle des algorithmes décentralisé (non parallélisé).

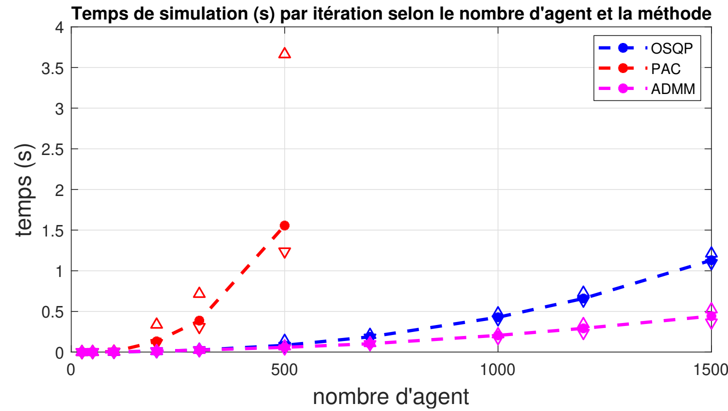


FIGURE 3.6 – Temps de calcul mesuré des différents algorithmes (C++ sur CPU, AMD RYZEN 5 5600H)

Les résultats suivants sont réalisés sur des cas AC (avec puissance active et réactive). Le nombre d'agents indiqué est le nombre réel d'agents. Par rapport à un problème en DC, la dimension (M) est au moins deux fois plus grande pour un même nombre d'agents.

TABLE 3.6 – Temps moyen (s) (nombre d'itération) selon la méthode et la taille

N agents	25	125	700	1500
OSQP	0.007 (31)	0.12 (31)	0.32 (31)	35 (31)
PAC	0.312 (2317)	315 (39 975)	*	*
ADMM	0.008 (31)	0.14 (31)	0.33 (31)	14 (32)

Le tableau Tab. 3.6 représente les moyennes des temps de calculs pour les différentes méthodes décentralisées. On remarque que la simulation via PAC devient rapidement impossible. En effet, une multiplication par 5 de la taille du problème entraine une multiplication par 100 du temps de calcul. Cependant, cela est principalement dû à une explosion du nombre d'itérations pour converger. D'un autre côté, la décentralisation par ADMM ne nécessite qu'une trentaine d'itérations pour converger qu'elle soit résolue localement via une OSQP ou une ADMM sous forme de partage. Afin de pouvoir afficher le temps par itération de toutes les méthodes, le nombre d'itérations maximales a été réduit à 50. On aura ainsi accès aux temps pour la méthode PAC à plus grande dimension. Cependant, il est important de noter que dans ce cas là, cet algorithme n'aura pas convergé. La figure Fig. 3.6 représente le temps par itération des différentes méthodes.

Le temps par itération de la méthode PAC augmente rapidement et est très irrégulier. Une étude plus poussée sera réalisée dans le chapitre suivant pour évaluer où se situe l'irrégularité du temps de calcul. Pour les deux autres méthodes, le temps par itération évolue avec une complexité proche de $O(N^2)$. Ainsi, les temps augmentent rapidement avec les dimensions du problème rendant les simulations à grande dimension impossibles. La parallélisation est donc nécessaire pour réduire les temps de simulations. On peut cependant dès à présent remarquer la supériorité de l'ADMM sur les autres méthodes sur CPU. C'est pourquoi, pour la suite, la décentralisation ou la résolution d'un problème local de marché seront respectivement résolues via une ADMM de consensus et de partage.

4 Optimal Power Flow

4.1 Mise en forme du problème

De manière générale, un Optimal Power Flow (OPF) est un problème cherchant à minimiser une fonction coût à l'échelle du réseau tout en respectant les contraintes dans les lignes. C'est un problème hautement non convexe. Ainsi, pour améliorer la robustesse des optimisations, de nombreuses relaxations ou approximations coexistent selon l'objectif ou le réseau considéré.

Dans notre cas, la fonction coût à minimiser sera la somme des fonctions coûts des agents. Cela pourrait être, de manière non exhaustive, la minimisation des pertes dans les lignes, la minimisation des échanges de puissances réactives, [55], [57]... Les véritables variables de décision sont la puissance des agents. Toutefois, pour faciliter la résolution (par exemple, la prise en compte des contraintes), le vecteur d'optimisation X pourra également contenir des variables du réseau, comme les tensions complexes, les puissances ou les courants dans les lignes. Le problème d'optimisation peut s'écrire ainsi :

$$\min_{\mathbf{X}} \sum_{n\in\Omega} f(\underline{s}_n) \tag{3.57a}$$

$$\text{t.q. } h(\mathbf{X}) = 0 \tag{3.57b}$$

$$g(\mathbf{X}) \leq 0 \tag{3.57c}$$

Contrairement à ce qui est souvent fait dans la littérature des OPF, ici, tous les agents sont flexibles. Ainsi, la fonction coût de tous les agents qui est considérée, et pas uniquement celles des générateurs. Celle des consommateurs représente le coût d'activation de leur flexibilité.

L'ensemble de ces contraintes peut s'écrire ainsi dans le cas où les tensions sont

en coordonnée polaire, (3.58a) et (3.58c) et que l'on considère le modèle en bus injection (3.58f) :

$$\underline{V_b} \leq V_b \leq \overline{V_b} \qquad b \in \mathcal{B} \tag{3.58a}$$
$$\underline{s_n} \leq |s_n| \leq \overline{s_n} \qquad n \in \Omega \tag{3.58b}$$
$$\underline{\theta_{ij}} \leq \theta_{ij} \leq \overline{\theta_{ij}} \qquad (\delta_{\theta 1-2}) \qquad (i,j) \in \mathcal{L} \tag{3.58c}$$
$$\underline{S_{ij}} \leq |S_{ij}| \leq \overline{S_{ij}} \qquad (\delta_{S1-2}) \qquad (i,j) \in \mathcal{L} \tag{3.58d}$$
$$\sum_{n \in \mathcal{N}_b} \mathbf{s}_n = \sum_{l \in \mathcal{L}_b} \mathbf{S}_l \qquad b \in \mathcal{B} \tag{3.58e}$$
$$\underline{S_l} = \underline{Y_l^*} \cdot \underline{E_i} \cdot (\underline{E_i} - \underline{E_j})^* \qquad l = (i,j) \in \mathcal{L} \tag{3.58f}$$
$$(V_0, \theta_0) = (V0, \theta 0) \qquad \textit{reference} \tag{3.58g}$$

La tension peut aussi être représentée en coordonnée rectangulaire ce qui donne :

$$\underline{\theta_{ij}} \leq \arctan(\frac{f_b}{e_b}) \leq \overline{\theta_{ij}} \qquad (\delta_{\theta 1-2}) \qquad (i,j) \in \mathcal{L} \tag{3.59a}$$
$$\underline{V_b}^2 \leq (e_b^2 + f_b^2) \leq \overline{V_b}^2 \qquad b \in \mathcal{B} \tag{3.59b}$$

Dans le cas d'un réseau radial, la contrainte (3.58f) peut être remplacée par les contraintes suivantes que l'on appellera les équations du flux (*DistFlow* ou *branch flow equations*) par opposition aux équations d'injections, avec l_{ij} la norme du courant au carré :

$$P_{ij} = R_{ij} l_{ij} - P_j + \sum_{m:j->m} P_{jm} \tag{3.60a}$$
$$Q_{ij} = X_{ij} l_{ij} - Q_j + \sum_{m:j->m} Q_{jm} \tag{3.60b}$$
$$V_j^2 = V_i^2 - 2(R_{ij} P_{ij} + X_{ij} Q_{ij}) + (R_{ij}^2 + X_{ij}^2) l_{ij} \tag{3.60c}$$
$$l_{ij} V_i^2 = P_{ij}^2 + Q_{ij}^2 \tag{3.60d}$$

Cette expression peut aussi être utilisée dans un réseau maillé, mais elle n'est plus exacte en état, c'est dans ce cas une relaxation non convexe.

Cette expression est linéaire par rapport à la norme de la tension au carré, un changement de variable est donc possible pour en tirer parti.

> La notation considère que la ligne i va vers le nœud i, comme il n'y a pas de boucle, cette définition est unique. On remarque donc que $S_0 = 0$ car on considère que le nœud de référence n'a pas d'ancêtres. S'il existe une puissance injectée via le réseau de transport, celle-ci apparaîtra dans s_0.

Enfin, on peut aussi noter la relation suivante sur les flux dans les lignes :

$$\begin{aligned} S_l^2 &= (G_l^2 + B_l^2) \cdot V_i^2 \cdot (V_i^2 + V_j^2 - 2V_iV_j \cos(\theta_i - \theta_j)) \\ &= (G_l^2 + B_l^2) \cdot V_i^2 \cdot ((e_i - e_j)^2 + (f_i - f_j)^2) \end{aligned} \tag{3.61}$$

Pour connaitre la forme des équations derrière les contraintes de (3.57), il faut fixer certaines représentations. Dans un premier temps, il faut décider si l'on représente la tension sous une forme polaire (angle et amplitude) ce qui nous donne les contraintes (3.58a) et (3.58c) ou en forme cartésienne (partie réelle ou imaginaire), ce qui implique les contraintes (3.59b) et (3.59a). Dans tous les cas, la tension au nœud de référence est fixée via la contrainte (3.58g). Ensuite, il faut contraindre les puissances dans les lignes (3.58d) et les puissances injectées par les agents (3.58b). Les lois de la physique impliquent que la puissance injectée dans un bus est égale à la somme des puissances des agents sur ce bus, (3.58e).

Pour obtenir la puissance dans les lignes, on peut considérer le modèle dit "en bus injection", (3.58f). Dans le cas d'un réseau radial, ou lorsque l'on accepte des approximations, on pourra considérer un modèle en "équation de flux" (3.60d).

Dans notre cas, on cherchera à résoudre spécifiquement des OPF sur des réseaux radiaux. On s'appuiera sur l'algorithme décentralisé présenté dans [59] pour ce faire. L'algorithme proposé sera appelé **OPFADMM**. Ensuite, une modification sera apportée pour prendre en compte plus facilement en compte le cas où il y a plusieurs agents par bus. Cette seconde version sera désignée sous le nom de **OPFADMM2**.

4.2 Résolution de référence

Dans le cas d'un réseau purement radial, une relaxation sur le cône de deuxième ordre peut être réalisée. Elle correspond au fait de transformer l'égalité (3.60d) en l'inégalité suivante [59] :

$$S_i^2 \leq V_i^2 l_i \tag{3.62}$$

Ici, les variables d'optimisation sont l'ensemble des variables des agents (p_n, q_n) et des inconnues du réseau constituées de la tension v_i, des courants (au carré) l_i, des puissances dans les lignes $\underline{S}_i$ et des puissances dans les bus $\underline{s}_i$. En utilisant le modèle du flux dans les branches, les angles ont été enlevés, on ne considérera donc

pas de contraintes sur cette variable. Par définition, on a $v_i = V_i^2$ l'amplitude au carré de la tension. Le problème que l'on cherchera à résoudre est donc le suivant :

$$\min_{\mathbf{X}} \quad \sum_{n\in\Omega} g_n(s_n) \tag{3.63a}$$

$$t.q.\ \underline{\mathbf{X}} \leq |\mathbf{X}| \leq \overline{\mathbf{X}} \qquad h \in \mathcal{H} \tag{3.63b}$$

$$\sum_{n\in\mathcal{N}_i} s_n = s_i \qquad i \in \mathcal{B} \tag{3.63c}$$

$$v_{A_i} - v_i + \underline{z}_i \underline{S}_i^* + \underline{S}_i \underline{z}_i^* - l_i|\underline{z}_i|^2 = 0 \qquad i \in \mathcal{B} \tag{3.63d}$$

$$\sum_{j\in C_i} (\underline{S}_j - l_j \underline{z}_j) + \underline{s}_i - \underline{S}_i = 0 \qquad i \in \mathcal{B} \tag{3.63e}$$

$$|\underline{S}_i|^2 \leq v_i l_i \qquad i \in \mathcal{B} \tag{3.63f}$$

$$(v_0, S_0) = (v0, 0) \qquad \textit{reference} \tag{3.63g}$$

Grâce à la relaxation, ce problème est convexe sous contraintes linéaires avec une contrainte sur le cône de second ordre pour chaque bus. Afin de résoudre ce problème d'optimisation, on séparera d'un côté la minimisation des fonctions coût avec les contraintes sur x et et $|\underline{S}_i|^2 < l_i v_i$ et de l'autre le respect des contraintes avec le reste du réseau et le consensus. Pour cela, on va utiliser une ADMM (section 2.4.1.1), on notera les variables slacks y et μ sera la variable duale associée à la contrainte $x = y$. L'application de cet algorithme permet de résoudre de manière itérative ce problème :

$$x^{k+1} = \underset{x}{\operatorname{argmin}}\, L_\rho(x, y^k, \mu^k) \tag{3.64a}$$

$$y^{k+1} = \underset{y}{\operatorname{argmin}}\, L_\rho(x^{k+1}, y, \mu^k) \tag{3.64b}$$

$$\mu^{k+1} = \mu^j + \rho \cdot (x - y) \tag{3.64c}$$

avec :

$$L_\rho = \sum_{i\in\mathcal{B}} \sum_{n\in\mathcal{N}_i} f(s_n) + \sum_j < \mu_{ij}, \mathbf{x}_i - \mathbf{y}_{ij} > + \frac{\rho}{2} ||\mathbf{x}_i - \mathbf{y}_{ij}||_2^2 \tag{3.65}$$

L'indice j représente l'ensemble des copies de la même variable x sur les différents bus. La démonstration et les détails se trouvent dans [59] et en annexe. L'idée de cet algorithme est de résoudre toutes les minimisations avec une forme fermée. On peut déjà remarquer que chaque minimisation peut être distribuée sur les bus. On note $\hat{\mathbf{x}}^k$, une constante dépendant de l'itération et représentant l'influence de $\mathbf{y}^k$ et μ^k sur la minimisation de x. De même, l'influence de $\mathbf{x}^{k+1}$ et μ^k sur la minimisation de $\mathbf{y}$ sera regroupé dans un vecteur noté $\mathbf{c}^T$.

On va donc réaliser 3 minimisations distinctes. En effet, la minimisation sur $\mathbf{x}$ fait intervenir 2 termes séparables. Le premier sous-problème est donc la minimisation du problème dépendant de $\mathbf{x}$. Pour résoudre, on commence par minimiser en fonction de l'injection de puissance $\mathbf{x}_i = (p_i, q_i, p_n, q_n)$ pour tous les agents n sur le bus. Donc, ici, on rajoute le fait que la puissance injectée est la somme des puissances des agents.

$$\begin{aligned} \mathbf{x}_i = \underset{x_i}{\operatorname{argmin}} \sum_{n \in \mathcal{N}_i} & g_n(p_n, q_n) + \rho/2(||p_i - \hat{p}_i||_2^2 + ||q_i - \hat{q}_i||_2^2) \\ \text{t.q. } & \underline{p_n} \leq p_n \leq \overline{p_n} & \forall n \in \mathcal{N}_i \\ & \underline{q_n} \leq q_n \leq \overline{q_n} & \forall n \in \mathcal{N}_i \\ & p_i = \sum_{n \in \mathcal{N}_i} p_n \\ & q_i = \sum_{n \in \mathcal{N}_i} q_n \end{aligned} \tag{3.66}$$

On peut remarquer que, si l'on considère qu'un seul agent se trouve par bus, comme c'est le cas dans [59], la minimisation est bien en forme fermée. En effet on a $p_i = p_n$ et $q_i = q_n$ que l'on peut remplacer dans la fonction coût et les contraintes correspondantes disparaissent. On doit minimiser une fonction coût quadratique avec uniquement des bornes en tant que contrainte, on retrouve la forme fermée présentée dans la section 2.4.3.3.

Dans le cas où il y a plusieurs agents, on reconnait un problème de partage, que l'on résoudra avec une ADMM de partage :

$$p_n^{j+1} = \underset{\underline{p_n} \leq p_n \leq \overline{p_n}}{\operatorname{argmin}} \left(f(p_n) + \frac{\rho_l}{2} \left\| p_n - p_n^j + \overline{p}^j - \tilde{p_i}^j + \mu_i^j \right\|_2^2 \right) \tag{3.67a}$$

$$\tilde{p_i}^{j+1} = \underset{\tilde{p_i}}{\operatorname{argmin}} \left(g(N_i \tilde{p_i}) + \frac{N_i \rho_l}{2} \left\| \tilde{p_i} - \mu^j - \overline{p}^{j+1} \right\|_2^2 \right) \tag{3.67b}$$

$$\mu^{j+1} = \mu^j + \overline{p}^{j+1} - \tilde{p_i}^{j+1} \tag{3.67c}$$

Tout comme on a fait pour la résolution locale du marché pair à pair on peut ainsi résoudre analytiquement chaque minimisation.

La deuxième minimisation pour la variable $\mathbf{x}$ considère les équations de la ligne i qui va de l'ancêtre A_i vers le bus i, on a donc $\mathbf{x}_i = (Pi, Qi, \sqrt{\frac{|C_i+1|}{2}}vi, li) = (x_1, x_2, x_3, x_4)$.

$$\begin{aligned} x_i = \underset{x_i}{\operatorname{argmin}} & \sum x_i^2 - 2 * \hat{x}_i x_i \\ \text{t.q. } & (P_i^2 + Q_i^2) \leq l_i v_i \\ & \underline{v_i} \leq v_i \leq \overline{v_i} \end{aligned} \tag{3.68}$$

En faisant une distinction entre les cas où les contraintes sont actives ou non, et en utilisant les conditions d'optimalité de KKT, on peut obtenir une expression analytique de l'optimum. Cet optimum existe de manière certaine, puisque le problème est convexe et que $(0, 0, \sqrt{\frac{|C_i+1|}{2}}, 0)$ est une solution respectant les contraintes. La résolution passe par la détermination des racines réelles d'un polynôme de degré 3 ou 4.

L'annexe démontre aussi la résolution dans le cas où on a une contrainte de la forme $l_i \leq \bar{l}_i$, même si cette contrainte n'a pas été utilisée dans les tests.

Il reste la minimisation sur les variables slacks et les contraintes associées $\mathbf{y}_i = (P_i, Q_i, v_i, l_i, v_{A_i}, (p_{ci}, q_{ci}, l_{ci})\ \forall c_i)$:

$$\begin{aligned} \mathbf{y}_{ij} = \underset{\mathbf{y}_i}{\operatorname{argmin}}\, & \mathbf{y}_i^T \mathbf{H} \mathbf{y}_i + \mathbf{c}_y^T \mathbf{y}_i \\ \text{t.q. } & v_{A_i} - v_i + \underline{z}_i \underline{S}_i^* + \underline{S}_i \underline{z}_i^* - l_i |\underline{z}_i|^2 = 0 \quad i \in \mathcal{L} \\ & \sum_{j \in C_i} \underline{S}_j - l_j \underline{z}_j + \underline{s}_i - \underline{S}_i = 0 \quad i \in \mathcal{B} \end{aligned} \tag{3.69}$$

Les contraintes de la dernière minimisation peuvent se mettre sous la forme $\mathbf{Ay} = \mathbf{0}$ ($\mathbf{A}$ est une matrice de 3 lignes), et comme $\mathbf{H}$ est une matrice diagonale, on peut trouver une solution de manière directe (section 4.3.2).

Les étapes restantes de l'algorithme consistent à mettre à jour les variables duales pour permettre la convergence :

$$\mu = \mu + \rho \cdot (\mathbf{x} - \mathbf{y}) \tag{3.70}$$

et à calculer les résidus pour respectivement la convergence entre les itérations (s) et le respect des contraintes (r) :

$$\begin{aligned} s^k &= \rho ||\mathbf{y}^k - \mathbf{y}^{k-1}|| \\ r^k &= ||\mathbf{x}^k - \mathbf{y}^k|| \end{aligned} \tag{3.71}$$

L'ensemble des étapes de l'algorithme avec la précision des communications nécessaires pour converger est représenté sur l'Alg. 6.

4.3 Déplacement de la contrainte des puissances injectées

Une autre manière de résoudre ce type de problème est de remarquer que la contrainte $p_i = \sum_{n \in \mathcal{N}_i} p_n$ est directement distribuable sur les bus, et que l'on n'a pas d'autres contraintes sur p_i. Pour la suite on ne parlera que de puissance active,

entrée : G, H, d, peers
sortie : a_{min}

```
1  while err > ϵ and j < j_max do
2      for i=1,...B do                         // Tous les bus en parallèle
3          x ← (3.66) & (3.68) ;
4      end
5      Communication x avec ancêtre et enfants ;
6      for i=1,...B do                         // Tous les bus en parallèle
7          y ← (3.69) ;
8          μ ← μ + ρ(x − y) ;
9      end
10     Communication y et μ avec ancêtre et enfants ;
11     for i=1,...B do                         // Tous les bus en parallèle
12         Calcul err ← (3.71)
13     end
14 end
```

Algorithme 6 : Algorithme ADMM pour de l'OPF

mais la même chose s'applique aux puissances réactives. Ainsi pour chaque bus on peut enlever p_i des variables et utiliser à la place la puissance de tous les agents sur le bus. En faisant la substitution, les modifications sont les suivantes. Dans les expressions de x et de y, il faut rajouter $2 * N_i$ variables (l'ensemble des p_n et q_n) et en enlever 2 (p_i et q_i). Pour (3.69) le problème devient :

$$\begin{aligned} \mathbf{y}_{ij} = \underset{\mathbf{y}_i}{\operatorname{argmin}}\, & \mathbf{y}_i^T \mathbf{H} \mathbf{y}_i + \mathbf{c}_y^T \mathbf{y}_i \\ \text{t.q. } & v_{A_i} - v_i + \underline{z}_i \underline{S}_i^* + \underline{S}_i \underline{z}_i^* - l_i |\underline{z}_i|^2 = 0 \quad i \in \mathcal{L} \\ & \sum_{j \in C_i} \underline{S}_j - l_j \underline{z}_j + \sum_{n \in \mathcal{N}_i} \underline{s}_n^{agent} - \underline{S}_i = 0 \quad i \in \mathcal{B} \end{aligned} \tag{3.72}$$

Le problème (3.68) demeure inchangé et le problème (3.66) devient :

$$\begin{aligned} \mathbf{x}_i = \underset{\mathbf{x}_i}{\operatorname{argmin}} & \sum_{n \in \mathcal{N}_i} \Big[g_n(p_n, q_n) + \rho/2 (p_n^2 + q_n^2 - 2\hat{p}_n p_n \quad -2\hat{q}_n q_n) \Big] \\ \text{t.q. } & \underline{p_n} \leq p_n \leq \overline{p_n} \qquad \forall n \in \mathcal{N}_i \\ & \underline{q_n} \leq q_n \leq \overline{q_n} \qquad \forall n \in \mathcal{N}_i \end{aligned} \tag{3.73}$$

avec $\hat{p}_n$ et $\hat{q}_n$ qui sont calculés de manière analogue que $\hat{p}_i$ ou $\hat{q}_i$ précédemment (en effet, la puissance injectée n'étant connue que du bus, le consensus a lieu dans les 2 cas dans le bus uniquement entre x_i et y_i). Le problème d'optimisation étant

quadratique avec des variables indépendantes et les contraintes n'étant que des bornes, on peut résoudre directement le problème ainsi :

$$\begin{aligned} p_n &= \min(\max(\frac{\rho\hat{p}_n - b_n}{\rho + a_n}, \underline{p}_n), \overline{p}_n) \\ q_n &= \min(\max(\frac{\rho\hat{q}_n - b_n}{\rho + a_n}, \underline{q}_n), \overline{q}_n) \end{aligned} \tag{3.74}$$

On remarque que, dans le cas où l'on a qu'un seul agent par bus, cette formulation n'a aucune différence avec la précédente. Si plusieurs agents sont présents dans un bus, on effectue quelques calculs supplémentaires dans la résolution de l'équation (3.69) en échange d'une forme fermée dans la résolution de l'équation (3.66). Enfin, dans le cas où il n'y a pas d'agent sur le bus, on réduit la taille de la minimisation sur y.

4.4 Validation fonctionnelle

Dans cette partie, la résolution d'OPF a été réalisée sur des cas fixes. Les versions avec ADMM ne sont faites que pour résoudre des cas radiaux. Dans notre cas, les puissances des agents sont initialisées au milieu de leurs bornes. Les tensions sont initialisées avec des tensions plates ($e_i = 1$ et $f_i = 0$ pour tous les bus i). La précision demandée est $\epsilon = 1e^{-4}$.

Table 3.7 – Nombre d'itération et temps (ms) pour converger pour les différents algorithmes et cas

Cas		OPFADMM		OPFADMM2	
Nom	Taille N/B	temps	iter	temps	iter
Cas 2 bus	6/2	14	1201	11	1201
Matpower 10ba	24/10	10	401	8	401
Matpower 85	122/85	262	2351	210	2351
TestFeeder	114/906	$15e^3$	17401	$10e^3$	15951

On peut remarquer que les deux algorithmes sont plutôt semblables à faible dimension. La deuxième représentation est légèrement plus rapide, et l'est de plus en plus avec l'augmentation des dimensions du problème résolu. On peut aussi voir que le nombre d'itérations pour converger est très variable.

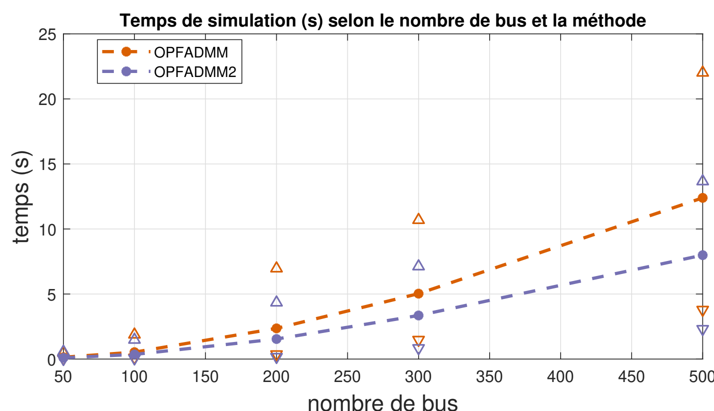


FIGURE 3.7 – Temps de calcul mesuré des différents algorithmes

4.4.1 Passage à l'échelle

Afin d'étudier le passage à l'échelle de ces méthodes, le même générateur de réseau radial sera utilisé que pour les **PF**. Afin que les simulations soient plus rapides, la précision demandée est de 0.01 pour un nombre d'itérations maximal à 50 000. Le facteur de pénalité a été fixé à $\rho = 2$ pour toutes les simulations. Pour ces générateurs, le nombre d'agents est fixé égal au nombre de bus. Cependant, leur position est tirée aléatoirement, donc il peut y avoir des bus avec plus d'agents et d'autres sans.

Les deux méthodes ont convergé sur tous les cas générés. On peut voir sur la figure 3.7 les temps de simulations des différentes méthodes. On peut y voir une augmentation rapide des temps de calcul avec l'augmentation des dimensions du problème. De plus, on remarque qu'une méthode passe mieux à l'échelle avec l'augmentation du nombre de bus. La Fig. 3.8 montre que cela est dû à des itérations plus rapides à calculer (puisque les deux méthodes ont besoin du même nombre d'itérations). En effet, les deux méthodes ont besoin d'autant d'itérations pour converger. On peut aussi remarquer que le nombre d'itérations augmente avec les dimensions du problème.

5 Marché endogène

Dans cette section, les algorithmes présentés précédemment seront appliqués pour résoudre le problème de **marché endogène**. La première manière présentée sera le fait de relaxer les contraintes de réseau dans la fonction coût (**EndoPF**). Ensuite, on résoudra en réalisant un consensus entre un **OPF** et un marché pair à pair (**EndoConsensus**). Pour enfin montrer une résolution directe (**EndoDirect**) dans le cas d'un réseau radial.

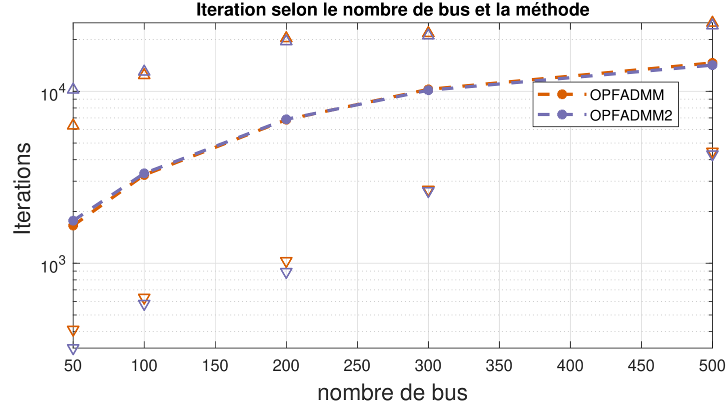


FIGURE 3.8 – Nombre d'itérations mesuré des différents algorithmes

5.1 Résolution avec relaxation d'un PF

Dans cette partie, on notera $G_{PF}(P^{agent}) \leftarrow Y$ la fonction qui à la puissance des agents associe les grandeurs dans le réseau. Dans le cas général, on a donc, pour tous les bus et lignes, $G_{PF}(P^{agent}) = Y = (\theta, V, \phi)$. On cherche donc à respecter les contraintes réseau en ayant $|G_{PF}(P^{agent})| < \overline{Y}$. On va chercher à utiliser une ADMM pour relaxer cette contrainte. Pour cela, on transforme les inégalités en égalité grâce aux variables *slack* :

$$-\mathbf{G}_{PF}(\mathbf{P}^{agent}) + \overline{\mathbf{Y}} - \mathbf{z}_1 = \mathbf{0}|_{\mathbf{\Delta}_1}, \ \mathbf{z}_1 \geq \mathbf{0} \tag{3.75a}$$
$$\mathbf{G}_{PF}(\mathbf{P}^{agent})) + \overline{\mathbf{Y}} - \mathbf{z}_2 = \mathbf{0}|_{\mathbf{\Delta}_2}, \ \mathbf{z}_2 \geq \mathbf{0} \tag{3.75b}$$

On peut utiliser une contrainte sous la forme $|G_{PF}(P^{agent})| < \overline{Y}$, car les contraintes du réseau sont symétrique. En effet, les signes du déphasage ou des flux de puissances ne sont que des conventions et, pour chaque bus, on accepte un écart avec la tension de référence (écart positif ou négatif).

Les Lagrangiens augmentés associés aux contraintes sont (on considère un seul facteur de pénalité pour toutes les contraintes) :

$$L_{add}(\mathbf{P}, \mathbf{\Delta}_1, \mathbf{\Delta}_2) = \frac{\rho_1}{2}(||-\mathbf{G}(\mathbf{P})+\overline{\mathbf{Y}}-\mathbf{z}_1+\mathbf{\Delta}_1||^2+||\mathbf{G}(\mathbf{P}_n)+\overline{\mathbf{Y}}-\mathbf{z}_2+\mathbf{\Delta}_2||^2) \tag{3.76}$$

On peut remarquer qu'en enlevant les termes n'intervenant pas dans la mini-

misation on peut écrire :

$$||-\mathbf{G}(\mathbf{P})+\overline{\mathbf{Y}}-\mathbf{z}_1+\mathbf{\Delta}_1||^2 = \sum_{h\in H}\Big(\overline{Y_h}-z_{1h}+\delta_{1h}-G(P)_h\Big)^2$$
$$= \sum_{h\in H}\Big((G(P)_h)^2 - 2\cdot G(P)_h\cdot(\overline{Y_h}-z_{1h}+\delta_{1h})\Big) \quad (3.77)$$

Avec les mises à jour suivantes :

$$\mathbf{z}_1^{k+1} = max(\mathbf{0}, -\mathbf{G}(\mathbf{P})+\overline{\mathbf{Y}}+\mathbf{\Delta}_1) \quad (3.78a)$$
$$\mathbf{z}_2^{k+1} = max(\mathbf{0}, \mathbf{G}(\mathbf{P})+\overline{\mathbf{Y}}+\mathbf{\Delta}_2) \quad (3.78b)$$
$$\mathbf{\Delta}_1^{k+1} = \mathbf{\Delta}_1^k - \mathbf{G}(\mathbf{P})+\overline{\mathbf{Y}}-\mathbf{z}_1 \quad (3.78c)$$
$$\mathbf{\Delta}_2^{k+1} = \mathbf{\Delta}_2^k + \mathbf{G}(\mathbf{P})+\overline{\mathbf{Y}}-\mathbf{z}_2 \quad (3.78d)$$

Dans le cas général, les expressions $(G(P_n)_h)^2$ et $G(P_n)_h$ ne peuvent pas être distribuées sur les différents agents. L'idée ici sera de réaliser différentes approximations pour être capable de résoudre l'optimisation de manière décentralisée. Les méthodes qui se basent sur ce principe s'appelleront **EndoPF**.

5.1.1 Approximation DC

Dans cette partie, on utilise l'approximation DC pour simplifier l'expression. En notant $\mathbf{I}$ la matrice de correspondance entre les agents et les bus on obtient que $\mathbf{P}^{bus} = \mathbf{I}\cdot\mathbf{P}^{agent}$. À partir de [26] et de la partie PF on obtient que :

$$\mathbf{G}(\mathbf{P}) = \phi = \mathbf{G}_{sensi}^{bus}\mathbf{P}^{bus} = \mathbf{G}_{sensi}^{bus}\cdot\mathbf{I}\cdot\mathbf{P}^{agent} = \mathbf{G}_{sensi}^{agent}\cdot\mathbf{P}^{agent} \quad (3.79)$$

Ceci nous permet de simplifier les équations précédentes :

$$[G(P_n)_l]^2 = \big(\sum_{n\in\Omega} G_{ln}p_n\big)^2 = \sum_{n\in\Omega}(G_{ln}p_n)^2 + 2\sum_{n\in\Omega}\sum_{i>n}(G_{ln}p_n)\cdot(G_{li}p_i) \quad (3.80)$$

En considérant les termes dépendant de p_i comme égaux à ceux de l'itération précédente p_j^k, on peut séparer les différents termes relaxés sur chaque agent ce qui donne après simplification :

$$\mathbf{T}_n^{k+1} = \underset{T_n}{\operatorname{argmin}}\, g_n(p_n) + \sum_{m\in\omega_n}\beta_{nm}t_{nm} + \frac{\rho}{2}\sum_{m\in\omega_n}(t_{nm}-z_{nm}+\lambda_{nm})^2$$
$$+ \rho_1\sum_{l\in L}\Big[(G_{ln}p_n)^2 + \big(z_1^k - z_2^k + \delta_2^k - \delta_1^k + 2\sum_{i>n}G_{li}p_i^k\big)G_{ln}p_n\Big] \quad (3.81)$$

En notant toute la partie provenant du marché endogène (*i.e.* la deuxième ligne) $g_{endo}(p_n)$ on peut remarquer que l'on peut résoudre de la même manière qu'un marché via une ADMM sous la forme de partage :

$$\begin{aligned} \mathbf{T}_n^{k+1} = \underset{t_{nm}}{\text{argmin}} \quad & g\Big(\sum_{m\in\omega_n} t_{nm}\Big) + \sum_{m\in\omega_n} f_{nm}(t_{nm}) \\ \text{t.q.} \quad & t_{nm} \in \mathbf{C}' \end{aligned} \tag{3.82}$$

avec :

$$\begin{aligned} f_{nm}(t_{nm}) &= \beta_{nm} t_{nm} + \frac{\rho}{2}\Big(t_{nm} - \frac{t_{nm}^k - t_{mn}^k}{2} + \lambda^k\Big)^2 \\ g\Big(\sum_{m\in\omega_n} t_{nm}\Big) &= g_n\Big(\sum_{m\in\omega_n} t_{nm}\Big) + g_{endo}\Big(\sum_{m\in\omega_n} t_{nm}\Big) \end{aligned} \tag{3.83}$$

Tout comme pour la résolution d'un marché pair à pair, on peut résoudre ce problème de partage en suivant itérativement les étapes suivantes pour chaque agent n :

$$t_i^{j+1} = \underset{lb_n<t_i<ub_n}{\text{argmin}} \Big(f(t_i) + \frac{\rho_l}{2}\Big\|t_i - t_i^j + \overline{t}^j - \tilde{p}^j + \mu^j\Big\|_2^2\Big) \tag{3.84a}$$

$$\tilde{p}^{j+1} = \underset{\underline{p_n}<M_n\tilde{p}<\overline{p_n}}{\text{argmin}} \Big(g(M_n\tilde{p}) + \frac{M_n\rho_l}{2}\Big\|\tilde{p} - \mu^j - \overline{t}^{j+1}\Big\|_2^2\Big) \tag{3.84b}$$

$$\mu^{j+1} = \mu^j + \overline{t}^{j+1} - \tilde{p}^{j+1} \tag{3.84c}$$

$$\begin{aligned} f_i(t_i) &= \frac{\rho}{2}(t_i - \frac{t_{ni}^k - t_{in}^k}{2} + \lambda_{ni}^k)^2 + \beta_{ni}\cdot t_i \\ g(M_n\tilde{p}) &= M_n^2 \cdot \Big[0.5a_n + \rho_1 \sum_{l\in L} G_{ln}^2\Big]\cdot \tilde{p}^2 \\ &+ M_n\Big[b_n + \rho_1\sum_{l\in L}\big(z_1^k - z_2^k + \delta_2^k - \delta_1^k + 2\sum_{j>n} G_{lj}p_j^k)G_{ln}\Big)\Big]\tilde{p} \end{aligned} \tag{3.85}$$

On remarque que l'on doit réaliser la minimisation de deux fonctions scalaires de la forme :

$$\sum_j 0.5\cdot a_j \cdot (y - b_j)^2 + \sum_j c_j \cdot y \tag{3.86}$$

La majorité des coefficients sont constants. En utilisant respectivement les indices t et p pour les coefficients des équations (3.84a) et (3.84b), les coefficients qui vont

changer selon l'itération globale (k) ou locale (j) sont les suivants :

$$b_{t1} = 0.5(t_{ni}^k - t_{in}^k) - \lambda_{ni}^k \tag{3.87a}$$
$$b_{t2} = t_i^j - \bar{t}^j + \tilde{p}^j - \mu^j \tag{3.87b}$$
$$b_{p1} = \mu^j + \bar{t}^{j+1} \tag{3.87c}$$
$$c_{p2} = \rho_1 M_n \sum_{l \in \mathcal{L}} \left(z_1^k - z_2^k + \delta_2^k - \delta_1^k + 2 \sum_{j>n} \left(G_{lj} p_j^k \right) G_{ln} \right) \tag{3.87d}$$

Et l'ensemble des coefficients constants sont les suivants :

$$a_{t1} = \rho \tag{3.88a}$$
$$a_{t2} = \rho_l \tag{3.88b}$$
$$c_{t1} = \beta_{ni} \tag{3.88c}$$
$$a_{p1} = M_n \cdot \rho_l \tag{3.88d}$$
$$a_{p2} = M_n^2 \cdot \left[a_n + 2 * \rho_1 \sum_{l \in L} G_{ln}^2 \right] \tag{3.88e}$$
$$c_{p1} = M_n \cdot b_n \tag{3.88f}$$

Des résidus peuvent être calculés pour vérifier la convergence anticipée de l'algorithme :

$$\begin{aligned} r_l^{j+1} &= \left\| \bar{t}^{j+1} - \tilde{p}^{j+1} \right\| \\ s_l^{j+1} &= \left\| t_i^{j+1} - t_i^j \right\| \end{aligned} \tag{3.89}$$

Pour prendre en compte les bornes sur les variables, les deux solutions t_i et $\tilde{p}$ doivent être respectivement projetées dans leur intervalle admissible $[lb, ub]$ (qui dépend du type de l'agent) et $[\underline{p_n}/M_n, \overline{p_n}/M_n]$.

Pour simplifier les notations, les variables suivantes sont rajoutées, avec Π^- étant la projection dans les nombres négatifs :

$$\alpha_{ln}^k = G_{ln} p_n^k \tag{3.90}$$

$$Q_l^{tot} = \sum_{n \in \Omega} \alpha_{ln} = \phi_l \ ; Q_{ln}^{part} = \sum_{j>n} \alpha_{lj} \tag{3.91}$$

$$\begin{aligned} \kappa_{1l}^k &= \bar{l}_l + \delta_{l1}^{k-1} - \sum_{n \in \Omega} G_{ln} p_n^k = \bar{l}_l + \Pi^-(\kappa_{1l}^{k-1}) - Q_l^{tot} \\ \kappa_{2l}^k &= \bar{l}_l + \delta_{l2}^{k-1} + \sum_{n \in \Omega} G_{ln} p_n^k = \bar{l}_l + \Pi^-(\kappa_{2l}^{k-1}) + Q_l^{tot} \end{aligned} \tag{3.92}$$

Les résidus indiquant la convergence de l'algorithme sont les suivants :

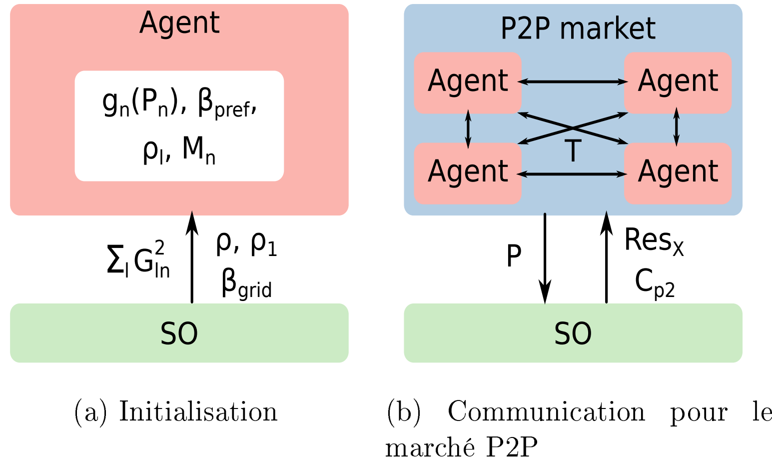
$$\begin{aligned} r^{k+1} &= \left\| t_{nm}^{k+1} + t_{mn}^{k+1} \right\| \\ s^{k+1} &= \left\| t_{nm}^{k+1} - t_{nm}^{k} \right\| \\ x^{k+1} &= \left\| [\Pi^-(\kappa_1^{k+1}) - \Pi^-(\kappa_1^k)]^2 + [\Pi^-(\kappa_2^{k+1}) - \Pi^-(\kappa_2^k)]^2 \right\| \end{aligned} \tag{3.93}$$

entrée : j_{max}, ub_n, lb_n, peers
sortie : t_{min}

1 **while** $((r, s, x) > \epsilon_{g,x}$ and $k < k_{max})$ **do**
2 **while** $((r_l, s_l) > \epsilon_l$ and $j < j_{max})$ **do**
3   $B_{t2} \leftarrow$ (3.87b) ;
4   $T^{j+1} \leftarrow$ (3.39a) ;
5   $T^{j+1} \leftarrow max(min(T^{j+1}, Ub), Lb)$;
6   $\overline{T} \leftarrow mean(T^{j+1})$;
7   $B_{p1} \leftarrow$ (3.87c) ;
8   $\tilde{p} \leftarrow$ (3.39b);
9   $\tilde{p} \leftarrow max(min(\tilde{p}, \overline{p_n}/M_n), \underline{p_n}/M_n)$;
10 **end**
11 $\mu \leftarrow$ (3.39c) ;
12 $(s_l^{k+1}, r_l^{k+1}) \leftarrow$ (3.44) ;
13 $\Lambda, B_{t1} \leftarrow$ (3.33), (3.87a) ;
14 $p_n^{k+1} \leftarrow \overline{T} \cdot M_n$;
15 $(\alpha_{ln}^{k+1}, Q^{partial}, Q^{tot}) \leftarrow$ ((3.90),(3.91)) ;
16 $(\kappa_{1l}^k, \kappa_{2l}^k) \leftarrow$ (3.92);
17 $C_{p2} \leftarrow$ (3.87d) ;
18 $(r^{k+1}, s^{k+1}, x^{k+1}) \leftarrow$ (3.93) ;
19 **end**

Algorithme 7 : Algorithme DC - EndoPF

L'algorithme complet est présenté dans Alg. 7. Pour se représenter le fonctionnement de cet algorithme s'il était implémenté en vrai, la figure Fig. 3.9 représente l'initialisation et les échanges pour chaque étape de l'algorithme. Lorsqu'un nouvel agent rejoint le marché, Fig : 3.9a, le gestionnaire de réseau (SO) lui envoie les constantes dont l'agent a besoin, tandis que le SO n'a besoin que de la localisation de l'agent. Pendant le calcul Fig : 3.9b, chaque agent envoie ses échanges avec les autres (comme un marché P2P classique) et envoie sa puissance totale au SO. Le gestionnaire renvoie aux agents un prix de pénalité (3.87d) pour déplacer l'optimum vers une solution qui respecte les contraintes dans les lignes et son résidu (ou juste un booléen) pour indiquer s'il a convergé pour que l'ensemble s'arrête.



(a) Initialisation

(b) Communication pour le marché P2P

FIGURE 3.9 – Fonctionnement du marché P2P DC-endogène

5.1.2 Approximation autour d'un point de fonctionnement

L'idée de cette partie est de remarquer que la mise à jour des variables duales peut se faire directement par l'utilisation d'un PF. Ainsi, la difficulté est de déterminer l'influence de la puissance des agents sur les valeurs de $\mathbf{G}(\mathbf{P})$. La résolution étant itérative, la connaissance exacte de l'influence est inutile, la valeur des puissances allant changer dans la prochaine itération. Ainsi soit $\mathbf{P}^k$ la puissance des agents déterminée à l'itération k, et soit $\mathbf{dP} = \mathbf{P} - \mathbf{P}^k$ une petite variation de puissance permise à l'optimisation ; on a en faisant un développement limité au premier ordre :

$$\mathbf{G}(\mathbf{P}^k + \mathbf{dP}) = \mathbf{G}(\mathbf{P}^k) + \frac{\mathbf{dG}(\mathbf{P^k})}{\mathbf{dP}} \cdot \mathbf{dP} \tag{3.94}$$

Par analogie avec l'approximation DC où l'on avait $\mathbf{Y} = \mathbf{G}(\mathbf{P}) = \mathbf{G}_{sensi}\mathbf{P}$, on notera la dérivée $\frac{\mathbf{dG}(\mathbf{P^k})}{\mathbf{dP}} = \mathbf{G}^k_{sensi}$. Le résultat du **PF** à l'itération k est noté $\mathbf{G}(\mathbf{P}^k) = \mathbf{Y}^k$. On va pouvoir substituer cette équation dans la fonction coût. On peut remarquer que, dans l'expression, seul le terme $\mathbf{G}^k_{sensi} \cdot \mathbf{P}$ a une influence sur l'optimum du terme linéaire. Ainsi, on peut écrire que :

$$\begin{aligned}\underset{\mathbf{P}}{\operatorname{argmin}}\, \mathbf{G}(\mathbf{P}) &= \underset{\mathbf{P}}{\operatorname{argmin}}\, \mathbf{G}(\mathbf{P}^k + \mathbf{dP}) = \underset{\mathbf{P}}{\operatorname{argmin}}\, \mathbf{Y}^k + \frac{\mathbf{G}(\mathbf{P}^k)}{\mathbf{dP}} \cdot \mathbf{dP} \\ &= \underset{\mathbf{P}}{\operatorname{argmin}}\, \mathbf{G}^k\mathbf{dP} = \underset{\mathbf{P}}{\operatorname{argmin}}\, \mathbf{G}^k\mathbf{P}\end{aligned} \tag{3.95}$$

Le terme quadratique nous donne :

$$\begin{aligned}[G(P^k + dP)_h]^2 &= [Y_h^k + \sum_n G_{hn}^k \cdot dP_n]^2 \\ &= [\sum_n (G_{hn}^k dP_n)]^2 + 2Y_h^k \sum_n G_{hn}^k \cdot dP_n\end{aligned} \tag{3.96}$$

Le premier terme est quadratique par rapport à dP et peut donc être négligé. La fonction à minimiser devient donc :

$$||\overline{\mathbf{Y}} - \mathbf{G}(\mathbf{P}) - \mathbf{z}_1 + \mathbf{\Delta}_1||^2 = 2\sum_{h\in H}\Big(Y_h^k(G^kP)_h - (G^kP)_h(\overline{Y_h} - z_{1h} + \delta_{1h})\Big) \tag{3.97}$$

$$||\overline{\mathbf{Y}} + \mathbf{G}(\mathbf{P}) - \mathbf{z}_2 + \mathbf{\Delta}_2||^2 = 2\sum_{h\in H}\Big(Y_h^k(G^kP)_h + (G^kP)_h(\overline{Y_h} - z_{2h} + \delta_{2h})\Big) \tag{3.98}$$

La fonction coût à optimiser devient donc :

$$\begin{aligned}\mathbf{T}_n^{k+1} = &\underset{T_n}{\operatorname{argmin}}\, g_n(p_n) + \sum_{m\in\omega_n}\beta_{nm}t_{nm} + \frac{\rho}{2}\sum_{m\in\omega_n}(t_{nm} - z_{nm} + \lambda_{nm})^2 \\ &+ \frac{\rho_1}{2}\sum_{l\in L}\Big[\big(z_1^k - z_2^k + \delta_2^k - \delta_1^k + 4Y_L^k\big)G_{ln}^k p_n\Big]\end{aligned} \tag{3.99}$$

Cette fonction peut se résoudre via une ADMM de partage. Les coefficients qui diffèrent sont :

$$c_{p2} = \rho_1 M_n \sum_{l\in\mathcal{L}}(z_1^k - z_2^k + \delta_2^k - \delta_1^k + 2Y^k)G_{ln}^k \tag{3.100a}$$

$$a_{p2} = M_n^2 a_n \tag{3.100b}$$

Détermination de la sensibilité $\mathbf{G}^k$ Le calcul de cette dérivée n'est pas trivial puisque l'on n'a pas accès à la relation analytique entre les grandeurs physiques du réseau et les puissances.

En linéarisant au premier ordre on sait que $\mathbf{dW} = \mathbf{Jac}\cdot\mathbf{dE}$ (section 2.3). On a $\mathbf{dP}^{agent} = \mathbf{IdW}$ et en inversant la Jacobienne on peut obtenir que $\mathbf{dE} = \mathbf{Jac}^{-1}\mathbf{dW} = \mathbf{Jac}^{-1}\cdot\mathbf{I}\cdot\mathbf{dP}^{agent}$. De plus on a $\frac{\partial\phi}{\partial P} = \frac{\partial\phi}{\partial E}\cdot\frac{\partial E}{\partial P}$. Ces relations nous permettent de déterminer la matrice $\mathbf{G}^k$. Cependant ce calcul demande une inversion de matrice, ce qui n'est pas adapté au fait de faire une résolution itérative (celle-ci changeant à chaque pas).

Dans notre cas, pour limiter les calculs, la sensibilité sera déterminée avec une dérivée numérique. Avec la méthode d'Euler explicite. Le calcul de la dérivée devient $\mathbf{G}^k = \frac{dG(P^k)}{dP} = \frac{\mathbf{G}(\mathbf{P}^k)-\mathbf{G}(\mathbf{P}^{k-1})}{\mathbf{P}^k-\mathbf{P}^{k-1}} = \frac{\mathbf{Y}^k-\mathbf{Y}^{k-1}}{\mathbf{P}^k-\mathbf{P}^{k-1}}$.

Cette méthode sera appelée dans la suite **AC-EndoPF**.

La formulation actuelle ne prend pas en compte les pertes dans les lignes. Pour que cela soit le cas, il faut récupérer les pertes dans les lignes pendant le **PF** et utiliser l'agent des pertes pour forcer les producteurs à produire plus pour compenser ses pertes.

5.2 Résolution par consensus avec un OPF

Le marché endogène peut être résolu en réalisant un consensus entre un OPF et un marché pair à pair (**EndoConsensus**). On considère le système suivant [21] :
- un marché P2P et un SO qui vont essayer d'atteindre un consensus sur les puissances à injecter et à soutirer (3.101) ;
- le marché P2P résout le problème (3.24) (marché sans contraintes réseau) auquel la relaxation de la contrainte de consensus a été ajoutée ;
- le SO résout le problème (3.57) (un OPF) avec pour fonction coût la relaxation de la contrainte de consensus.

La contrainte qui est ajoutée est donc la suivante :

$$p_n^{SO} = p_n^{P2P} \qquad (\eta_{SO}) \qquad n \in \Omega \tag{3.101}$$

L'idée est d'utiliser une ADMM sous forme de consensus pour partager la variable de puissance de l'agent n entre l'agent n et le **SO**. La variable duale associée du point de vue de l'agent est :

$$\eta_{SO}^{k+1} = \eta_{SO}^{k} + \frac{1}{2}(p_n^{[n]} - p_n^{[SO]}) \tag{3.102}$$

Pour obtenir la variable duale du point de vue du SO, il faut prendre l'opposé de celle de l'agent.

Cette résolution a plusieurs avantages.
- Elle permet la séparation concrète entre le marché décentralisé et la gestion du réseau.
- Cette organisation permet la résolution de différents marchés indépendants en parallèle avec l'OPF, qui centralise l'ensemble des résultats pour vérifier que le réseau tienne.
- Cette résolution peut se faire, quel que soit l'algorithme utilisé, du côté du marché ou de l'OPF.

On peut écrire que résoudre le problème de Marché revient à minimiser la

fonction suivante :

$$\begin{aligned} \mathbf{T}_n^{k+1} = \underset{\mathbf{T}_n}{\text{argmin}} \quad & g_n(p_n) + \sum_{m \in \omega_n} \beta_{nm} t_{nm} \\ & + \sum_{m \in \omega_n} \frac{\rho}{2} \left(t_{nm} - \frac{t_{nm}^k - t_{mn}^k}{2} + \frac{\lambda_{nm}^k}{\rho} \right)^2 \\ & + \frac{\rho_{SO}}{2} (p_n - \frac{p_n^{SO\ k} + p_n^k}{2} + \eta_{SO}^{[n],k})^2 \\ \text{t.q.} \quad & (3.24b) - (3.24g) \end{aligned} \tag{3.103}$$

La résolution peut donc être la même que le marché pair de ce chapitre (section 3.3). Les seules différences sont les suivantes en AC :

— il faut rajouter un agent dans le marché pour acheter les pertes ;
— il faut rajouter des coefficients a_{p3} et b_{p3}

Le coefficient a_{p3} est constant et prend pour valeur :

$$a_{p3} = M_n^2 \rho_{SO} \tag{3.104}$$

L'autre coefficient variera toutes les itérations globales k :

$$b_{p3} = (\frac{p_n^{SO\ k} + p_n^k}{2} - \eta_{SO}^k)/M_n \tag{3.105}$$

Le nombre de voisins de l'agent n noté M_n provient du fait que l'on fait une ADMM sous forme de partage et que donc on n'optimise pas la puissance, mais la moyenne de cette puissance.

Pour rappel la modélisation de l'agent devant acheter les pertes se fait ainsi :

— C'est un consommateur donc il partage les caractéristiques des consommateurs (relié aux producteurs, $u_b = 0$).
— On ne cherche pas à donner un coût aux pertes donc $b_n = 0$, $a_n = 0$, $\overline{p_n} = \underline{p_n} = P_{pertes}$
— il n'a pas de contrainte sur la puissance échangée : $l_b = -\infty$

Comme l'infini est une valeur pouvant avoir de mauvais comportements en numérique, on choisira en réalité une valeur finie très grande qui devrait ne jamais être atteinte en tant que perte dans les lignes. Cela nous permet de ne pas changer la valeur maximale des échanges à chaque itération.

Même si l'on réalise un consensus entre le marché et l'OPF, le comportement n'est pas symétrique pour la gestion des pertes. En effet, cette puissance influence le marché, mais la valeur obtenue par l'agent lors du marché n'affecte pas l'OPF. En effet, on ne souhaite pas que les producteurs incitent l'ensemble à avoir plus de pertes en ligne, pour produire plus.

Tandis que le problème du SO peut s'écrire ainsi (par abus de notation $P_0^{[SO]} = P_{loss}^{[SO]}$) :

$$\begin{aligned} \mathbf{P}^{k+1} = & \underset{\mathbf{P}}{\text{argmin}} \frac{\rho_{SO}}{2} \sum_{n \in \omega} (p_n - \frac{p_n^{SO\ k} + p_n^k}{2} + \eta_{SO}^{[SO],k})^2 \\ & \text{t.q.} \quad (3.58a) - (3.58g) \end{aligned} \tag{3.106}$$

C'est un OPF tout à fait classique qui peut se résoudre de manière centralisée ou décentralisée. Cependant, il y a plusieurs points de différence. Tout d'abord, le **SO** n'a aucune raison de connaitre les puissances limites des agents. Ainsi, il faut imposer des valeurs non atteignables à $\overline{p_n}$ et $\underline{p_n}$. Selon l'algorithme, il peut être plus intéressant d'enlever les contraintes correspondantes. La contrainte $p_{loss}^{[SO]} = -\sum_n p_n^{[SO]}$ doit être ajoutée.

Dans le cas radial, cela peut se faire en gardant la même forme de problème que **OPFADMM**. Pour cela, on rajoute un "bus fictif" qui devra respecter la contrainte sur les puissances. Cela modifie légèrement les valeurs de $\hat{p}$ et $\hat{q}$, mais la résolution reste identique.

On peut aussi imposer que $p_{loss}^{[SO]} = -\sum_i z_i * l_i$. Cela permet que le calcul des pertes ne soit pas influencé par le déséquilibre en puissance pendant la résolution. Cependant à cause de la relaxation, il est possible que le courant ne soit plus lié aux puissances et donc que cette relation soit fausse.

5.3 Résolution directe

Pour la résolution de ce problème, on se placera dans un réseau radial. On résoudra de la même manière que le problème d'OPF présenté précédemment avec la relaxation en *Second cone order*, dans la version où l'on a substitué les puissances des bus par celle des agents. On appellera cette méthode **EndoDirect**. Pour rappel

le problème à résoudre est donc :

$$\begin{aligned}
\min_{\mathbf{T},\mathbf{P}} \quad & \sum_{n\in\Omega}\Big(g_n(p_n,q_n)+\sum_{m\in\omega_n}\beta_{nm}t_{nm}\Big) && && (3.107a)\\
\text{s.t.} \quad & \mathbf{T}=-{}^t\mathbf{T} \qquad (\Lambda) && && (3.107b)\\
& (p_n,q_n)=(\sum_{m\in\omega_n}t^p_{nm},\sum_{m\in\omega_n}t^q_{nm}) \qquad (\mu) && n\in\Omega && (3.107c)\\
& (\underline{p_n},\underline{q_n})\le(p_n,q_n)\le(\overline{p_n},\overline{q_n}) && n\in\Omega && (3.107d)\\
& lb\le t_{nm}\le ub && n\in\Omega_p && (3.107e)\\
& \underline{V_b}\le V_b\le\overline{V_b} && b\in\mathcal{B} && (3.107f)\\
& \underline{l_{ij}}\le l_{ij}\le\overline{l_{ij}} \qquad (\delta_{S1-2}) && (i,j)\in\mathcal{L} && (3.107g)\\
& P_{ij}=R_{ij}l_{ij}-\sum_{n_i n\mathcal{N}_j}p_n^{agent}+\sum_{m:j->m}P_{jm} && (i,j)\in\mathcal{L} && (3.107h)\\
& Q_{ij}=X_{ij}l_{ij}-\sum_{n_i n\mathcal{N}_j}q_n^{agent}+\sum_{m:j->m}Q_{jm} && (i,j)\in\mathcal{L} && (3.107i)\\
& V_i^2=V_j^2-2(R_{ij}P_{ij}+X_{ij}Q_{ij})+(R_{ij}^2+X_{ij}^2)l_{ij} && b\in\mathcal{B} && (3.107j)\\
& l_{ij}V_i^2\le P_{ij}^2+Q_{ij}^2 && && (3.107k)\\
& V_{ref}=V_0 && \textit{reference} && (3.107l)
\end{aligned}$$

Tout comme pour la résolution de l'OPF de la section précédente, on décentralisera sur les bus. Pour cela, comme pour les **OPF**, on devra résoudre une optimisation itérative. La minimisation de la variable duale Y permet de prendre en compte les contraintes entre les bus et agents.

On pose comme variable $\mathbf{x}_i=(t_{nm},p_n,v_i,S_i,l_i)$ $\forall i\in\mathcal{B}$, $\forall n\in\mathcal{N}_b$, $\forall m\in\omega_n$ et $\mathbf{x}_{Loss}=(p_{loss},q_{loss},t_{0n})$. Ainsi, par rapport à l'OPF, on rajoute les échanges économiques.

L'algorithme de résolution est donc de la même forme que pour les **OPF** :

$$x^{k+1}=\underset{x\in K_x}{\text{argmin}}\,L_\rho(x,y^k,\mu^k) \tag{3.108a}$$

$$y^{k+1}=\underset{y\in K_y}{\text{argmin}}\,L_\rho(x^{k+1},y,\mu^k) \tag{3.108b}$$

$$\mu^{k+1}=\mu^k+\rho(x^{k+1}-y^{k+1}) \tag{3.108c}$$

On remarque que les différentes minimisations peuvent être séparées sur les différents bus i, car toutes les contraintes et fonctions coûts peuvent être séparées sur les bus. En effet sur la minimisation de la variable slack, les contraintes entre les bus concernent les copies des variables et chaque bus a les copies dont il a besoin

tout comme c'était le cas lors de l'OPF. Le détail des résolutions est disponible en Annexe.

Plutôt que de garder les variables des échanges dans les vecteurs X et Y pour chaque bus, elles seront stockées à part. Ceci permet de récupérer la résolution des marchés pair à pair tel quel. En effet, cette résolution pour le problème duale revient à appliquer une ADMM sous forme de consensus sur les échanges.

5.4 Validation fonctionnelle

Pour vérifier la validité de nos méthodes, celles-ci ont été testées sur des cas fixes. Comme deux méthodes ne sont utilisables que sur des réseaux radiaux, la majorité des tests aura lieu sur ces cas. L'ensemble des cas a été initialisé en résolvant un marché pair à pair pour fixer les puissances et échanges des agents. Ensuite, tout comme pour l'OPF, les tensions sont initialisées avec des tensions plates ($e_i = 1$ et $f_i = 0$ pour tous les bus i). Les pertes étant faibles, les précisions demandées sont de $\epsilon_x = 1e^{-4}$ et de $\epsilon_g = 5e^{-3}$. Le facteur de pénalité est fixé à 1 pour tous les cas, sauf pour *MatPower30* et *TestFeeder* où il vaut respectivement 200 et 10. Le facteur de pénalité de la méthode **AC-EndoPF** est multiplié par 20 pour tous les cas. Ces facteurs de pénalité ne sont pas optimaux, mais permettent la convergence. Il est important de noter que, dans le cas de non-convergence, le changement de facteur de pénalité (à 1, 10, 15 ou 20) n'a pas l'air de changer la convergence. La notation X indique que la résolution n'est pas possible (un réseau maillé pour des méthodes sur réseau radial). Tandis que $*$ indique que la méthode ne converge pas quel que soit le nombre d'itérations permises.

TABLE 3.8 – Nombre d'itération et temps (ms) pour converger pour les différents algorithmes et cas

Cas	EndoConsensus		EndoDirect		AC-EndoPF		DC-EndoPF	
Nom	temps	iter	temps	iter	temps	iter	temps	iter
Cas 2 bus	21	1613	15	1130	4	76	1.3	21
Cas 3 bus	X	X	X	X	3	10	1.4	15
Matpower 10ba	23	286	25	189	*	*	2	8
Matpower 30	X	X	X	X	50	26	230	570
Matpower 85	$3.0e^3$	906	$2.2e^3$	1752	$1.7e^5$	3859	3	8
TestFeeder	$2.1e^5$	6597	$4.0e^3$	910	*	*	12	11

On peut voir que toutes les méthodes n'ont pas du tout besoin du même nombre

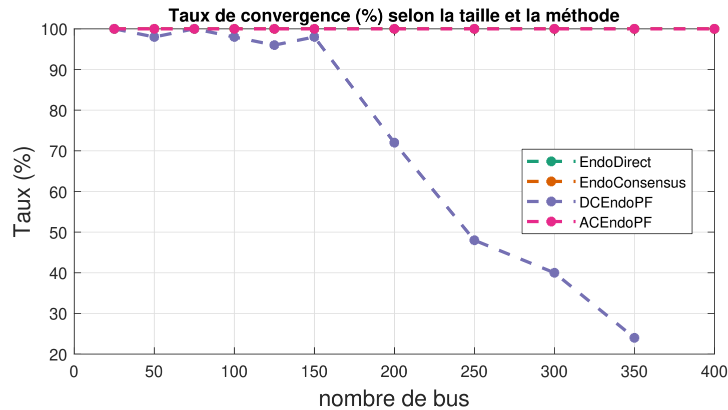


FIGURE 3.10 – Taux de convergence des différents algorithmes

d'itérations pour converger. Les méthodes **EndoPF** ne convergent pas vers le même point, étant des approximations. Cependant, on peut voir que le temps de calcul est extrêmement bas pour ces méthodes. Cette situation est en partie due au fait que tous les cas n'ont pas de restrictions de débit dans les lignes, ce qui revient à résoudre un marché sans contraintes de réseaux pour **DC-EndoPF**. La méthode **AC-EndoPF** ne converge pas sur tous les cas. Notamment elle est incapable de descendre en dessous d'une certaine précision. Par exemple, le cas *MatPower30* avec $\epsilon_g = 1e^{-3}$ ne converge plus.

5.4.1 Passage à l'échelle

L'objectif de cette partie est d'étudier le passage à l'échelle des différentes résolutions du marché endogène. Tout comme précédemment, on génèrera des réseaux radiaux aléatoirement pour comparer les différentes méthodes. Dans un premier temps, on génèrera autant d'agents que de bus. Pour éviter un temps de simulation trop long sur des cas potentiellement infaisables, le nombre d'itérations maximal est fixé à 10000 pour une précision demandée de 0.01.

Le taux de convergence des différents algorithmes en fonction des dimensions du problème est représenté sur la Fig. 3.10. On peut y voir que la méthode **DC-EndoPF** ne converge pas toujours et cela empire avec l'augmentation des dimensions. Comme cette méthode est sur un problème convexe, cela est dû à un mauvais réglage du facteur de pénalité et donc la méthode ne converge pas assez vite. Cela se montre grâce à la Fig. 3.11. Sur cette figure sont représentés les résidus représentant l'avancement de la convergence. Dans notre cas $ResR$ représente le respect de la contrainte d'antisymétrie du marché, $ResS$ représente la convergence, et $ResV$ le respect des contraintes du réseau (ou le consensus avec le gestionnaire de réseau). On peut voir que, pour **DC-EndoPF**, les résidus n'ont pas divergé. Les résidus

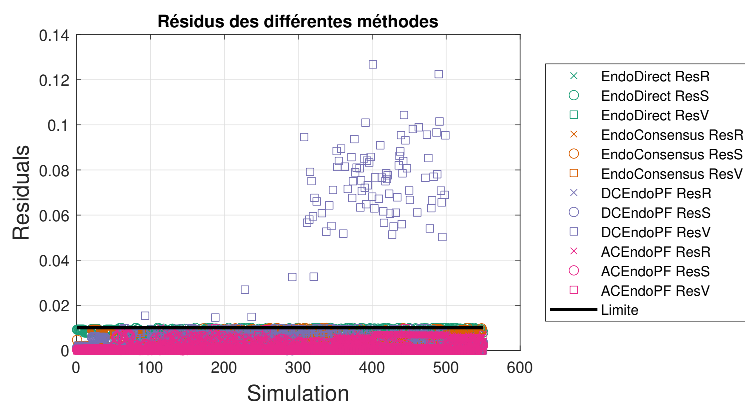


FIGURE 3.11 – Résidus des différentes méthodes sur les cas (par taille croissante en nombre de bus, avec 50 simulations par nombre de bus)

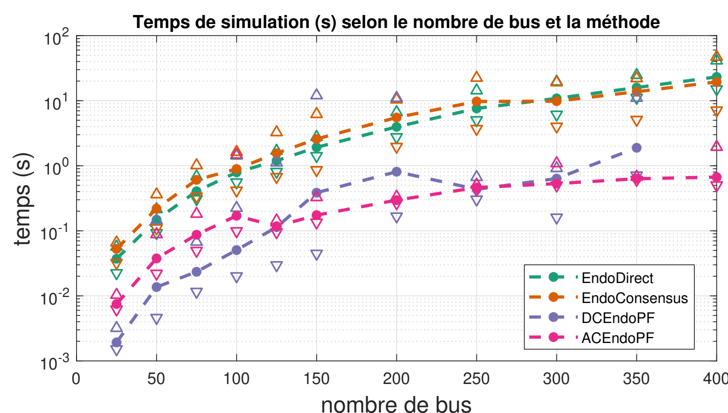


FIGURE 3.12 – Temps de calcul mesuré des différents algorithmes

trop grands sont toujours $ResV$, ce qui signifie que, pour ces cas-là, le facteur de pénalité associé aux contraintes de réseau, ρ_1, est trop faible.

C'est pourquoi les résultants suivants ne concernent que les cas où les méthodes ont convergé. La figure 3.12 représente les temps de simulations des différentes méthodes. On peut y voir une augmentation rapide des temps de calcul avec l'augmentation des dimensions du problème pour les méthodes **EndoConsensus** et **EndoDirect**. Si **EndoConsensus** est la méthode la plus lente à petite dimension, cette méthode devient plus rapide que **EndoDirect** à partir de 300 bus. Pour les autres méthodes, la résolution est beaucoup plus rapide. Cependant, le temps de calcul augmente aussi très rapidement.

On peut y voir aussi une grande variabilité des temps mesurés pour simuler. Si le temps diminué pour **DC-EndoPF** c'est parce que l'on ne regarde que les cas qui ont convergé. Ainsi, il suffit qu'il y ait des cas qui convergent proche du

nombre maximal d'itérations pour que la moyenne soit bien plus haute. En effet, la majorité des simulations avec cette méthode convergent en quelques itérations.

6 Conclusion

Dans cette partie, les algorithmes ont pu être appliqués sur les différents problèmes de la gestion du réseau. L'implémentation de ces problèmes a permis d'effectuer la résolution d'un marché endogène de différentes manières.

L'implémentation sur CPU a démontré la fonctionnalité des algorithmes pour atteindre les optimums pour les problèmes convexes. La méthode **DC-EndoPF** est très sensible au paramétrage des facteurs de pénalité [1]. Ainsi, sa convergence peut être extrêmement lente. Dans le cas où l'optimisation n'est pas convexe (**PF** et **AC-EndoPF**) la convergence n'est pas garantie.

Cependant, quel que soit le problème, le temps de calcul ne permet pas de passer la simulation à l'échelle pour la plupart des algorithmes. Les exceptions sont les PF **Cur** et **BackForPQ** qui ont besoin de moins de 0.1s pour des cas de plus de 1500 bus. Pour tous les autres cas une parallélisation est nécessaire pour réduire le temps de simulation et réduire l'augmentation du temps nécessaire en fonction du nombre d'agents. Afin de permettre une parallélisation efficace, une analyse et une adaptation, des algorithmes sont nécessaires.

Table 3.9 – Résumé de l'ensemble des algorithmes et problème

Problème	Algorithme	Hypothèse	Remarques
Marché	ADMM	DC/AC	Meilleure méthode sur CPU
	OSQP	DC/AC	Solveur open Source
	PAC	DC/AC	beaucoup d'itérations
Power-Flow	NR	AC	peu d'itération
	GS	AC	beaucoup d'itérations
	Curr	AC + radial	
	BackPQ	AC + radial	meilleure vitesse et convergence
Optimal Power Flow	OPFADMM	AC + radial	
	OPFADMM2	AC + radial	légèrement plus rapide
Marché Endogène	EndoDirect	AC + radial	plus rapide que **EndoConsensus**
	EndoConsensus	AC + radial	plus lente que **EndoDirect**
	DC-EndoPF	DC	réglage de ρ et ρ_1 très sensible
	AC-EndoPF	AC	Problème de convergence

1. voir Section 4.2.1 du chapitre 5 pour plus de détail

Le tableau 3.9 regroupe l'ensemble des algorithmes et des problèmes qui ont été considérés. Les hypothèses sur lesquelles les algorithmes sont utilisables et les premiers résultats sur CPU sont aussi précisés.

Chapitre 4

A3 sur une architecture de calcul CPU-GPU

Sommaire

1 **Introduction** **168**
2 **Partitionnement en bloc fonctionnel** **168**
2.1 Marché pair à pair 169
2.2 Power Flow 173
2.3 Optimal Power Flow 176
2.4 Marché endogène 178
3 **Étude de complexité** **182**
3.1 Vision globale 182
3.2 Marché pair à pair 185
3.3 Power Flow 187
3.4 Optimal Power Flow 191
3.5 Marché endogène 195
4 **Réécriture algorithmique pour partitionnement sur CPU-GPU** **197**
4.1 Lien entre le langage de programmation et l'architecture de calcul 197
4.2 Optimisation générale 200
4.3 Prise en compte de la sparsité 201
4.4 Problème de partage 203
4.5 Stockage de la sensibilité 208
4.6 Fonction avec dépendance 215
4.7 Parcours des bus 216
4.8 Calcul de la tension 217
5 **Conclusion** **219**

1 Introduction

Dans le chapitre précédent, les différents algorithmes ont pu être implémentés sur CPU pour résoudre les différents problèmes. À partir des métriques définies dans le chapitre 2, ces algorithmes ont pu être comparés et analysés. Il en ressort ainsi que, pour la majorité des méthodes, un partitionnement CPU-GPU est nécessaire afin de lever le verrou scientifique du temps de calcul rédhibitoire. Afin de réaliser le partitionnement CPU-GPU, les étapes suivantes sont nécessaires.

Tout d'abord, les différents algorithmes seront partitionnés en différents blocs fonctionnels. Ceci nous permettra d'étudier les dépendances entre les différentes opérations pour déterminer comment paralléliser. Ceci permettra aussi en mesurant le temps passé dans chaque bloc d'identifier les goulots d'étranglement des fonctions. C'est sur ces goulots d'étranglement que les efforts d'optimisation doivent être réalisés.

Ensuite, la complexité théorique des différents blocs fonctionnels sera déterminée. Cela nous permettra d'identifier la réduction théorique de complexité induite par la parallélisation sur GPU.

Enfin, le GPU est un matériel connu pour avoir une structure adaptée aux applications hautement parallélisables grâce à ces milliers de cœurs. Cependant, afin de tirer le maximum des capacités du GPU, il est important de prendre en compte les caractéristiques matérielles du GPU dès la conception de l'algorithme et continuer lors de l'implémentation. On détaillera les études et optimisations réalisées pour optimiser les principaux blocs fonctionnels.

2 Partitionnement en bloc fonctionnel

L'objectif de cette partie est de séparer l'algorithme en différents blocs fonctionnels. Cette opération permet de remplir plusieurs objectifs, tels que l'analyse des dépendances entre les différentes parties de l'algorithme en ce qui concerne la séquentialité et de transfert de données. Cela nous permettra ensuite, dans la partie suivante, la détermination de la complexité en série ou parallèle en calcul et en mémoire. L'analyse de la charge de calcul de chaque bloc par la mesure (dans cette section) puis par le calcul, pour en déduire l'intensité algorithmique ; et enfin l'identification des goulots d'étranglement.

De manière générale, quel que soit l'algorithme, celui-ci commence par une **initialisation**. Celle-ci correspond à la création, définition et mise en forme des variables et paramètres en se basant sur les cas d'études. Pour toutes les méthodes, cela correspond à la création de vecteurs regroupant les données nécessaires. C'est à ce stade que la sparsité des échanges du marché ou du réseau est prise en compte pour transformer par exemple la matrice des échanges T et les caractéristiques des

agents en vecteurs. Quelques fois, ces correspondances sont directement réalisées lors de la création du cas d'étude. Pour la suite, on ne précisera pour chaque méthode que ses opérations spécifiques.

Lorsque la résolution possède plusieurs pas temporels, il n'est pas utile de refaire l'initialisation complète. En effet, on considèrera dans ce cas là que le réseau ne change pas, et que seuls les agents changent. Toutes les méthodes possèdent une manière de se mettre à jour entre deux pas temporels en changeant le moins de variables possible.

À la fin de simulation, les résultats sont récupérés. Les résultats sont pour toutes les méthodes (lorsque ces variables existent) :

- la valeur optimale de la variable d'optimisation,
- les variables duales des contraintes,
- le temps de simulation,
- le nombre d'itérations,
- l'évolution des résidus selon les itérations,
- et enfin la valeur de la fonction objectif.

Toutes les autres variables ne sont pas récupérées, mais sont conservées dans l'objet jusqu'à sa destruction.

2.1 Marché pair à pair

2.1.1 Blocs fonctionnels

Dans cette partie, la résolution d'un marché pair à pair décentralisé sera étudiée. Les trois méthodes (**ADMM**, **OSQP**, **PAC**) présentées dans la section 3.3 partagent les mêmes étapes. L'organisation générale est représentée sur la Fig. 4.1.

- **FB 0** est l'**initialisation**. La méthode d'**ADMM** n'a besoin d'aucune opération particulière. Pour pouvoir utiliser **OSQP**, des objets correspondants doivent être créés et initialisés. Dans le cas de **PAC**, il faut aussi inverser les matrices définissant le problème local.
- **FB 1** est la résolution du problème local de chaque agent. Pour l'**ADMM** cela se fait par la répétition des trois étapes (noté a, b, c) permettant la minimisation. Pour **OSQP** cela consiste en l'appel du solveur correspondant. Tandis que, pour **PAC** cette étape est la multiplication d'une matrice et d'un vecteur pour chaque agent, et d'une projection du vecteur dans ses bornes accessibles. Pour finir, il faut calculer $\hat{x}$ et μ pour chaque agent.
- **FB 2** est une étape présente uniquement pour l'**ADMM**. Elle consiste dans le calcul des résidus (pouvant avoir lieu à chaque itération ou moins souvent) du problème local. Dans le cas de **OSQP**, cette étape est incluse dans le solveur et n'est donc pas mesurable. L'algorithme **PAC** résout le problème local avec une forme fermée et n'a donc pas besoin de calculer de

résidus.

— **FB 3** est appelé le problème global. Il consiste en la communication des résultats des problèmes locaux et la mise à jour des variables duales pour toutes les méthodes. C'est-à-dire la mise à jour de λ pour **ADMM** et **OSQP** la mise à jour de ν pour **PAC**. Ensuite il faut mettre à jour des variables nécessaires pour la résolution du problème local de chaque agent. Pour l'**ADMM** cela consiste en la mise à jour de B_{t1}. Pour **OSQP** et **PAC**, il faut calculer Q qui correspond au terme linéaire de l'optimisation quadratique.
— **FB 4** est le calcul des résidus de l'algorithme complet pour les trois méthodes.
— **FB 5** est la récupération et la mise en forme des résultats.

Le problème de marché pair à pair est simulé séquentiellement sur un seul CPU. Ainsi, la "communication" est en réalité juste une lecture de variable correspondante. Dans le cas de résolution parallélisée sur une architecture unique, ces étapes de communications correspondent à une synchronisation et à des lectures/écriture de données. Dans le cas d'un calcul parallèle en cluster, comme celui proposé dans [20] ou de sa mise en œuvre réelle, l'étape de communication exige vraiment un échange de messages entre les unités de calcul.

Le tableau 4.1 regroupe les correspondances entre les équations étudiées dans le chapitre précédent et les blocs fonctionnels. Tandis que la Fig. 4.1 représente les blocs fonctionnels et les variables qui sont calculées.

TABLE 4.1 – Correspondance entre blocs fonctionnels et équations

<table>
<tr><th>Block</th><th>Description</th><th>ADMM</th><th>OSQP</th><th>PAC</th></tr>
<tr><td>FB 0</td><td>Initialisation</td><td></td><td></td><td></td></tr>
<tr><td rowspan="4">FB 1</td><td rowspan="4">Problème local</td><td>(3.42b) , (3.39a)</td><td rowspan="4">(3.32)</td><td>(3.51)</td></tr>
<tr><td>$\overline{T}_n = moy(T_n)\ \forall n$</td><td>(3.52)</td></tr>
<tr><td>(3.42c), (3.39b), (3.39c)</td><td rowspan="2">$\hat{x}$, μ</td></tr>
<tr><td>(3.44)</td></tr>
<tr><td rowspan="2">FB 2</td><td rowspan="2">Problème global</td><td colspan="2">(3.33)</td><td>ν</td></tr>
<tr><td>(3.42a)</td><td>(3.35a)</td><td>(3.49)</td></tr>
<tr><td>FB 3</td><td>Résidus globaux</td><td colspan="2">(3.34)</td><td></td></tr>
<tr><td>FB 4</td><td>Résultats</td><td></td><td></td><td></td></tr>
</table>

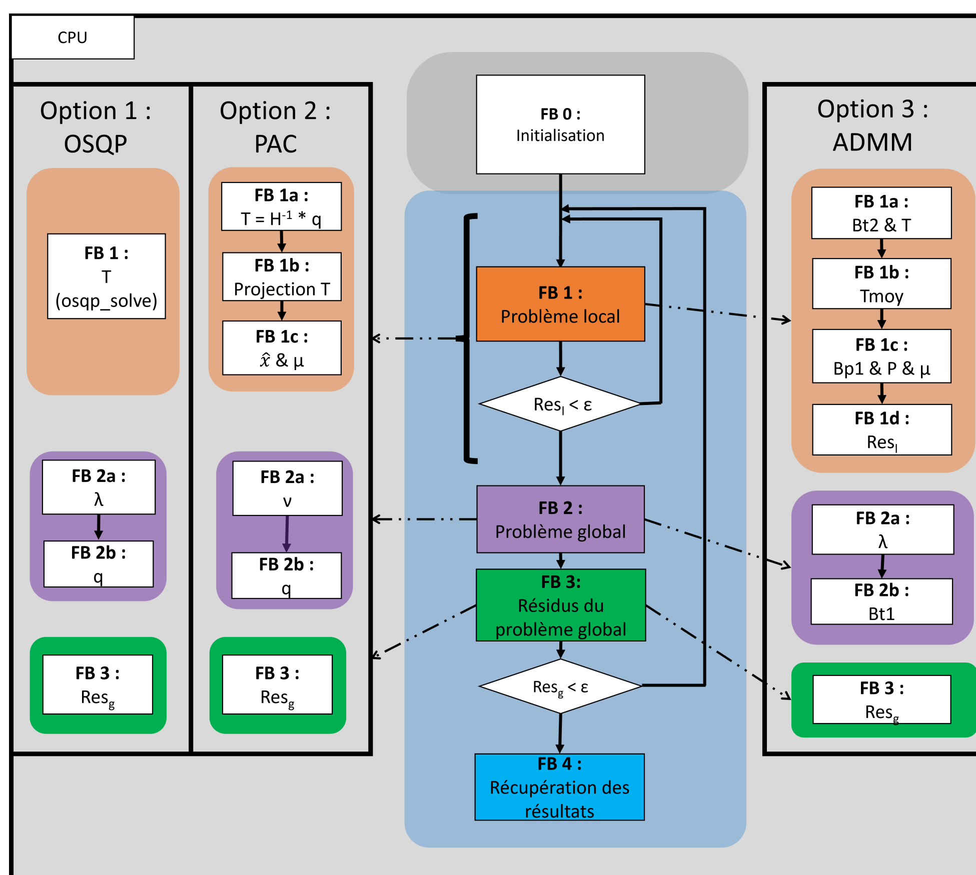
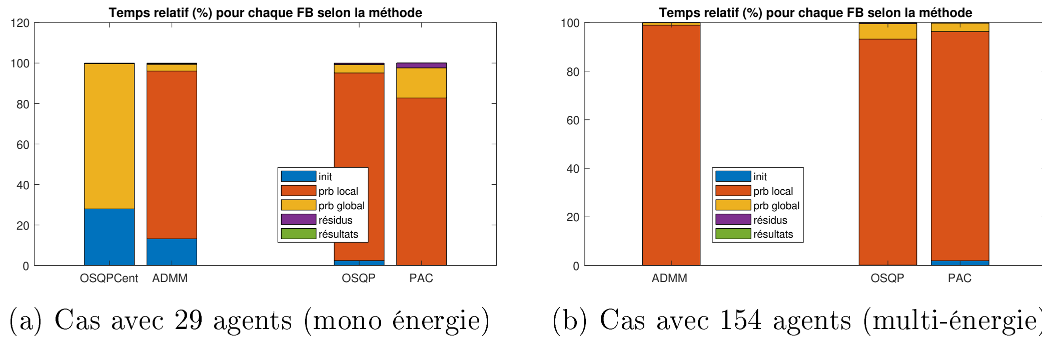


FIGURE 4.1 – Partitionnement en bloc fonctionnel pour le marché pair à pair

2.1.2 Dépendance

Quelle que soit la méthode choisie, chaque bloc fonctionnel a besoin que le précédent soit fini avant de pouvoir être réalisé. L'ensemble du bloc **FB 1** peut se réaliser en parallèle sur les agents. Le calcul des résidus du problème local peut être spécifique à chaque agent ou un seul résidu pour l'ensemble. Pour rappel, dans notre cas, on considère un marché pair à pair synchrone. Cela signifie que le bloc **FB 2** ne peut être effectué par un agent qu'une fois que l'ensemble de ses pairs a fini. Ce marché est complètement connecté, donc cela correspond au fait d'attendre à chaque fois tous les agents. Pour **ADMM** et **OSQP**, le bloc **FB 2b** peut être fait par chaque agent en parallèle et n'a besoin que du résultat du bloc **FB 2a** de l'agent correspondant. Ce qui n'est pas le cas pour **PAC** où une autre communication est nécessaire entre les deux étapes.



(a) Cas avec 29 agents (mono énergie) (b) Cas avec 154 agents (multi-énergie)

FIGURE 4.2 – Répartition des temps de calcul pour les marchés P2P

Marché multiénergie Le marché considéré peut être constitué de plusieurs énergies sans considération des couplages entre elles. Ce cas revient au fait d'avoir plusieurs marchés pair à pair qui n'échangent pas entre eux. Dans ce cas-là, les résolutions sont indépendantes et peuvent donc être réalisées en parallèle sans avoir besoin de synchroniser les blocs fonctionnels.

2.1.3 Répartition de la charge de travail

La figure Fig. 4.2 représente la répartition des temps de calcul entre les différents blocs fonctionnels pour les différentes méthodes. Les temps totaux de calcul sont ceux obtenus lors de la validation fonctionnelle dans la section 3.3.5. Ces répartitions sont obtenues en résolvant le cas 39 nœuds en modélisation DC (c'est-à-dire en échangeant que de la puissance active) ou le cas Matpower 118 avec une précision de $0,001$ [1]. On peut remarquer que, quelle que soit la méthode décentralisée utilisée, la majorité du temps est utilisée pour résoudre le problème local. On remarque que les répartitions sont différentes selon les dimensions du problème. L'initialisation ne nécessite qu'un temps négligeable pour le cas le plus grand.

Ainsi, il est clair que le goulot d'étranglement de cet algorithme se trouve dans la résolution de la minimisation de chaque agent. Paralléliser sur GPU, pour que les résolutions soient réalisées en même temps, devrait réduire grandement les temps de calcul. Cependant, cela n'est pas possible pour la méthode **OSQP**. En effet, celle-ci ne peut pas être implémentée sur GPU, utilisant un solveur sur CPU. Il existe une version sur GPU d'**OSQP** mais elle ne permet pas de lancer plusieurs résolutions en parallèle et n'est plus efficace que sa version CPU qu'à partir d'une taille de 10^5 [107].

La taille pour **OSQP** correspond au nombre de voisins de chaque agent. Une taille de 10^5 ne sera jamais atteinte dans le cadre de cette thèse.

1. Voir Section 2.?? pour plus de détails sur les cas d'études

La parallélisation de **OSQPCen** sur GPU permet bien de réduire les temps de calcul à plus grande dimension. Mais elle reste plus lente que les autres méthodes [110]. Elle ne sera donc pas plus étudiée dans cette thèse.

2.2 Power Flow

2.2.1 Blocs fonctionnels

Cette section présentera le partitionnement en bloc fonctionnel des résolutions de **PF**. Pour rappel on considère 4 types de méthodes qui sont **NR**, **GS**, et en load flow en courant (**Cur**) et en puissance **BackPQ**. Ces méthodes réalisent les mêmes opérations, mais pas dans le même ordre. C'est pourquoi dans cette partie on nommera les blocs plutôt que de les numéroter.

L'initialisation est globalement la même entre les différents algorithmes. Les seules différences notables sont le besoin ou non de calculer certaines constantes telles que $\frac{1}{\underline{Y}_{ii}}$ et $\frac{\underline{Y}_{ij}}{\underline{Y}_{ii}}$ pour **GS**. Dans notre cas les grandeurs d'entrée sont les puissances des agents, il faut donc passer de ces puissances à celles des bus.

Le calcul de la puissance repose sur les équations (3.3), (3.4) qui peuvent être adaptées pour être réalisées avec la tension sous sa forme cartésienne. 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 (dont lui même). Les méthodes **Cur** et **BackPQ** ne réalisent ce calcul qu'une fois la convergence atteinte pour déterminer les puissances au nœud de référence.

Le calcul de la tension, dans le cadre de l'algorithme de **Cur** ou **BackPQ**, correspond à l'équation (3.21) ou (3.22). Pour l'algorithme de **GS** cela correspond à chaque itération à un calcul de (3.16) pour chaque bus. Pour l'algorithme de **NR**, le calcul de tension nécessite dans un premier temps le calcul de la Jacobienne (3.12). Ensuite il faut réaliser la factorisation LU de la Jacobienne. Enfin il faut obtenir le déplacement de la tension en résolvant le système.

Le calcul du flux de **Cur** ou **BackPQ** correspond aux équations (3.17) et (3.19) ou (3.18) (3.20).

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

Le tableau 4.2 regroupe les correspondances entre les équations vues le chapitre précédent et les blocs fonctionnels. Tandis que la Figure 4.3 représente les blocs fonctionnels.

TABLE 4.2 – Blocs fonctionnels

Bloc	NR	GS	Cur	BackPQ
Initialisation				
Flux			(3.17) (3.19)	(3.18) (3.20)
Tension	(3.12) Factorisation Résolution	(3.16)	(3.21)	(3.22)
Puissance	(3.3), (3.4)			
Erreur	(3.13)		(3.23)	

2.2.2 Dépendance

Quelle que soit la méthode considérée, un bloc fonctionnel ne peut être commencé que si l'entièreté du bloc fonctionnel précédent a été terminée. Contrairement au calcul de la tension, le calcul du flux n'a qu'une dépendance partielle pour ses étapes intermédiaires. Cela signifie que l'on pourrait commencer la deuxième étape du calcul du flux avant la fin de la précédente en ne calculant que les lignes qui ont déjà été calculées. Ainsi le parallélisme devra être exploité pour chaque étape indépendamment avec une synchronisation entre chacune.

2.2.3 Répartition de la charge de travail

La figure Fig. 4.4 représente la répartition des temps de calcul entre les différents blocs fonctionnels pour les différentes méthodes. Les temps totaux de calcul sont ceux obtenus lors de la validation fonctionnelle dans la section 3.2.6. Cette simulation a été réalisée sur le cas Matpower 85 (voir section 2.6.2.1) avec la même précision que pour les tests fonctionnels. Les méthodes avec un **D** sont calculées en double précision. On peut voir que la précision des variables n'a aucun effet sur la répartition des temps.

Il est clair que le goulot d'étranglement de **NR** se trouve dans le calcul de la tension. Plus précisément, l'étape nécessitant la majorité du temps de calcul est la résolution du système.

La résolution d'un système linéaire sur GPU est un domaine de recherche en soi [108]. Ainsi une méthode sera implémentée afin d'avoir une résolution sur GPU, mais celle-ci ne sera pas optimisée, cela ayant déjà été fait.

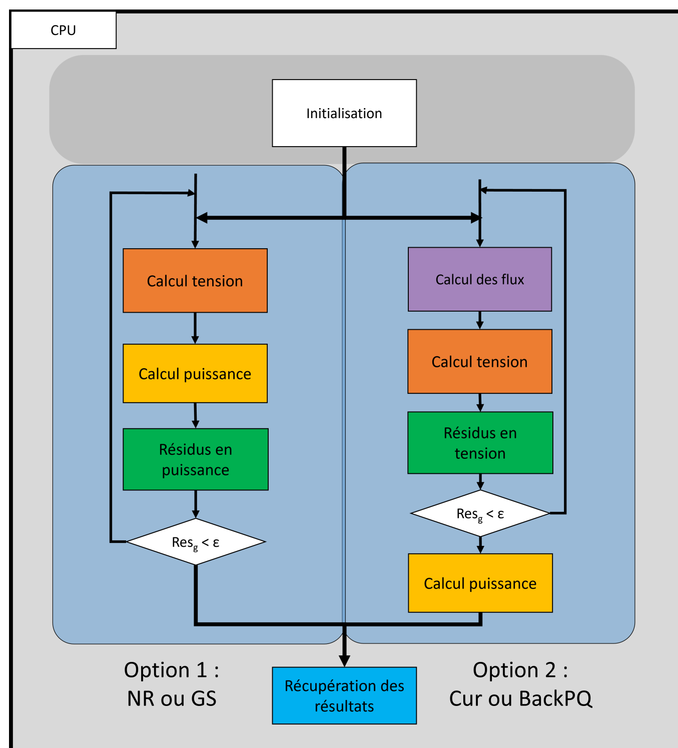


FIGURE 4.3 – Partitionnement en bloc fonctionnel pour le Power Flow

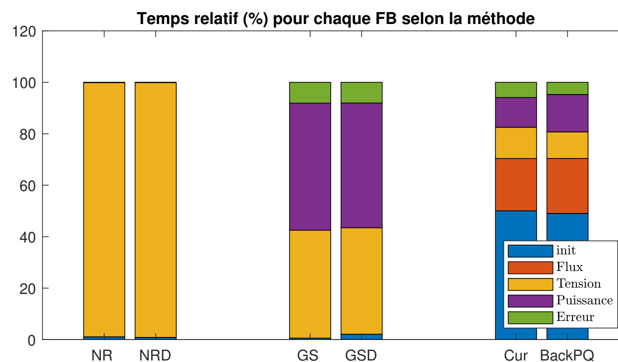


FIGURE 4.4 – Répartition des temps selon la méthode, cas Matpower 85

Pour **GS** le temps est partagé entre le calcul de la tension et de la puissance.

Le calcul de la puissance entre **NR** et **GS** est très similaire. Cette étape prend plus de temps pour **GS** à cause du grand nombre d'itérations.

Enfin pour les méthodes **Cur** et **BakPQ**, le temps est assez bien réparti entre

les blocs, avec une majorité du temps à l'initialisation.

Le passage sur GPU risque d'augmenter le temps d'initialisation des méthodes **Cur** et **BackPQ**. De plus ces méthodes ont déjà été parallélisées sur GPU [54]. Ainsi une implémentation GPU de ces méthodes sera réalisée, mais ne sera pas optimisée. L'objectif de ces implémentations est d'avoir un Power Flow fonctionnel sur GPU dans les cas radiaux.

2.3 Optimal Power Flow

2.3.1 Blocs fonctionnels

Les méthodes **OPFADMM** et **OPFADMM2** sont très proches. En ce qui concerne les blocs fonctionnels, la seule différence se situe lors de la minimisation de x.

On se basant sur l'algorithme suivant Alg. 6 de la section 3.4, on peut distinguer différents blocs fonctionnels qui sont représentés sur le schéma suivant, Fig. 4.5. Les blocs fonctionnels peuvent être définis ainsi.

- Le bloc fonctionnel **FB 1** correspond au calcul de x pour tous les bus. Ce bloc peut être séparé entre la détermination de (P_i, Q_i, l_i, v_i) d'un côté et celle de (p_n, q_n) de l'autre. Pour **OPFADMM2**, les deux étapes sont des formes fermées. La détermination de (p_n, q_n) correspond, pour la méthode **OPFADMM**, à la répétition de quatre étapes qui sont les trois étapes de l'algorithme ADMM de partage et le calcul des résidus. Pour les deux méthodes, la détermination de (P_i, Q_i, l_i, v_i) repose sur la résolution de polynôme du 3e et 4e degré.
- Le suivant (**FB 2**) correspond à ce qui dans l'implémentation réelle sera la communication de la variable x. Dans le cas de notre simulation cela correspond à la copie de la valeur de variables dans notre variable augmentée et au calcul du terme linéaire de la minimisation de y (noté c_y).
- Le bloc **FB 3** correspond au calcul des variables de consensus noté y. On y ajoute aussi le calcul de la variable duale μ.
- Le bloc suivant **FB 4** est aussi une communication dans l'implémentation réelle (de y et de μ), mais dans le cadre de notre simulation il correspond juste au calcul de grandeurs utiles, noté $\hat{C}$ pour la minimisation de x.
- Enfin le bloc **FB 5** contient le calcul des différents résidus.

2.3.2 Dépendance

Les blocs sont ordonnés par dépendance temporelle, c'est-à-dire que l'on ne peut pas faire pour un bus donné l'étape **FB N+1** avant d'avoir fini l'étape

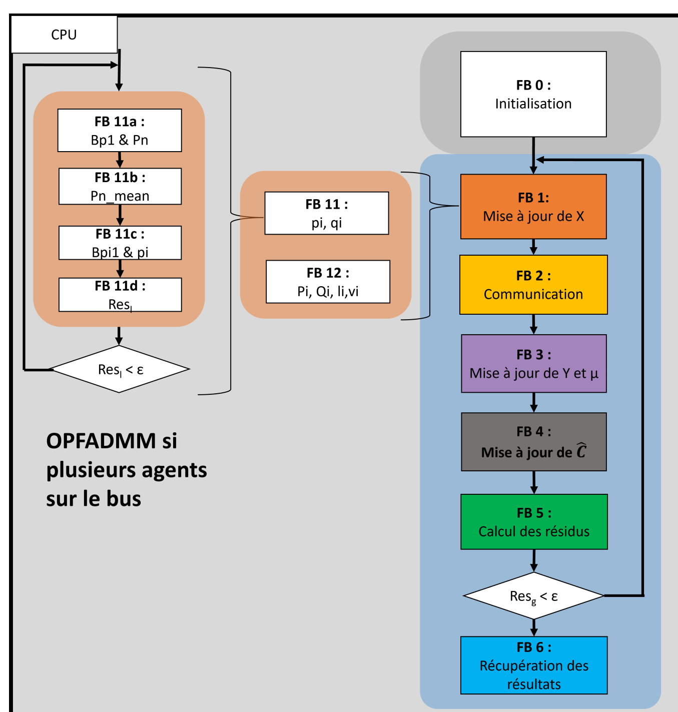


FIGURE 4.5 – Partitionnement en bloc fonctionnel pour l'Optimal Power Flow

N. Les bus ne communiquant qu'avec leurs enfants et leur ancêtre, on pourrait différencier chaque bloc fonctionnel par bus pour que les bus n'attendent que les données nécessaires pour la suite de leur calcul. Cela reviendrait à faire de l'a-synchronisme entre les bus. Cependant le GPU n'est pas fait pour faire de l'a-synchronisme tout en faisant des échanges d'informations. De plus comme on ne considère qu'un seul réseau, il existe un chemin entre chaque couple de bus, donc si un bus est bloquant tous les bus devront attendre que ce bus ait fini lors de l'itération suivante. Ainsi chaque bloc aura bien lieu les uns après les autres pour tous les bus en même temps. Les blocs **FB 11** et **FB 12** sont complètement indépendants, ils peuvent être réalisés en parallèle. Les étapes de **FB 11a** à **FB 11d** (correspondant à un problème de partage) pour **OPFADMM1** n'ont besoin d'être réalisées que lorsqu'il y a plusieurs agents sur le bus considéré. Dans le cas contraire le bloc **FB 11** n'est qu'une seule opération.

Les blocs **FB 1** et **FB 3** n'ont besoin des informations que du bus considéré. Le bloc **FB 5** est une réduction permettant de calculer les résidus et a donc besoin des variables x et y de tous les bus. Les blocs **FB 2** et **FB 4** ont besoin pour chaque bus de respectivement les valeurs de x ou de y de l'ancêtre du bus et des ses enfants. Les blocs **FB 11a** n'a besoin que des informations de chaque agent, là où les autres parties du bloc **FB 11** ont besoin des informations de tous les agents

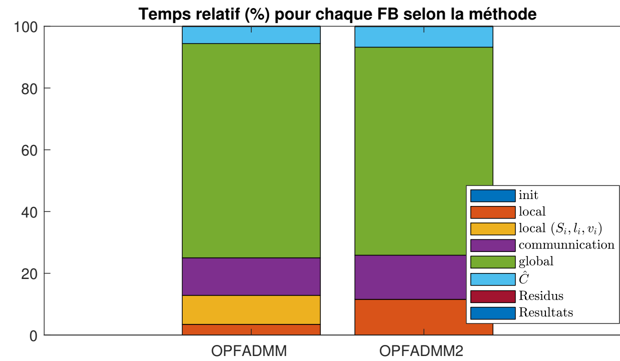


FIGURE 4.6 – Répartition des temps selon la méthode, cas TestFeeder

sur un bus donné.

2.3.3 Répartition de la charge de travail

La figure Fig. 4.6 représente la répartition des temps de calcul entre les différents blocs fonctionnels pour les deux méthodes. Cette simulation a été réalisée sur les deux premières heures (avec un pas d'une minute) du cas *TestFeeder* avec une précision de 0.0005. Pour rappel, ce cas est défini dans la section 2.6.2.3. Le temps total de calcul est d'environ 350s pour **OPFADMM** et de 250s pour **OPFADMM2**. Cette différence s'explique par le fait que ce cas contienne peu d'agent par rapport au nombre de bus. Ainsi la deuxième version de l'algorithme réduit le nombre de variable à calculer et donc diminue grandement le temps pour calculer **FB 3**.

On peut remarquer que les deux méthodes ont des répartitions des temps similaires. Les étapes d'initialisation et de récupération des résultats ont des temps négligeables. La majorité du temps est utilisée pour le problème global (c'est-à-dire le calcul de Y).

> Le calcul de Y correspond à B multiplications entre une matrice et un vecteur. Cette opération est très efficacement parallélisable sur GPU.

2.4 Marché endogène

2.4.1 Blocs fonctionnels

Le contenu des blocs fonctionnels des différentes méthodes va grandement dépendre de la méthode considérée. Cependant, on peut représenter l'organisation de ceux-ci de manière similaire. En effet quelle que soit la méthode étudiée, on

peut représenter l'algorithme comme étant une communication entre les agents et un **SO**.

De la partie marché pair à pair, on a pu déterminer que l'ADMM était meilleur pour décentraliser le problème de marché et pour résoudre le problème de partage. Ainsi, lorsque l'on parlera de marché, on fera référence à la version ADMM. Pour l'utilisation d'un **PF** ou **OPF** théoriquement n'importe quel algorithme pourrait être utilisé. Dans notre cas le PF sera résolu via un **NR** dans le cas maillé et via un **BackPQ** dans le cas radial. Pour l'OPF, ce sera **OPFADMM2** légèrement modifié pour calculer les pertes et avoir moins d'informations sur les agents.

FB 0 : l'initialisation est assez similaire entre les méthodes. Pour toutes les méthodes, il faut initialiser un marché. Pour **EndoConsensus** et **EndoDirect** il faut aussi initialiser un OPF et pour **AC-EndoPF** un **PF**. Enfin pour pour **DC-EndoPF** il faut manipuler la matrice de sensibilité, notamment faire des multiplications terme à terme pour obtenir G^2_{sensi}. Les méthodes peuvent aussi s'initialiser via la résolution d'un marché sans contraintes de réseau.

FB 1 : la résolution de chaque agent du marché endogène est ce que l'on appellera le problème local. Quelle que soit la méthode, ce problème local est une résolution d'un problème de partage. La seule différence provient de la valeur des coefficients de la fonction à minimiser localement. Pour la méthode **EndoDirect** on appellera **FB 1b** la mise à jour des variables locales physiques (S_i, l_i, v_i) de chaque bus.

FB 2 : cette étape correspond à la communication du problème local vers le problème global. Pour toutes les méthodes, cela correspond à la mise à jour des puissances totales des agents. Pour **EndoDirect** il faut calculer le terme linéaire de la minimisation du problème dual et copier les valeurs de x selon les valeurs dont les autres bus ont besoin. Pour **EndoConsensus** cette étape a lieu après le bloc **FB 3** car l'OPF de l'itération k dépend du marché de l'itération $k-1$.

FB 3 : le calcul du **SO** va dépendre de la méthode considérée. Pour **AC-EndoPF** cela correspond à la résolution d'un AC Power Flow. Pour **DC-EndoPF** cela correspond à la résolution d'un **DC-PF** qui est dans notre cas une multiplication entre la matrice de sensibilité et le vecteur de puissance des agents. Pour **EndoConsensus** cela correspond à la résolution d'un OPF. Enfin, cela correspond à une itération d'un OPF pour **EndoDirect** avec la mise à jour des variables slacks de tous les bus.

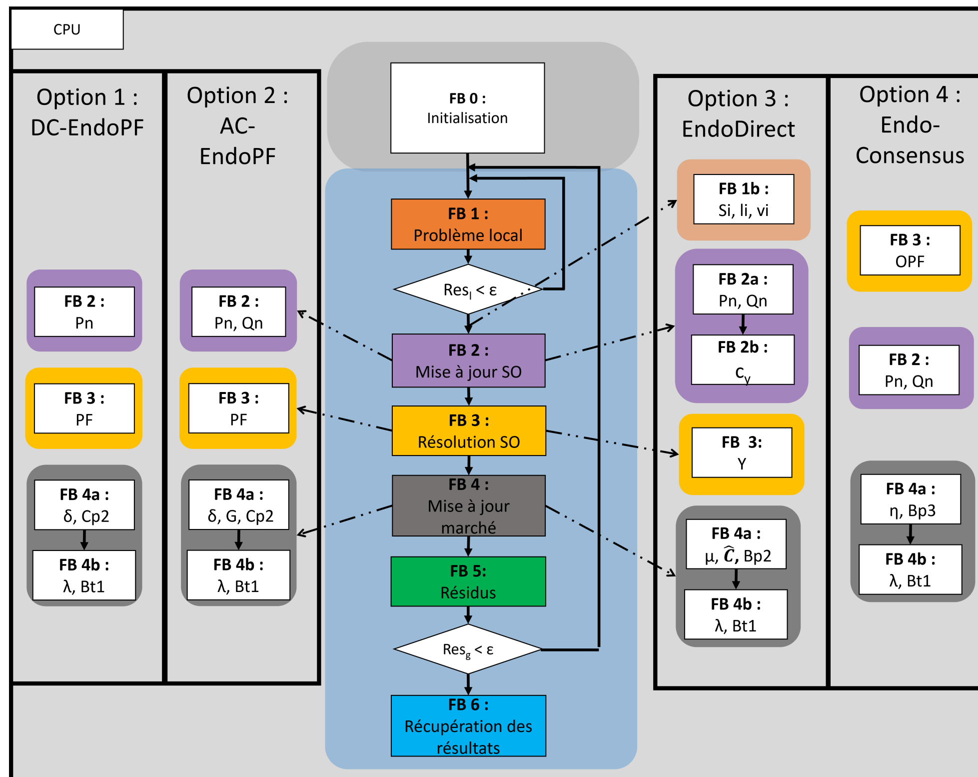


FIGURE 4.7 – Partitionnement en bloc fonctionnel pour le marché endogène

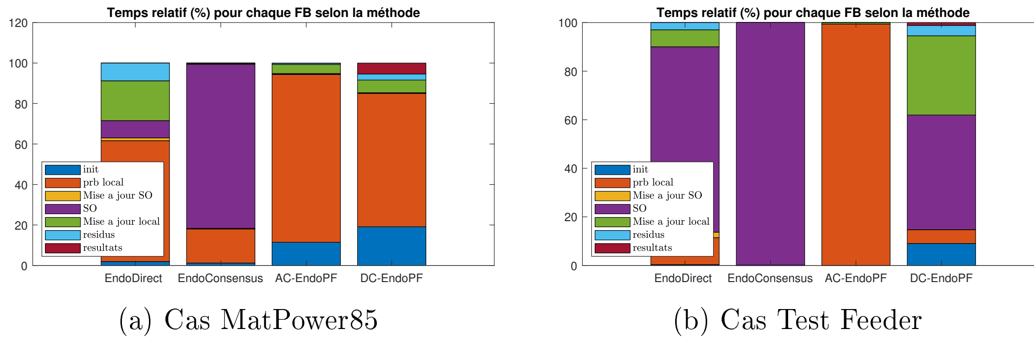
FB 4 : ce bloc correspond à la mise à jour du problème local en fonction du problème global. Pour toutes les méthodes, cela correspond à la mise à jour des variables duales et des coefficients de la fonction coût du problème local.

FB 5 : cette étape est le calcul des résidus qui représente la convergence de la solution, du respect des contraintes du marché et de celle du réseau (ou du consensus).

Le partitionnement en bloc fonctionnel des différentes méthodes est représenté sur la Fig. 4.7. L'ordre des opérations n'est pas le même pour **EndoConsensus** puisque l'OPF doit se mettre à jour en fonction de l'itération précédente du marché.

2.4.2 Dépendance

De manière générale, un bloc fonctionnel ne peut être réalisé que si le précédent est fini. Cependant pour la méthode **EndoDirect**, les étapes **Fb1a** et **Fb1b** sont indépendantes. De plus pour **EndoConsensus** les mises à jour du marché et de l'OPF dépendent de l'itération précédente. Ainsi les étapes **FB1** et **FB3** sont indépendantes et pourraient être réalisées en parallèle.



(a) Cas MatPower85 (b) Cas Test Feeder

FIGURE 4.8 – Répartition des temps de simulations selon les méthodes et le cas simulé

Pour **EndoConsensus**, l'étape **FB 2** nécessite la fin de l'étape **FB 1** et ne doit pas modifier la résolution de **FB 3**. Ainsi il faut soit la réaliser après le bloc **FB3** soit implémenter une sorte de *buffer* pour retarder la mise à jour de l'OPF. Dans notre cas, il a été choisi de réaliser la mise à jour du SO après sa résolution.

2.4.3 Répartition de la charge de travail

Pour étudier les répartitions des temps, la Fig. 4.8 représente les résultats sur CPU des quatre méthodes. La simulation est réalisée sur 5pas du cas *TestFeeder* (section 2.6.2.3) et sur le cas Matpower 85 (section 2.6.2.1). Les précisions demandées pour les cas *Matpower85* et *TestFeeder* sont respectivement de $\epsilon_g = 0.005$ et $\epsilon_g = 0.05$. La précision est plus faible que lors de la validation fonctionnelle en section 3.5.4 résultant en des temps plus faible. Ainsi les temps des méthodes **EndoDirect**, **EndoConsensus**, **AC-EndoPF** et **DC-EndoPF** sont respectivement d'environ $5s$, $60s$, $34s$ et $0.02s$.

À faible dimension, Fig. 4.8a la plupart du temps de calcul est alloué pour la résolution du problème local pour les différentes méthode. Pour **EndoConsensus**, le problème local n'est que le deuxième bloc le plus important, la majorité du temps de calcul est utilisée pour la résolution du SO.

On peut remarquer que la répartition est très différente entre les méthodes lorsque le réseau est plus grand. Dans **EndoConsensus**, la majorité du temps de simulation se situe dans la résolution de l'OPF. Ce temps pourrait donc être réduit en ayant une résolution OPF plus rapide. Les résultats sont cohérent avec le fait que le cas *TestFeeder* a beaucoup de ligne et bus (905) mais peu d'agent (55).

Pour limiter le temps de résolution de l'OPF plusieurs options sont accessibles. Il est possible d'améliorer la méthode de résolution. Mais il est aussi possible de faire varier la précision demandée notamment au début, ou ne pas résoudre à toutes les itérations.

Cependant pour **AC-EndoPF**, c'est la résolution locale qui prend la majorité du temps de résolution. Pour réduire ce temps, il est possible de demander moins de précision pour le problème local. Il est aussi possible de trouver une manière d'améliorer la convergence de cette méthode pour qu'il y ait moins d'itération.

Avec l'augmentation du nombre de bus, le calcul du SO prend une plus grande importance pour les méthodes **EndoDirect** et **DC-EndoPF**.

3 Étude de complexité

3.1 Vision globale

3.1.1 Introduction

L'objectif de cette partie est d'analyser et de comparer la complexité asymptotique en termes de mémoire nécessaire et de nombre de calcul. La connaissance de ces grandeurs nous permettra d'avoir un ordre d'idée du comportement entre les différents algorithmes avec l'augmentation de la taille du problème. Cela permettra aussi de comparer l'efficacité de la parallélisation selon les algorithmes. Cependant, cela ne nous permettra pas a priori de déterminer les meilleurs algorithmes par rapport à leur vitesse de convergence puisqu'un algorithme avec une plus grande complexité de calcul peut avoir de bien meilleures propriétés de convergence et ainsi converger bien plus rapidement ou être capable d'atteindre de meilleures précisions.

Pour simplifier le calcul, on considéra comme équivalent une addition, une multiplication ou une affectation lors de l'évaluation de la complexité. Par contre on calculera à part les calculs complexes tels que les fonctions trigonométriques, les logarithmes ou exponentielles. On négligera le temps pour accéder aux données pour un calcul sur un matériel donné, mais les transferts entre CPU et GPU seront identifiés.

Les paramètres qui seront pris en compte sont les dimensions du problème, qui sont pour rappel : N le nombre d'agents, M le nombre total de voisins, B le nombre de bus, et L le nombre de lignes. Comme le nombre d'itérations ne peut pas être déterminé a priori et dépend plus de la précision demandée et de l'activation ou non des contraintes que de la dimension du problème, on ne considérera que la complexité pour une itération. Dans le cas où l'algorithme possède deux boucles

imbriquées, on ne considérera aussi dans l'écriture de la complexité, qu'une seule itération de cette boucle interne.

3.1.2 Complexité parallèle

Pour calculer la complexité théorique en parallèle on se basera sur le théorème de Brent [89]. On a :

$$C_{para} = O(\frac{o}{p} + t) \tag{4.1}$$

avec o le nombre total d'opérations (donc la complexité en série, sauf en cas de redondance des calculs), p le nombre de processeurs sur lequel on parallélise, et t le nombre d'étapes.

On peut remarque que p pour une parallélisation sur CPU vaut au plus le nombre de cœurs. Ce nombre est bien trop faible pour permettre de réduire la complexité des calculs. Ainsi pour la suite on ne considérera que la complexité en série sur CPU. On appellera complexité en parallèle celle sur GPU.

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. Selon le calcul et la parallélisation choisie que l'on réalisera, le nombre de processeurs p pourra être égal au nombre de *thread* ou au nombre de blocs. Sauf exception, dans tous les problèmes on a fixé $N_{t/B} = 512$ pour que la valeur soit assez importante sans atteindre le maximum possible.

Dans le cas où l'on a atteint les limites physiques de notre GPU, il se peut que p ne soit plus égal au nombre de *threads*. Lorsque cette valeur sera atteinte, on aura atteint pour une implémentation donnée la parallélisation maximale (ce qui n'est pas forcément le plus rapide).

3.1.3 Résultats généraux

Le problème global consiste en la combinaison des différents blocs qui seront présentés ci-après. Il est possible que chaque bloc fasse plusieurs itérations pour chaque itération du problème global. Cependant, sauf en cas de grandes différences, cela n'influencera pas la complexité. Soit K le nombre total d'itérations. On a donc pour l'algorithme complet la complexité suivante :

$$C_{global} = K \cdot (C_{Marche} + C_{PF} + C_{OPF} + C_{interaction}) + C_{init} \tag{4.2}$$

L'initialisation (de même que la récupération des résultats) n'a lieu qu'une fois par résolution. De plus dans le cas où l'on simule plusieurs pas temporels, son influence peut être encore plus réduite en ne changeant que ce qui doit l'être. Dans tous les cas, la complexité de cette étape (en série ou en parallèle sur GPU) est au moins linéaire avec les dimensions du problème (pour créer et copier les différentes

grandeurs). On ne présentera donc dans les parties suivantes que les opérations qui peuvent avoir une complexité plus grande.

On a pu voir dans la partie précédente que, pour la grande majorité des méthodes et des problèmes, le temps de l'initialisation était négligeable devant le temps de simulation. La parallélisation et l'optimisation de ce bloc ne sont donc pas des priorités.

De nombreuses opérations se retrouvent dans les différents algorithmes. Les opérations de bases seront définies ici. On note temporairement pour cette partie I une taille de vecteur et on considérera des matrices de taille $J \cdot I$. Dans cette partie on ne considérera que des vecteurs et matrices pleins. Différentes techniques permettent de réduire la complexité dans le cas de matrices creuses. Elles seront présentées lors de leurs utilisations.

Les opérations linéaires sur un vecteur ont une complexité linéaire avec la taille du vecteur, $O(I)$. Ces opérations concernent les additions, soustractions de vecteur, mais aussi les multiplications et divisions terme à terme. Lorsque ce type d'opération est parallélisé sur GPU la complexité devient en temps constant. En effet la simplicité de l'opération et le grand nombre de processeurs disponibles permettent de répartir les calculs sur les différents threads. En ayant un thread pour chaque opération on a $o = I$, $p = I$ et $t = 1$ d'où $C_{para} = O(1)$.

Les réductions ont une complexité linéaire en série, $O(I)$. Les moyennes, les sommes ou normes d'un vecteur et la recherche de maximum/minimum dans un vecteur sont toutes des réductions. Lorsque l'on parallélise, on peut atteindre une complexité logarithmique en suivant la méthode de Nvidia [109]. Dans ce cas-là on a $t = log(I)$, $o = O(I)$ et $p = I$ ou $p = \frac{I}{log(I)}$, ce qui donne $C_{para} = O(log(I))$.

Les multiplications matrices/vecteurs ont une complexité quadratique en série, $O(2J \cdot I)$. En effet pour chaque ligne de la matrice, il faut effectuer la multiplication terme à terme entre cette ligne et le vecteur et réaliser la somme de ces termes. Lors de la parallélisation sur GPU on peut associer un bloc de thread par ligne de la matrice. Dans ce cas là, pour chaque ligne, on réalise une opération linéaire et une réduction. On a ainsi comme précédemment $t = log(I) + 1$, $o = O(2J \cdot I)$ et $p = 512J$. La complexité parallèle est donc $C_{para} = O(log(I))$.

Le résultat précédent n'est vrai que si I est du même ordre de grandeur que le nombre de threads par bloc (512). De même il faut que le GPU soit capable de supporter J blocs en même temps (en termes de nombre de processeurs ou de mémoire).

La résolution d'un système linéaire peut se faire de différentes manières. Que l'on passe par une factorisation **LU** ou un pivot de Gauss la complexité est cubique, $O(I^3)$. Dans les deux cas les méthodes recherchent pour chaque colonne le meilleur pivot ($O(I)$). Puis il faut échanger les lignes ($O(I)$), et ensuite on met à jour la sous-matrice ($O(I^2)$). Ces étapes étant réalisées pour chaque colonne, il faut multiplier la complexité par I. Ce qui donne bien la complexité cubique en série. Pour l'implémentation en parallèle sur GPU, la recherche du pivot est une réduction ($O(log(I))$, l'échange de ligne est juste une affectation $O(1)$. Pour la dernière étape, on parallélise avec un bloc qui gère une ligne, et chaque thread qui calcule un terme. Ce qui permet de mettre à jour la sous-matrice en temps constant ($O(1)$). La complexité totale est donc en $O(I \cdot log(I))$ en parallèle. On pourrait aussi paralléliser sur les colonnes et utiliser la sparcité pour encore réduire la complexité, comme dans [108]. La résolution du système ensuite est quadratique en série et linéaire en parallèle. Dans le cas où l'on a inversé la matrice, la résolution du système en parallèle est une réduction, donc logarithmique.

L'inverse d'une matrice correspond globalement aux mêmes étapes que la résolution d'un système. Cependant dans cette thèse, cette opération ne sera réalisée que pendant les initialisations. On choisira donc de toujours la réaliser sur CPU. Ceci permet d'utiliser une bibliothèque (Eigen) pour réaliser efficacement et précisément ce calcul.

3.2 Marché pair à pair

Pour déterminer la complexité totale des algorithmes de marché on va évaluer la complexité bloc par bloc. Les grandeurs importantes dans ce problème sont le nombre d'agent N, le nombre de voisin de chaque agent M_n, et le nombre total d'échange M. Dans le cas complètement connecté, le nombre de voisin dépend de la répartition entre consommateur, producteur et consommateur. Sauf cas extrême, on peut considérer autant dans chaque catégorie, ainsi on a $M_n = O(N)$ et $M = O(N^2)$. Toutes les quantités sont données en ordre de grandeur. Comme le solveur **OSQP**, utilise des matrices, il est assez complexe de déterminer à priori la complexité de la résolution. On notera donc cette complexité α, tel que la complexité soit en $O(N^\alpha)$. Comme **OSQP** nécessite la manipulation de vecteur et la résolution de système on peut s'attendre à une complexité plus élevé que linéaire $\alpha \geq 2$ [110].

TABLE 4.3 – Blocs fonctionnels et complexité en série et en parallèle des algorithmes pour le marché pair à pair

Bloc Fonctionnel	OSQP	ADMM		PAC	
	CPU	CPU	GPU	CPU	GPU
FB0	$O(N^3)$	$O(M)$	$O(M)$	$O(N^4)$	$O(N^4)$
FB1	$O(N^{\alpha+1})$	$O(M)$	$O(log(M))$	$O(N^3)$	$O(log(M))$
FB2	$O(M)$	$O(M)$	$O(1)$	$O(M)$	$O(log(N))$
FB3	$O(M)$	$O(M)$	$O(log(N))$	$O(M)$	$O(log(N))$
FB4	$O(M)$	$O(M)$	$O(M)$	$O(M)$	$O(M)$
FB 0 bis	$O(M)$	$O(M)$	$O(M)$	$O(M)$	$O(M)$
FB1&2&3	$O(N^{\alpha+1})$	$O(M)$	$O(log(N))$	$O(N^3)$	$O(log(N))$
Total	$O(N^{\alpha+1})$	$O(M)$	$O(M)$	$O(N^4)$	$O(N^4)$

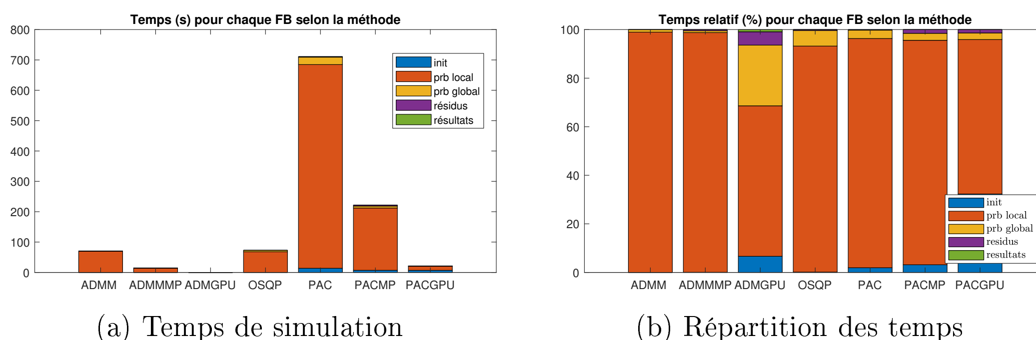
Pour l'initialisation de **OSQP**, il faut créer N matrices de taille $M_n \cdot M_n$. Cela signifie une complexité $O(N^3)$. Pour **PAC**, l'opération dominante est l'inverse de N matrices de taille $M_n \cdot M_n$, ce qui donne une complexité de $O(N^4)$. Ces opérations sont réalisées par la bibliothèque Eigen dans toutes les implémentations de PAC. La complexité de cette étape est donc identique ente la version sur CPU et celle sur GPU.

La complexité de la mise à jour du problème local dépend de l'algorithme. Ainsi **PAC** a N multiplications matrice-vecteur. Lors de la version GPU, l'ensemble des matrices peuvent être concaténées pour ne réaliser qu'une seule multiplication et ainsi paralléliser plus efficacement. Pour **OSQP**, cela consiste en N appels du solveur, donc une complexité en $O(N^{\alpha+1})$. Pour l'ADMM cela consiste en des opérations linéaires sur des vecteurs et une réduction par agent. En réalisant une réduction par bloc de thread, on peut complètement paralléliser cette réduction comme si cela en était qu'une seule.

La mise à jour du problème global sont des mises à jour de vecteurs pour tous les algorithmes. Tandis que le calcul des résidus globaux sont des réductions pour toutes les méthodes.

Le tableau Tab. 4.3 regroupe les complexités des différents algorithmes. On peut y voir que l'utilisation du GPU permet de réduire la complexité en parallélisant les calculs.

La figure Fig. 4.9 représente l'évolution de la répartition des temps avec le passage sur GPU pour le cas Matpower 118 (section 2.6.2.1).



(a) Temps de simulation (b) Répartition des temps

FIGURE 4.9 – Évolution de la répartition des temps sur le cas 154 agents

> Afin de pouvoir mesurer les temps des appels kernels sur GPU, il faut rajouter des barrières de synchronisations. En effet, le comportement asynchrone entre le CPU et le GPU empêche de mesurer les bons temps sans cela. Ainsi les temps du GPU sont légèrement surestimés puisqu'on force le CPU à attendre le GPU avant de continuer. Ceci est vrai pour toutes les méthodes de tous les problèmes.

On peut d'abord remarquer sur la Fig. 4.9a que la parallélisation a très fortement réduit les temps de calcul. La méthode **PAC** est bien plus lente que les autres méthodes mais la parallélisation réduit d'autant plus le temps. Ces résultats sont cohérents avec la réduction de complexité démontrée précédemment. On peut voir sur la Fig. 4.9b la nouvelle répartition des temps. On peut voir que la parallélisation de **ADMMGPU** a clairement permis de réduire l'importance du problème local sur le temps total. La répartition des temps pour les méthodes décentralisant avec **PAC** reste plutôt similaire. La parallélisation a permis de réduire le temps de toutes les étapes dans les mêmes proportions.

3.3 Power Flow

Cette partie évaluera la complexité des différentes manières de résoudre un **PF** sur CPU. Les grandeurs caratéristiques de ce type de problème sont le nombre de bus B et le nombre de ligne L. Dans un cas radial ou peu connecté, on a $L = O(B)$. Dans un cas fortement connecté on a $L = O(B^2)$. Soit L_b le nombre de ligne partant ou arrivant au bus b, ce nombre varie entre 1 et $B-1$. En pratique, même dans un réseau maillé, il n'y a que quelques lignes par bus, ainsi on pourra aussi considérer que $L = O(B)$. Le nombre d'agent N intervient dans un calcul de l'initialisation pour passer des puissances des agents aux puissances des bus.

Même s'il on considère que $L = O(B)$, on pourra dans la suite tester le comportement de certains appels kernels dans le cas où le réseau est complètement connecté.

L'initialisation est globalement la même entre les différents algorithmes. On peut pré-calculer des constantes comme par exemple $\frac{1}{\underline{Y}_{ii}}$ et $\frac{\underline{Y}_{ij}}{\underline{Y}_{ii}}$. Dans tout les cas les calculs sont au plus linaires ($O(L)$). Afin de réaliser le passage des puissances des agents à celles des bus, on ajoute séquentiellement la puissance de l'agent à la puissance du bus sur lequel il est :

$$\underline{s}_b^{bus} = \sum_{n \in \mathcal{N}_b} \underline{s}_n^{agent} \tag{4.3}$$

Le calcul est de complexité linéaire $O(N)$. Cette opération est en réalité une réduction. Ainsi sur GPU on associe un bloc par bus, et les threads du bloc réalisent la somme des différentes puissances des agents sur le bus. La complexité est donc en $O(log(N_b))$. Dans l'article [51], le calcul était fait avec un thread par bus, leur complexité est donc en $O(N_b)$. Si notre méthode permet de mieux paralléliser, elle demande plus de ressources. Ainsi cette méthode est plus adaptée à un fort nombre d'agent par bus, là où la méthode de [51] est plus adaptée à l'augmentation du nombre de bus.

Le calcul de la puissance consiste en 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 (l'équation sur la puissance réactive est similaire) :

$$P_i = V_i \sum_{l_i} V_{k(l_i)} (G_{l_i} \cos \theta_{ik(l_i)} + B_{l_i} \sin \theta_{ik(l_i)}) \tag{4.4}$$

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 nœud de référence.

Pour paralléliser sur GPU, dans un premier temps on peut calculer chaque terme de la somme dans un thread $C_{para} = O(1)$. Ensuite on peut faire une réduction $C_{para} = O(log(max(L_i)))$ avec un bloc par bus. 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.

Le calcul de la tension, dans le cadre de l'algorithme de **Cur**, correspond à l'équation (3.21) et a une complexité en $O(B)$. Pour l'algorithme de **GS** il faut réaliser le calcul (3.16) pour chaque bus. 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 $\underline{Y}_{il}$ 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 (3.12). 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 $O(B^2)$.

Pour paralléliser (3.16) sur GPU, on sépare le calcul en deux étapes. La première pré-calcule les termes qui dépendent de l'itération précédente, c'est donc une réduction de complexité $O(log(max(L_i)))$. La deuxième calcule séquentiellement ce qui change pour chaque bus est de complexité en $O(B)$ (plus de détail dans la section 4.4.6).

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 [108]. Enfin il faut obtenir le déplacement de la tension en résolvant le système, qui est un calcul en $O(B)$.

Le calcul du courant de **Cur** correspond aux équations (3.17) et (3.19). Ces deux calculs sont en $O(B)$.

Le calcul de l'erreur est une réduction dans tous les algorithmes. Ce calcul a donc une complexité en $O(B)$ sur CPU et $O(log(B))$ sur GPU.

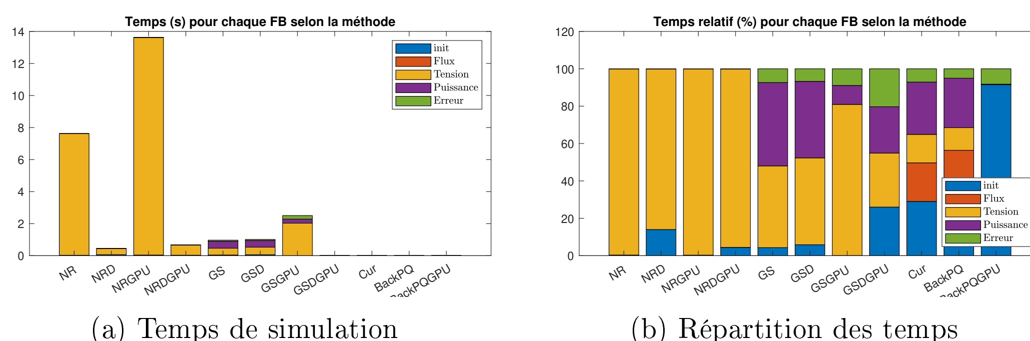
L'ensemble des complexités des méthodes est regroupé dans le Tab. 4.4. La ligne appelée "Total calcul" considère tous les blocs fonctionnels sauf l'initialisation.

La figure Fig. 4.10 représente l'évolution de la répartition des temps avec le passage sur GPU pour le cas *TestFeeder* (section 2.6.2.3).

On peut d'abord remarquer sur la Fig. 4.10a que la parallélisation n'a pas permis de réduire les temps de calcul. Cela peut s'expliquer par le fait que le cas est un réseau radial. Comme on a $L = B$, la majorité des bus n'ont que quelques

TABLE 4.4 – Blocs fonctionnels et complexité en série et en parallèle des algorithmes pour le Power Flow

Bloc Fonctionnel	CPU			GPU	
	Curr	NR	GS	NR	GS
init	$O(L+N)$				
tension	$O(L)$	$O(B^3)$	$O(B^2)$	$O(Blog(B))$	$O(B)$
Puissance	$O(L)$			$O(log(L_i))$	
flux	$O(L)$				
Erreur	$O(B)$			$O(log(B))$	
Total calcul	$O(L)$	$O(B^3)$	$O(B^2)$	$O(Blog(B))$	$O(B)$



(a) Temps de simulation

(b) Répartition des temps

FIGURE 4.10 – Évolution de la répartition des temps sur le cas Test Feeder

lignes. Or on associe un bloc de thread par bus, pour que chaque thread calcule une valeur associée à une ligne. Ainsi on utilise beaucoup de ressources du GPU pour paralléliser quelques calculs.

Les méthodes **NRD** et **NRDGPU** (implémentation de **NR** en double précision) sont bien plus rapides que **NR** et **NRGPU**. Cela s'explique par le fait que sur ce cas, l'algorithme de Newton-Raphson ne converge qu'en double précision. Il y a ainsi que 2 itérations de ces méthodes contre 20 (le nombre d'itérations maximales permises) pour les méthodes en simple précision.

Sur la Fig. 4.10a, les méthode utilisant Gauss Seidel semblent plus rapide. En réalité pour ce cas, seules les méthodes **NRD**, **NRGPU**, **Cur** et **BackPQ** convergent. Le temps total des autres méthodes dépendent uniquement du nombre d'itérations maximales.

Les trois méthodes en *load flow* sont bien plus rapides que les autres. Ces algorithmes seront donc utilisés lorsque le réseau sera radial. La version sur GPU

est légèrement plus lente. Cela est dû à une initialisation et un calcul des résidus plus lents à cause des transferts de mémoires.

Toutes les implémentations sur GPU sont plus lentes que leur version sur CPU pour ce cas. L'utilisation du GPU sera donc déterminée en fonction du contexte où est utilisé le Power Flow. Par exemple si les données sont déjà sur GPU, on effectuera le calcul du **PF** sur GPU. Le temps perdu par la résolution sur GPU sera compensé par le fait qu'il n'y a pas besoin de faire des transferts mémoires.

3.4 Optimal Power Flow

Pour déterminer la complexité totale des **OPF** on évaluera la complexité bloc par bloc. Pour l'utilisation de cet algorithme on se base avec un réseau radial, on a donc $B = L + 1$, et on ne considère pas de marché donc N a un impact mais pas M. Soit c_b le nombre d'enfant d'un bus, on remarquera que chaque bus (sauf le bus de référence) est l'enfant d'un unique ancêtre, ainsi on a $\sum_b c_b = B - 1$ (que l'on approximera par B). La taille d'un vecteur pour les variables x, y et μ est sera noté o_b pour chaque bus b :

$$\begin{aligned} o_b &= 3 * c_b + 7 \qquad && OPFADMM \\ o_b &= 3 * c_b + 2 * N_b + 5 \qquad && OPFADMM2 \end{aligned} \tag{4.5}$$

Ainsi leurs tailles totales sont de $\sum_b o_b = 10 * B$ ou $\sum_b o_b = 8 * B + 2 * N$. Pour la suite on aura besoin du calcul de $\sum_b o_b^2$. Globalement plus la répartition des enfants sera déséquilibré (tous les bus reliés au bus de référence) plus cette valeur sera proche de $9B^2$. Dans le cas où tous les bus n'ont qu'un seul enfant $C_b = 1$ (une ligne droite) cela vaut $100 * B$. Le cas général se situe entre ces deux extrêmes. Pour la méthode **OPFADMM2**, cette valeur varie entre $O(B + N)$ lorsque la répartition des enfants et des agents sont équilibrés et $O(B^2 + N^2)$ lorsque tout est sur le même bus.

Lors de l'initialisation (**FB 0**) il faut définir les B matrices de taille $o_b \cdot o_b$ permettant la résolution de (3.69). Dans la version sur GPU, une seule matrice est définie pour contenir toutes les matrices. Cette matrice est de taille $\sum_b o_b \cdot o_b^{max}$. Plus le cas est équilibré, plus la taille est proche sur celle sur CPU. Pour obtenir ces matrices, il faut faire B multiplications de matrice de taille $3 \cdot o_b$. Ce qui donne une complexité totale en $O(B^2)$. D'autres opérations sont nécessaires pour initialiser les variables, mais ces opérations ont au plus une complexité linaire.

Quand le bloc **FB 1** est une forme fermée à résoudre pour chaque bus, la complexité est linéaire $O(B + N)$ en série. En parallélisant on peut faire en sorte de réaliser cet opération en temps constant $O(1)$. Il peut être intéressant de noter que les threads étant synchrones par groupe de 32 (warps), le nombre d'étape est celui maximal du warp même si certains bus auraient pu s'arrêter avant. Lorsqu'il

y a plusieurs agents sur un bus dans la méthode **OPFADMM** il faut résoudre un problème de partage. La complexité reste linaire même si la solution devient itérative en série. Lors de la parallélisation on ne peut pas garder un temps constant, la complexité devient logarithmique $O(ln(N_b))$.

Il est important de noter que même si la résolution est en temps constant, la recherche de racines de polynôme demande l'utilisation de fonctions très couteuses.

Pour le bloc **FB 2**, on a une affectation par enfant et ancêtre ce qui correspond à un calcul en série d'environ $C_{serie,2} = O(4 \cdot B)$. Chaque affectation est indépendante donc en théorie on pourrait avoir chaque thread allouant une valeur. Cependant en faisant ainsi il est assez dur d'affecter un thread par valeur (en sachant que selon les cas il faut accéder à l'ancêtre du bus ou à ses enfants). Ainsi dans ce cas là on associera un bloc de thread par bus, $N_B = B$. Ensuite un thread par bloc s'occupera d'affecter la valeur de l'ancêtre et puis c_b threads affectent les trois valeurs pour les enfants. En faisant ainsi on a beaucoup de threads inactifs (dépendant de la largeur du réseau) mais on accède facilement aux données de manière coalescente. On a donc $o = 4 \cdot B$, $p = B$ (on parallélise sur plus mais une grande partie ne fait presque rien) et on pourrait tout faire en une étape donc $t = 1$, ce qui nous donne $C_{para,2} = O(1)$.

L'étape suivante, **FB 3**, correspond à B multiplications matrice-vecteur pour la minimisation de y et le calcul de μ correspondant à B opérations sur des vecteurs. La complexité est de $O\left(\sum_b o_b^2\right)$ en série et $O(ln(o_b))$ en parallèle.

Le calcul de $\hat{\mathbf{C}}$ dans **FB 4** correspond à des opérations sur des vecteurs pour le calcul de $\hat{\mathbf{P}}$, $\hat{\mathbf{Q}}$, $\hat{\mathbf{p}}$, $\hat{\mathbf{q}}$, $\hat{\mathbf{l}}$. Le calcul de $\hat{\mathbf{v}}$ correspond à une réduction sur les enfants pour chaque bus. Ainsi la complexité de ce bloc est en $C_{serie,4} = O(B)$. Pour réaliser en parallèle l'ensemble des opérations, on associe chaque bus à un bloc de thread. Les 6 premiers threads calculent les différentes constantes, puis l'ensemble des threads réalise une réduction. Ainsi comme pour le bloc précédent on a une complexité en $C_{para,4} = O(ln(c_b^{max}))$.

> Dans ce cas là, on créera beaucoup de blocs de taille $N_{t/B}$. Si on garde $N_{t/B} = 512$, chaque réduction sera parallélisée totalement jusqu'à 1024 enfants. Ce nombre d'enfant par bus ne sera atteint que sur un nombre limité de bus et dans des cas extrêmement rares. Dans les cas avec moins d'enfants on fera beaucoup de sommes inutiles (de la forme $0+0$) et on allouera de la mémoire pour rien. Il vaut donc mieux avoir un $N_{t/B}$ bien plus petit, cela permettra en plus d'avoir plus de blocs effectivement exécutés en parallèle (puisqu'il y aura besoin de moins de mémoire). On prendra ainsi $N_{t/B} = 32$ (un warp) pour ce calcul.

Le bloc **FB 5** est une réduction qui est donc en $O(B)$ en série et en $O(ln(B))$ en parallèle. A chaque fois que cette étape est réalisée, un transfert GPU vers CPU permet de récupérer les valeurs des résidus, ainsi on transfère deux flottants.

La dernière étape (**FB 6**) consiste dans le calcul de la valeur de la fonction objectif qui est donc un calcul en $C_{serie,6} = O(N)$ et à la récupération du résultat en puissance des agents (un vecteur de dimension en $O(N)$). Dans le cas de la version parallélisé, le calcul est une réduction donc en $C_{para,6} = O(ln(N))$ avec transfert du résultat (un flottant) et on doit réaliser le transfert entre CPU et GPU du vecteur des puissances de taille $O(N)$.

Lorsque l'on résout plusieurs pas temporels consécutifs, plutôt que d'initialiser de 0 à chaque pas, on peut adapter cette initialisation. Entre deux pas, la seule donnée qui change est la fonction coût des agents et leurs bornes (de taille $O(N)$ à re-transférer). Pour avoir des meilleures propriétés de convergence, on change les puissances des agents pour qu'elles soient au milieu de leurs nouvelles bornes et on initialise les flux de puissance à partir de ces puissances. On met à jour le courant dans les lignes et on initialise les variables de consensus $y = x$. On n'a pas besoin de recalculer les matrices ou les vecteurs de correspondance, ce qui fait une complexité de $C_{serie,0bis} = O(B + N)$ en série et $C_{para,0bis} = O(ln(c_b^{max}))$.

Les résultats théoriques sur la complexité de la parallélisation sur GPU sont regroupés dans le Tab. 4.5. Pour plus de clarté, on supposera dans le tableau que les agents et enfants sont répartis de manière égale entre les bus.

La figure Fig. 4.11 représente l'évolution de la répartition des temps avec le passage sur GPU pour le cas *TestFeeder* (section 2.6.2.3). Pour obtenir ces données, les 2 premières heures du cas d'études ont été simulées avec un pas d'une minute.

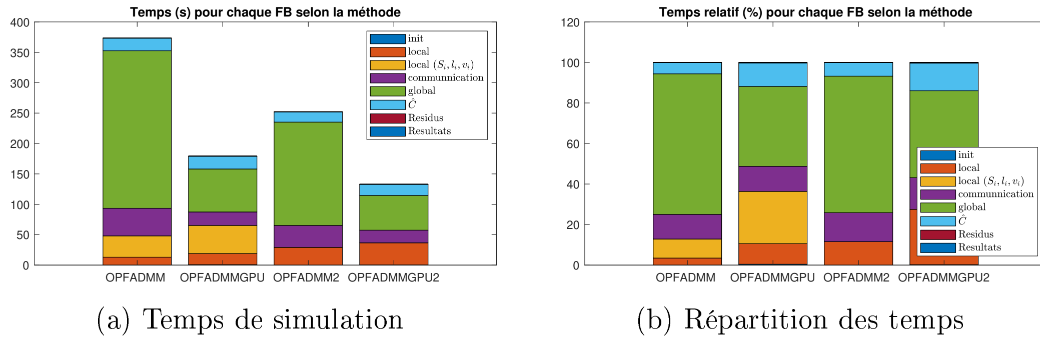
Sur la Fig. 4.11b, on peut remarquer que le calcul global nécessite une proportion de temps bien plus petite. La Fig. 4.11a montre que la parallélisation a permis de réduire le temps total de simulation.

Cependant on peut remarquer que le fait de passer sur GPU a augmenté le temps de calcul du problème local.

Pour le bloc $FB12$ (calcul de S_i, l_i, v_i) cela peut s'expliquer par le fait que l'on doivent trouver des racines de polynômes. En effet, ces types d'opérations utilisent

Table 4.5 – Blocs fonctionnels et complexité en série et en parallèle des algorithmes pour l'optimal power Flow

Bloc	ADMMOPF		ADMMOPF2	
	CPU	GPU	CPU	GPU
FB0	$O(B^2+N)$	$O(B^2+N)$	$O(B^2+N^2)$	$O(B^2+N^2)$
FB 0 bis	$O(B+N)$	$O(ln(c_b))$	$O(B+N)$	$O(ln(c_b))$
FB6	$O(B+N)$	$O(B+N)$	$O(B+N)$	$O(B+N)$
FB1	$O(B+N)$	$O(ln(N))$	$O(B+N)$	$O(1)$
FB2	$O(B)$	$O(1)$	$O(B+N)$	$O(1)$
FB3	$O(B^2)$	$O(ln(c_b))$	$O(B^2+N^2)$	$O(ln(c_b+N_b))$
FB4	$O(B)$	$O(ln(c_b))$	$O(B)$	$O(ln(c_b))$
FB5	$O(B)$	$O(ln(B))$	$O(B)$	$O(ln(B))$
FB 1->5	$O(B^2+N)$	$O(ln(N)+ln(B))$	$O(B^2+N^2)$	$O(ln(N)+ln(B))$
Total	$O(B^2+N)$	$O(B+N)$	$O(B^2+N^2)$	$O(B^2+N^2)$



(a) Temps de simulation

(b) Répartition des temps

Figure 4.11 – Évolution de la répartition des temps sur le cas Test Feeder

des opérateurs complexes ($\sqrt[3]{a}$, $\sqrt{a}$, $\cos(a)$...) qui sont très lents sur GPU. De plus les calculs étant très sensibles aux erreurs numériques, les calculs sont effectués en double précision.

Pour le bloc $FB11$, cela s'explique par le fait que l'on ait peu d'agent et beaucoup de bus. En effet sur ce cas, il y a au plus 1 agent par bus. Ainsi ce bloc peut être résolu avec une forme fermée. Cependant on parallélise avec un bloc par bus. Il en résulte que la majorité des bus sont lancés mais ne doivent rien faire. Les quelques blocs qui doivent calculer ne doivent faire qu'un seul calcul. Il en résulte que la majorité des threads sont inactifs tout en utilisant les ressources du GPU.

3.5 Marché endogène

La complexité des différentes méthodes du marché endogène va dépendre des blocs choisis à l'intérieur de chaque méthode. La partie de marché pair à pair a la même structure entre les différentes méthodes. Par contre la complexité de la partie du SO va dépendre de la méthode utilisée pour résoudre sa partie.

Les dimensions importantes pour évaluer la complexité du marché endogène sont N, M, B, L respectivement les nombres d'agents, d'échanges, de bus et de lignes. On considèrera dans la suite pour limiter le nombre de variable que le marché est complètement connecté et que le réseau est faiblement maillé. Ainsi on a $M = O(N^2)$ et $L = O(B)$.

Comme les méthodes **AC-EndoPF**, **EndoDirect** et **EndoConsensus** repose directement sur des blocs des autres méthodes. On reportera directement leur complexité dans le Tab. 4.6. On ne détaillera que **DC-EndoPF** et les interactions entre les blocs.

Mise à jour du SO : correspond au calcul des puissances totales des agents pour toutes les méthodes. Comme $p_n = M_n * \tilde{p}_n$, le calcul est une opération linéaire sur un vecteur. Pour **AC-EndoPF** il faut calculer $p_b = \sum_{n \in \mathcal{N}_n} p_n$. L'étape **Fb 2b** de la méthode **EndoDirect** correspond à l'étape **Fb 2** des OPF. Enfin pour **EndoConsensus** il faut mettre à jour les fonctions coûts des agents de l'OPF à partir des variables duales de consensus. Ce qui correspond à des opérations linéaires sur des vecteurs.

Calcul du SO : est un DC-PF pour la méthode **DC-EndoPF**. Cela correspond aux calculs de α_{ln}, Q_{ln}^{part} et Q_{ln}^{tot}. La complexité est donc en $O(N * L)$ en série. Le calcul de α et Q^{tot} peuvent se faire en temps constant sur GPU. Le calcul de Q^{part} se parallélise mal sur GPU (étant $N * L$ réductions vers $N * L$ valeurs). Le choix a été fait de mettre un bloc par ligne l et que chaque thead calcule un ou plusieurs coefficients. La complexité est donc en $O(N)$.

Mise à jour du problème local : est dans tous les cas des mises à jour de variables duales. Cela a donc une complexité linaire avec la taille des vecteurs. La mise à jour de la fonction coût (coefficient c_{p2}) du problème local de **DC-EndoPF** et de **AC-EndoPF** est une réduction de $L \cdot N$ vers N. Cela a donc une complexité en $O(L \cdot N)$ en série et en $O(ln(L))$ en parallèle. Pour le calcul de la sensibilité G de la méthode **AC-EndoPF** a une complexité de $O(NL)$ en série et en $O(1)$ en parallèle. Pour toutes les méthodes il faut mettre à jour l'antisymétrie des échanges.

La figure Fig. 4.12 représente l'évolution de la répartition des temps avec le passage sur GPU pour le cas *TestFeeder* (section 2.6.2.3). La simulation ne concerne

TABLE 4.6 – Blocs fonctionnels et complexité en série et en parallèle des algorithmes pour le marché endogène sur CPU

Bloc	DC - EndoPF	AC-EndoPF	EndoConsensus	EndoDirect
FB0	$O(L \cdot N + M)$	$O(L + M)$	$O(B^2 + M)$	$O(B^2 + M)$
FB 0 bis	$O(M)$		$O(B + M)$	
FB6	$O(B + M)$			
FB1	$O(M)$			$O(M + B)$
FB2	$O(N)$			$O(N + B)$
FB3	$O(L \cdot N)$	C_{PF}	C_{OPF}	$O(B^2 + N^2)$
FB4	$O(N \cdot L + M)$		$O(M)$	$O(B + N + M)$
FB5	$O(L + M)$	$O(L + B + M)$	$O(M)$	$O(B + M)$
FB 1->5	$O(L \cdot N + M)$	$O(C_{PF} + M)$	$O(C_{OPF} + M)$	$O(B^2 + N^2)$

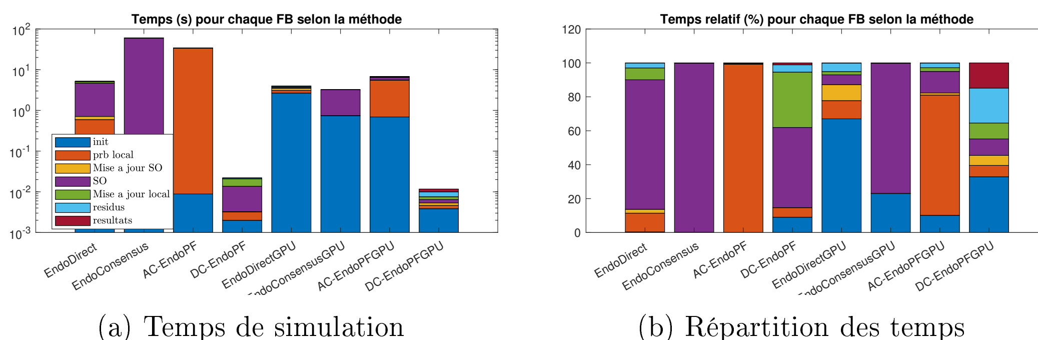
TABLE 4.7 – Blocs fonctionnels et complexité en série et en parallèle des algorithmes pour le marché endogène sur GPU

Bloc	DC - EndoPF	AC-EndoPF	EndoConsensus	EndoDirect
FB0	$O(L \cdot N + M)$	$O(L + M)$	$O(B^2 + M)$	$O(B^2 + M)$
FB 0 bis	$O(ln(M))$		$O(ln(B) + ln(M))$	
FB6	$O(B + M)$			
FB1	$O(ln(N))$			$O(ln(N))$
FB2	$O(1)$			
FB3	$O(N)$	C_{PF}	C_{OPF}	$O(ln(c_b + N_b)$
FB4	$O(ln(L))$		$O(1)$	$O(ln(c_b)))$
FB5	$O(ln(L + M))$	$O(ln(L + B + M))$	$O(ln(M))$	$O(ln(B + M))$
FB 1->5	$O(N + ln(L))$	$O(C_{PF} + ln(N))$	$O(C_{OPF} + ln(M))$	$O(ln(B + M)))$

les cinq premiers pas du cas d'étude.

La Fig. 4.12a représente les temps de calcul des différentes méthodes en échelle logarithmique. Les détails sur les temps totaux sont dans le Tab 4.8. Il apparait que la méthode **DC-EndoPF** est bien plus rapide que les autres méthodes. Le passage sur GPU permet de diviser par presque 2 le temps lors de cette mesure. Cependant, cette méthode repose sur une approximation qui n'a pas vraiment de sens sur un réseau de distribution. Les temps ne sont pas vraiment comparables.

La méthode **EndoDirect** n'est que légèrement plus rapide sur GPU que sur CPU. Cela s'explique par le fait que l'initialisation est une étape qui prend 60% du temps total pour la version sur GPU. Ainsi les transferts mémoires sont dominants, il faudra trouver un moyen de les réduire ou simuler plus de pas de temps.



(a) Temps de simulation (b) Répartition des temps

FIGURE 4.12 – Évolution de la répartition des temps sur le cas Test Feeder

TABLE 4.8 – Temps total (s) de résolution, cas TestFeeder

Methode	DC - EndoPF		AC-EndoPF		EndoConsensus		EndoDirect	
Matériel	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU
Temps	0.02	0.01	34	7	60	3	5	4

Pour toutes les méthodes le passage sur GPU augmente le temps passé lors de l'initialisation mais réduit grandement le temps de résolution.

4 Réécriture algorithmique pour partitionnement sur CPU-GPU

4.1 Lien entre le langage de programmation et l'architecture de calcul

4.1.1 Lien logiciel et physique

Si l'utilisation de Cuda permet de programmer en faisant abstraction de l'architecture de la carte, il reste important d'être conscient de son utilisation pour pouvoir optimiser. Ainsi la grille est l'ensemble des coeurs utilisés. En fonction du nombre de threads et de la mémoire dont ils ont besoin, les blocs sont répartis sur les différents Multi-streaming processors (**MP**). La mémoire partagée de chaque bloc correspond à la mémoire partagée du MP, ainsi plusieurs blocs peuvent physiquement utiliser le même bloc mémoire sans que cela soit possible d'utiliser ce fait lors de la programmation. Un thread utilise un coeur cuda lorsqu'il est actif. Il est possible que l'on demande plus de threads qu'il y a de coeurs cuda, que pendant le calcul certains threads ne font rien ou utilisent de la mémoire non prête. Dans ces cas là, c'est la partie contrôle du GPU qui indique (*schedule*) quels sont les

threads actifs et leur coeurs cuda associés pour chaque instant.

Même si cela n'apparaît nulle part dans le code cuda, les coeurs cuda (et donc les threads) sont matériellement commandés par wraps (groupe de 32). Ainsi chaque groupe de 32 threads exécute exactement la même opération en même temps (sur des données différentes). Si le code présente un embranchement entre les threads d'un même wrap, l'exécution sera sérialisée pour chaque opération différente pour ces threads. Pendant qu'un des embranchements est réalisé, les threads qui n'ont pas cet embranchement seront "actifs" (dans le sens où ils utiliseront un coeur cuda) mais ne feront rien [44].

Le fait que les threads d'un warp soient synchrones n'est plus garanti dans les dernières architectures de GPU. Il est donc possible de rajouter des barrières de synchronisations entre les threads d'un même warp. Dans cette thèse, on cherchera lorsque c'est possible à garder ces threads non divergents pour avoir de bonnes performances même sur les cartes plus anciennes.

4.1.2 Accès mémoires

Comme ce qui a été présenté dans la partie architecture, le temps pour réaliser les différents accès mémoire dépend du type de mémoire. Cependant la manière dont la mémoire est accédée est aussi très importante.

Ainsi, pour la mémoire globale il est important de réaliser des accès mémoires dit *coalescents* c'est à dire que des threads adjacents (en numérotation) doivent accéder à de la mémoire adjacente. En faisant ainsi les accès mémoires pourront effectivement être réalisés en parallèle, sinon ils seront sérialisés.

Par exemple, dans le cas où la mémoire est stocké en 2D ou ligne par ligne (*row major*), il faut que chaque thread lise soit un élément, soit une colonne : *i.e* le thread i accède à l'élément $M[i]$ (tout se fait en une fois) ou $M[:, i]$ (un appel par ligne de donnée). Si le thread i accède à $M[i, :]$ les accès ne sont pas coalescents et on a un accès par donnée !

L'accès mémoire n'est réellement coalescent que si le premier thread accède à une donnée alignée avec un multiple de 32 mots mémoires de 4 octets.

D'un autre côté pour la mémoire partagée, la coalescence n'a pas d'impact mais un autre point peut ralentir grandement l'exécution d'un code cuda : le *bank conflit*. La mémoire partagée est organisée en plusieurs modules de 4 octets appelés *bank*. Si plusieurs threads essaient d'accéder à la même *bank*, des conflits surviennent et les accès sont sérialisés. La seule exception survient lorsque tous les threads d'un

warp accèdent à la même *bank*, dans ce cas ci, la donnée est *broadcast* et un seul accès est nécessaire.

4.1.3 Prise en compte des limites physiques

De plus la mémoire disponible localement pour chaque thread (et donc coeur) est assez limitée. Si celle-ci est remplie, c'est la mémoire globale du GPU qui sera utilisée (de manière totalement transparente pour l'utilisateur). Cependant les accès mémoires sont extrêmement plus longs pour la mémoire globale que pour les registres, la mémoire partagée ou des constantes. Il peut donc être intéressant d'utiliser ces deux dernières mémoires pour éviter de surcharger les registres et par conséquent d'utiliser la mémoire globale.

Enfin la quantité de matériel disponible est limitée. Ainsi même si on peut demander n'importe quel nombre de blocs (et donc de threads), il n'y en aura qu'un certain nombre qui pourront être actifs à la fois. Il est conseillé d'appeler plus de threads qu'il y a de cœurs car pendant que certains sont actifs, le GPU peut préparer la mémoire pour les autres threads et ainsi on utilise le GPU au maximum de sa capacité en permanence.

4.1.4 Synchronisme ou asynchronisme CPU-GPU

Le CPU et le GPU fonctionne en mode asynchrone, cela signifie que lorsque l'appel kernel est lancé, le CPU continue son exécution sans attendre la fin du calcul du GPU sauf mention explicite de synchronisation. Il est ainsi possible de faire des calculs sur CPU et sur GPU en même temps pour paralléliser encore plus l'application. Dans cette configuration une opération sur GPU peut être représentée par la partie haute du schéma Fig. 4.13. En effet même si l'appel kernel du GPU est asynchrone avec le CPU, par défaut les transferts mémoire sont synchrones et donc bloquants, de plus par défaut un seul appel Kernel ne peut être actif à la fois.

Afin d'accélérer les calculs, il est possible de lancer plusieurs appels kernels en même temps et de faire des transferts mémoires asynchrones. Pour que cela soit possible on peut lancer plusieurs *streams* dans le GPU, et demander un transfert asynchrone. Ce type de transfert n'est possible que si la mémoire est *pinned* du côté du CPU. Comme ce qui est montré dans la partie basse de la Fig. 4.13, ceci est particulièrement utile lorsque l'on veut commencer à traiter des données pendant qu'elles sont en train d'être transférées sur le GPU et de les renvoyer au CPU à la fin de leur calcul. On peut remarquer que dans l'encadré ce sont quatre opérations différentes qui sont exécutées en même temps. Pour réaliser plus d'opérations en même temps il faut définir plus de *streams* et séparer chaque appel kernel en plusieurs appels kernel.

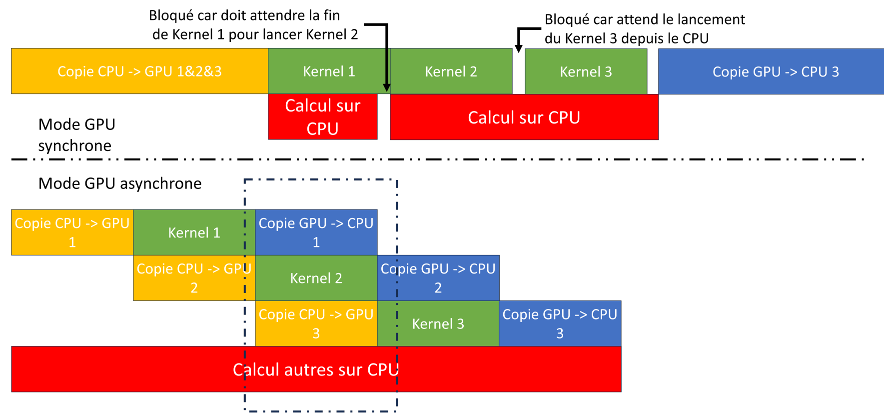


FIGURE 4.13 – Fonctionnement synchrone (haut) et asynchrone (bas) par rapport au CPU

Dans notre cas, le temps d'échange de données en initialisation et en récupération des résultats est négligeable devant le temps de calcul total. Ainsi le fait de devoir *pinned* la mémoire pour permettre l'a-synchronisme risque de faire perdre du temps. De plus on souhaite que tous les calculs soient réalisés sur GPU. Ainsi on conservera un GPU en mode synchrone sans chercher à occuper le CPU pendant les calculs du GPU.

4.2 Optimisation générale

Une fois que les faiblesses des algorithmes sont identifiées des optimisations peuvent être réalisées. Celles-ci peuvent être algorithmiques (changer l'algorithme, régler des paramètres), logicielles (optimiser le code) ou matérielles (utiliser les spécificités du GPU-CPU).

De manière non exhaustive les principales optimisations algorithmiques qui ont été réalisées sont :

— changer la norme pour calculer les résidus pour éviter la sommation d'erreur numérique,
— rendre lorsque possible le facteur de pénalité ρ variable pour accélérer la convergence
— changer la manière de calculer pour prendre en compte la sparsité des données (plus de détail ci-après).
— dans le cas de **DC-EndoPF** réduire la précision pour le respect d'une contrainte tout en contraignant plus pour être sûr de respecter la contrainte originale.

Les optimisations logicielles sont les suivantes :

— permettre un *warm Start* (partir d'une solution précédente),

— utiliser et "ré-utiliser" des objets pour éviter des allocations et des transferts mémoires inutiles,
— changer la manière de représenter les données pour prendre en compte la sparsité (vecteur plutôt que matrice...)
— optimiser les réductions selon [109]

De manière générale, la prise en compte de la sparsité a un effet beaucoup plus bénéfique sur CPU, où les calcul sont sérialisés que sur GPU. En effet la parallélisation diminue l'effet de la réduction du nombre de calcul et on risque de perdre en coalescence.

Pour optimiser une fonction sur GPU, les règles suivantes doivent être suivies le plus possible :
— paralléliser pour tirer le maximum des caractéristiques du GPU (selon l'intensité algorithmique),
— les threads d'un même warp (et si possible bloc) doivent accomplir la même opération pour éviter des divergences qui séquentialiseraient l'exécution,
— les accès à la mémoire globale doivent être minimisés en utilisant la mémoire locale ou partagée lorsque possible,
— limiter le nombre d'appel kernel pour limiter le nombre de synchronisation global du GPU,
— les accès à la mémoire globale doivent être coalescents *i.e* des threads adjacents doivent accéder à de la mémoire adjacente,
— éviter les *bank conflicts* lors des accès en mémoires partagées,
— utiliser les coeurs cuda en simple précision ou avec des calculs entiers.

Ces règles ne sont pas toujours compatibles, et des mesures sont donc nécessaires pour déterminer comment trancher entre deux solutions. De plus le GPU a beaucoup plus d'unités de calcul en flottant simple précision qu'en flottant double précision. C'est pourquoi les calculs seront par défaut réalisés en simple précision. Cependant certains calculs peuvent être très sensibles aux erreurs numériques, ces calculs seront réalisés en double. Cela concerne les PF **NR** et **GS** où la résolution avec que des doubles ou que des simples précisions existent. De plus la recherche de racine de polynômes sont faites sur des doubles précisions pour les **OPF** et marché endogène **EndoDirect** et **EndoConsensus**.

4.3 Prise en compte de la sparsité

Par défaut la création de données pour l'ensemble de l'application se fait avec des matrices pleines. La seule exception est que l'utilisation d'OSQP nécessite de définir les matrices sous un format sparce. Cependant dans plusieurs étapes des algorithmes, le fait d'avoir des matrices pleines impose de nombreux calculs inutiles car de résultat nul de manière prévisible. Cette partie explique comment la sparsité inhérente au cas d'étude a pu être prise en compte de manière efficace

aussi bien par le CPU que par le GPU. Pour cela, on utilisera des structures de données permettant de conserver des accès coalescents en mémoire.

4.3.1 Sparsité pour le marché

Cette partie concerne toutes les méthodes faisant intervenir un marché pair à pair. Cela concerne donc les méthodes de marché endogène ou sans contrainte de réseau.

Dans le cas de base, le nombre de variable du marché est en $O(N^2)$ puisque l'on suppose que tous les agents sont reliés entre eux. Cependant il est possible que certains liens soient inutiles. En effet deux producteurs (ou deux consommateurs) ne peuvent pas échanger entre eux. Ainsi plus il y a un déséquilibre entre le nombre de consommateurs et de producteurs plus le nombre de liens est faible. Le cas extrême étant $N-1$ d'un type et un seul de l'autre. Dans ce cas le nombre de lien vaut N. Dans le cas général, cela permet de passer d'une taille de problème en $O(N^2)$ à une taille en $O(M)$ avec $M = \sum_{n\in\omega} M_n$.

Pour ce faire, la matrice des échanges T de taille $N \cdot N$ a été remplacée par un vecteur T_{lin} de taille M. Pour passer de l'un à l'autre on utilise différents vecteurs :
- le premier indique à quel indice commencent les échanges d'un agent V_{debut},
- un autre indique à quel indice se situe le transposé de l'échange V_{trans},
- deux vecteurs permettent d'indiquer pour chaque échange l'agent et le voisin considéré V_{from} et V_{to}.

On a aussi besoin de connaitre le nombre de voisin de chaque agent M_n. Un exemple avec trois agents peut être vu dans la Fig. 4.14.

Cette méthode est inspirée du format CSC (Compressed Sparse Column) mais nécessite 6 vecteurs au lieu de 3. En effet le format CSC n'aurait eu besoin que de T_{lin}, V_{from} et V_{debut} (aussi appelé *data*, *indices* et *indptr*) dans le cas de Python. Le fait d'avoir plus de vecteurs nous permet d'accéder directement à toutes les

$$T = \begin{pmatrix} 0 & 0 & t_{02} \\ 0 & 0 & t_{12} \\ t_{20} & t_{21} & 0 \end{pmatrix}$$

$T_{lin} = [t_{02}\ t_{12}\ t_{20}\ t_{21}]$ $\quad V_{From} = [0\ 1\ 2\ 2]$

$V_{Trans} = [2\ 3\ 0\ 1]$ $\quad V_{To} = [2\ 2\ 0\ 1]$

$V_{debut} = [0\ 1\ 2]$ $\quad M_n = [1\ 1\ 2]$

FIGURE 4.14 – Exemple de vecteurs créés à partir de la matrice des échanges pour trois agents

informations nécessaires pour chaque calcul. Cela facilite la parallélisation des calculs. De plus cette manière de faire permet d'avoir des accès coalescents pour la lecture et l'écriture pour toutes les opérations. La seule exception est lorsque l'on a besoin d'obtenir la transposé de l'échange où là il est impossible d'avoir à la fois la lecture et l'écriture de coalescent sans pré-chargé en mémoire partagée l'ensemble des données.

L'ensemble des données reliées aux échanges (variables duales, préférences hétérogènes...) sont linéarisées de la même manière, les correspondances sont identiques.

4.3.2 Sparsité pour le réseau

Cette partie concerne les méthodes impliquant un réseau maillé, c'est à dire les méthodes de Power Flow **NR** et **GS** (et donc indirectement **AC-EndoPF**).

La même logique sur les liens entre les agents peut être appliquée sur les liens entre les bus. En effet, même dans un cas de réseau maillé, tous les bus ne sont pas reliés entre eux. Dans le cas extrême d'un réseau radial, il y a autant de bus que de lignes (à un près).

On peut donc définir les mêmes vecteurs que précédemment pour linéariser les impédances. Les différences sont les suivantes.

— Il n'y a pas besoin de connaitre la transposée (donc il n'y a pas besoin de V_{trans}).
— En plus de l'impédance de chaque ligne, chaque bus a sa propre impédance. Ainsi le vecteur représentant la taille vaut $V_{taille} = [(L_b + 1)_{b \in \mathcal{B}}]$ avec L_b le nombre de ligne arrivant ou partant du bus b.

Dans le cas d'un réseau radial, la manière de stocker les informations est différente. En effet dans ce cas là, les informations importantes sont les enfants de chaque bus. Les impédances ne sont pas nécessaires car soit elles sont déjà incluses dans une matrice représentant le système à résoudre (cas des **OPF** et **EndoDirect**) soit on peut directement utiliser les impédances de chaque lignes (cas des **PF** radiaux).

4.4 Problème de partage

4.4.1 Introduction

Pour rappel un problème de partage peut s'écrire sous la forme suivante :

$$\underset{x}{\operatorname{argmin}} \sum_i f_i(x_i) + v\big(\sum_i x_i\big) \tag{4.6}$$

La méthode itérative permettant de résoudre ce problème est :

$$x_i^{k+1} = \underset{x_i}{\operatorname{argmin}}\, f_i(x_i) + \frac{\rho}{2}||x_i - x_i^k + \overline{x}^k - \overline{z}^k + u_i^k||_2^2 \tag{4.7a}$$

$$z^{k+1} = \underset{z}{\operatorname{argmin}}\, v(N\overline{z}) + \frac{N\rho}{2}||\overline{z} - u^k - \overline{x}^{k+1}||_2^2 \tag{4.7b}$$

$$u^{k+1} = u^k + \overline{x}^{k+1} - \overline{z}^{k+1} \tag{4.7c}$$

Dans notre cas, la minimisation en fonction de x_i et de z peut se mettre sous la forme d'un polynôme du second degré, et donc le minimum peut être trouvé analytiquement.

Cette formulation se retrouve à chaque fois qu'un agent doit résoudre son problème local d'un marché pair à pair. Cela concerne donc l' algorithme **ADMM** du marché pair à pair et les algorithmes **EndoPF**, **EndoConsensus** et **EndoDirect** des marchés endogènes.

Cette partie s'attellera à démontrer la meilleure manière de paralléliser cette résolution. Pour cela, chaque manière de paralléliser sera analysée en fonction des particularités matérielles du GPU. Les méthodes les plus prometteuses seront implémentées pour être comparées.

4.4.2 Parallélisation sur les agents

La manière de décentraliser la plus naturelle serait de paralléliser sur les agents par analogie avec ce qui serait réalisé lors du déploiement réel. La partie gauche du schéma Fig. 4.15 représente la parallélisation du problème local pour deux agents, le premier ayant 2 pairs et le deuxième ayant qu'un pair.

Les avantages de cette manière de faire est qu'il n'y a besoin d'aucune synchronisation entre les threads à l'intérieur du bloc fonctionnel. Ainsi toute la résolution peut se faire en un seul appel kernel. Cette configuration pourrait être adaptée à une parallélisation sur CPU, mais lève plusieurs problème dans le cas d'un parallélisation sur GPU.

Tout d'abord, si les agents n'ont pas le même nombre de pairs, les différents threads ne devront pas faire le même nombre de calcul. De plus, chaque agent termine le calcul lorsqu'il atteint son critère de terminaison. Mais cela provoque des embranchements divergents entre les threads (ceux qui doivent continuer la simulation et ceux qui doivent la terminer). L'étape suivante (**FB 3**) est bloquante ; les agents qui terminent ne feront rien, attendant les autres. Il y aura donc beaucoup de threads inactifs, ce qui ne permettra pas de gagner du temps. En effet, il y a deux possibilités pour un agent qui a terminé. Soit le thread de l'agent est lancé dans le cadre d'un warp actif et ne fait rien. Cela utilisera de toute façon des ressources GPU. Soit on fait en sorte qu'il ne soit pas lancé, mais dans ce cas, les

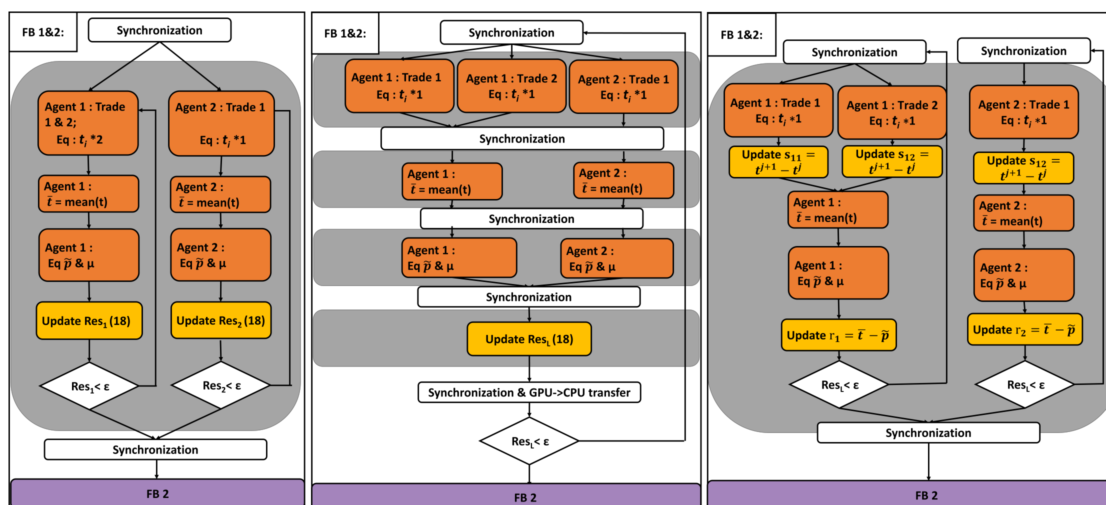


FIGURE 4.15 – Parallélisation sur les agents (gauche), sur les échanges (milieu), ou sur les échanges par block (droite). Zoom sur deux agents, un avec 2 pairs et l'autre avec un unique pair. Les contours gris représentent les appels kernels

threads contigus ne peuvent pas accéder aux mémoires adjacentes (no-coalescent memory access).

Finalement on peut remarquer que l'on n'a pas parallélisé autant que l'on pouvait. En effet on peut paralléliser sur chaque x_i et le calcul de $\overline{x}$ et des résidus sont des réductions qui sont donc parallélisables.

4.4.3 Parallélisation maximale

Dans cette partie on cherchera à paralléliser le plus possible les calculs.

Au milieu de la figure 4.15, le calcul des transactions est parallélisé afin d'éviter la boucle for variable en fonction de la taille, et le calcul des résidus sur l'ensemble des agents. Tous les agents s'arrêtent lorsque les résidus maximaux sont suffisamment faibles. Chaque étape est réalisée par un appel kernel (rectangle gris), et la taille est toujours adaptée au calcul, de sorte que le calcul est entièrement parallélisé. Chaque thread traite exactement la même opération que les autres.

Néanmoins, cela se fait en ajoutant une synchronisation entre chaque étape (plusieurs appels kernels). Ainsi en forçant tous les agents à calculer en même temps et à s'attendre, on peut perdre énormément de temps car on s'aligne sur le plus lent. De plus comme on utilise plusieurs appels kernel, chaque lancement peut faire perdre du temps, puisqu'il faut à chaque début d'appel relire dans la mémoire globale.

4.4.4 Synchronisation minimale

Dans cette partie on cherchera à limiter au maximum le nombre d'appel kernel (et donc le nombre de synchronisation). Cela nous permettra en plus de limiter les accès mémoires puisque l'on pourra garder les informations en local ou dans la mémoire partagée.

<La partie gauche de la fig. 4.15 montre le compromis entre la parallélisation et l'utilisation d'un unique appel kernel. Comme dans la première proposition, un seul appel au noyau est utilisé, et chaque agent a son résidu. Toutes les étapes sont réalisées en ayant un bloc de thread par agent et en utilisant la mémoire partagée pour calculer une réduction par bloc. Ainsi, toutes les étapes sont entièrement parallélisées, mais le **FB1c** n'utilise qu'un seul thread par bloc (tous les autres threads doivent attendre). Chaque thread écrit dans la mémoire partagé si le calcul doit être continuer ou non. Cette variable est mise à faux si un calcul résiduel est supérieur à la précision demandée et remise à vrai à chaque début de boucle. Ensuite, une réduction est réalisée pour passer de ce vecteur à une unique valeur pour le bloc.Cette fois, l'agent peut s'arrêter une fois qu'il a atteint la convergence souhaitée. La divergence se fait entre les blocs ce qui ne pose pas de problème. Cependant cette configuration demande beaucoup de ressources du GPU.

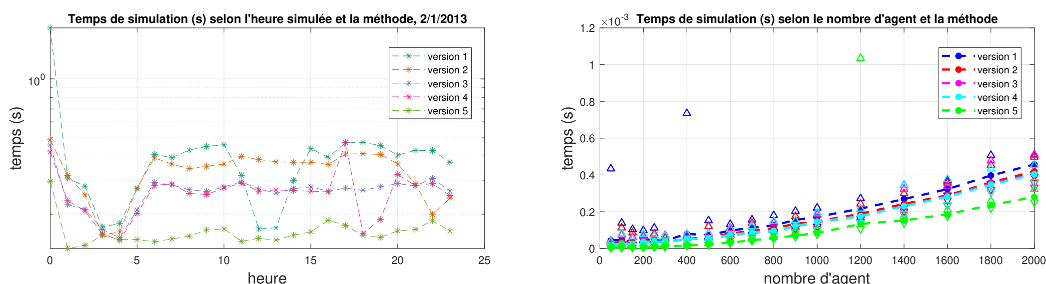
Une solution intermédiaire a été de garder séparé le calcul du résidus et le reste du problème local. En faisant ainsi cela permet de faire plusieurs itérations du problème local en un seul appel kernel avant de calculer le résidu unique à tous les agents.

4.4.5 Conclusion

Comme le bloc de résolution du problème de partage est une optimisation, il est difficile de le tester de manière indépendante. En effet le test de cette fonction sur des données aléatoires risque de donner des résultats aberrants. De plus dans le cadre de la résolution du marché, le problème local doit être résolu à chaque itération. Ainsi la méthode la plus performante doit l'être sur la résolution totale et non pas sur une unique itération.

Ainsi pour tester la meilleure manière de paralléliser, chaque méthode a été intégrée dans une résolution de marché pair à pair. La seule différence entre ces méthodes étant le problème local,cela devrait nous permettre de discriminer la meilleure méthode. Les cinq versions qui ont été testées sont les suivantes.

- Version 1 : la version avec le parallélisme maximal, c'est à dire avec un appel kernel par sous-bloc.
- Version 2 : une version intermédiaire où les trois étapes sont réalisées en un seul appel kernel. Dans cette version une seule itération du problème local est réalisée par l'appel.



(a) Temps de simulation pour le cas Européen

(b) Temps de simulation sur des cas aléatoires

FIGURE 4.16 – Comparaison des différentes résolutions du problème local

— Version 3 : la même que précédemment sauf que plusieurs itérations sont réalisées dans le même appel kernel. Entre chaque itération les données nécessaires sont gardées dans la mémoire locale.
— Version 4 : cette fois les données sont stockées dans la mémoire partagée pour éviter de surcharger la mémoire locale (ce qui forcerait l'utilisation de la mémoire globale).
— Version 5 : c'est la version avec le moins de synchronisation. Toutes les itérations du problème local avec le calcul des résidus sont réalisées dans un seul appel kernel.

La Fig. 4.16a représente le temps de simulation pour les 24h de la journée du 2 janvier 2021 pour le cas Européen (section 2.6.2.2). On peut remarquer de la variabilité entre les performances des différentes méthodes. Cependant la version avec le moins de synchronisation est clairement la plus efficace et reste toujours plus rapide que les autres.

Afin de vérifier ce résultat dans le cas global, on réalise aussi la comparaison sur des cas aléatoires. Les paramètres choisis sont identiques pour toutes les méthodes. La figure obtenue est Fig. 4.16b. On peut y voir que sur toutes les méthodes ont globalement le même comportement pour le passage à l'échelle. Cependant la dernière méthode a des temps de calculs qui augmentent plus lentement avec les dimensions des cas.

Ainsi pour toute la suite on considèrera que le problème de partage sera résolu via la méthode avec le moins de synchronisation. Ainsi tous les problèmes locaux de marchés seront résolus en un seul appel kernels.

Pour rappel, les méthodes concernées sont **ADMM** pour le marché pair à pair et toutes les méthodes de marché endogènes.

4.5 Stockage de la sensibilité

Ce problème est spécifique à la résolution de l'algorithme **DC-EndoPF**, même si certains types de calculs peuvent se retrouver dans d'autres algorithmes.

La matrice de sensibilité du réseau G_{sensi} de taille $L \cdot N$ est une matrice constante qui ne change pas pendant la simulation. Cette matrice sera donc stockée directement sur le GPU. Or le calcul sur GPU est très dépendant de l'efficacité des accès mémoires. Cette matrice est utilisée dans de nombreux calculs, il est donc important d'optimiser la manière de stocker cette matrice. Il apparaît donc que pour réaliser des calculs avec appel coalescent de la mémoire il pourrait être intéressant d'utiliser principalement la transposé de cette matrice. En réalité stocker la transposée de la matrice revient à stocker cette matrice en "*column-major*", là où toutes les autres matrices sont par défaut stockées en "*row-major*".

4.5.1 Méthodologie

L'objectif de cette partie est de déterminer la meilleur manière de paralléliser sur GPU tout en utilisant la manière la plus efficace de stocker la matrice de sensibilité. La méthodologie est basé sur celle de [103].

Pour cela, on étudiera le passage à l'échelle de chaque calcul dépendant de la manière de stocker. On étudiera à la fois le cas de l'augmentation du nombre d'agent et le cas d'augmentation du nombre de lignes contraintes.

Pour optimiser, les paramètres variables sont l'organisation des threads du GPU (par bloc selon les agents ou les lignes, en une ou deux dimensions avec utilisation de mémoire partagée ou non) et le stockage de la matrice de sensibilité G.

Pour cela des matrices d'une taille donnée sont générées aléatoirement et le temps de calcul de l'appel kernel est mesuré. Ce calcul est répété de nombreuses fois en copiant les données à chaque fois pour éviter certaines optimisations du GPU. Enfin pour chaque méthode le travail, les données transférées sont évaluées pour pouvoir calculer l'intensité algorithmique et ainsi pouvoir placer les différentes méthodes sur la RoofLine (section 2.2.1).

4.5.2 Calculs utilisant la matrice

Pour chacun des calculs utilisant la matrice, on détaillera les différentes manières de paralléliser. Pour chacune de ces manières on indiquera le nombre d'opérations et d'accès mémoire.

Coefficient α Pour rappel le calcul de α est le suivant :

$$\alpha_{ln} = G_{ln} * P_n;$$

Dans l'objectif d'optimiser le temps de calcul de ces coefficients, six méthodes de calcul de α ont été testées via des micro-benchmarks :

- un bloc par agent : permet l'utilisation de mémoire partagée pour la récupération de P_n
- une répartition en 2D : permet l'accès mémoire en utilisant les coordonnées x et y des threads
- une répartition en 1D : il suffit de calculer $n = i\%N$ pour obtenir le bon coefficient de P_n (avec i le i-ème coefficient de alpha en parcourant par ligne)
- les trois précédentes mais avec G transposée (dans le dernier cas, $n = i/N$).

Nombre d'opération Quelle que soit la méthode, le calcul est le même, sauf pour les méthodes en 1D où il faut rajouter le calcul de l'indice pour récupérer la bonne valeur de Pn. On a donc le nombre de calcul en $O(2L \cdot N)$ pour les méthodes en 1D et $O(L \cdot N)$ pour toutes les autres méthodes.

Accès mémoire Il y a deux cas possibles. Lorsque les calculs sont regroupés par bloc d'agent, alors l'utilisation de mémoire partagée est possible, ainsi le nombre de lecture est en $O(N \cdot L + N)$. Lorsqu'il n'y pas de mémoire partagée, le nombre de lecture est égale à deux fois le nombre de calcul (donc $O(2L \cdot N)$). Dans tous les cas le nombre d'écriture est en $O(N * L)$. On remarque que lors que l'on a un bloc par agent, chaque bloc calcule une colonne de α ce qui correspond à un accès non coalescent de la mémoire, sinon toutes les autres méthodes utilisent des accès coalescents. Ainsi le fait d'utiliser la transposé de G sera surtout efficace pour la méthode en regroupant par bloc.

Comme changer le sens de G change aussi celui de α, il faut regarder l'effet de ce changement sur les calculs qui dépendent de α :

- calcul de Q^{tot}
- calcul de Q^{part}

Coefficent Qpart Pour rappel le calcul de Q^{part} est le suivant :

$$Q^{part}_{ln} = \sum_{j>n} \alpha_{lj}$$

Ce calcul ressemble à une réduction puisque l'on réalise une somme d'un ensemble de terme. Ce calcul est difficilement parallélisable car soit on réalise un calcul avec beaucoup de dépendance, soit on réalise beaucoup de fois les mêmes calculs. C'est cette deuxième solutions qui a été choisie. Lors du test des appels kernel, huit méthodes de calcul de Q^{part} ont été testées.

- Un bloc par ligne l : on utilise la mémoire partagée pour stocker α_{ln}.
- Un bloc par agent.

- Les méthodes précédentes avec G et alpha transposés.
- Un bloc par ligne, en calculant la boucle dans le sens descendant avec mémoire.
- Un bloc par ligne, en calculant la boucle dans le sens descendant avec mémoire, G et alpha transposé.
- Un bloc par ligne, en calculant la boucle dans le sens descendant.
- Un bloc par ligne, en calculant la boucle dans le sens descendant, G et alpha transposé.

Nombre d'opération Dans tous les cas sauf ceux dit avec mémoire, le nombre de calcul est le même. Soit un agent n fixé, le nombre de calcul est (N-n). Il faut le faire pour toutes les lignes et pour tous les agents :

$$\begin{aligned} W &= L \cdot \sum_{0<n\leq N} (N-n) \\ &= L \cdot (N^2 - \frac{N(N+1)}{2}) \\ &= L \cdot (\frac{N^2-N}{2}) \end{aligned} \tag{4.8}$$

Donc le nombre d'opération est en $O(L \cdot N^2)$.

Dans les cas dit avec mémoire, le nombre de calcul va dépendre du nombre de thread par bloc par rapport au nombre d'agent. S'il y a autant ou plus de thread que d'agent, alors le nombre de calcul sera le même que précédemment. Soit $K = N/T$ avec T le nombre de thread par bloc, chaque thread s'occupera de K agents. On remarque que soit n_1 et n_2 deux agents d'un même thread ($n_1 = n_2 + T$) on a :

$$Q_{n_2}^{part} = \sum_{k=n_2+1}^{N} \alpha_k = \sum_{k=n_2+1}^{n_1} \alpha_k + \sum_{k=n_1+1}^{N} \alpha_k = \sum_{k=n_2+1}^{n_1} \alpha_k + Q_{n_1}^{part} \tag{4.9}$$

Cette équation est en réalité vraie pour tous les agents, mais le fait de ne le faire que pour des agents étant fait par le même thread permet de ne pas ajouter des problèmes de synchronisation et d'accès mémoire puisque le résultat précédent peut être stocké en local.

Dans ce cas-ci le nombre d'opération devient $O(L \cdot T \cdot N)$.

Accès mémoire Il y a de nouveau deux cas possibles. Lorsque les calculs sont regroupés par bloc de ligne, alors l'utilisation de mémoire partagée est possible, ainsi le nombre de lecture est en $O(N * L)$. Lorsqu'il n'y pas de mémoire partagée (un bloc par agent), le nombre de lecture est égale au nombre de calcul (donc $O(L \cdot N^2)$). Dans tous les cas le nombre d'écriture est en $O(N * L)$.

Selon si la matrice est transposée ou pas et si les blocs sont sur les agents ou les lignes, les threads consécutifs accèdent à la mémoire de manière consécutifs ou pas. De plus on peut remarquer que si on a un bloc par agent n, tous les threads d'un bloc font le même nombre d'opération, il n'y a donc pas de divergence dans un warp. La variable Q^{part} intervient dans les calculs de C_{p2b}. Il faut donc aussi étudier le calcul de ces coefficients.

Coefficient Qtot Pour rappel le calcul de Q^{tot} est le suivant :

$$Q_l^{tot} = GP = \sum_{n\in\Omega} \alpha_{ln}$$

Cependant on peut remarquer que :

$$Q_l^{tot} = Q_{0l}^{part} + \alpha_{0l} \tag{4.10}$$

Lors du test des appels kernel, 3 méthodes de calcul de Qtot ont été testées :
- calcul à partir de alpha seulement
- calcul à partir de alpha et de Qpart
- calcul à partir de alpha et de Qpart avec G transposée

Nombre d'opération Pour la première méthode, on a une réduction d'une matrice de taille $N \cdot L$ vers L en sommant chaque ligne. Il y a donc $O(N \cdot (L-1))$ calculs. Pour les deux autres méthodes, on calcule directement la valeur de chaque coefficient en faisant une somme, donc $O(L)$ calculs.

Accès mémoire Dans le cas de la première méthode on lit toutes les valeur de alpha, donc il y a $O(N * L)$ lectures. Dans le deuxième cas on ne lit que $2 * L$ éléments. Dans tous les cas le nombre d'écriture est en $O(L)$.

Dans le premier cas le calcul de Qtot consiste en une réduction partielle de $L \cdot N$ vers L. Pour ce calcul-ci il est intéressant d'avoir d'avoir les lignes l (du réseau) représentées par les lignes (de la matrice), et les agents par les colonnes puisque l'on somme sur tous les agents, avec la ligne fixée par bloc. Faire l'inverse serait très peu efficace, c'est pourquoi cela n'a pas été testé. Quel que soit le sens de calcul, Q^{tot} est un vecteur, cela n'influence en rien les calculs suivants.

Cp2 Pour rappel le calcul de C_{p2} est le suivant :

$$C_{p2} = \rho_1 M_n \sum_{l\in L} \left((|\kappa_{l1}^k| - |\kappa_{l2}^k|)G_{ln} + 2G_{ln}Q_{ln}^{part} \right)$$

On va d'abord tester l'efficacité en séparant le calcul de C_{p2} en trois parties :

$$C_{p2} = \rho_1 M_n (C_{p2a} + C_{p2b}) \tag{4.11}$$

avec :

$$c_{p2a} = \sum_{l \in L} (|\kappa_{l1}^k| - |\kappa_{l2}^k|) G_{ln} \tag{4.12}$$

en supposant pour la suite déjà connaître la différence $|\kappa_{l1}^k| - |\kappa_{l2}^k|$ en entrée de la fonction et :

$$c_{p2b} = \sum_{l \in L} 2 G_{ln} Q_{ln}^{part} \tag{4.13}$$

Pour le calcul de ces deux quantités, à chaque fois deux méthodes ont été testées :
- un bloc par agent
- un bloc par agent avec G transposée

Nombre d'opération Quelle que soit la manière de faire, le nombre d'opérations est le même pour C_{p2a}. Dans un premier temps il faut réaliser la multiplication entre G et la différence des κ, soit $N * L$ calculs. Ensuite il faut réaliser la réduction de $N * L$ vers N, donc $N \cdot (L-1)$. Le nombre total de calcul est donc $O(N \cdot (2L-1))$. En ce qui concerne le calcul de C_{p2b}, dans un premier temps il faut réaliser la multiplication entre G et Q^{part} soit $N * L$ calculs. Ensuite il faut réaliser la réduction de $N * L$ vers N, donc $N \cdot (L-1)$ calculs. Puis enfin le résultat est multiplié par 2, ce qui rajoute N calculs. Le nombre total de calcul est donc $O(N \cdot 2L)$.

Accès mémoire Quelle que soit la méthode, le nombre de lecture est le même, et il est de $O(2 * N * L)$ Dans tous les cas le nombre d'écriture est en $O(N)$.

4.5.3 Conclusion

Dans le tableau suivant sont regroupées les différentes valeurs du *work* en nombre d'opération et les transferts de données. Ce qui nous permet d'obtenir l'intensité algorithmique de chaque appel kernel. Le temps indiqué en seconde est celui nécessaire pour réaliser tous les tests (en comptant uniquement les appels kernels sans les transferts ou génération de donnée). Ainsi on peut noter que le temps est en seconde mais la simulation dure plusieurs minutes. En effet chaque mesure de temps est une moyenne sur 10 essais, et la majorité du temps de simulation est utilisé pour créer les matrices et allouer leur valeur sur le GPU.

On peut voir dans le tableau que les temps peuvent être extrêmement différents selon la manière de réaliser un même calcul. Cela démontre l'importance de réaliser cette démarche lorsque la parallélisation n'est pas évidente.

Table 4.9 – étude Rooofline (asymptotique)

Méthodes	W	D	I	t
Qpart0	$0.5 \cdot L \cdot N(N-1)$	$2 \cdot L \cdot N$	N/16	70
Qpart1	$0.5 \cdot L \cdot N(N-1)$	$0.5 \cdot L \cdot N(N+1)$	1/4	>> 105
Qpart2	$0.5 \cdot L \cdot N(N-1)$	$2 \cdot L \cdot N$	N/16	71.6
Qpart3	$0.5 \cdot L \cdot N(N-1)$	$0.5 \cdot L \cdot N(N+1)$	1/4	105
Qpart4	$L \cdot T \cdot N$	$2 \cdot L \cdot N$	T/8	7.77
Qpart5	$L \cdot T \cdot N$	$2 \cdot L \cdot N$	T/8	8.53
Qpart6	$0.5 \cdot L \cdot N(N-1)$	$2 \cdot L \cdot N$	N/16	71
Qpart7	$0.5 \cdot L \cdot N(N-1)$	$2 \cdot L \cdot N$	N/16	70.7
alpha0	$L \cdot N$	$L \cdot (N+1)$	1/4	5.7
alpha1	$L \cdot N$	$2 \cdot L \cdot N$	1/8	0.86
alpha2	$2 \cdot L \cdot N$	$2 \cdot L \cdot N$	1/4	0.849
alpha3	$L \cdot N$	$L \cdot (N+1)$	1/4	0.86
alpha4	$L \cdot N$	$2 \cdot L \cdot N$	1/8	0.88
alpha5	$2 \cdot L \cdot N$	$2 \cdot L \cdot N$	1/4	0.85
Cp2a0	$N \cdot (2L-1)$	$2 \cdot N \cdot L + N$	1/4	1.89
Cp2a1	$N \cdot (2L-1)$	$2 \cdot N \cdot L + N$	1/4	0.5
Cp2b0	$2 \cdot L \cdot N$	$2 \cdot N \cdot L + N$	1/4	7.6
Cp2b1	$2 \cdot L \cdot N$	$2 \cdot N \cdot L + N$	1/4	0.84
Qtot0	$N \cdot (L-1)$	$L \cdot (N+1)$	1/4	0.536
Qtot1	L	$3 \cdot L$	1/12	0.151
Qtot2	L	$3 \cdot L$	1/12	0.145

On peut voir sur la Fig.4.17 le position des méthodes sur la Roofline. On remarque que pour toutes ces méthodes l'intensité opérationnelle est très faible. Cependant certaines atteignent de très bonnes performances. Le dépassement est sûrement dû à une optimisation du GPU lors du calcul qui a réussis à mieux gérer la mémoire ou à augmenter la fréquence au delà de la valeur nominale. La figure suivante, Fig.4.18 montre le positionnement sur la roofline des méthodes calculant Q^{part}. On peut y remarquer que pour certaines méthodes l'intensité augmente très rapidement pour que le problème devienne "computationnal bound". Cependant ce ne sont pas ces méthodes qui sont les plus rapides pour réaliser le calcul, ces méthodes faisant beaucoup de calculs "redondants".

Enfin sur la dernière figure, Fig. 4.19, on y compare les temps cumulés selon si c'est la transposée qui est utilisée ou non (dans le cas où il y a plusieurs méthodes pour le même calcul, c'est la meilleure qui a été gardée). On peut y voir que même

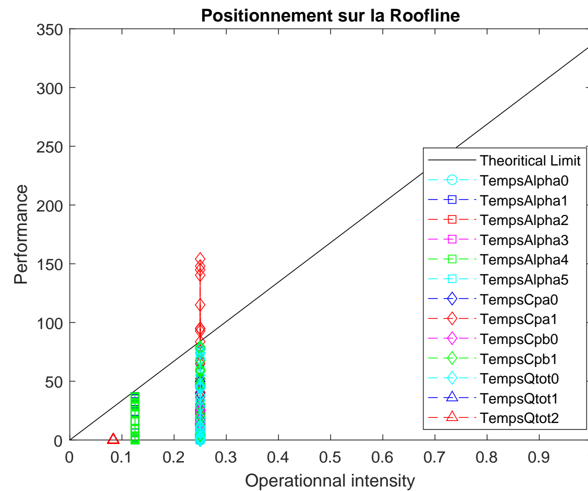


FIGURE 4.17 – Position sur la Roofline des méthodes (sauf Q^{part})

si la différence n'est pas très importante, utiliser la transposée permet d'être de plus en plus rapide avec l'augmentation de la taille du problème Tab. 4.10. Il est important de noter que le nombre d'agent étant indépendant du nombre de bus, il est possible d'avoir peu d'agents et beaucoup de lignes ou inversement.

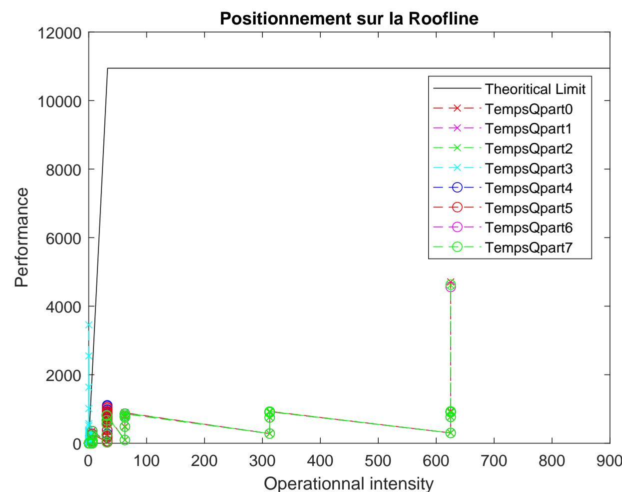


FIGURE 4.18 – Position sur la Roofline du calcul de Q^{part}

Ainsi en stockant la transposée plutôt que la matrice de base, on peut diviser le temps par deux pour une taille de 10000 lignes et plus de 5000 agents.

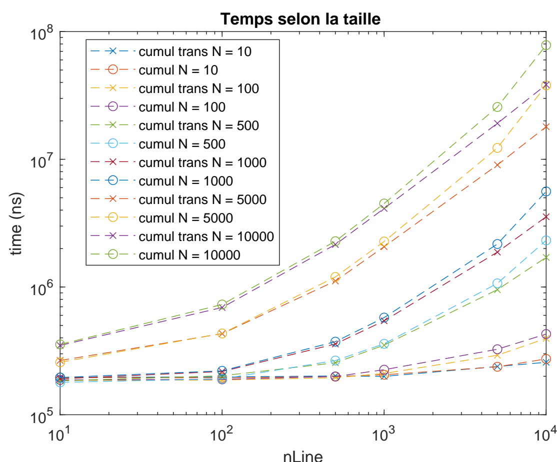


FIGURE 4.19 – Temps du cumul des meilleurs méthodes pour la version avec ou sans transposée

TABLE 4.10 – Speed-up (%) d'utiliser la transposée

NAgent / Nline	10	100	500	1000	5000	10000
10	-0.27	-1.6	-1.3	2.9	-0.5	5.9
100	0.90	1.2	2.3	6.6	10	7.6
500	-2.7	-5.8	3.9	2.4	**11**	**26**
1000	2.3	1.2	4.1	5.7	**14**	**37**
5000	-3.3	0.30	7.1	9.0	**26**	**53**
10000	1.8	5.2	5.8	8.5	**26**	**51**

4.6 Fonction avec dépendance

L'objectif de cette partie est de montrer comment des fonctions peu parallélisables ont été implémentées sur GPU. Ce style de fonction est plus rapide sur CPU. Cependant, le fait de tout réaliser sur GPU permet d'éviter des transferts mémoires entre CPU et GPU.

De manière générale, cela concerne les étapes où il faut parcourir les bus d'un réseau dans un sens particulier. Cela peut être des enfants vers les ancêtres ou inversement. Le calcul de la tension de la méthode de **GS** est aussi dans cette catégorie. En effet, pour une itération donnée, chaque coefficient de la tension dépend de calcul des précédents coefficients.

4.7 Parcours des bus

Le parcours des bus est utilisé dans les méthodes de PF en *load flow* (**Cur** et **BackPQ**). On peut aussi l'utiliser pour toutes les méthodes à base d'OPF sur un réseau radial pour initialiser les flux de puissances dans les lignes en fonction de celles qui sont injectées ou soutirées.

Tout d'abord, on peut remarquer que la dépendance entre les calculs impose de faire une synchronisation. Sur GPU, il y a deux manières de synchroniser. Il faut soit refaire un autre appel kernel, dans ce cas là on peut être sûr que tous les threads ont fini leurs calculs. L'autre méthode est d'ajouter une barrière de synchronisation. Ceci ne peut être fait que pour synchroniser des threads dans un même bloc. Comme le nombre de synchronisations peut être très important (égale au nombre de bus dans le cas d'un réseau en ligne droite), il a été décidé de choisir la deuxième solution. Ceci nous impose donc de calculer le parcours du réseau sur un unique bloc. Cependant, on pourrait utiliser plusieurs blocs dans le cas où l'on aurait plusieurs réseaux indépendants.

Ainsi pour paralléliser on associe à chaque bus un thread. Ce thread garde en mémoire locale son nombre d'enfants, où lire les identifiant de ses enfants et s'il peut calculer. On utilise la mémoire partagée pour stocker la liste de l'ensemble des enfants et une liste de booléen indiquant pour chaque bus s'il a calculé.

Ainsi on commence par l'ensemble des bus n'ayant pas d'enfant. Une fois qu'ils ont réalisés les calculs nécessaires, ils modifient la mémoire partagée pour indiquer que leurs calculs sont finis. Ensuite tous les bus qui n'ont pas fini mettent leur variable indiquant qu'il doivent calculer sur vrai. Ensuite il parcourt leurs enfants. Si au moins un enfant n'a pas fini, cette variable repasse à faux.

Si l'on doit parcourir le réseau dans l'autre sens, on commence par le bus 0 qui n'a aucun ancêtre. Chaque bus regarde si son ancêtre a fini avant de calculer. On considère le calcul global comme étant fini par défaut. Tant qu'il existe un bus qui n'a pas fini, il indique que le calcul global n'est pas fini.

Dans le cas où l'on a plus de bus que de threads, il faut que chaque thread gère plusieurs bus. Dans ce cas là, on crée statiquement des tableaux pour stocker les différentes valeurs locales. Ensuite pour chaque passe, il faut que les threads parcourent l'ensemble des bus dont ils ont la charge.

4.8 Calcul de la tension

Cette section se concentre sur le calcul suivant pour l'algorithme de **GS**, pour un bus i et une itération k :

$$\underline{E}_i^{k+1} = \frac{1}{\underline{Y}_{ii}} \cdot \left(\frac{P_i - jQ_i}{\underline{E}_i^*} - \sum_{l=1}^{i-1} \underline{Y}_{il}\underline{E}_l^{k+1} - \sum_{l=i+1}^{B} \underline{Y}_{il}\underline{E}_l^{k} \right) \tag{4.14}$$

On remarque que le terme central dépend de termes de la même itération k que celle que l'on doit calculer. Ainsi quelle que soit la manière de paralléliser, une boucle pour parcourir les bus est nécessaire.

Tout comme le parcours du réseau, on peut mettre une boucle dans un unique bloc pour gérer l'ensemble des bus. On peut précharger en mémoire partagée les valeurs des tensions pour éviter de devoir lire plusieurs fois dans la mémoire globale. En modifiant directement la valeur de la tension après son calcul, on peut faire en sorte que les deux sommes valent : $\sum_{l=1}^{B} \underline{Y}_{il}\underline{E}_l$. Ainsi pour chaque itération il faut réaliser une réduction.

Cette manière de faire a l'avantage d'être simple, puisque, quelle que soit l'itération, le calcul effectué est identique. La seule différence repose sur les données qui doivent être lues. Cependant, on remarque que cela nous force à n'utiliser qu'un seul bloc, alors que certains calculs sont complètement indépendants.

Ainsi on peut remarquer que le calcul précédent peut être séparé en plusieurs étapes. À chaque itération globale k, on initialise les tensions pour chaque bus :

$$E_i^0 = \frac{P_i - jQ_i}{Y_{ii}E_i^*} - \sum_{l=i+1}^{B} \frac{Y_{il}}{Y_{ii}} E_l^k \quad \forall i \in B \tag{4.15}$$

Ici, les calculs sont indépendants et peuvent donc être parallélisés avec un bloc par bus. Ceci permet de réaliser les réductions sur chaque bloc. Ensuite, il faut séquentiellement calculer les tensions ainsi (avec p l'itération jusqu'à p = B) :

$$E_i^p = E_i^{p-1} - \frac{Y_{ip-1}}{Y_{ii}} E_{p-1}^{p-1} \quad \forall i > p \in B \tag{4.16}$$

Cette opération est une opération linéaire sur un vecteur, on peut donc y associer un thread par bus.

Le calcul précédent est un calcul en complexe. Il faut donc récupérer la partie réelle et imaginaire pour réaliser le calcul.

Les termes $\frac{Y_{il}}{Y_{ii}}$ peuvent être précalculés lors de l'initialisation, comme ils ne changent pas entre les itérations.

Un problème se pose si l'on utilise la sparcité du réseau. En effet le terme $\frac{Y_{ip-1}}{Y_{ii}}$ est non nul uniquement s'il existe une ligne ente le bus i et le bus $p-1$. Ainsi le fait d'accéder pour chaque bus $i > p$ directement au terme $\frac{Y_{ip-1}}{Y_{ii}}$ n'est pas facile. Il faudrait pour chaque bus stocker sa position pour chacun de ses voisins comme cela a été fait pour représenter les réseaux radiaux.

Pour résoudre trois méthodes ont été proposées :

- Calculer la tension en un seul appel kernel dans un bloc. La mémoire partagée est utilisée pour stocker la tension et pour faire les réductions.
- Calculer en deux appels distincts. Le premier calcule l'initialisation E_i^0 avec un bloc par bus. Le deuxième appel kernel fait l'ensemble des itérations en stockant la tension dans la mémoire partagée.
- Même type de calcul que la méthode précédente. Mais les impédances sont préchargées dans la mémoire partagée.

Pour évaluer les performances de chaque méthode, on va utiliser la même méthode que celle que nous avons utilisée pour déterminer le stockage de la sensibilité. Nous allons donc tester les trois implémentations avec des données aléatoires de différentes tailles. Comme pour chacune des méthodes on doit faire B itérations, les trois méthodes sont globalement *computationnal bound*. On va donc plutôt se concentrer directement sur les performances plutôt que de se positionner sur la Roofline.

Ainsi les cas qui ont été testés sont les suivants. Le nombre de bus prend les valeurs de 10, 100 et 500. Et pour chaque nombre de bus 5 configurations ont été testées :

- le cas radial avec $L = B - 1$,
- le cas avec $L = 0.25 * (B * (B - 1)/2)$
- le cas avec $L = 0.5 * (B * (B - 1)/2)$
- le cas avec $L = 0.75 * (B * (B - 1)/2)$
- le cas avec $L = (B * (B - 1)/2$.

Ce qui donne un total de 15 cas, avec le nombre de bus changeant tout les 5 cas. La figure en résultant est la Fig. 4.20.

On peut remarquer que globalement le temps de calcul augmente avec les dimensions du problèmes. Le fait de séparer les calculs en deux parties permet d'effectivement réduire les temps de calcul. A petite dimension les différentes méthodes sont plutôt équivalentes. Tandis qu'à grande dimension les méthodes en deux étapes sont jusqu'à 30 fois plus rapides. Le 11ème cas est assez particulier. On peut voir que les méthodes en deux étapes mettent énormément de temps pour ce cas. Ce cas correspond au cas radial avec 500 bus. Il est possible que le fait de n'avoir que très peu de ligne par bus, rendent l'exécution très peu efficace sur GPU.

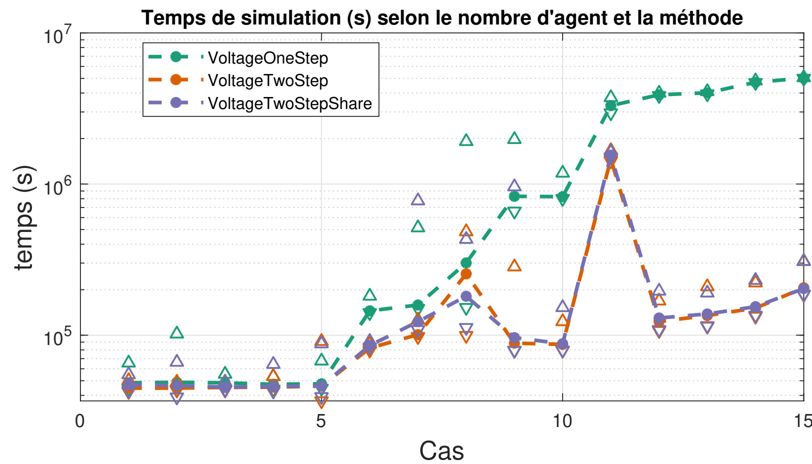


FIGURE 4.20 – Temps moyen minimal et maximal pour chaque méthode de calcul

5 Conclusion

Dans ce chapitre, la décomposition en blocs fonctionnels de l'ensemble des algorithmes a pu être réalisée. L'analyse des dépendances et l'identification des goulots d'étranglement ont pu être conduites. Le passage sur GPU a permis de diminuer la complexité des blocs les plus importants.

La diminution de la complexité a permis une amélioration des performances des algorithmes pour les algorithmes de Marché P2P, marché endogène et OPF. En effet, les parties dominantes sur CPU sont devenues bien plus rapides sur GPU.

Cependant, le passage sur GPU n'a pas permis de réduire les temps de calcul pour les problèmes de PF. En effet, le calcul est déjà rapide sur CPU et n'est pas parfaitement parallélisable. Ainsi, les dimensions des cas tests ne sont pas assez grandes pour permettre une accélération en passant sur GPU.

Les résultats du passage sur GPU du marché endogène sont très dépendants de la méthode. Quelle que soit la méthode considérée, le passage sur GPU réduit les temps de calcul en augmentant les temps de l'initialisation. Ainsi, il est fort probable que le fait de résoudre plusieurs pas de temps successifs permettra d'améliorer les temps de calcul, notamment celles sur GPU.

Chapitre 5

Performances atteintes et sensibilité

Sommaire

1 **Introduction** **221**
2 **Évaluation de la complexité** **222**
2.1 Marché Pair à Pair 223
2.2 Marché endogène DC 225
2.3 Marché endogène AC 227
2.4 Conclusion 230
3 **Performance sur cas tests** **231**
3.1 Marché Pair à Pair 231
3.2 Marché endogène DC 233
3.3 Marché endogène AC 235
3.4 Conclusion et état de l'art 237
4 **Limites et études paramétriques** **238**
4.1 Paramètres du cas d'études 239
4.2 Paramètres algorithmiques et architecturaux 242
5 **Conclusion** **249**

1 Introduction

Dans le chapitre précédent le partitionnement CPU-GPU a pu être présenté pour toutes les méthodes. Les algorithmes ont été séparés en différents blocs fonctionnels qui ont pu être analysés en terme de dépendance, de charge de travail et de complexité.

L'objectif de cette partie sera de montrer les performances réelles des implémentations sur GPU. En effet dans le chapitre précédent les méthodes ont été instrumentées afin de mesurer les temps pour chaque bloc fonctionnel. Ces mesures ont imposées du synchronisme entre le CPU et le GPU, ce qui a pu diminuer les performances de ce dernier.

Ainsi la première partie de ce chapitre évaluera les différentes méthodes sur plusieurs pas temporels des cas Européen ou du cas *TestFeeder*. La méthode **AC-EndoPF** ne convergeant pas sur ce dernier cas (voir section 3.5), elle ne sera pas évaluée sur ces cas dans ce chapitre. Comme les différents PF ne sont utilisés que dans ce type de résolution endogène, les résultats avec les implémentations sur GPU des PF ne seront pas présentés. L'évaluation de la complexité des méthodes sur GPU sera aussi réalisée avec des cas générés aléatoirement.

Ensuite une étude paramétrique sera réalisée afin d'étudier l'évolution des performances selon certains paramètres. On étudiera ainsi les performances selon la valeur des différents facteurs de pénalités (lorsqu'il y en a), selon la configuration du réseau (nombre de bus et profondeur/largeur du réseau radial) et selon les agents. On étudiera par exemple l'impact d'avoir une grande hétérogénéité entre les fonctions coûts ou les puissances des agents. Enfin on pourra regarder l'influence du rapport entre les dimensions du réseau (nombre de bus) et du marché (nombre d'agent).

Pour toute la suite les paramètres de simulations sont :

- le nombre d'itérations maximales du problème global k_{max} ;
- le nombre d'itérations maximales du problème local j_{max} ;
- la précision visée pour le problème global ϵ_g ;
- la précision visée pour les contraintes réseau ϵ_x ;
- la précision visée pour le problème local ϵ_l ;
- le nombre d'itération entre chaque calcul du résidu pour le problème global $step_g$;
- le nombre d'itération entre chaque calcul du résidu pour le problème local $step_l$;
- le facteur de pénalité du problème global ρ_g (décentralisation) ;
- le facteur de pénalité associé à la relaxation des contraintes réseaux ρ_x ;
- le facteur de pénalité du problème local ρ_l (ADMM de partage) ;

2 Évaluation de la complexité

La section 4.3 avait étudié la réduction de la complexité qu'il était possible d'obtenir grâce à une parallélisation sur GPU. Cette section évaluera l'évolution du temps de calcul avec les dimensions du cas simulé. Ces mesures de temps nous permettent d'évaluer la complexité concrète des algorithmes. Cette complexité mesurée sera comparée à celle théorique.

Pour toute la suite, on utilise respectivement 1 et 2 pour indice indiquant la plus petite et la plus grande taille simulées. On cherche à déterminer α tel que la complexité soit en $O(N^\alpha)$. Pour s'affranchir de possibles mauvais réglages de facteur de pénalité, on considèrera les temps par itération. L'approximation choisie

sera la suivante :

$$\alpha = \frac{\log(t_2/t_1)}{\log(N_2/N_1)} \tag{5.1}$$

2.1 Marché Pair à Pair

Dans cette partie les différentes implémentations seront évaluées dans un ordre aléatoire sur 50 cas générés aléatoirement pour chaque nombre d'agents testé. Comme l'objectif est d'étudier le passage à l'échelle et non pas la convergence sur des cas potentiellement absurdes, le nombre maximal d'itérations est limité à 50. Le générateur est identique à celui utilisé dans a section 3.3. Pour rappel, on considère un marché Pair à Pair avec deux types de puissances échangées. Ainsi un cas généré de 10 agents correspond (en dimension du problème) à un marché mono-énergie de 20 agents.

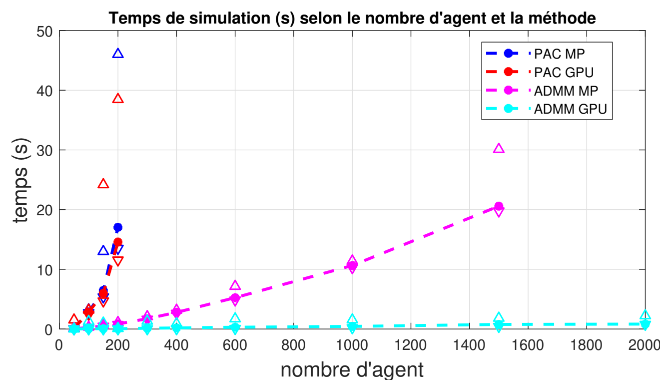


FIGURE 5.1 – Évolution des temps de calcul avec le nombre d'agent, marché P2P parallélisé

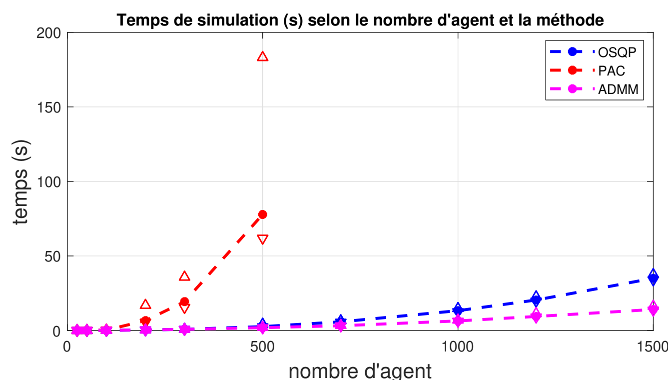


FIGURE 5.2 – Évolution des temps de calcul avec le nombre d'agent, marché P2P en série

La Fig. 5.1 montre les temps moyens, maximum et minimum pour chaque algorithme parallélisé selon le nombre d'agents. Pour rappel, les temps des méthodes en série sont sur la Fig. 5.2. On peut y voir que les algorithmes parallélisés sur CPU ont des temps de calcul augmentant rapidement avec les dimensions du problème. Ainsi une parallélisation sur CPU permet bien de réduire les temps de calcul mais ne permet pas de résoudre les problèmes de passage à l'échelle.

TABLE 5.1 – Étude du passage à l'échelle des différentes implémentations

Méthode		OSQP	ADMM			PAC		
Architecture		CPU	CPU	CPUMP	GPU	CPU	CPUMP	GPU
k_{max} 50	t_1	0.0208	0.028	0.058	0.0077	0.037	0.086	0.095
	t_2	35	14	21	0.83	78	17	15
	N_2	1500	1500	1500	2000	500	200	200
	$\frac{t_2}{t_1}$	1680	490	357	108	2090	199	154
	α	1.8	1.5	1.6	1.3	1.8	1.4	1.3

Le Tab. 5.1 regroupe l'évaluation de la complexité des différentes méthodes. Pour toutes les méthodes, le premier temps t_1 est pour $N_1 = 50$. Pour le calcul de la complexité α, les temps utilisés sont ceux par itération. On remarque bien que l'utilisation du GPU permet bien de réduire la complexité apparente de l'ensemble de l'algorithme.

L'accélération est bien plus faible que celle obtenue lors de l'étude par block en section 4.3. Cela est dû au très faible nombre d'itérations permises. De même le faible nombre d'itérations ne permet pas d'atteindre la réduction de complexité que l'on devrait théoriquement obtenir lorsque l'on ne considère pas les étapes de gestion des données pré et post calcul [114].

Les implémentations **ADMM** étant clairement meilleures, on étudiera plus précisément leur complexité. Ainsi la Fig. 5.3 représente l'évolution du temps de calcul sur des cas aléatoires. Contrairement à ce qui vient d'être fait, le nombre d'itérations maximal autorisé sera assez haut pour permettre la convergence. Pour éviter d'avoir une dépendance au nombre d'itérations pour converger, on s'intéressera uniquement aux temps par itération.

On peut voir que cette fois les méthodes parallélisées sont bien plus rapides. De plus, on peut remarquer que les méthodes parallélisées passent mieux à l'échelle avec une augmentation du temps de calcul avec la taille du problème plus faible. La complexité mesurée dans ce cas est représentée dans le Tab. 5.2. Dans le cas

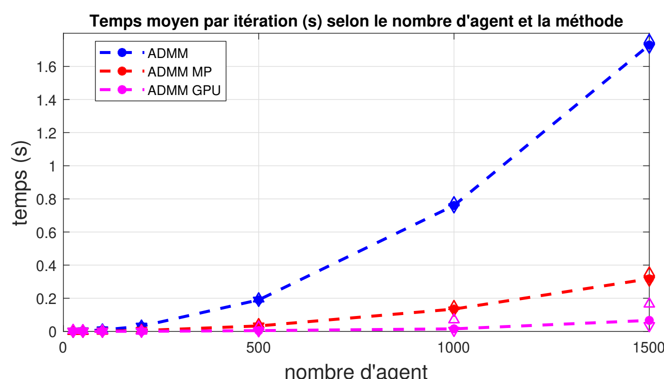


FIGURE 5.3 – Évolution des temps de calcul avec le nombre d'agent, marché P2P résolu par ADMM

où l'on considère un marché mono-énergie (marché DC), l'analyse de complexité a été faite dans [110] et est rappelé dans ce même tableau.

TABLE 5.2 – Augmentation du temps et complexité théorique dans le cas d'un marché selon s'il est mono-énergie ou poly-énergies

Type de marché	Methode	α_{poly}	C_{the}
DC	OSQP	2.1	$O(N^3)$
	ADMM	1.9	$O(N^2)$
	ADMM - GPU	1.1	$O(\frac{N}{K} + log(N))$
AC	ADMM	1.86	$O(N^2)$
	ADMM - MP	1.9	$O(N^2)$
	ADMM - GPU	1.4	$O(\frac{N}{K} + log(N))$

Il peut être remarqué que pour les deux types de marché, la complexité de la résolution sur CPU est presque quadratique. Cependant la complexité mesurée pour la version parallélisée sur GPU est plus élevée lors d'un marché multi-énergie.

2.2 Marché endogène DC

Pour évaluer la complexité de cette méthode, il faut faire varier le nombre d'agents et le nombre de lignes contraintes. On pourra aussi étudier l'effet des dimensions du problème sur les temps de calcul du marché P2P et du SO. On cherchera à simuler sur un réseau de transport où l'approximation DC a du sens. Pour cela, le réseau a été fixé à partir de celui défini dans les données open source européen [105]. Une loi aléatoire homogène a été utilisée pour sélectionner les lignes

contraintes et leurs limites, les puissances, préférences et fonctions coûts des agents et leur positionnement sur le bus. Les paramètres de l'aléatoire et de simulation sont respectivement définis dans les tableaux, Tab. 5.3 et Tab. 5.4.

TABLE 5.3 – Caractéristique de la génération aléatoire pour le cas d'étude

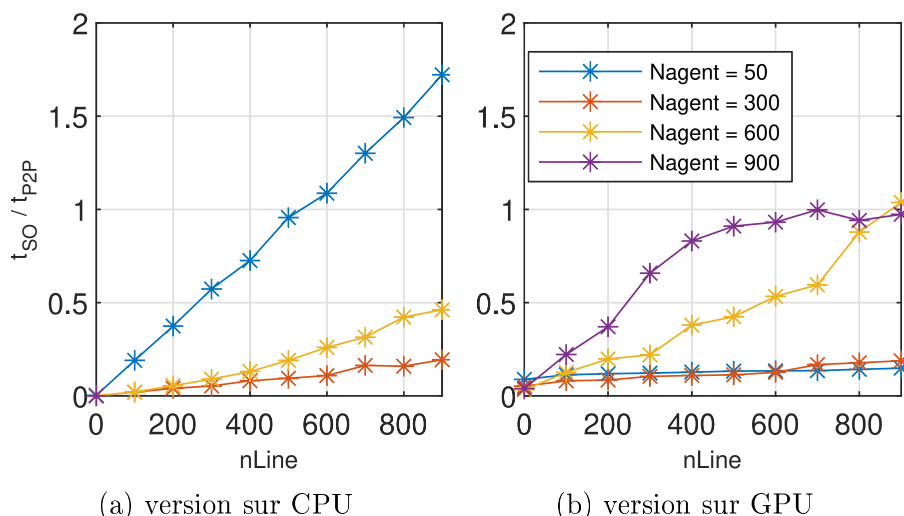
	Consommateur		Générateur	
Caractéristique	moyenne	variation	moyenne	variation
Proportion	0.6	0	0.4	0
P0 (MW)	60	50	300	250
$\overline{p_n}$ (MW)	-0.9 P0	0	P0	0
$\underline{p_n}$ (MW)	-1.1 P0	0	0	0
a_n (MW^{-2})	1	0	0.01	0
b_n (MW^{-1})	P0	0	20	18
$\|\beta\|$ (MW^{-1})	4	2	4	2
$\bar{l}$ (MW)	1000	300	1000	300

TABLE 5.4 – Paramètres de simulation

caractéristiques	valeur	caractéristique	valeur
k_{max}	50000	j_{max}	5000
$step_g$	10	$step_L$	1
ϵ_g	0.01	ϵ_l	0.001
ρ_g	10	ρ_l	ρ_g
ρ_1	0.004	ϵ_x	1
nSimu	20	$offset$	2

Les facteurs de pénalités étant fixés à une valeur arbitraire, on peut s'attendre à ce qu'ils soient très mal choisis selon les cas, de plus certains cas pourraient ne pas être faisables à cause des contraintes de lignes. Ainsi les résultats ne seront présentés que pour les cas ayant convergé. La résolution a été faite sur CPU et sur GPU pour permettre de comparer les temps de résolution.

Sur la Fig. 5.4 on peut voir la proportion du temps passé pour le calcul du SO par rapport à celui du marché P2P pour plusieurs dimensions du problème. Dans le cas du CPU, Fig. 5.4a, on peut voir que plus le nombre d'agents est grand, plus le marché P2P prend du temps à être résolu par rapport au SO. Alors que pour le GPU c'est l'inverse. Ceci est bien cohérent avec ce qui est montré dans le chapitre



(a) version sur CPU (b) version sur GPU

FIGURE 5.4 – Ratio entre les temps de calcul du SO et du marché P2P

précédent, Tab. 4.3. En effet sur CPU, les complexités de respectivement le marché et le SO sont $O(M) \approx O(N^2)$ et $O(N \cdot L)$. Ainsi le nombre de calculs augmente plus vite pour le marché que pour le SO. Alors que sur GPU, les complexités sont respectivement de $O(ln(N))$ et $O(N)$, la variation est donc inversée.

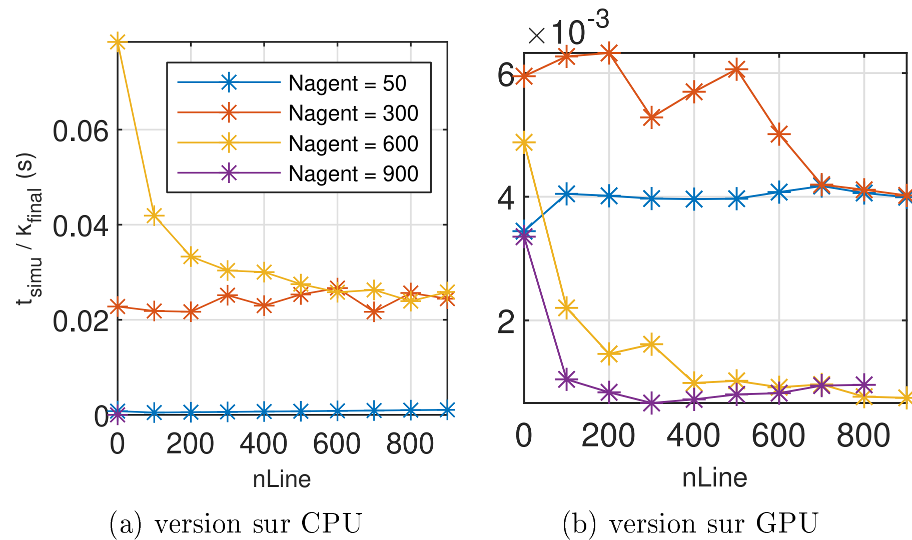
Sur la Fig. 5.5, on peut voir le temps par itération globale en fonction de la taille du problème. On voit que pour le CPU, Fig. 5.5a, le temps par itération augmente grandement avec le nombre d'agents alors que ce n'est pas le cas pour le GPU, Fig. 5.5b grâce à la parallélisation.

On peut remarquer que le temps diminue avec l'augmentation du nombre de lignes pour de grands nombres d'agents. Cela provient du fait que ces problèmes sont plus contraints à grande dimension. Ainsi, il faut plus d'itérations globales pour converger et celles-ci sont de plus en plus rapides comme le problème local ne cherche à changer que les agents actionnant les contraintes de ligne.

Ce graphe permet aussi d'évaluer le speed up entre la version sur CPU et celle sur GPU comme les deux implémentations ont besoin d'autant d'itération pour converger. Ainsi même si cela dépend des dimensions, on peut observer un facteur **5** entre les temps CPU et les temps GPU pour 300 agents et **25** pour 600 agents, Fig. 5.6.

2.3 Marché endogène AC

Les versions implémentées fonctionnelles de marché endogène AC sur les générateurs de cas aléatoires sont **AC-EndoPF**, **EndoDirect** et **EndoConsensus**. Les deux dernières ne peuvent actuellement qu'être utilisées sur des réseaux ra-



(a) version sur CPU (b) version sur GPU

FIGURE 5.5 – Temps de calcul (s) divisé par le nombre d'itérations

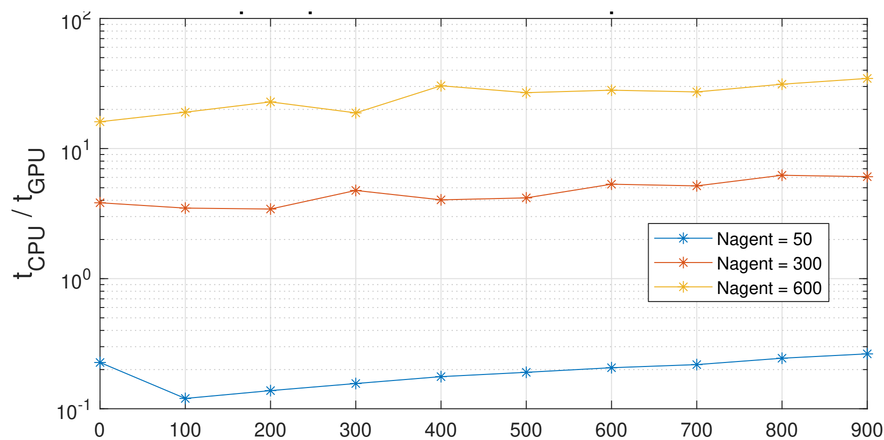


FIGURE 5.6 – Ratio entre les temps de calcul sur CPU et sur GPU

diaux. Ainsi pour évaluer la complexité des méthodes il faut faire varier la taille du réseau $B = L + 1$ et le nombre d'agents N. Pour ce faire, on utilisera le même générateur que pour la validation fonctionnelle (section 3.5). On fera aussi varier pour le nombre d'agent pour différents nombres de bus et lignes.

On cherchera à se concentrer sur les temps de calcul en s'affranchissant de la convergence des différentes méthodes sur des cas potentiellement impossibles. Ainsi la précision demandée sera mauvaise et l'on affichera le temps par itération des cas ayant convergé. Les paramètres sont dans le Tab. 5.5. Les résultats lorsque les nombres de bus et d'agents sont identiques sont dans la Fig. 5.7. On peut

TABLE 5.5 – Paramètres de simulation

caractéristiques	valeur	caractéristique	valeur
k_{max}	5000	j_{max}	5000
$step_g$	1	$step_L$	1
ϵ_g	0.1	ϵ_l	0.001
ρ_g	10	ρ_l	ρ_g
ρ_x	5	ϵ_x	0.05

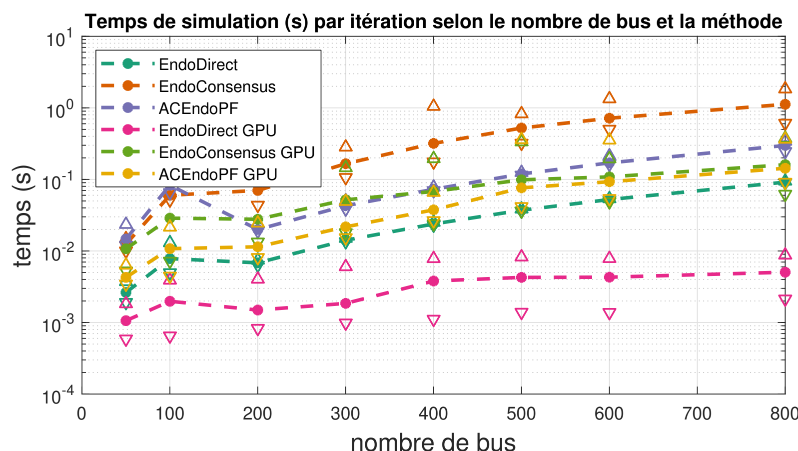


FIGURE 5.7 – Évolution des temps de calcul avec le nombre de bus (ou agents), marché AC-Endogène

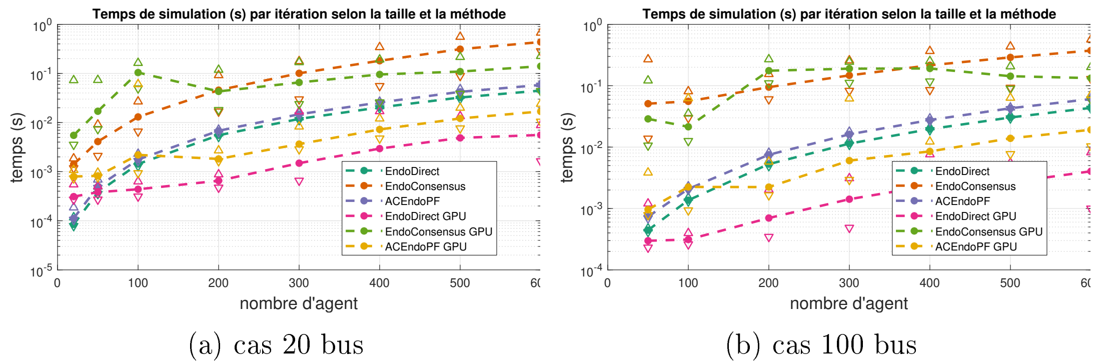
remarquer qu'à partir de 100 bus (ou agents) les méthodes GPU sont toutes plus rapides que leurs versions sur CPU.

L'évaluation de la complexité est dans le Tab. 5.6. On peut y voir que la parallélisation sur GPU a bien permis de réduire la complexité sauf pour **AC-EndoPF**. La complexité des méthodes sur CPU sont plus faible que celle théorique. Tandis que les méthodes sur GPU ont des complexités plus élevée que celles à lesquelles on pourrait s'attendre.

On peut maintenant étudier dans le cas où le nombre de bus est fixé et que l'on rajoute de plus en plus d'agents. On étudiera le cas où $B = 20$ et $B = 100$. En faisant varier le nombre d'agents, on risque d'aussi faire varier la charge sur le réseau. Ainsi pour se concentrer sur le passage à l'échelle de l'algorithme, on fixera la puissance des agents à $P = P_0 * B/N$ avec P_0 la puissance des agents lorsque l'on avait $B = N$ dans les simulations précédentes. Ainsi on devrait avoir le même ordre de grandeur de puissance injectée sur le réseau à chaque bus avec un nombre d'agents par bus qui varie.

TABLE 5.6 – Augmentation du temps et complexité théorique (N=B)

Method	η	α_{poly}	C_{the}
EndoDirect	35	1.3	$O(N^2)$
EndoConsensus	90	1.6	$O(N^2)$
AC EndoPF	20	1.1	$O(N^2)$
EndoDirect GPU	4.7	0.6	$O(log(N))$
EndoConsensus GPU	15	1	$O(log(N))$
AC EndoPF	33	1.3	$O(log(N) + B)$



(a) cas 20 bus

(b) cas 100 bus

FIGURE 5.8 – Temps de calcul (s) divisé par le nombre d'itérations, marché AC-Endogène

On peut remarquer que le comportement pour le passage à l'échelle en fonction du nombre d'agents est similaire pour les deux tailles de réseau. Malgré une multiplication par 5 du nombre de bus, les temps par itération sont très proches. Des différences sont visibles lorsque le nombre d'agent est faible. En effet s'il y a peu d'agent, le nombre de bus a bien plus d'importance.

On peut voir l'approximation de la complexité mesurée dans le Tab. 5.7. On remarque que les méthodes sur CPU ont une plus grande complexité lorsque le nombre d'agents augmente indépendamment du nombre de bus. Alors que pour le GPU, la complexité reste proche du linéaire. On remarque que le fait d'avoir plus de bus fait chuter la complexité des méthodes **EndoConsensus** (sur CPU ou sur GPU).

2.4 Conclusion

On a ainsi pu montrer dans cette partie que, quel que soit le problème considéré l'utilisation du GPU a permis d'effectivement réduire la complexité de l'algorithme.

TABLE 5.7 – Augmentation du temps et complexité théorique

Nombre de bus		20		100	
Méthode	C_{the}	η	α_{poly}	η	α_{poly}
EndoDirect	$O(N^2)$	530	1.8	100	1.8
EndoConsensus	$O(N^2)$	307	1.7	7	0.8
AC EndoPF	$O(N^2)$	520	1.8	82	1.8
EndoDirect GPU	$O(log(N))$	18	0.85	14	1
EndoConsensus GPU	$O(log(N))$	25	0.95	5	0.6
AC EndoPF	$O(log(N) + B)$	21	0.9	20	1.2

Si, dans tous les cas le CPU est plus rapide à petite dimension, la réduction de complexité permet au GPU d'être plus rapide à partir d'une certaine taille. Cette taille peut grandement dépendre du nombre d'itérations (donc du cas d'étude et de la précision demandée). Pour les cas qui ont été testés et générés, l'ordre de grandeur se situe autour de la centaine d'agents.

3 Performance sur cas tests

L'objectif de cette section est de déterminer les performances des différentes méthodes sur des cas à grande dimension. Les simulations seront multi-pas afin de pouvoir profiter du fait de pouvoir garder la majorité des données en place. Ainsi la quantité de transfert mémoire entre CPU et GPU peuvent rester limitée. Le fait de simuler l'ensemble des pas permet de réaliser des warm-start entre chaque pas de temps et conserver une cohérence temporelle des données.

L'apport de cette thèse étant centré sur la résolution de marché endogène, cette section se concentrera sur les performances sur des cas tests de marché pair à pair avec ou sans contraintes de réseau.

3.1 Marché Pair à Pair

La résolution d'un marché pair à pair ne dépend que du nombre d'agents. Ainsi on utilisera le cas Européen pour simuler un nombre d'agents sur une grande plage de temps. Les méthodes **PAC** nécessitent trop de mémoire pour réaliser la résolution, ces méthodes ne seront donc pas testées dans ce cas. Les paramètres de simulations sont dans le Tab. 5.8. Le facteur de pénalité a été choisi parmi les valeurs ρ_g = [100 150 200 250 300 400] pour qu'il y ait le moins d'itérations possible.

TABLE 5.8 – Paramètres de simulation pour le marché pair à pair

paramètre	valeur	paramètre	valeur
k_{max}	10000	j_{max}	1000
$step_g$	2	$step_L$	2
ϵ_g	0.01	ϵ_l	0.005
ρ_g	250	ρ_l	ρ_g

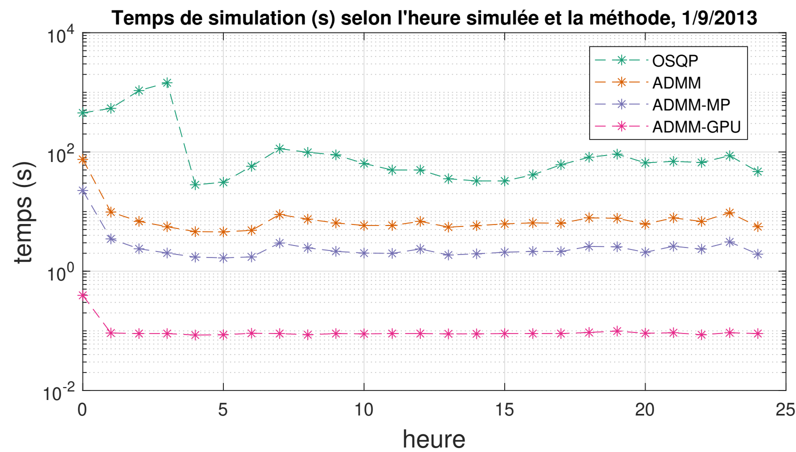


FIGURE 5.9 – Temps de calcul selon la méthode et l'heure simulée, cas Européen (2463 agents)

Pour évaluer, comparer et présenter les performances des différentes méthodes, la date du 1 septembre 2013 a été choisie arbitrairement. Il faut simuler des pas consécutifs pour pouvoir utiliser le warm-start. En effet sans celui-ci le temps de calcul est bien plus long.

La Fig. 5.9 représente les temps de calcul pour les différents pas de temps de la date considérée. Le temps varie très légèrement selon le pas de temps considéré. Cela peut provenir d'un équilibre plus dur à atteindre (avec un prix plus élevé) ou d'une plus grande variation des puissances optimales avec le pas précédent. Pour plus de clarté, les temps des différentes méthodes sont reportés dans le Tab. 5.9.

Ainsi la méthode **ADMMGPU** résout une simulation de 2463 agents en moins de $0.2s$ par pas. Dans le Tab. 5.10 peut être lu le *Speedup* atteint grâce à la parallélisation sur CPU et sur GPU. L'accélération est calculée pour chaque pas de temps pour ensuite pouvoir prendre la moyenne, le minimum et le maximum.

La version sur GPU est clairement plus rapide que toutes les autres. L'accélération apportée par la parallélisation sur GPU permet de réaliser des simulations sur des cas de grandes dimensions. Ainsi la Fig. 5.10 représente les temps de calcul

TABLE 5.9 – Moyenne, médiane, minimum, et maximum pour le temps de calcul (s) selon les méthodes

Methode	Moyenne	Médiane	Min	Max	Std
OSQP	192	66	28	1450	348
ADMM	9.3	6.5	4.6	75	14
ADMM-MP	3.1	2.2	1.7	22.6	4.1
ADMM-GPU	0.1	0.09	0.085	0.39	0.06

TABLE 5.10 – Speedup de la parallélisation par rapport à la version en série

Méthode	Moyenne	Médiane	Min	Max	Std
ADMMMP	65.9 %	66.2 %	62.6 %	69.9 %	0.015
ADMMGPU	98.6 %	98.6 %	98.12 %	99.48 %	0.0031

sur un mois de simulation complet.

On peut y voir que le temps de calcul reste compris entre 0.1 et 0.3s lorsque l'on conserve les mêmes paramètres que pour la simulation précédente.

3.2 Marché endogène DC

L'approximation DC n'ayant réellement de sens que sur un réseau de transport, cette méthode sera testée sur un cas différent des autres méthodes de marché endogène. Ainsi la méthode **DC-EndoPF** sera évaluée sur le cas Européen.

Pour donner un ordre de grandeur de l'accélération sur ce cas, une simulation avec $k_{max} = j_{max} = 100$ a été réalisée sur les 3 premières heures du mois de juin. Sur le premier pas, les temps du CPU et du GPU sont respectivement de $265s$ et de $1.7s$. Sur les 2 autres pas, les temps sont d'environ $95s$ et $0.18s$. Ainsi pour le pas *coldstart* l'accélération est de **150** et pour les pas suivants utilisant un *warmstart* l'accélération est de **515**. Le temps de calcul étant beaucoup trop lent sur CPU, seule la méthode sur GPU sera vraiment simulée jusqu'à la convergence.

Afin de s'affranchir des problèmes de précision sur le respect des contraintes dans les lignes, la précision demandée pour cette contrainte sera mauvaise. Une marge sur les limites des lignes sera donc ajoutée pour être sûr de respecter les contraintes réelles. Les paramètres sont dans le Tab. 5.11. Pour plus de détails sur l'influence des paramètres ϵ_{gc} et $offset$ voir la Section 5.4.2.2.

Les temps pour chaque pas sont dans la Fig. 5.11. Le temps est clairement plus lent que dans le cas d'un marché pair à pair sans contraintes réseau. Cependant

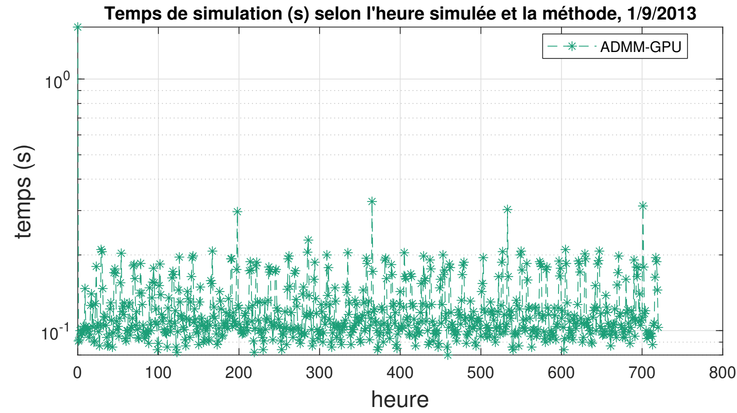


FIGURE 5.10 – Temps de calcul d'un marché P2P pour chaque heure simulée sur GPU, début du jeu de test le 01/09/2013

TABLE 5.11 – Paramètres de simulation pour le marché endogène DC

paramètre	valeur	paramètre	valeur
k_{max}	30000	j_{max}	2000
$step_g$	5	$step_L$	1
ϵ_g	0.01	ϵ_l	0.00001
ρ_g	125	ρ_l	ρ_g
ϵ_{gc}	1	ρ_1	0.001
$offset$	1		

cela provient principalement d'un nombre bien plus important d'itérations nécessaires pour converger. En effet le marché Pair à Pair avec ADMM ne nécessite que quelques dizaines d'itérations. Le nombre d'itérations nécessaire pour converger de **DC-EndoPF** est visible sur la Fig. 5.12.

Ainsi le temps par itération est compris entre 1.2 et 1.7 ms ce qui est tout aussi rapide que pour la méthode précédente [1]. Un temps moyen de 9s permet de réaliser ce genre de simulation qui n'aurait pas été possible sur CPU. En effet, en considérant un facteur d'accélération de 515, il faudrait plus d'une heure par pas simulé par le CPU.

1. Le lecteur très attentif pourra remarquer que le temps des derniers pas augmente pour un nombre d'itérations constant. Ce phénomène sera explicité dans la section 5.4.2.4

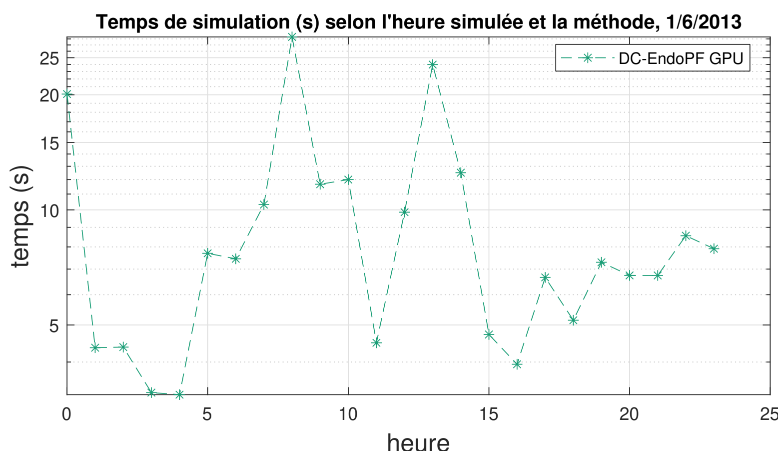


FIGURE 5.11 – Temps de calcul selon l'heure simulée pour le **DC-EndoPF** sur GPU

3.3 Marché endogène AC

Les deux dernières méthodes, **EndoDirect** et **EndoConsensus** ont pu être testées sur le cas *TestFeeder*. Si ce cas a peu d'agents, il est en grande dimension pour le nombre de bus et permet de simuler plusieurs pas de temps. Les paramètres pour la simulation sont les suivants, Tab. 5.12.

TABLE 5.12 – Paramètres de simulation pour le marché endogène AC

paramètre	valeur	paramètre	valeur
k_{max}	20 000	j_{max}	5000
$step_g$	1	$step_L$	1
ϵ_g	0.05	ϵ_l	0.0005
ρ_g	10	ρ_l	ρ_g
ϵ_x	0.005	ρ_x	1.5

On peut donc regarder le temps de simulation sur les 30 premières minutes du cas *TestFeeder* sur la Fig. 5.13. On y observe une grande variabilité des temps de simulation. Cependant les méthodes par consensus sont clairement globalement plus lentes. On peut remarquer que la minute 9 est le cas nécessitant le plus de temps pour converger.

Les statistiques pour les temps et nombres d'itérations des différentes méthodes sont dans le Tab. 5.13. On peut remarquer que les temps et le nombre d'itération sont très variables avec une médiane très loin de la moyenne pour toutes les

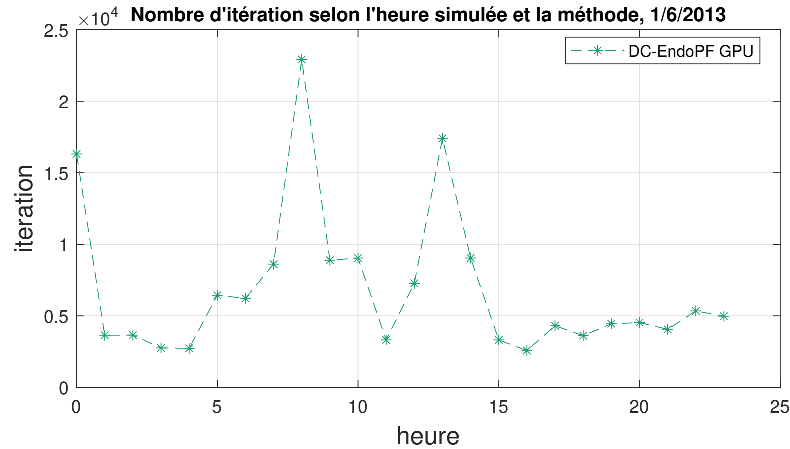


FIGURE 5.12 – Nombre d'itération selon l'heure simulée pour le **DC-EndoPF** sur GPU

méthodes. On peut remarquer que les méthodes utilisant un consensus nécessite beaucoup moins d'itérations pour converger.

> Comme indiqué dans la partie sur les métriques (Section 3), le nombre d'itérations lors d'une simulation en place comme il est fait dans cette thèse n'est pas important. Cependant, un trop grand nombre d'itérations peut rendre prohibitive une implémentation réellement distribuée. En effet à chaque itération des messages doivent être échangés. L'envoi et la réception des messages entre les agents [19] ou entre processeur d'un cluster [20] nécessite un temps non négligeable.

Le temps moyen par itération est représenté sur la Fig. 5.14. On peut remarquer que pour la méthode directe le temps est plutôt constant, quelle que soit la minute simulée. Une exception existe pour la version sur GPU sur le premier pas, démontrant l'impact des transferts mémoires sur les temps de calcul. Pour les méthodes avec consensus, le temps par itération est extrêmement variable avec environ un facteur 100 entre le plus rapide et le plus lent.

L'implémentation des méthodes **EndoDirect** et **EndoConsensus** a permis de simuler ce cas de grande dimension avec de faibles temps de calcul. Le passage sur GPU a permis de diviser par 10 les temps de calcul. Ceci permet d'envisager de faire des simulations à long terme même sur de grand réseau. L'accélération induite par la parallélisation sur GPU devrait être d'autant plus importante si le nombre d'agents participant au marché augmente.

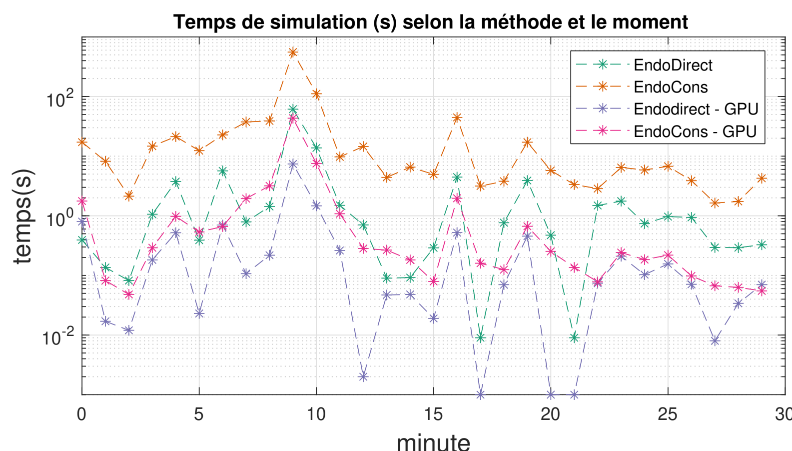


FIGURE 5.13 – Temps de calcul (s) selon la méthode et la minute simulée

TABLE 5.13 – Moyenne, médiane, minimum, et maximum pour le temps de calcul (s) (nombre d'itération) selon les méthodes

Méthode **Endo**	Moyenne	Médiane	Min	Max	Std
Direct	3.6 (818)	0.75 (168.5)	0.009 (2)	61 (14263)	11 (2603)
Consensus	33 (10)	6.7 (8.5)	1.6 (4)	557 (41)	101 (7.7)
Direct-GPU	0.45 (774)	0.07(139.5)	0.001 (2)	7.4 (13431)	1.3 (2445)
Consensus-GPU	2.2 (10)	0.25 (8)	0.048 (3)	43 (44)	7.9 (8.7)

3.4 Conclusion et état de l'art

Dans cette partie, les différentes résolutions du marché pair à pair avec et sans contraintes de réseau ont pu être appliquées sur des cas de grandes dimensions. La parallélisation sur GPU réduit les temps de calcul et rend possibles des simulations sur de grandes plages temporelles.

La comparaison avec l'état de l'art est complexe puisque ce genre de résolution n'a pas été réalisé à grande échelle. Cependant, on peut se référer au chapitre 1 Section 3.1 pour effectuer une comparaison.

Pour le marché pair à pair, dans [25] un cas de 39 agents est résolu en $9.5s$. Ce même cas est résolu en respectivement $0.1s$ et 1.3s sur CPU et sur GPU avec la méthode **ADMM**. Même si pour cette taille-ci l'implémentation sur CPU est plus rapide que pour GPU, les deux implémentations sont plus rapides que celle de [25]. Dans l'article [20], un cas de 300 agents est résolu avec un temps de $0.014s$ par itération. Dans notre cas il faut respectivement $0.03s$ et $0.0027s$ par itération pour la version parallélisée avec openMP et celle parallélisée sur GPU.

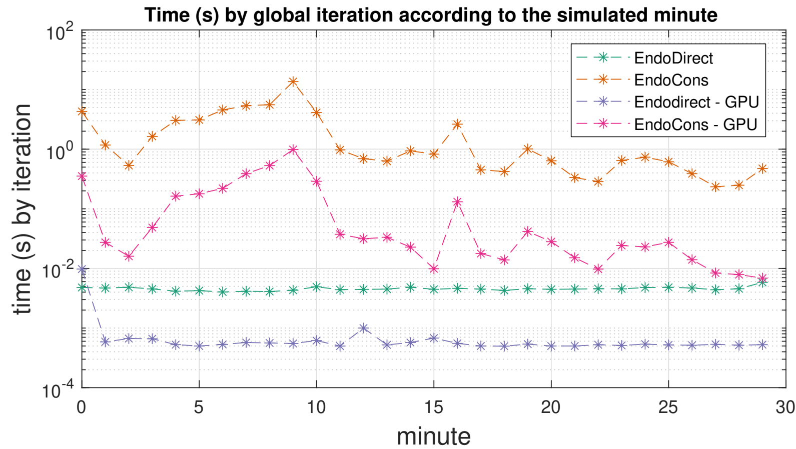


FIGURE 5.14 – Temps par itération (s) selon la minute simulée et la méthode

La comparaison avec [20] est difficile puisque leurs cas nécessitent beaucoup d'itérations (plusieurs milliers) pour converger. C'est pourquoi on compare le temps par itération.

Il est aussi possible de comparer le marché endogène DC avec [26]. Dans cet article le cas IEEE 39 bus est résolu en 1min sur Matlab. Il est résolu en 2 et 6 secondes respectivement sur CPU et sur GPU, le CPU étant plus rapide à faible dimension.

4 Limites et études paramétriques

L'objectif de cette partie est de montrer la sensibilité des résultats précédents aux cas d'études, paramètres de simulation et au matériel utilisé. Le but de cette partie n'est pas d'être exhaustif mais plutôt d'identifier les paramètres les plus impactant ou ceux qui sont négligeables.

De plus on pourra présenter dans cette partie les limites des implémentations réalisées. Ces limites peuvent être algorithmiques et ainsi se retrouver à la fois sur la version CPU et la version GPU. Le problème peut aussi venir de l'interaction avec le matériel, avec ainsi une différence de comportement entre l'implémentation sur CPU et sur GPU.

Comme il serait illusoire de tester tous les paramètres de toutes les méthodes sur tous les problèmes, seuls les triptyques paramètre-algorithmes-architectures les plus pertinents ne seront présentés. Ainsi on se concentrera comme sur les parties précédentes sur les marchés P2P et marché endogène. De plus on ne testera pas toutes les méthodes, ainsi on n'utilisera que les méthodes **ADMM** pour le marché

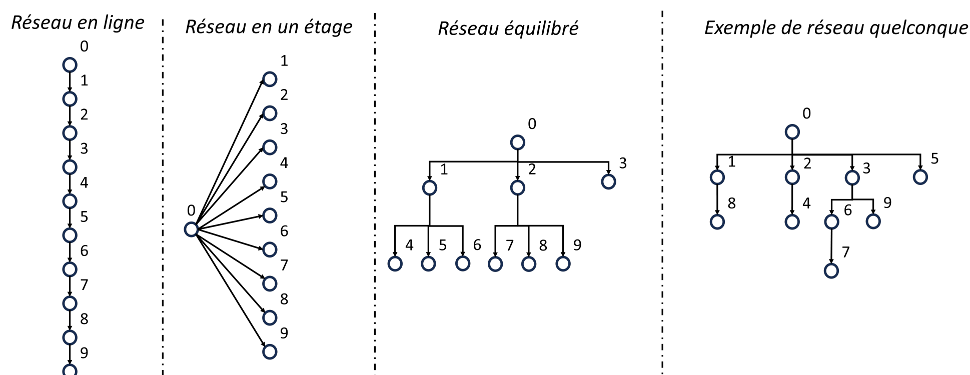


FIGURE 5.15 – Type de réseau généré

Pair à pair, cet algorithme étant clairement le meilleur. Pour le marché endogène on testera **DC-EndoPF**, **EndoConsensus** et **EndoDirect** qui sont les méthodes étudiées précédemment.

4.1 Paramètres du cas d'études

4.1.1 Configuration du réseau

Afin de tester l'influence de la configuration du réseau sur les temps de résolution, on proposera quatre types de réseaux radiaux :

— le réseau **en ligne**, est le réseau où chaque bus a un unique enfant ;
— le réseau **à un étage**, est le réseau où tous les bus sont les enfants du bus 0 ;
— le réseau **équilibré** est un réseau où l'on essaie d'avoir autant d'enfants par bus que d'étage du réseau
— le réseau **quelconque** ou **Normal** sera généré comme les chapitres précédents (un bus a autant de chance de se positionner en bout d'une branche que d'en créer une nouvelle).

Les réseaux spéciaux dans le cas 10 bus sont représentés sur la Fig. 5.15. Il est important de noter que pour le réseau **quelconque**, on n'a représenté qu'un exemple de ce qui peut être généré. Rien n'empêche le générateur quelconque de générer un réseau spécial (en une ligne ou un étage par exemple) , c'est juste extrêmement peu probable.

On fera aussi varier le nombre d'agents par bus à $n_b = [0.5\ 1\ 2\ 5]$ tel que $N = n_b * B$. Pour chaque combinaison réseau-agents, on réalisera 50 générations.

Le taux de convergence selon le cas d'étude est représenté sur la Fig. 5.16. On peut remarquer que les méthodes sur CPU convergent dans tous les cas. D'un autre côté, les méthodes sur GPU ne convergent pas toujours (se référer à l'annexe pour

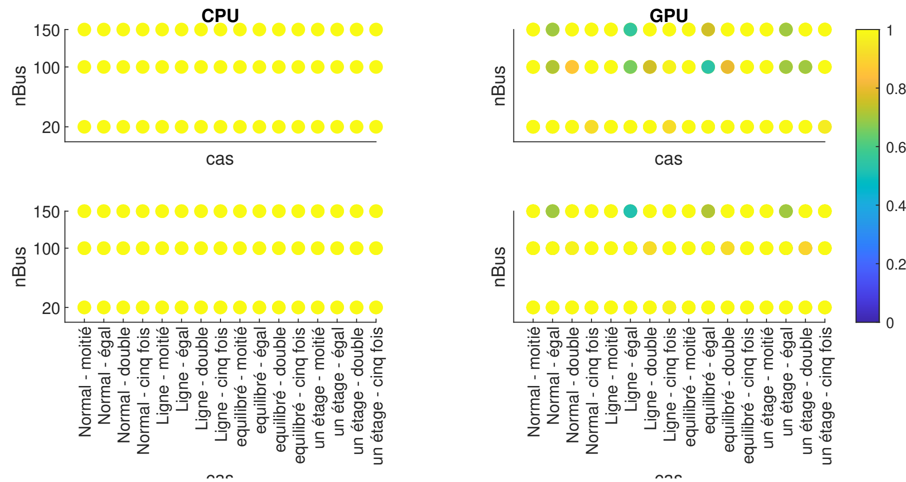


FIGURE 5.16 – Taux de convergence (**EndoDirect** en haut puis **EndoConsensus** en bas), cas de la forme "type de réseau - ration N/B"

plus de détails). On peut remarquer que plus il y a de bus, moins les méthodes sur GPU convergent. Cependant, la configuration du réseau (ou le nombre d'agents par bus) n'a pas l'air d'avoir d'impact clair.

Pour les cas qui ont convergé, le temps moyen est représenté sur la Fig. 5.17. On peut remarquer que pour les méthodes sur CPU les temps augmentent avec l'augmentation du nombre de bus et du nombre d'agents par bus. Cependant, la configuration du réseau n'a aucun impact visible. Pour les méthodes sur GPU, ces mêmes paramètres ont toujours un effet sur le temps de calcul, mais moindre. En effet, seul le cas avec moins d'agents que de bus est clairement plus rapide que les autres. Cependant, la configuration du réseau a l'air d'avoir un effet notable sur la méthode **EndoConsensusGPU**.

4.1.2 Paramètres du marché

Pour rappel les paramètres définissant les agents du marché sont :

— la fonction coût $g(p_n) = 0.5a_n p_n^2 + b_n p_n$ avec son minimum en $P_0 = -\frac{b_n}{a_n}$,

— le type de l'agent (consommateur, producteur...) et ses limites sur les puissances ($\overline{p}_n$, $\underline{p}_n$) ou sur ses échanges lb et ub,

— le nombre d'énergie échangé (réseau multi énergie ou non) et les liens entre les agents.

Dans cette partie on considèrera un marché mono énergie de 300 agents (comme dans [20]). On générera 1000 cas en tirant aléatoirement pour chaque cas la proportion de consommateur. On aura au minimum 10% de consommateur et au maximum 90%. Tous les agents qui ne sont pas des consommateurs sont des

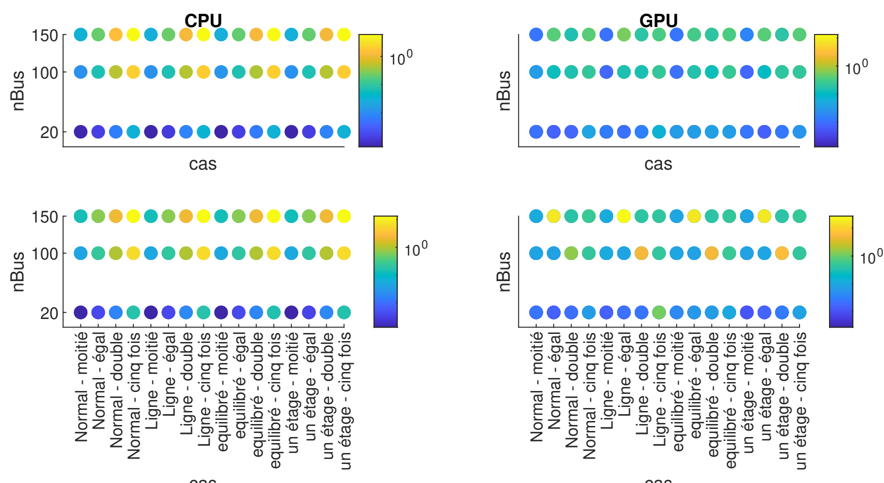
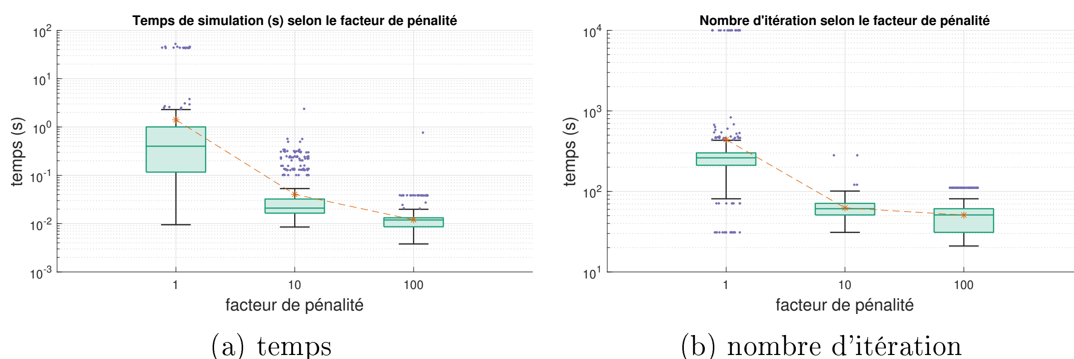


FIGURE 5.17 – Temps de simulation selon le cas et la taille (**EndoDirect** en haut puis **EndoConsensus** en bas)



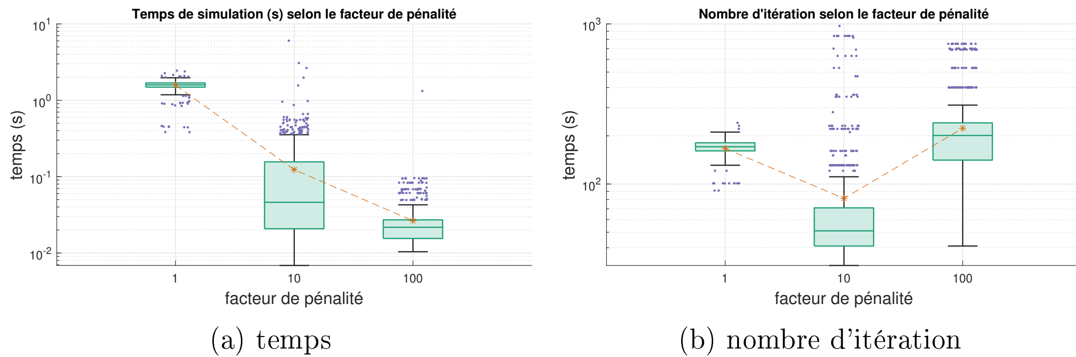
(a) temps (b) nombre d'itération

FIGURE 5.18 – Répartition et moyenne des temps de calculs et nombre d'itération selon le facteur de pénalité, grande plage de variation de a

producteurs. Ensuite on choisira aléatoirement la fonction coût de chaque agent $g_n(p_n) = 0.5a_n(p_n - P0)^2$ avec a_n entre 0.001 et 100 et $|P0| \leq 100$. On générera aussi des préférences hétérogènes $|\beta| \leq 100$. La précision demandée sera de $\epsilon_g = 0.0001$. Les bornes des agents seront fixées entre 0 et $10P0$ permettant d'assurer la faisabilité des cas (car 0 est une solution possible).

On peut remarquer sur la Fig. 5.18 que les répartitions des temps et du nombre d'itérations est très dépendante du facteur de pénalité. La variabilité du temps provient principalement du nombre d'itération. Les points les plus extrêmes lorsque $\rho = 1$ sont des cas qui n'ont pas convergé en 10 000 itérations. Pour les autres facteurs de pénalité, tous les cas ont convergé.

Tous les paramètres énoncés pourrait être modifiés pour étudier leur influence



(a) temps (b) nombre d'itération

FIGURE 5.19 – Répartition et moyenne des temps de calculs et nombre d'itération selon le facteur de pénalité, petite plage de variation de a

sur les résultats. On pourrait par exemple changer les bornes ou changer la distribution. Cela ne sera pas fait ici, mais il est important de remarquer que grâce à la vitesse de calcul du GPU en parallèle ce genre d'étude de Monté Carlo est possible même avec beaucoup d'agents. Par exemple ici la simulation avec $\rho = 100$ ne prend que quelques secondes.

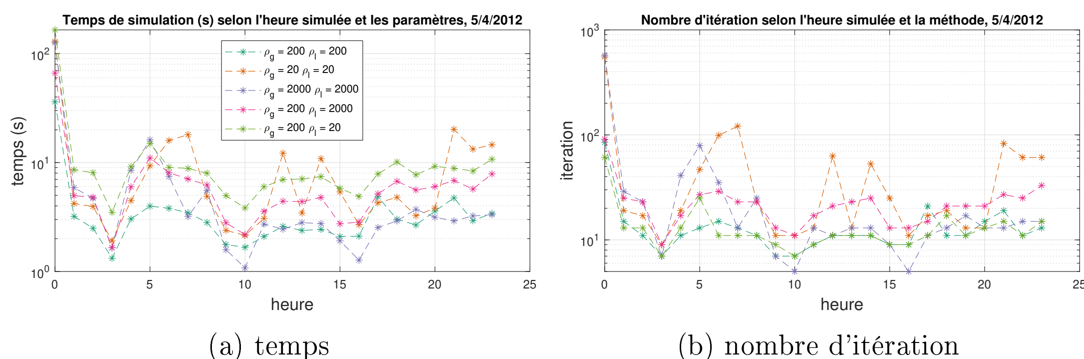
On pourra tout de même remarquer que dans le cas où l'on ne fait varier a qu'entre 0.1 et 10, les répartitions sont bien différentes, Fig. 5.19. On peut par exemple remarquer sur la Fig. 5.19a que pour $\rho = 1$ tous les temps de résolution sont assez proches. Avec cette plage de variation de a et de ρ tous les cas converges. Cependant, le temps de chaque résolution est plus long que précédemment (la moyenne est équivalente, mais la médiane est plus haute).

4.2 Paramètres algorithmiques et architecturaux

4.2.1 Facteur de pénalité

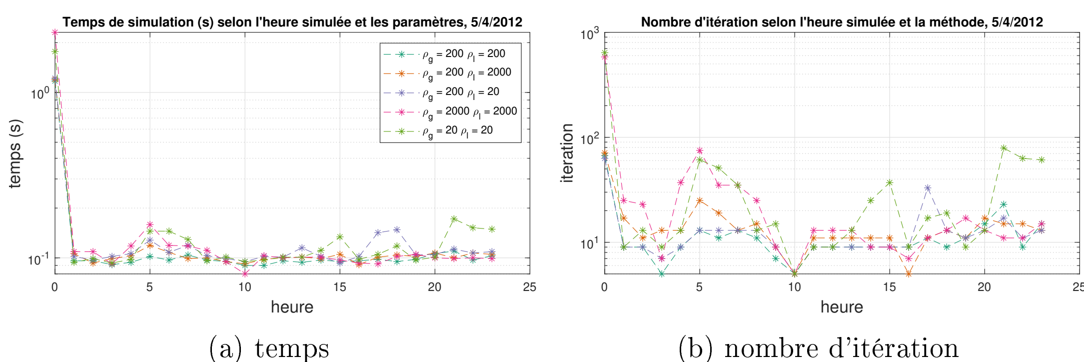
Les facteurs de pénalités sont extrêmement importants quelle que soit la méthode. En effet, comme il a pu être observé dans la section précédente ; ces paramètres conditionnent la convergence [90]. De plus selon leurs valeurs l'algorithme convergera en plus ou moins d'itérations. Comme le nombre d'itération est limité, cela peut à nouveau provoquer des problèmes de convergence. En effet si le problème local (le problème de partage) est mal paramétré, cela peut augmenter grandement le temps de calcul puisque chaque itération du problème global prendra plus de temps. Enfin si le problème local ne converge pas ou n'est pas assez précis, cela peut empêcher le problème global de converger.

Toutes les méthodes considérées dans cette partie ont au moins deux facteurs de pénalité. En effet pour la résolution du marché, il faut ρ_g pour réaliser le consensus et ρ_l pour résoudre le problème de partage. La méthode **DC-EndoPF** utilise aussi



(a) temps (b) nombre d'itération

FIGURE 5.20 – Influence du facteur de pénalité pour le marché P2P sur CPU



(a) temps (b) nombre d'itération

FIGURE 5.21 – Influence du facteur de pénalité pour le marché P2P sur GPU

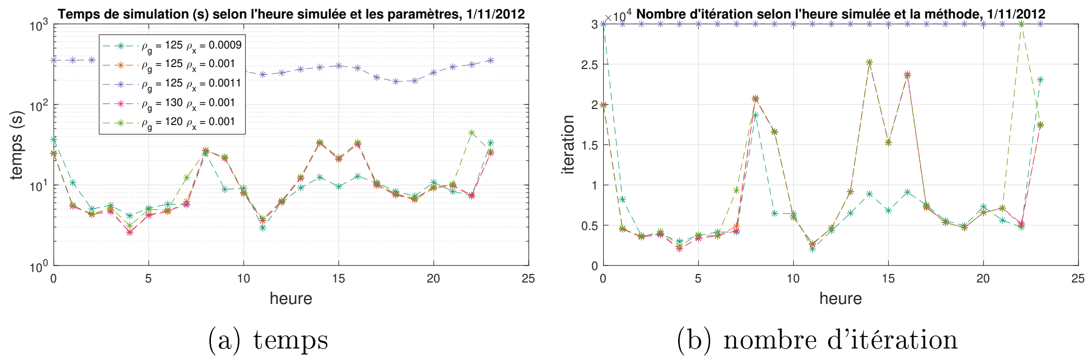
ρ_x pour relaxer les contraintes de flux.

L'objectif de cette partie n'est pas de trouver les paramètres optimaux, mais d'étudier leur impact sur les performances.

On va dans un premier temps étudier l'influence des valeurs de ρ_g et ρ_l. Comme la résolution du problème de partage est différent sur CPU et sur GPU on simulera les deux. On utilisera le problème de marché sans contraintes réseaux pour cette étude.

Pour rappel sur CPU, on peut paralléliser le problème local sur les agents mais le calcul du résidu est commun à tous. Tandis que sur GPU le résidu est spécifique à chaque agent. Chaque agent s'arrête de calculer une fois qu'il a convergé.

Les figures Fig. 5.20 et Fig. 5.21 représentent le temps et le nombre d'itérations nécessaire pour résoudre le cas Européen pour plusieurs valeurs de facteur de pénalité pour respectivement l'implémentation sur CPU avec openMP et l'implémentation sur GPU. On peut tout d'abord remarquer que l'algorithme converge



(a) temps (b) nombre d'itération

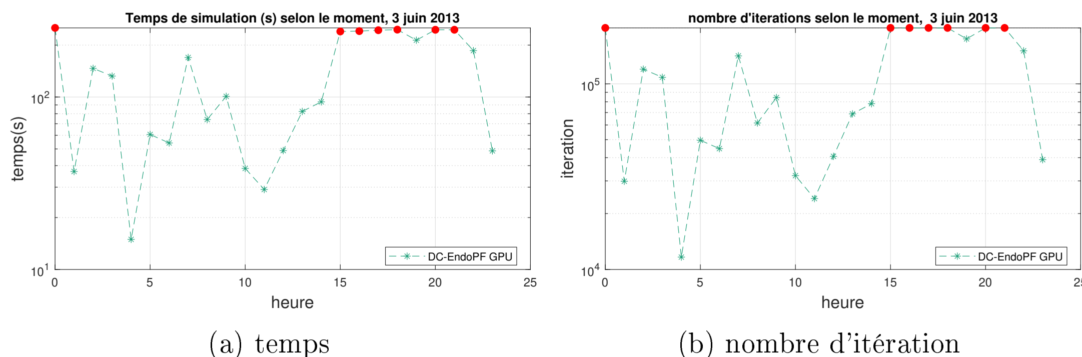
FIGURE 5.22 – Influence du facteur de pénalité pour le marché endogène DC sur GPU

au moins entre $\rho_g = 20$ et $\rho_g = 2000$. Ainsi on peut avoir une grande plage de convergence. Sur CPU, certain cas sont jusqu'à 10 fois plus rapides selon la valeur des paramètres. Pour le GPU, le facteur de pénalité a la même influence sur le nombre d'itération, Fig. 5.21b. Cependant l'effet sur le temps de calcul est bien plus faible, Fig. 5.22a. Dans les deux cas l'influence du facteur de pénalité sur le temps du premier pas est importante.

Les résultats sont très proches même en empêchant le facteur de pénalité de changer pendant la simulation. Ainsi pour le marché pair à pair, le réglage du facteur de pénalité n'est pas très sensible. La rapidité de l'implémentation sur GPU permet de rapidement identifier l'ordre de grandeur qu'il faut choisir via quelque tests. Une heuristique utilisable est de regarder les valeurs des différents résidus au bout d'un certains nombre d'itérations. Si c'est le résidu associé à la contrainte d'antisymétrie qui est le plus grand, alors il faut augmenter ρ. Dans le cas contraire il faut le diminuer. S'ils sont équivalent alors ρ est bien choisis et il faut laisser plus d'itération pour converger.

Dans un second nous allons utiliser **DC-EndoPF** pour étudier l'influence de ρ_g et de ρ_x. On choisira $\rho_l = \rho_g$. On peut voir sur la Fig. 5.22 les temps pour la résolution d'un jour du cas Européen pour différentes valeurs de facteur de pénalité. On peut y voir une bien plus grande sensibilité, notamment à la valeur de ρ_x. En effet avec $\rho_x = 0.0011$ plus aucun pas ne converge en 30 000 itérations. De même si $\rho_x = 0.0009$ le premier pas ne converge pas, cependant les suivants sont bien plus rapides. La variation de ρ_g semble avoir peu d'impact. Cependant le 22eme pas ne converge pas pour $\rho_g = 120$.

On a ainsi pu voir que selon le problème le facteur de pénalité pouvait avoir un grand impact sur la vitesse de convergence. Ce facteur a une importance moindre sur GPU grâce à la rapidité de calcul. Ainsi lorsque les paramètres sont mal réglés, l'accélération entre CPU et GPU sera plus importante que lorsque les paramètres



(a) temps (b) nombre d'itération

FIGURE 5.23 – Simulation d'un DC marché endogène avec des cas non convergeant

sont bien réglés. Dans le cas de **DC EndoPF** la très grande sensibilité du paramètre impose une phase de réglage sur chaque nouveau type de cas. De plus cela compromet les tests sur des cas complètement aléatoires.

4.2.2 Marge sur les contraintes de flux

Il existe un problème majeur à la méthode **DC-EndoPF** qui est détaillé dans [112]. En effet l'algorithme n'arrive pas dans certains cas à atteindre la précision demandée pour les contraintes de flux. Dans ces cas là quelles que soient les valeurs prises par les facteurs de pénalités ou de nombre d'itérations maximales permises, l'algorithme ne convergera pas.

Dans ce cas là l'algorithme ne converge pas mais il ne diverge pas non plus. Il arrive juste que le résidu associé aux contraintes de flux atteint une valeur minimale (petite) qui ne diminue plus au cours des itérations. Puis finalement les puissances ne changent plus non plus au cours des itérations.

Ainsi en simulant le cas européen avec une précision de $\epsilon_g = 0.01$, $\epsilon_x = 0.1$ et $\epsilon_l = 0.0002$, on obtient la Fig. 5.23. Les points rouges représente les cas qui n'ont pas convergé en 200 000 itérations. Ce nombre étant déjà absurde, il ne sert à rien de continuer à permettre plus d'itérations. Un réglage plus fin du facteur de pénalité ne permet pas de résoudre ce problème.

On est donc obligé de réduire la précision demandée à la contrainte de flux. Cependant si on fait cela, on pourrait avoir certaines contraintes de flux non respectées à la fin de la simulation. La figure 5.24 représente pour chaque ligne l'ensemble des valeurs prises pendant la simulation du mois de Juin 2013 pour $\epsilon_x = 1$. Les points dépassant la limites sont en rouge. C'est pourquoi il a été proposé de décaler les contraintes de flux d'un "$offset$" pour prendre de la marge.

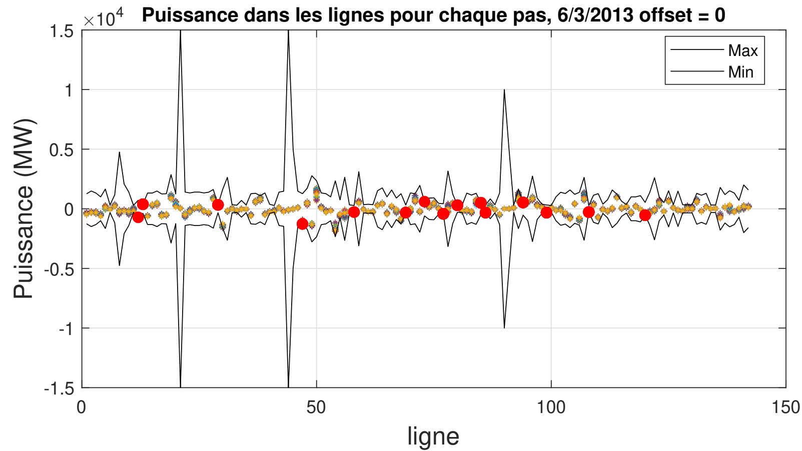
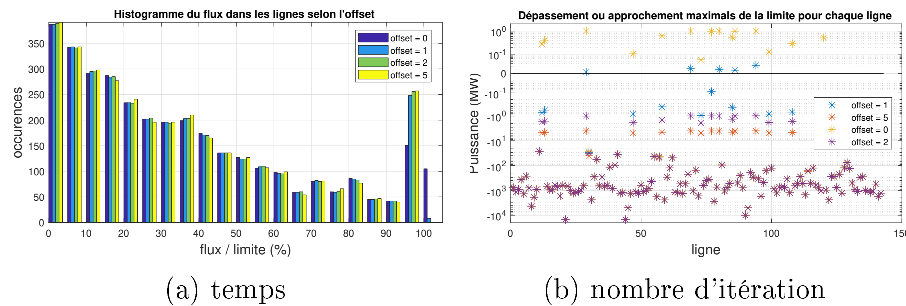


FIGURE 5.24 – Flux dans les lignes



(a) temps (b) nombre d'itération

FIGURE 5.25 – Analyse des flux dans les lignes selon le décalage

> Le fait de converger aussi rapidement avec ou sans offset nous permet de montrer que l'on a bien affaire à un problème de précision et non pas à un problème de faisabilité.

Le choix de la valeur du décalage n'est pas évidente. S'il est trop faible on ne respectera pas toujours les contraintes de flux. Cependant si trop de marge est prise, la solution sera sous-optimale. Ainsi la Fig. 5.25 présente les résultats pour différentes valeurs de décalage. L'histogramme de la charge de chaque ligne selon le décalage est représenté dans la Fig. 5.25a. On peut y voir qu'utiliser un décalage de 1 pu ne permet pas d'éviter de dépasser les limites. Cependant en réalité, le dépassement est très faible comme il peut être vu sur la Fig. 5.25b.

4.2.3 Influence du pas de calcul

La détermination du pas pour le calcul des résidus est assez direct. En effet ce paramètre n'influe que sur le nombre de fois où le résidu est calculé. Ainsi plus la

simulation nécessite d'itérations plus il est intéressant d'avoir un grand pas pour économiser. Cependant avoir un pas trop grand risque de demander de faire plus d'itérations potentiellement plus couteuses que de calculer les résidus.

Ainsi pour la majorité des méthodes sur GPU avoir un pas plus grand que 1 peut être intéressant car chaque itération est rapide et que chaque calcul de résidus est couteux car implique un transfert $GPU \rightarrow CPU$. Pour les méthodes **EndoConsensus** chaque itération étant très couteuse (résolution complète d'un OPF), économiser des calculs de résidus n'est pas rentable.

Un pas de $step_g = 2$ divise par 2 le nombre de calcul et transfert de GPU au prix dans le pire des cas d'une seule itération.

Ainsi la question qui sera étudiée dans cette partie sera la suivante *est -il intéressant de ne pas réaliser le calcul du SO (l'OPF) toutes les itérations pour la méthode* ***EndoConsensus*** *?*.

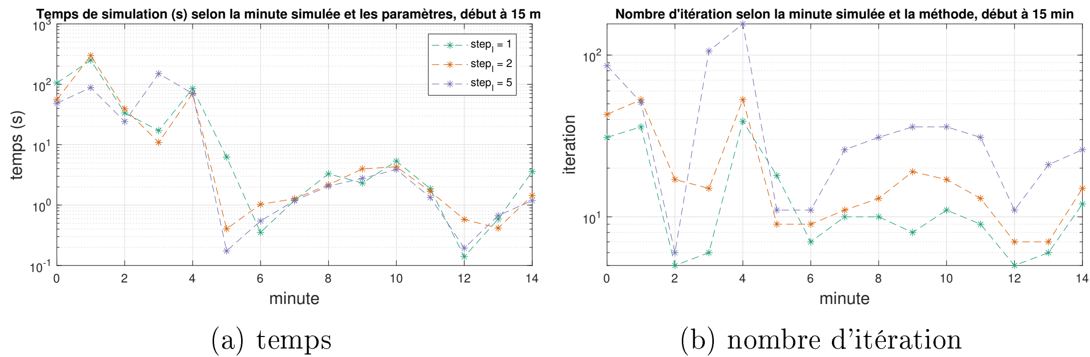
Pour répondre à cette question une simulation sur plusieurs pas du cas *TestFeeder* a été réalisée. Le pas entre chaque résolution du SO, noté $step_l$ a été changé. Pour ces simulations, le pas pour le calcul du résidu est choisi tel que $step_g = step_l$. Cela garantit que la résolution ne s'arrête que lors d'une itération où un OPF a été effectué et donc que le consensus est vraiment respecté à la fin.

Les résultats sont visibles sur la Fig. 5.26. Ces simulations ont été réalisées avec une précision de $\epsilon_g = 0.05$, $\epsilon_x = 0.005$ et $\epsilon_l = 0.0005$. On peut remarquer que malgré une grande variation du nombre d'itération sur la Fig.5.26b, les temps reste très proches sur la Fig. 5.26a. Globalement il semblerait qu'augmenter le pas, augmente le nombre d'itérations nécessaire pour converger. Mais même si le nombre d'itérations augmente, le temps total est plus faible. Cependant ce résultat semble très dépendant du cas d'étude. Ainsi $step_l = 5$ est bien plus rapide sur les 3 premiers pas mais extrêmement plus lent pour le 4ième.

En pratique, les résolutions du SO et du marché étant indépendantes, elles pouront être réalisées en parallèle. Ainsi il pourrait être intéressant de fixer le pas pour faire en sorte que le marché se mette à jour dès que l'OPF a fini ses calculs. En effet la résolution d'une itération du marché est plus rapide que la résolution complète d'un OPF.

4.2.4 Influence du matériel

Les performances d'un implémentation donnée dépend des caractéristiques physique des CPU et GPU. Cela peut englober aussi bien la mémoire disponible, la fréquence nominale ou le nombre de cœur. Cependant il est aussi possible que des paramètres extérieurs influent sur les performances d'un algorithme.



(a) temps (b) nombre d'itération

FIGURE 5.26 – Temps de simulation et nombre d'itérations selon le pas et la minute simulée

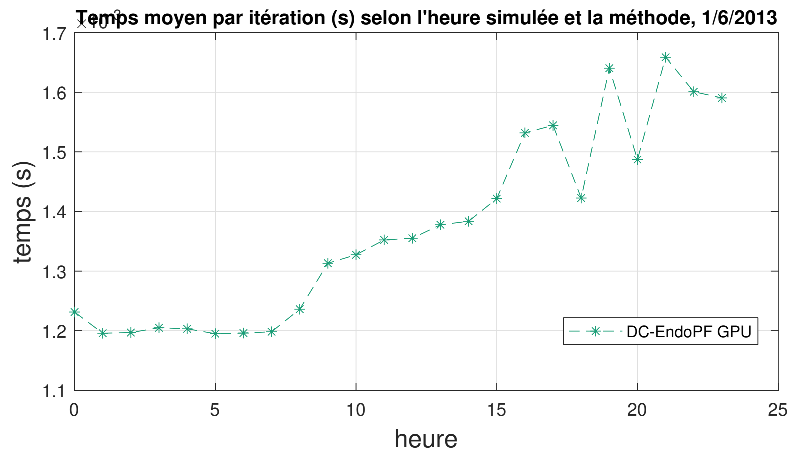


FIGURE 5.27 – Temps par itération selon l'heure simulée pour le **DC-EndoPF** sur GPU avec refroidissement

Ainsi la Fig. 5.27 représente les temps de calcul par itération pour l'algorithme **DC-EndoPF** lors de la simulation du cas Européen. On peut remarquer que si ce temps est constant sur les premiers pas, le temps augmente progressivement à partir de la neuvième heure simulée (qui est calculé en environ 1 min). Or cela correspond à l'heure simulé qui a besoin du plus d'itérations pour converger.

L'implémentation a été conçue pour utiliser le plus de ressources possibles du GPU. Cela conduit ainsi à une longue utilisation intensive du GPU pour les calculs. Cette utilisation provoque un échauffement du GPU qui doit donc réduire sa fréquence de fonctionnement afin de limiter la température. Cet étranglement thermique est particulière visible sur la Fig. 5.28. Cette figure a été obtenue en lançant exactement la même simulation que précédemment sauf que les ventilateurs servant à refroidir l'ordinateur ont été limités. On peut y voir des temps plus

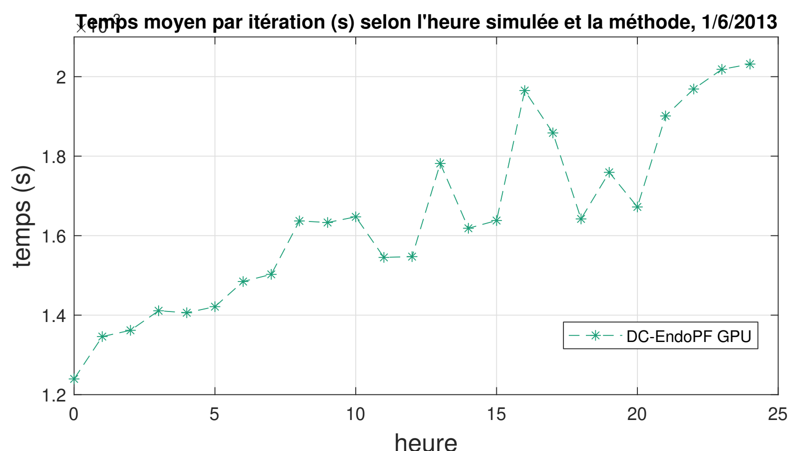


FIGURE 5.28 – Temps par itération selon l'heure simulée pour le **DC-EndoPF** sur GPU sans refroidissement

importants dès la première simulation et qui augmente plus rapidement.

Ce genre de phénomène reste assez limité dans les autres problèmes et type de test. En effet pour les tests aléatoires les résolutions sur CPU et GPU sont mélangés. De plus la création du cas d'étude qui est réalisé sur CPU a besoin d'un temps non négligeable pour être réalisée. Ainsi le GPU a le temps de refroidir entre chacune de ses utilisations. Lors du tests des autres algorithmes sur le cas Européen ou *TestFeeder*, le nombre d'itération est bien plus bas.

On remarque même dans le cas où le refroidissement est réalisé, les performance se dégradent au cours de la simulation. Il peut donc être intéressant de refroidir avec d'autres moyens le GPU. Les cartes FPGA sont connues pour moins consommer et donc moins s'échauffer que les GPU. Leur utilisation pourrait donc être intéressante sur ce cas.

5 Conclusion

Dans cette partie, la complexité des différents partitionnements CPU-GPU a pu être évaluée. Il a ainsi été démontré que la parallélisation sur GPU permettait de réduire l'explosion des temps avec l'augmentation des dimensions du problème. En effet, celle-ci a bien permis de réduire la complexité des optimisations réalisées pour le marché pair à pair ou pour le marché endogène.

Les différentes implémentations ont été testées sur des cas tests possédant plusieurs pas de données. Ces tests ont permis de démontrer l'accélération apportée par la parallélisation sur GPU. En effet, si l'accélération de la parallélisation sur CPU est limitée par le nombre de cœurs, résultant en une accélération entre 2 et

10. L'architecture hautement parallèle du GPU permet d'atteindre une accélération de l'ordre de la centaine. Cette accélération rend possible la simulation de grands cas d'étude sur de longs horizons temporels.

Il a aussi été montré que, dans la majorité des cas, le GPU permettait d'accélérer la simulation, quels que soient les cas d'études. De plus, le fait que les versions sur GPU soient plus rapides permet de réaliser plus facilement des études paramétriques pour trouver les paramètres optimaux de simulation.

Cependant, les algorithmes utilisés ont quelques faiblesses pouvant empêcher la convergence. Ce problème de précision se retrouve aussi bien sur CPU et GPU dans le cas du **DC-EndoPF**. Le problème de précision de l'OPF est spécifique au GPU. Cependant, ce problème peut être partiellement résolu en changeant la manière de résoudre l'OPF comme il a été réalisé dans cette thèse.

Chapitre 6

Conclusion et perspectives

Bilan

La gestion du réseau électrique est un défi de tous les instants nécessitant de bonnes capacités de prévision et du contrôle en temps réel. Cependant, la transition énergétique, avec les augmentations des moyens de production renouvelable intermittents et du nombre de consommateurs actifs, risque de bouleverser ce mode de gestion. L'utilisation de *SmartGrid* (pour réseau intelligent) pour une gestion plus locale et plus précise de l'énergie semble nécessaire pour éviter une surcharge des opérateurs centraux. Cependant, des simulations de grandes échelles sont nécessaires pour évaluer l'impact de cette gestion local de l'énergie sur le réseau global. De plus, avec la multiplication des agents, leur influence totale sur le réseau physique ne peut pas être négligée. La prise en compte des contraintes du réseau rend le problème d'autant plus difficile à résoudre. Les algorithmes de décentralisation complexifiant les calculs, la simulation fait face à un verrou calculatoire. En effet, les temps de simulation des systèmes décentralisés sont prohibitifs à grande échelle.

Une manière de lever ce verrou serait d'utiliser des supercalculateurs de plus en plus puissants. Cependant, les développements de GP-GPU (*General Purpose Graphic Prcessing Unit*) ou de FPGA permettent à tous d'avoir accès à des architectures hautement parallèles. Le réseau et les algorithmes de décentralisation étant intrinsèquement parallèles, l'utilisation de ces nouvelles architectures peut permettre de réduire les temps de calcul. Cependant, la fréquence d'un GPU ou d'un FPGA étant plus faible que celle d'un CPU, il est important de prendre en compte les spécificités matérielles du GPU pour effectivement réduire les temps.

L'objectif de cette thèse est de mener une démarche d'Adéquation Algorithme Architecture sur le problème de marché endogène pour permettre des simulations à grande dimension sur un long horizon temporel.

Le premier chapitre a posé les bases de la gestion des réseaux électriques et de l'accélération des calculs en général ou dans ce domaine spécifique. Il a ainsi pu être montré que le problème de marché décentralisé n'était jamais simulé à grande échelle. Même en utilisant un super calculateur, le nombre maximal d'agents simulés était de 300 à cause du temps de calcul. Il est donc nécessaire de suivre la démarche Adéquation Algorithme Architecture afin de résoudre le verrou du temps de calcul.

Ainsi, le deuxième chapitre a présenté la méthode d'Adéquation Algorithme

Architecture et le travail préliminaire de cette méthode. Différents algorithmes, métriques et architectures pouvant être utilisés ont été analysés et comparés. Enfin, les cas d'études qui seront utilisés pour réaliser la validation fonctionnelle, l'analyse de la complexité et enfin, les performances ont été décrits.

Les différents problèmes du réseau électrique ont été présentés et résolus dans le chapitre 3 avec différents algorithmes. Ainsi, la résolution sur CPU de Power Flow, de marché pair à pair, d'Optimal Power Flow et de marché endogène a pu être effectué. En testant sur différentes tailles, il a été montré que le passage à l'échelle est compromis en série. En effet, même si le temps de calcul est de l'ordre de la seconde, cela reste trop lent dans le cas d'une simulation long terme ou lors d'un dimensionnement. De plus, ce temps augmente rapidement avec les dimensions du problème rendant impossible la simulation à très grande échelle.

Une analyse de la complexité grâce à un partitionnement en bloc fonctionnel des différents algorithmes a été réalisée dans le chapitre 4. L'étude de la répartition des temps a permis de déterminer où sont les goulots d'étranglement des différents algorithmes. Ces parties ont ainsi pu être optimisées pour gagner le plus de temps possible. Ceci a permis d'implémenter des algorithmes optimisés spécifiquement pour une exécution sur GPU. La combinaison de ces algorithmes a permis de résoudre le problème de marché endogène sur GPU de manière optimisée.

Ces implémentations ont ainsi été testées dans le dernier chapitre. La parallélisation sur GPU a permis d'effectivement réduire la complexité des algorithmes et donc d'améliorer le passage à l'échelle. De plus, sur des cas de plusieurs milliers de bus ou d'agents, le passage sur GPU a extrêmement réduit les temps de simulation. Une implémentation optimisée sur une architecture GPU a permis de réduire de plus de 98% les temps de simulation d'un marché sans contraintes réseau. Comparée à un modèle de calcul sur une architecture conventionnelle type PC, la démarche d'adéquation algorithme-architecture a permis de définir un modèle de calcul sur GPU 1000 fois plus rapide lors de la simulation d'un DC-marché endogène et 10 fois plus rapide sur un marché AC-endogène sur un réseau radial. Ce dernier chapitre a aussi permis d'analyser la sensibilité des résultats aux cas d'études, aux paramètres de simulations et aux matériels. Ceci a permis de montrer que, dans tous les cas, le GPU permettait de réduire les temps de calcul de nos méthodes implémentées.

Les simulations à grande échelle de marché pair à pair tout en prenant en compte les contraintes réseau sont donc rendues possibles. Cependant, la surchauffe du GPU lors de son utilisation intensive et le manque de précision de certains calculs sur GPU sont deux points de vigilance à ne pas négliger.

Perspectives

L'application d'une démarche d'Adéquation Algorithme Architecture a permis de réduire grandement les temps de simulations à grande échelle. Cependant, ce travail appelle à d'autres recherches.

En effet, le code source ouvert démontre la faisabilité des études à grande échelle. Cependant, des ajustements mineurs doivent être apportés pour permettre l'utilisation pratique de ce code. Dans la version actuelle du code, la méthode PAC n'est pas fonctionnelle à grande dimension à cause d'une trop grande complexité mémoire. Il faudrait donc trouver d'autres structures de mémoire adaptées afin de permettre la simulation, notamment sur l'OPF où l'algorithme est censé avoir de meilleures propriétés de convergence que si l'on décentralise la résolution avec ADMM. De plus, l'algorithme permet d'être paramétré pour limiter les informations transmises lors de la résolution, ce qui devrait aussi être étudié. Afin de rendre ce code utilisable facilement par des gestionnaires du réseau ou des chercheurs, des modifications sont nécessaires. Ainsi, il faudra le refactoriser et réaliser une interface avec d'autres langages. Il serait aussi intéressant de tester les algorithmes avec des cas d'étude plus proche de ce que chercherait à simuler le gestionnaire de réseau.

Ces travaux de thèse ouvrent aussi des perspectives à moyen terme. On pourra ainsi implémenter l'application dans un autre langage (Julia) pour comparer l'impact du choix du langage sur les performances. L'objectif est d'aussi pouvoir utiliser les bibliothèques disponibles (PowerModels) pour réaliser des comparaisons de performance. Cela permettra aussi de faciliter l'utilisation de ce code par les chercheurs. Il faudra aussi, lors du changement de langage, rendre le code plus robuste afin de pouvoir garantir ses performances à un utilisateur extérieur. Ceci passera par l'ajout de nombreux tests pour, par exemple, vérifier les limites de précision atteignable selon l'algorithme, le problème et le matériel. On se concentrera notamment sur les problèmes de convergence de la méthode **OPFADMMGPU** pour identifier sur quels cas ils apparaissent. Enfin, ce code devrait permettre de réaliser de multiples études paramétriques afin de déterminer les paramètres déterminants, notamment pour déterminer *a priori* la valeur des facteurs de pénalité empiriquement ou en utilisant de l'IA. L'étude des effets de la normalisation des données selon l'architecture ou l'algorithme pourra aussi être réalisée.

Des perspectives à long terme pourraient être l'utilisation de ce code en tant que base pour pouvoir réaliser des études plus poussées sur le futur du réseau électrique. On pourra ainsi utiliser ce code pour étudier l'impact sur le réseau à l'échelle globale de la multiplication des productions renouvelables, des communautés d'autoconsommations, etc. Une démarche AAA pourra être appliquée sur un marché endogène sur un marché de transport. Cela permettra ainsi de l'appliquer dans le cas d'un marché endogène hétérogène qui réalise un consensus entre

le réseau de transport et tous les réseaux de distribution en parallèle. Permettant ainsi de contrôler optimalement le réseau en réalisant une coordination entre le GRT et le GRD. L'application pourrait notamment être adaptée pour être utilisée en tant que jumeau numérique en temps réel en passant sur FPGA. Enfin, le problème pourrait être étendu aux réseaux multiénergie couplés (avec un réseau de gaz, thermique ou mixte AC-DC).

Chapitre 6

Annexe

1 Démonstration formulation ADMM pour marché

1.1 Méthode 1

Pour simplifier les notations on ne considérera pas les contraintes qui bornes les échanges, et on n'utilisera pas les puissances totales des agents. Le problème global à optimiser est le suivant :

$$\underset{\mathbf{T}}{\operatorname{argmin}} \sum_n f_n(\mathbf{T}) \tag{6.1}$$

Avec pour contrainte que $\mathbf{T} = -\mathbf{T}^t$. Afin de permettre la distribution du calcul sur chaque agent on utilise une ADMM sous forme de consensus. Pour cela chaque agent n va avoir pour variable locale x_n une copie d'une partie des variables globale noté z_n. L'ensemble des variables globales est noté z. Chaque agent pour optimiser en respectant les contraintes a besoin de ses échanges $t_{nm}^{[n]}$ et d'une estimation/copie/volonté du symétrique de cet échange $t_{mn}^{[n]}$ pour tous ses pairs m. On note entre crochet pour chaque variables à quel agent elle appartient. L'ADMM sous forme de consensus nous dit que résoudre le premier problème peut se faire via la méthode itérative suivante :

$$\mathbf{x}_n^{k+1} = \underset{\mathbf{x}_n}{\operatorname{argmin}} f_n(\mathbf{x}_n) + \frac{\rho}{2}||\mathbf{x}_n - \mathbf{z}_n^k + \frac{\lambda_n^k}{\rho}||_2^2 \qquad \forall n \tag{6.2a}$$

$$\mathbf{z}^{k+1} = moy(\mathbf{x}_n^{k+1}) \tag{6.2b}$$

$$\lambda_n^{k+1} = \lambda_n^k + \rho(\mathbf{x}_n - \mathbf{z}_n) \qquad \forall n \tag{6.2c}$$

On peut remarquer que les contraintes sont donc distribuées sur chaque agent, ceux-ci pouvant l'appliquer sur leurs valeurs locales. La mise à jour des z étant la moyenne des copies de chacune de ses composantes ainsi dans notre cas on a :

$$z_{nm} = \frac{t_{nm}^{[n]} + t_{nm}^{[m]}}{2} \tag{6.3}$$

De part les contraintes on a pour tout agent n et pour tous les pairs m, la relation suivante $t^{[n]}_{nm} = -t^{[n]}_{mn}$. Cela permet donc de modifier la relation précédente :

$$z_{nm} = \frac{t^{[n]}_{nm} - t^{[m]}_{mn}}{2} \tag{6.4}$$

De plus :

$$\begin{aligned}
\lambda^{[n],k+1}_{nm} &= \lambda^{[n],k}_{nm} + \rho(t^{[n],k+1}_{nm} - z^{k+1}_{nm}) \\
&= \lambda^{[n],k}_{nm} + \rho(t^{[n],k+1}_{nm} - \frac{t^{[n],k+1}_{nm} - t^{[m],k+1}_{mn}}{2}) \\
&= \lambda^{[n],k}_{nm} + \frac{\rho}{2}(t^{[n],k+1}_{nm} + t^{[m],k+1}_{mn})
\end{aligned} \tag{6.5}$$

La minimisation pour obtenir x_n peut s'écrire ainsi (pour alléger les notations on explicitera pas ici que les variables appartiennent à l'agent n). Pour simplifier les notations on utilisera la contrainte $t_{nm} = -t_{mn}$. Ensuite on pourra remarquer que grâce à la contrainte précédente $z_{nm} = -z_{mn}$. Enfin on note $\rho' = 2 * \rho$ et $\mu^k_{nm} = \lambda^k_{nm} - \lambda^k_{mn}$.

$$\begin{aligned}
h_n(x_n) &= \frac{\rho}{2}||\mathbf{x}_n - \mathbf{z}^k_n + \frac{\lambda^k_n}{\rho}||^2_2 \\
&= \frac{\rho}{2}\Big[\sum_m t^2_{nm} + t^2_{mn} + 2 * t_{nm}(\frac{\lambda^k_{nm}}{\rho} - z^k_{nm}) + 2 * t_{mn}(\frac{\lambda^k_{mn}}{\rho} - z^k_{mn})\Big] \\
&= \frac{\rho}{2}\Big[\sum_m 2 * t^2_{nm} + 2 * t_{nm}(\frac{\lambda^k_{nm}}{\rho} - z^k_{nm} - \frac{\lambda^k_{mn}}{\rho} + z^k_{mn})\Big] \\
&= \frac{\rho}{2}\Big[\sum_m 2 * t^2_{nm} + 4 * t_{nm}(\frac{\lambda^k_{nm} - \lambda^k_{mn}}{2 * \rho} - z^k_{nm})\Big] \\
&= \frac{\rho'}{2}||t_{nm} - z^k_{nm} + \frac{\mu^k_{nm}}{\rho'}||^2_2
\end{aligned} \tag{6.6}$$

On remarque que l'on a :

$$\begin{aligned}
\mu^k_{nm} &= \lambda^k_{nm} - \lambda^k_{mn} \\
&= \lambda^{k-1}_{nm} - \lambda^{k-1}_{mn} + \frac{\rho}{2}((t^{[n],k+1}_{nm} + t^{[m],k+1}_{mn} - t^{[n],k+1}_{mn} - t^{[m],k+1}_{nm}) \\
&= \mu^{k-1}_{nm} + \frac{2 * \rho}{2}(t^{[n],k+1}_{nm} + t^{[m],k+1}_{mn}) \\
&= \mu^{k-1}_{nm} + \frac{\rho'}{2}(t^{[n],k+1}_{nm} + t^{[m],k+1}_{mn})
\end{aligned} \tag{6.7}$$

On peut remarquer que l'on a $\mu^k_{nm} = -\mu^k_{mn}$ (de l'agent n). On peut donc ne conserver qu'une variable duale par échange pour chaque agent. Ainsi en remplaçant ρ

par ρ' et λ par μ on remarque que la minimisation de x_n ne fait intervenir que les échanges directement dépendant de l'agent (les $t_{nm}^{[n]}$). Ainsi l'utilisation, le calcul et le stockage des puissances des vis-à-vis $t_{mn}^{[n]}$ n'est plus nécessaire. Ainsi pour alléger les notations l'échange t_{nm} appartient à l'agent n. De même on n'a besoin que d'une variable duale par pair de chaque agent. Ainsi contrairement à la version classique de l'ADMM, seuls les échanges voulus par les agents sont communiqués, le reste des grandeurs peuvent être déduit de ces valeurs. Ce qui nous permet de conclure que l'utilisation d'un ADMM sous forme de consensus revient à minimiser (en revenant à une notation de ρ pour le facteur de pénalité et de λ pour la variable duale souhaitée) :

$$\begin{aligned} \mathbf{T}_n^{k+1} = & \underset{\mathbf{T}_n}{\text{argmin}} \quad g_n(p_n) + \sum_{m \in \omega_n} \gamma_{nm} t_{nm} \\ & + \sum_{m \in \omega_n} \Big(\frac{\rho}{2} (t_{nm} - \frac{t_{nm}^k - t_{mn}^k}{2} + \frac{\lambda_{nm}^k}{\rho})^2 \Big) \\ & \text{t.q.} \quad t_{nm} \in C \end{aligned} \tag{6.8}$$

Cas avec les puissances Si les puissances sont des variables, dans le cas du marché, elles n'ont pas besoin d'être partagées. Ainsi on a donc $z_n = p_n^{[n]}$. La variable duale associée est donc nulle. Ainsi cela rajoute le terme $||p_n - z_n^k||_2^2$. Ce terme tend vers 0 avec la convergence du problème. Ce terme induit une "inertie" sur le changement de la valeur de la puissance dû au caractère itératif de la méthode. Ainsi plutôt que de relaxer cette contrainte on peut directement considérer p_n comme la valeur de z_n à tout instant.

Dans le cas où l'on rajoute un **SO** pour gérer d'autres contraintes, on doit faire un consensus entre l'agent et le **SO** pour les puissance. Ainsi on a $z_n = \frac{p_n^{[n]} + p_n^{[SO]}}{2}$. La variable duale vaut $\lambda_n^{k+1} = \lambda_n^k + \rho(p_n^{[n],k+1} - z_n^{k+1}) = \lambda_n^k + \frac{\rho}{2}(p_n^{[n],k+1} - p_n^{[SO],k+1})$ du point de vue de l'agent. Du point de vu du gestionnaire, $\lambda_n^{k+1} = \lambda_n^k + \frac{\rho}{2}(p_n^{[SO],k+1} - p_n^{[n],k+1})$.

Le terme à ajouter dans la fonction coût d'un agent est donc :

$$||p_n - z_n^k + \frac{\lambda_n^k}{\rho}||_2^2 = ||p_n - \frac{p_n^{[n]} + p_n^{[SO]}}{2} + \frac{\lambda_n^k}{\rho}||_2^2 \tag{6.9}$$

D'un autre côté la fonction coût du gestionnaire est :

$$\sum_n ||p_n - z_n^k + \frac{\lambda_n^k}{\rho}||_2^2 = \sum_n ||p_n - \frac{p_n^{[SO]} + p_n^{[n]}}{2} + \frac{\lambda_n^k}{\rho}||_2^2 \tag{6.10}$$

Remarque : Si les valeurs duales des puissances sont initialisées à 0 on peut remarquer que $\lambda_n^{[n]} = -\lambda_n^{[SO]}$ pour toutes les itérations. Ainsi on n'a besoin de ne conserver qu'une variable duale. Cependant dans une implémentation réelle il apparait plus logique que chaque agent (dont le SO) calcule sa variable duale et que seules les puissances soient échangées.

Prise en compte des pertes Les pertes peuvent être partagées d'un côté par l'agent des pertes qui va les racheter sur le marché et de l'autre sur le **SO** qui va déterminer sa valeur physique. On rajoute donc un terme dans la fonction coût $(p_{loss} - z_{loss}^k + \frac{\lambda_{loss}^k}{\rho})^2$. Avec $z_{loss} = \frac{p_{loss}^{[0]} + p_{loss}^{[SO]}}{2}$ et du point de vu du gestionnaire, $\lambda_{loss}^{k+1} = \lambda_{loss}^k + \frac{\rho}{2}(p_{loss}^{[SO],k+1} - p_{loss}^{[0],k+1})$.

Du point de vue de l'agent des pertes, c'est directement sa fonction coût qu'il doit optimiser. Cependant du point de vue de **SO**, la relation suivante doit être vérifiée $p_{loss}^{[SO]} = -\sum_n p_n^{[SO]}$.

1.2 Méthode 2

Dans cette version là, on n'utilise pas l'ADMM sous forme de consensus. On va plutôt définir x comme l'ensemble des variables des échanges et z comme la copie de ces variables et sur laquelle doit s'appliquer les contraintes d'antisymétrie. Le problème d'ADMM se resout donc ainsi :

$$x^{k+1} = \underset{x}{\operatorname{argmin}}\, L_\rho(x, z^k, y^k) \tag{6.11a}$$

$$z^{k+1} = \underset{z}{\operatorname{argmin}}\, L_\rho(x^{k+1}, z, y^k) \tag{6.11b}$$

$$y^{k+1} = y^j + \rho \cdot (Ax + Bz - c) \tag{6.11c}$$

Avec

$$L_\rho(x, z, y) = f(x) + v(z) + y^T(Ax + Bz - c) + \frac{\rho}{2}||Ax + Bz - c||_2^2 \tag{6.12}$$

Dans notre cas on a $c = 0$ et $A = -B = I$ la matrice identité. De plus aucune fonction coût n'est associée aux z, ainsi $v(z) = 0$. On utilisera la notation réduite, ainsi $y^T(Ax + Bz - c) + \frac{\rho}{2}||Ax + Bz - c|| = \frac{\rho}{2}||Ax + Bz - c + \frac{y^T}{\rho}||$. Le problème s'écrit donc :

$$x^{k+1} = \underset{x}{\operatorname{argmin}}\, f(x) + \frac{\rho}{2}||x - z^k + \frac{y^k}{\rho}||_2^2 \tag{6.13a}$$

$$z^{k+1} = \underset{z}{\operatorname{argmin}}\, \frac{\rho}{2}||x^{k+1} - z + \frac{y^k}{\rho}||_2^2 \tag{6.13b}$$

$$y^{k+1} = y^j + \rho \cdot (x^{k+1} - z^{k+1}) \tag{6.13c}$$

L'optimisation de z nécessite le fait de respecter la contrainte $z_{nm} = -z_{mn}$. Ainsi on doit minimiser une somme avec des termes indépendant sauf 2 à deux. La minimisation devient donc :

$$\begin{aligned}\frac{\rho}{2}||\mathbf{x}^{k+1} - \mathbf{z} + \frac{\mathbf{y}^k}{\rho}||_2^2 &= \sum_n \sum_m (x_{nm} - z_{nm} + \frac{y_{nm}}{\rho})^2 \\ &= \sum_{(n,m)} (x_{nm} - z_{nm} + \frac{y_{nm}}{\rho})^2 + (x_{mn} - z_{mn} + \frac{y_{mn}}{\rho})^2\end{aligned} \tag{6.14}$$

En substituant grâce à la contrainte, on obtient qu'il faut minimiser chaque terme de la forme $(x^{k+1}_{nm} - z_{nm} + \frac{y^k_{nm}}{\rho})^2 + (x^{k+1}_{mn} + z_{nm} + \frac{y^k_{mn}}{\rho})^2$. Or le minimum d'une somme de deux polynômes de même coefficients directeur a son minimum au milieu des deux minimum. Ainsi on a pour solution :

$$z^{k+1}_{nm} = \frac{1}{2}(x^{k+1}_{nm} + \frac{y^k_{nm}}{\rho} - x^{k+1}_{mn} - \frac{y^k_{mn}}{\rho}) \tag{6.15}$$

On peut utiliser cette expression dans le calcul des variables duales :

$$\begin{aligned}y^{k+1}_{nm} &= y^k_{nm} + \rho \cdot (x^{k+1}_{nm} - z^{k+1}_{nm}) \\ &= y^k_{nm} + \rho \cdot (x^{k+1}_{nm} - \frac{1}{2}(x^{k+1}_{nm} + \frac{y^k_{nm}}{\rho} - x^{k+1}_{mn} - \frac{y^k_{mn}}{\rho})) \\ &= \frac{y^k_{nm} + y^k_{mn}}{2} + \rho(\frac{x^{k+1}_{nm} + x^{k+1}_{mn}}{2}) \\ &= y^{k+1}_{mn} \\ &= y^k_{nm} + \rho(\frac{x^{k+1}_{nm} + x^{k+1}_{mn}}{2})\end{aligned} \tag{6.16}$$

La symétrie des variables duales permet de simplifier le calcul de z.

$$z^{k+1}_{nm} = \frac{x^{k+1}_{nm} - x^{k+1}_{mn}}{2} \tag{6.17}$$

Le stockage des valeurs de z ne sont pas nécessaire il suffit de remplacer son expression dans le calcul de x.

2 Démonstration de la résolution directe du marché Endogène

Cette partie démontre la résolution directe du marché endogène. En enlevant toute référence au échanges économiques entre les agents, cette partie démontre aussi la résolution des **OPF**.

2.1 Problème global

Les variables de ce problème sont :
- les échanges entre agent t_{nm} ;
- les puissances des agents p_n et q_n ;
- les variables du réseau : puissance dans les lignes P, Q, les courants (normes) dans les lignes l et les tensions (normes) sur les bus v.

Pour la résolution de ce problème, on se placera dans un réseau radial. Dans le cas où l'on cherchera à résoudre un OPF, les variables d'échanges t_{nm} seront ignorées. Pour rappel le problème à résoudre est donc :

$$
\begin{aligned}
\min_{\mathbf{T},\mathbf{P}} \quad & \sum_{n\in\Omega} \Big(g_n(p_n,q_n) + \sum_{m\in\omega_n} \beta_{nm} t_{nm} \Big) && && (6.18a)\\
\text{t.q.} \quad & \mathbf{T} = -{}^t\mathbf{T} && && (6.18b)\\
& (p_n,q_n) = (\sum_{m\in\omega_n} t^p_{nm}, \sum_{m\in\omega_n} t^q_{nm}) \qquad (\mu) && n\in\Omega && (6.18c)\\
& (\underline{p_n},\underline{q_n}) \leq (p_n,q_n) \leq (\overline{p_n},\overline{q_n}) && n\in\Omega && (6.18d)\\
& lb \leq t_{nm} \leq ub && n\in\Omega_p && (6.18e)\\
& \underline{V_b} \leq V_b \leq \overline{V_b} && b\in\mathcal{B} && (6.18f)\\
& \underline{l_{ij}} \leq l_{ij} \leq \overline{l_{ij}} \qquad (\delta_{S1-2}) && (i,j)\in\mathcal{L} && (6.18g)\\
& l_{ij}V_i^2 \leq P_{ij}^2 + Q_{ij}^2 && && (6.18h)\\
& P_{ij} = R_{ij}l_{ij} - \sum_{n\in\mathcal{N}_j} p_n^{agent} + \sum_{m:j->m} P_{jm} && (i,j)\in\mathcal{L} && (6.18i)\\
& Q_{ij} = X_{ij}l_{ij} - \sum_{n\in\mathcal{N}_j} q_n^{agent} + \sum_{m:j->m} Q_{jm} && (i,j)\in\mathcal{L} && (6.18j)\\
& V_i^2 = V_j^2 - 2(R_{ij}P_{ij} + X_{ij}Q_{ij}) + (R_{ij}^2 + X_{ij}^2)l_{ij} && b\in\mathcal{B} && (6.18k)\\
& \underline{S}_{loss} = \sum_{(i,j)\in\mathcal{L}} l_{ij}\underline{z}_{ij} && && (6.18l)\\
& V_{ref} = V_0 && \textit{reference} && (6.18m)
\end{aligned}
$$

Pour résoudre on cherche à décentraliser sur les bus. Ainsi on utilise des variables (vecteur) slack $\mathbf{y}_b$ qui représente les valeurs dont le bus b a besoin pour résoudre son propre problème. On a donc $\mathbf{x}_i = (t_{nm}, p_n, v_i, S_i, l_i)$ $\forall i \in \mathcal{B}$, $\forall n \in \mathcal{N}_b$, $\forall m \in \omega_n$ et $x_{Loss} = (p_{loss}, q_{loss})$. On conserve pour la minimisation de x la fonction coût (6.18a) et les contraintes du côté des puissances : (6.18c)-(6.18e), et du côté du réseau (6.18f)-(6.18h).

Les variables slacks doivent prendre en compte les autres contraintes, c'est à dire (6.18b) et (6.18l) pour les puissances et (6.18i)- (6.18k). Ainsi par rapport à

l'OPF, on rajoute les échanges économique. En posant le Lagrangien suivant :

$$\begin{aligned} L_\rho(\mathbf{x},\mathbf{z},\mathbf{y}) &= f(\mathbf{x}) + v(\mathbf{y}) + \mu^T(\mathbf{A}\mathbf{x}+\mathbf{B}\mathbf{y}-\mathbf{c}) + \frac{\rho}{2}||\mathbf{A}\mathbf{x}+\mathbf{B}\mathbf{y}-\mathbf{c}||_2^2 \\ &= f(\mathbf{x}) + v(\mathbf{y}) + \frac{\rho}{2}||\mathbf{A}\mathbf{x}+\mathbf{B}\mathbf{y}-\mathbf{c}+\frac{\mu}{\rho}||_2^2 \end{aligned} \tag{6.19}$$

Dans notre cas on ne vise que des égalités entre les variables de x et y, ainsi $(Ax + Bz - c)$ revient à faire $x - y$. Lorsque la variable apparait plusieurs fois, alors pour chaque copie j on doit faire $x - y_j$.

Le problème d'optimisation est donc :

$$\mathbf{x}^{k+1} = \underset{\mathbf{x}\in K_x}{\operatorname{argmin}}\, L_\rho(\mathbf{x},\mathbf{y}^k,\mu^k) \tag{6.20a}$$

$$\mathbf{y}^{k+1} = \underset{\mathbf{y}\in K_y}{\operatorname{argmin}}\, L_\rho(\mathbf{x}^{k+1},\mathbf{y},\mu^k) \tag{6.20b}$$

$$\mu^{k+1} = \mu^k + \rho(\mathbf{x}^{k+1} - \mathbf{y}^{k+1}) \tag{6.20c}$$

On pose pour les variables slacks $\mathbf{y}_i = (t_{nm}, p_n, v_i, S_i, l_i, t_{mn}, S_c, l_c, v_{A_i})$ $\forall i \in \mathcal{B}$, $\forall c \in \mathcal{C}_i$, $\forall n \in \mathcal{N}_b$, $\forall m \in \omega_n$ et $\mathbf{y}_{Loss} = (p_{loss}, q_{loss}, l_i)$ $\forall i \in \mathcal{B}^*$. On remarque que les différentes minimisations peuvent être séparé sur les différents bus i car toutes les contraintes et fonctions coûts peuvent être séparé sur les bus. En effet sur la minimisation de la variable duales les contraintes entre les bus concernent les copies des variables et chaque bus a les copies dont il a besoin tout comme c'était le cas lors de l'OPF.

2.2 Problème dual

Le problème dual est le suivant (à noter que par abus de notation on considère que l'on compte x^{k+1} autant de fois qu'il a été copié) :

$$\begin{aligned} min \quad & \frac{\rho}{2}||\mathbf{y} - \mathbf{x}^{k+1} - \frac{\mu^k}{\rho}||_2^2 \\ & \text{t.q. } (6.18b), (6.18l), (6.18i) - (6.18k) \end{aligned} \tag{6.21}$$

On doit donc minimiser une fonction quadratique de la forme $\frac{1}{2}\mathbf{y}^T\mathbf{P}\mathbf{y} + \mathbf{c}^T\mathbf{y}$ soumise à des contraintes d'égalités. Plutôt que de créer une grande matrice de contrainte tel que $\mathbf{A}\mathbf{y} = \mathbf{0}$ on va séparer le problème en deux parties. La première sera sur les variables de réseau, on ne conserve que les contraintes (6.18i)- (6.18k) pour tous les bus et (6.18l) pour un bus fictif. Cela permet de garder une matrice dont le nombre de ligne ne varie pas avec les dimensions du problèmes. On résout donc en utilisant la forme fermée (section 4.3.2). On a donc :

$$\mathbf{y} = \mathbf{H}^{-1}\mathbf{c}^T \tag{6.22}$$

avec $\mathbf{c}^T = \rho\mathbf{x}^{k+1} + \mu^k$ et $\mathbf{H} = (\mathbf{P}^{-1}\mathbf{A}^T(\mathbf{A}\mathbf{P}^{-1}\mathbf{A}^T)^{-1}\mathbf{A}\mathbf{P}^{-1} - \mathbf{P}^{-1})$.

La deuxième partie considèrera les échanges. Pour la prise en compte de (6.18b), on va substituer pour résoudre. On retrouve les mêmes formules que pour l'ADMM par consensus on a donc $t_{nm}^{[y]} = \frac{t_{nm}^{[x]} - t_{mn}^{[x]}}{2}$ et $\mu_{nm}^{k+1} = \mu_{nm}^k + \rho\frac{t_{nm}^{[x]} + t_{mn}^{[x]}}{2}$.

Ainsi tout comme pour le marché, on ne stockera pas y pour les échanges. De plus on conservera les variables du marché (échanges et variables duales associés) dans des variables à parts.

Pour respecter la contrainte physique des pertes tout en gardant l'équilibre des puissances pour le marché, la gestion de la variable $\underline{s}_{loss}$ est particulière. On ne peut pas faire un consensus entre la physique et le marché parce qu'il ne faut pas qu'un déséquilibre en puissance implique une augmentation ou diminution des pertes physiques. Ainsi à chaque pas de temps on définit la fonction coût de l'agent des pertes pour qu'il soit d'accord avec les pertes physiques. Cependant les pertes physiques sont directement calculées et ne dépendent pas d'un consensus.

2.3 Problème direct

Le problème directe est le suivant :

$$\begin{aligned} min \quad & g_n(p_n, q_n, t_{nm}) + \frac{\rho}{2}||\mathbf{x}^k - \mathbf{y} + \frac{\mu}{\rho}||_2^2 \\ & \text{t.q. } (6.18c) - (6.18e), (6.18f) - (6.18h). \end{aligned} \tag{6.23}$$

On remarque que dans cette minimisation les variables de réseau $(\underline{S}_i, l_i, v_i)$ sont complètement indépendant des variables du marché (p_n, q_n, t_{nm}). On va donc séparer la résolution en deux parties, $H_i^{(1)}$ sera associé au réseau du bus i, et $H_i^{(2)}$ sera associé aux variables de marché du bus i. Le $\mathbf{x}^k$ doit être répété autant de fois qu'il a été copié, ainsi on a :

$$H_i^{(1)} = |S_i - \hat{S}_i|^2 + |l_i - \hat{l}_i|^2 + \frac{|C_i| + 1}{2}|v_i - \hat{v}_i|^2 \tag{6.24}$$

$$H_i^{(2)} = \sum_{n \in \mathcal{N}_i} g_n(p_n, q_n, t_{nm}) + \frac{\rho}{2}|s_n - \hat{s}_n|^2 + \rho \sum_{m \in \omega_n} |t_{nm} - \hat{t}_{nm}|^2 \tag{6.25}$$

La partie de marché correspond à un problème de partage, là où celle de réseau est identique à celle de l'OPF.

Les variables $\hat{c}$ représente l'influence de y^k et μ^k sur la minimisation de x. On a de manière générale $\hat{x}_i = moy(y_{ij}) - \frac{moy(\mu_{ij})}{\rho}$. C'est à dire qu'il faut faire la moyenne entre la valeur des différentes copies et des variables duales associées afin de trouver ce terme (comme on le ferait pour ADMM sous forme de consensus).

Les échanges sont utilisés deux fois (chez l'agent et chez son voisin) qui sont sur les bus i et j (cela peut être le même bus). On a donc $t_{nm}^{[y_i]} = \frac{t_{nm}^{[x]} - t_{mn}^{[x]}}{2} = t_{nm}^{[y_j]}$. Ainsi on peut directement calculer $\hat{t}_{nm} = t_{nm}^{[y]} - \frac{\mu_{nm}}{\rho} = \frac{t_{nm}^{[x]} - t_{mn}^{[x]}}{2} - \frac{\mu_{nm}}{\rho}$.

Les puissances des agents sont nécessaires sur le bus où est l'agent et sur le bus fictif pour calculer les pertes. Ainsi on a $\hat{p}_n = \frac{p_n^{[y_b]} + p_n^{[y_{loss}]}}{2} - \frac{\mu_n^{[y_b]} + \mu_n^{[y_{loss}]}}{2\rho}$.

Dans le cas de l'OPF, on n'a pas besoin du bus fictif. Dans ce cas là $\hat{p}_i = p_i^{y_i} + \frac{\mu^{[y_i]}}{\rho}$ pour la puissance du bus i de la méthode **OPFADMM**. De même pour la puissance des agents de la méthode **OPFADMM2** on a $\hat{p_n} = p_n^{y_i} + \frac{\mu_n^{[y_i]}}{\rho}$ pour l'agent n sur le bus i.

Pour le flux de courant ou de puissance, il faut faire un consensus entre le bus et son ancêtre. On a donc pour l_i, P_i et Q_i la relation suivante :

$$\hat{c}_i = \frac{c_i^{[y_i]} + c_i^{[y_{A_i}]}}{2} - \frac{\mu_{c_i}^{[y_i]} + \mu_{c_i}^{[y_{A_i}]}}{2\rho} \tag{6.26}$$

Enfin pour les tensions v_i il faut réaliser un consensus entre un bus et l'ensemble de ses enfants. Ce qui donne :

$$\hat{v}_i = \frac{v_i^{[y_i]} + \sum_{j \in \mathcal{C}_i} v_j^{[y_j]}}{|C_i| + 1} - \frac{\mu_{v_i}^{[y_i]} + \sum_{j \in \mathcal{C}_i} \mu_{v_j}^{[y_j]}}{\rho(|C_i| + 1)} \tag{6.27}$$

2.3.1 Problème de partage

On cherche à minimiser le problème suivant :

$$\begin{aligned}
&\sum_{n \in \mathcal{N}_i} g_n(p_n, q_n) + \frac{\rho}{2}||s_n - \hat{s}_n||_2^2 + \sum_{m \in \omega_n} \quad \beta_{nm} t_{nm} + \rho||t_{nm} - \hat{t}_{nm}||_2^2 \\
s.t \quad & lb_n \leq t_{nm} \leq ub_n && \forall (n, m) \in (\mathcal{N}_i, \omega_n) \\
& (\underline{p}_n \underline{q}_n) \leq (p_n, q_n) \leq (\overline{p}_n \overline{q}_n) && \forall n \in \mathcal{N}_i \\
& p_n = \sum_{m \in \omega_n} t_{nm} && \forall n \in \mathcal{N}_i
\end{aligned} \tag{6.28}$$

Dans le cas où les agents n'ont qu'un seul partenaire commercial, on peut faire une résolution directe (car $p_n = t_{nm}$ dans ce cas). Cependant dans le cas général on pourra résoudre ce problème avec une ADMM sous la forme de partage (tout comme on le ferait dans un marché). On peut donc récrire la problème précédemment ainsi :

$$\sum_n \Big[\sum_m f_{nm}(x_{nm}) + g_n(\sum_m x_{nm}) \Big] \tag{6.29}$$

Les contraintes sur les puissances et les échanges sont sur d'autres optimisions. Ainsi chaque terme de la somme est indépendant. On peut donc résoudre agent par agent. Selon [90], soit k l'itération globale et j l'itération de cette minimisation, t_i le i^{ime} terme du vecteur T_n correspondant au i^{ime} échange de l'agent considéré, $\overline{t}$ est la valeur moyenne des échanges de l'agent n, et enfin $\tilde{p} \approx \frac{p_n}{M_n}$. Pour simplifier les notations on n'écrira que la puissance active, mais il faut garder à l'esprit que l'on a la même chose pour les puissance réactives. La minimisation est indépendante sur chaque agent n, ainsi on peut résoudre sur chaque agent indépendamment. La solution (où $\tilde{p} = \overline{t}$) est atteinte itérativement en suivant les étapes suivantes pour chaque agent n :

$$t_i^{j+1} = \underset{lb_n < t_i < ub_n}{\text{argmin}} \left(f(t_i) + \frac{\rho_l}{2} \left\| t_i - t_i^j + \overline{t}^j - \tilde{p}^j + \mu^j \right\|_2^2 \right) \tag{6.30a}$$

$$\tilde{p}^{j+1} = \underset{\underline{p_n} < M_n\tilde{p} < \overline{p_n}}{\text{argmin}} \left(g(M_n\tilde{p}) + \frac{M_n\rho_l}{2} \left\| \tilde{p} - \mu^j - \overline{t}^{j+1} \right\|_2^2 \right) \tag{6.30b}$$

$$\mu^{j+1} = \mu^j + \overline{t}^{j+1} - \tilde{p}^{j+1} \tag{6.30c}$$

$$\begin{aligned} f_i(t_i) &= \rho(t_i - \hat{t}_i^k)^2 + \beta_i \cdot t_{ni} \\ g(M_n\tilde{p}) &= M_n^2 \cdot 0.5a_n \cdot \tilde{p}^2 + M_n b_n \tilde{p} + \frac{\rho}{2} ||M_n\tilde{p} - \hat{p}_n||_2^2 \end{aligned} \tag{6.31}$$

On remarque que l'on doit réaliser la minimisation de deux fonctions scalaires de la forme :

$$\sum_j 0.5 \cdot a_j \cdot (y - b_j)^2 + \sum_j c_j \cdot y \tag{6.32}$$

La majorité des coefficients sont constants, en utilisant respectivement les indices t et p pour les coefficients des équations précédentes, les coefficients qui vont changer selon l'itération globale de l'OPF (k) ou locale (j) sont les suivants :

$$b_{t1} = \hat{t}_{ni}^k \tag{6.33a}$$

$$b_{t2} = t_i^j - \overline{t}^j + \tilde{p}^j - \mu^j \tag{6.33b}$$

$$b_{p1} = \mu^j + \overline{t}^{j+1} \tag{6.33c}$$

$$b_{p2} = \frac{\hat{p}_n^k}{M_n} \tag{6.33d}$$

Et l'ensemble des coefficients constants sont les suivants :

$$a_{t1} = \rho \tag{6.34a}$$
$$a_{t2} = \rho_l \tag{6.34b}$$
$$c_{t1} = \beta_{ni} \tag{6.34c}$$
$$a_{p1} = M_n \cdot \rho_l \tag{6.34d}$$
$$a_{p2} = M_n^2 \cdot \rho \tag{6.34e}$$
$$a_{p3} = M_n^2 \cdot a_n \tag{6.34f}$$
$$c_{p1} = M_n \cdot b_n \tag{6.34g}$$

Sur le bus fictif le problème à résoudre est le même. La seule différence est que a_{p3} et c_{p1} sont nul s'il n'y a pas de fonction coût associée aux pertes.

2.3.2 Solution du problème local OPFADMM

Le problème que l'on cherche à résoudre est le suivant :

$$\begin{aligned} \underset{x_1,x_2,x_3,x_4}{\text{argmin}} & \sum_i x_i^2 + \hat{c}_i x_i \\ \text{t.q.} \quad & \frac{x_1^2 + x_2^2}{x_3} \leq k^2 x_4 \quad (\gamma) \\ & \underline{x}_3 \leq x_3 \leq \overline{x}_3 \quad (\lambda) \\ & x_4 \leq \overline{x}_4 \quad (\delta) \end{aligned} \tag{6.35}$$

avec $x_i = (P, Q, v, l)$. Le Lagrangien associé est le suivant :

$$\begin{aligned} L(x, \gamma, \lambda, \delta) = & \sum_i (x_i^2 + \hat{c}_i x_i) + \gamma(\frac{x_1^2 + x_2^2}{x_3} - k^2 x_4) \\ & + \overline{\lambda}(x_3 - \overline{x}_3) - \underline{\lambda}(x_3 - \underline{x}_3) + \overline{\delta}(x_4 - \overline{x}_4) \end{aligned} \tag{6.36}$$

Le problème que l'on cherche à résoudre est strictement convexe, ainsi trouver l'optimum est équivalent à trouver là où le gradient s'annule. Cela revient à trouver

la solution au système d'équation suivant :

$$2x_1 + c_1 + 2\gamma\frac{x_1}{x_3} = 0 \tag{6.37a}$$

$$2x_2 + c_2 + 2\gamma\frac{x_2}{x_3} = 0 \tag{6.37b}$$

$$2x_3 + c_3 - \gamma\frac{x_1^2 + x_2^2}{x_3^2} + \overline{\lambda} - \underline{\lambda} = 0 \tag{6.37c}$$

$$2x_4 + c_4 - k^2\gamma + \overline{\delta} = 0 \tag{6.37d}$$

$$\overline{\lambda}(x_3 - \overline{x}_3) = 0 \ \overline{\lambda} \geq 0 \ x_3 \leq \overline{x}_3 \tag{6.37e}$$

$$\underline{\lambda}(x_3 - \underline{x}_3) = 0 \ \underline{\lambda} \geq 0 \ x_3 \geq \underline{x}_3 \tag{6.37f}$$

$$\overline{\delta}(x_4 - \overline{x}_4) = 0 \ \overline{\delta} \geq 0 \ x_4 \leq \overline{x}_4 \tag{6.37g}$$

$$\gamma(\frac{x_1^2 + x_2^2}{x_3} - k^2x_4) = 0 \ \gamma \geq 0 \ \frac{x_1^2 + x_2^2}{x_3} \leq k^2x_4 \tag{6.37h}$$

Pour résoudre on peut faire une distinction de cas en fonction de l'activation ou non des contraintes.

Contraintes de cone inactive Dans ce cas là on a $\gamma = 0$, et il suffit de prendre le minimum pour chaque x_i et de projeter dans l'espace autorisé. Ainsi :

$$\begin{aligned} x_1 &= -\frac{c_1}{2} \\ x_2 &= -\frac{c_2}{2} \\ x_3 &= min(max(-\frac{c_3}{2}, \underline{x}_3), \overline{x}_3) \\ x_4 &= min(-\frac{c_4}{2}, \overline{x}_4) \end{aligned} \tag{6.38}$$

Si la contrainte (6.37h) est bien respectée, alors l'optimisation est résolue. Sinon cela signifie que $\gamma \neq 0$ et que donc $\frac{x_1^2+x_2^2}{x_3} - k^2x_4 = 0$.

Cas avec coefficients nuls Ce cas est assez particulier et suppose que $c_1 = c_2 = 0$. Dans ce cas là on remarque que les deux premières équations donne $2x_{1,2}(1 + \frac{\gamma}{x_3}) = 0$. Or on sait que que $\gamma \geq 0$ et $x_3 \geq 0$. Cela implique que l'on a $x_1 = 0$ et $x_2 = 0$. Si x_4 est négatif, il faut imposer $x_4 = 0$ pour que la contrainte de cone soit respectée. Pour la suite on supposer que les deux coefficients sont non nuls.

Flux à une borne Dans cette partie là on considère que $x_4 = \overline{x}_4$ et $x_3 = \overline{x}_3$ ou $x_3 = \underline{x}_3$. Il nous reste donc 3 équations pour déterminer les valeurs de x_1, x_2 et γ.

$$2x_1 + c_1 + 2\gamma\frac{x_1}{x_3} = 0 \tag{6.39a}$$

$$2x_2 + c_2 + 2\gamma\frac{x_2}{x_3} = 0 \tag{6.39b}$$

$$\frac{x_1^2 + x_2^2}{x_3} - k^2 x_4 = 0 \tag{6.39c}$$

On peut remarque que l'on a $\frac{c_1}{x_1} = \frac{c_2}{x_2}$, ainsi on pose $p = \frac{x_1}{c_1 x_3} = \frac{x_2}{c_2 x_3}$. On peut utiliser cette nouvelle variable pour simplifier la dernière équation précédentes.

$$p^2(c_1^2 + c_2^2)x_3 - k^2 x_4 = 0 \tag{6.40}$$

On obtient de la dernière relation que $p = \pm\sqrt{\frac{k^2 x_4}{(c_1^2+c_2^2)x_3}}$. Connaissant p on peut déterminer x_1 et x_2. On peut vérifier que la solution est valide en calculant δ, λ pour vérifier qu'ils sont bien positifs.

Dans le cas où la tension n'est pas contrainte, on a $\lambda = 0$ ce qui nous permet d'utiliser l'équation suivante :

$$2x_3 + c_3 - \gamma\frac{x_1^2 + x_2^2}{x_3^2} = 0 = 2x_3 + c_3 - \gamma p^2(c_1^2 + c_2^2) \tag{6.41}$$

De la première équation on a $2pc_1x_3 + c_1 + 2\gamma c_1 p = 0$, ce qui nous permet d'avoir $\gamma = -\frac{2px_3+1}{2p}$. En substituant la valeur de γ on obtient que

$$2x_3 + c_3 + \frac{2p^2x_3 + p}{2}(c_1^2 + c_2^2) = 0 \tag{6.42}$$

On sait que l'on a $p^2x_3(c_1^2 + c_2^2) = k^2x_4$ et que $p = K\frac{1}{\sqrt{x_3}}$, (avec $K = \pm\sqrt{\frac{k^2x_4}{(c_1^2+c_2^2)}}$). Ce qui nous permet d'écrire avec $K' = K * (c_1^2 + c_2^2)/2$ une constante) :

$$2x_3 + c_3 + k^2x_4 + \frac{K'}{\sqrt{x_3}} = 0 = 2\sqrt{x_3}^3 + (c_3 + k^2x_4)\sqrt{x_3} + K' \tag{6.43}$$

Cette équation nous donne un polynôme de troisième degré (en fonction de $\sqrt{x_3}$) à résoudre pour déterminer x_3 (dans ce cas là il ne faut garder que les racines réelles et positives). Une fois que x_3 est connu on peut déterminer p puis x_1 et x_2.

Flux libre Dans cette partie on suppose que le flux n'est pas contraint, donc $\delta = 0$. On se trouve donc dans les mêmes cas que [59]. Ceci nous permet d'écrire le système suivant en substituant x_4 et γ :

$$2 + \frac{c_1}{x_1} + 4\frac{x_1^2 + x_2^2}{k^4 x_3^2} + \frac{2c_4}{k^2 x_3} = 0 \quad (6.44a)$$

$$2 + \frac{c_2}{x_2} + 4\frac{x_1^2 + x_2^2}{k^4 x_3^2} + \frac{2c_4}{k^2 x_3} = 0 \quad (6.44b)$$

$$2 + \frac{c_3}{x_3} - \frac{(x_1^2 + x_2^2)^2}{k^4 x_3^4} - c_4\frac{x_1^2 + x_2^2}{x_3^3} + \frac{\overline{\lambda} - \underline{\lambda}}{x_3} = 0 \quad (6.44c)$$

Dans le cas où x_3 est à une borne, il suffit de substituer p dans une des 2 premières équations pour obtenir :

$$\frac{4(c_1^2 + c_2^2)}{k^4}p^3 + (2\frac{c_4}{k^2 x_3} + 2)p + \frac{1}{x_3} = 0 \quad (6.45)$$

On peut résoudre ce polynôme du troisième degré pour déterminer la valeur de p et ensuite en déduire x_1 et x_2.

Dans le cas où x_3 n'est pas contraint, on a $\lambda = 0$. Ceci nous permet d'utiliser la 3eme équation. En divisant la 2eme équation par la 3eme (les deux sous la bonne forme, voir [59] pour la démonstration plus poussée) on obtient :

$$x_3 = -\frac{(c_1^2 + c_2^2)p + 2c_3}{2((c_1^2 + c_2^2)p^2 + 2)} \quad (6.46)$$

En substituant l'expression de x_3 dans une des deux premières équations pour obtenir :

$$\frac{(c_1^2 + c_2^2)^2}{k^4}p^4 + \frac{c_1^2 + c_2^2}{k^2}(\frac{2c_3}{k^2} - c_4)p^3 + (c_3 - \frac{2c_4}{k_2})p - 1 = 0 \quad (6.47)$$

Cette equation est un polynôme du quatrième degré à résoudre. Comme précédemment la détermination de p permet d'ensuite déterminer x_3 puis x_1 et x_2.

Dans tous les cas, les racines sont réelles. Il faut tester pour chaque racine si la solution proposée respecte bien toutes les contraintes.

3 Recherche des racines

3.1 Polynôme du 3 ème degré

Soit un polynôme de 3 ème degré définit par :

$$P(x) = x^3 + bx^2 + cx + d \quad (6.48)$$

Dans toute la thèse, on considère que les coefficients $b,\ c,\ d$ sont réels. En utilisant un changement de variable $z = x{+}b/3$ on peut travailler avec un nouveau polynôme de la forme $P2(z) = z^3 + pz + q$, avec :

$$\begin{aligned} p &= -b^2/3 + c \\ q &= b/27 * (2b^2 - 9c) + d \end{aligned} \tag{6.49}$$

Un polynôme du 3 ème degré peut avoir soit 3 racines réelles (avec potentiellement une racine double ou triple) ou une racine réelle et 2 racines complexes conjuguées. Dans le cas d'un polynôme sans second terme, la seule racine triple possible est $z = 0$ et correspond au cas $p = q = 0$. Une fois que les racines en z ont été déterminées, on trouve les racines en x via la relation suivante :

$$x = z - \frac{b}{3} \tag{6.50}$$

3.1.1 Résolution exacte

En utilisant les formules de Cardan, on peut calculer les 3 racines complexes du polynômes par les relations suivantes :

$$\begin{aligned} z_k &= u_k + v_k \\ u_k &= j^k \sqrt[3]{0.5 * (-q + \sqrt{\frac{-\Delta}{27}})} \\ v_k &= j^{-k} \sqrt[3]{0.5 * (-q + \sqrt{\frac{-\Delta}{27}})} \\ \Delta &= -(4p^3 + 27q^2) \end{aligned} \tag{6.51}$$

Pour évaluer efficacement les racines, on pourra distinguer différents cas selon le signe de Δ.

Si Δ est positif ou nul, les solutions sont réelles. Si $\Delta = 0$, soit $p = q = 0$ et dans ce cas là, la racine est triple à $z = 0$, soit on a p et q non nul. Ce qui implique que $z_0 = \frac{3*q}{p}$ et $z_1 = z_2 = \frac{3*q}{2*p}$. Cependant dans le cas non nul, on remarque que l'on reste obligé de calculer les racines à partir de nombre complexe. Ainsi l'expression finales des racines est :

$$z_k = 2\sqrt{\frac{-p}{3}} cos((arcos(\frac{3q\sqrt{3}}{2p\sqrt{-p}}) + 2k\pi)/3) \tag{6.52}$$

En pratique, numériquement on ne trouvera presque jamais $\Delta = 0$ à cause des approximations numériques des calculs. Cependant, même si on ne détecte pas que l'on a une forme particulière, toutes les racines trouvées seront proches de la solution que l'on cherche. Ainsi cela ne devrait pas poser de problème particulier.

Si Δ est négatif, alors une seule racine est réelle et est calculée par $z_0 = v_0 + u_0$.

En factorisant par cette racine, on obtient un polynôme du second degré duquel il est plus simple de déterminer la partie imaginaire et réelle des racines complexes. En effet avec cette méthode les calculs sont entièrement en réel. Dans notre code on définira $z'_1 = real(z_1) = real(z_2)$ et $z'_2 = imag(z_1) = -imag(z_2)$. Pour toujours renvoyer 3 réels pour les racines, et un entier pour le nombre de racines réelles (ce qui permet de déterminer le signe de Δ).

3.1.2 Résolution itérative

L'objectif des différentes méthodes itératives est de trouver une racine réelle. Ainsi on construit une suite x_n qui convergera vers une racine. Ensuite on peut factoriser le polynôme par cette racine pour obtenir un polynôme de degré deux à résoudre. Deux méthodes ont été testé : la méthode de Newton et celle de Halley. Les deux méthodes utilise le développement Taylor pour approximer le polynôme. Pour la méthode de Newton on construit la suite :

$$x_{n+1} = x_n + \frac{f(x_n)}{f'(x_n)} \tag{6.53}$$

et pour la méthode de Halley, c'est la suite suivante qui est utilisé en approximant $x_{n+1} - x_n$ par $\frac{f(x_n)}{f'(x_n)}$ d'un côté de l'équation :

$$x_{n+1} = x_n - \frac{2f(x_n)f'(x_n)}{2f'(x_n)^2 - f''(x_n)f(x_n)} \tag{6.54}$$

Il est possible dans le cas où il n'y a qu'une seule racine que la méthode itérative parte dans la mauvaise direction (*i.e.* la tangente mène vers un point d'inflexion sans qu'il y ait de racine entre ce point et l'initialisation). Dans ce cas là il faut changer le point d'initialisation. Pour cela on cherchera un antécédent qui change le signe du polynôme. Ainsi si $f(x_0) \geq 0$ on cherchera x'_0 tel que $f(x'_0) \leq 0$ et inversement.

3.2 Polynôme du 4 ème degré

Soit un polynôme de 4 ème degré définit par :

$$P(x) = x^4 + bx^3 + cx^2 + dx + e \tag{6.55}$$

Dans toute la thèse, on considère que les coefficients b, c, d, e sont réels. En utilisant un changement de variable $z = x+b/4$ on peut travailler avec un nouveau polynôme de la forme $P2(z) = z^4 + pz^2 + qz + t$, avec :

$$\begin{aligned} p &= c - \frac{3b^2}{8} \\ q &= d - \frac{bc}{2} + \frac{b^3}{8} \\ t &= e - \frac{bd}{4} + \frac{cb^2}{16} - \frac{3b^4}{256} \end{aligned} \tag{6.56}$$

Un polynôme du 4 ème degré peut avoir soit 4 racines réelles, 2 racines réelles et 2 complexes conjugués ou $2 * 2$ racines complexes conjuguées. Une fois que les racines en z ont été déterminées, on trouve les racines en x :

$$x = z - \frac{b}{4} \tag{6.57}$$

3.2.1 Résolution exacte par Lagrange

Soit un polynôme de 3 ème degré définit par :

$$R(y) = y^3 + 2py^2 + (p^2 - 4t)y - q^2 \tag{6.58}$$

En posant $y_1,\ y_2,\ y_3$ les racines (réelles ou complexes) de ce polynôme, la méthode de Lagrange permet de déterminer les racines du polynôme de degré quatre par l'expression suivante :

$$\begin{aligned} z_1 &= 0.5(\ \sqrt{y_1} + \sqrt{y_2} + \sqrt{y_3}) \\ z_2 &= 0.5(\ \sqrt{y_1} - \sqrt{y_2} - \sqrt{y_3}) \\ z_3 &= 0.5(-\sqrt{y_1} + \sqrt{y_2} - \sqrt{y_3}) \\ z_4 &= 0.5(-\sqrt{y_1} - \sqrt{y_2} + \sqrt{y_3}) \end{aligned} \tag{6.59}$$

La notation $\sqrt{y}$ signifie un nombre qui lorsque qu'il est mis au carré donne y. Ainsi toutes les racines peuvent être remplacées par $-\sqrt{y}$. Le signe dépend de celui de q. En effet il faut que l'on ait $\sqrt{y_1} \cdot \sqrt{y_2} \cdot \sqrt{y_3} = -q$. Ainsi si $q > 0$ il faut remplacer tous les $\sqrt{y}$ par $-\sqrt{y}$.

Plusieurs configurations sont possible :

- $R(y)$ a trois racines positives $\rightarrow P(z)$ a quatre racines réelles ;
- $R(y)$ a trois racines dont 2 négatives $\rightarrow P(z)$ a $2 * 2$ racines complexes conjuguées ;
- $R(y)$ a une racine positive et 2 racines complexes conjuguées $\rightarrow P(z)$ a 2 racines réelles et 2 racines complexes conjuguées.

Comme l'objectif est de trouver les racines réelles, le deuxième cas est directement résolu. De même dans le premier cas, une application des formules permet de déterminer directement les 4 racines réelles à partir de réels.

Pour résoudre le dernier cas il faut remarque que la racine du conjugué est égale au conjugué de la racine d'un nombre. Ainsi dans ce cas là les deux racines réelles sont les suivantes :

$$\begin{aligned} z_1 &= 0.5(\sqrt{y_1} + \sqrt{y_2} + \sqrt{y_3}) = 0.5(\sqrt{y_1} + 2 * real(\sqrt{y_2})) \\ z_2 &= 0.5(\sqrt{y_1} - \sqrt{y_2} - \sqrt{y_3}) = 0.5(\sqrt{y_1} - 2 * real(\sqrt{y_2})) \end{aligned} \tag{6.60}$$

3.2.2 Résolution exacte par Ferrari

L'idée de cette méthode est de factoriser le polynôme de degré quatre en deux polynômes de degré deux. On supposera dans la suite que $q! = 0$ (dans le cas contraire le polynôme est bicarré donc peut être résolu par un changement de variable $Z = z^2$). Pour cela on va définir un paramètre λ dont on cherchera à déterminer une "bonne" valeur. Pour cela on peut remarquer que :

$$\begin{aligned} z^4 + pz^2 + qz + t &= (z^2 + \lambda)^2 - 2\lambda z^2 - \lambda^2 + pz^2 + qz + t \\ &= (z^2 + \lambda)^2 - [(2\lambda - p)z^2 - qz + \lambda - t] \end{aligned} \tag{6.61}$$

Afin de pouvoir factoriser sous la forme d'un produit de deux polynômes on va chercher à mettre le deuxième terme sous la forme d'un carré. Pour cela il faut que le deuxième polynôme ait une racine double (c'est à dire que le déterminant est nul). On cherche donc λ_0 tel que :

$$8\lambda_0^3 - 4p\lambda_0^2 - 8t\lambda_0 + (4tp - q^2) = 0 \tag{6.62}$$

On pose ensuite $a_0 = 2\lambda_0 - p$ et $b_0 = -\frac{q}{2a_0}$, ce qui nous permet d'écrire :

$$(z^2 + \lambda_0)^2 - (a_0 z + b_0)^2 = (z^2 + \lambda_0 + a_0 z + b_0)(z^2 + \lambda_0 - a_0 z - b_0) \tag{6.63}$$

Ce qui donne deux polynômes de degré 2 à résoudre, ce qui nous permet de déterminer les quatre racines du polynôme.

Le changement de variable permettant d'éliminer le terme devant le x^3 est inutile pour la méthode de Ferrari qui peut être appliquée aussi bien sur $P(x)$ que sur $P2(z)$. Par contre dans les deux cas il faut vérifier que le polynôme ne soit pas bicarré pour éviter de réaliser une division par 0 lors de la factorisation.

3.2.3 Résolution itérative

La résolution itérative d'un polynôme de degré 4 se fait de la même manière que lorsque le polynôme de degré 3. Un fois cette racine identifiée, le polynôme peut être factorisé. Il faut donc maintenant trouver les racines du polynôme résultant de degré 3. Ceci peut être fait de manière analytique ou de manière itérative comme vu précédemment.

Contrairement aux polynômes de degré 3, un polynôme de degré 4 n'a pas nécessairement de racines réelles. Cependant dans le contexte de l'optimisation convexe, on peut montrer qu'une solution existe de manière certaine. Ainsi dans ce cas là on peut être sûr de l'existence d'au moins deux racines réelles.

4 Sur la non convergence de l'OPFADMMGPU

Comme il a été présenté dans [115], l'implémentation sur GPU permet de réduire les temps de calcul de l'OPF sur un réseau radial. Cependant il existe différents cas où quel que soit le nombre d'itération permis, l'algorithme ne converge pas. En effet à partir d'un certain nombre d'itérations, les résidus de la version sur GPU oscillent sans diminuer.

Pour représenter ce phénomène le cas Matpower 69 nœuds est utilisé. La représentation des résidus selon le nombre d'itération est représenté sur la Fig. 6.1. La représentation sur les premières itérations est représenté sur la Fig. 6.2. On peut remarquer que les résidus sont identiques entre la version sur CPU et sur GPU sur les 150 premières itérations. Ceci démontre que le problème ne vient pas spécifiquement d'un problème dans le cas d'étude (dans ce cas cela devrait diverger dès le début). Il semblerait plutôt qu'une des étapes ne se passe pas de la même manière sur CPU et sur GPU. Dans le cas où le GPU arrive à converger, les résidus sont identiques que l'implémentation soit sur CPU et sur GPU.

Les étapes de communications, de mise à jour global (calcul de y) ou du calcul de $\hat{C}$ sont des opérations élémentaires (addition, soustraction, multiplication, division). Les divisions ne sont qu'avec des diviseurs constants (ρ ou $1 + |C_i|$ par exemple). Ainsi aucunes de ces opérations ne devraient causer ce genre d'instabilité. En effet même si les résultats ne sont pas exactes, cela pourrait empêcher les résidus de diminuer trop bas mais cela ne devrait pas causer de comportement oscillatoire.

Ainsi il est plus probable que le problème provienne du calcul de x, et plus particulièrement du calcul de (P_i, Q_i, l_i, v_i). En effet pour calculer ces termes il faut faire une distinction de cas en fonction de racines de polynômes. Or le calcul exacte de racines demandent de réaliser des opérations complexes (dans le sens

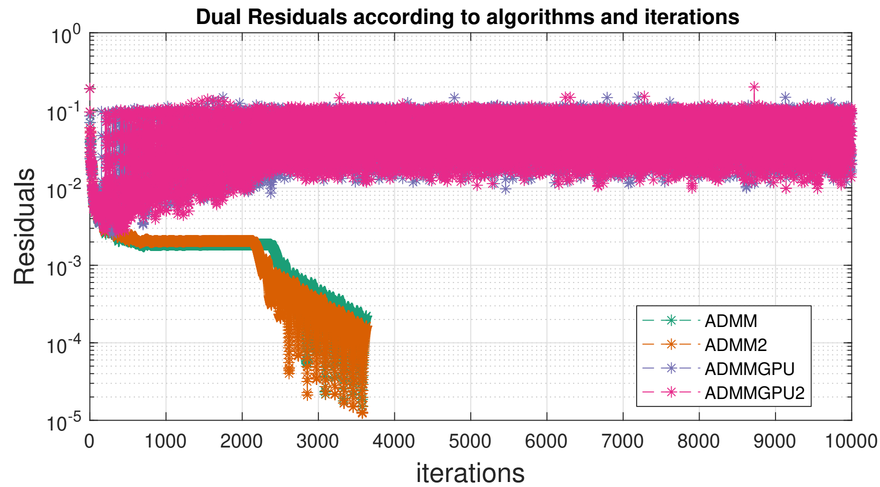


FIGURE 6.1 – Résidus dans le cas MatPower 69 bus (avec racines carrés en simple précision)

où ce ne sont pas juste des additions, multiplication etc... mais plutôt des racines cubiques, des opérations trigonométriques...). Ainsi selon les valeurs des racines les valeurs de x peuvent être complètement différentes avec des contraintes actives ou non.

En utilisant que des calculs en double précision sur GPU (ce qui est temporellement très peu efficace) lors du calcul des racines des polynômes, une convergence peut être atteinte sur le cas Matpower 69.

Cependant en testant sur des cas aléatoires, il apparait que la méthode **OPFADMMGPU** continue d'avoir des problèmes pour converger sur certains cas. La méthode **OPFADMMGPU2** convergent mieux, mais il existe des cas où la méthode ne convergent pas non plus.

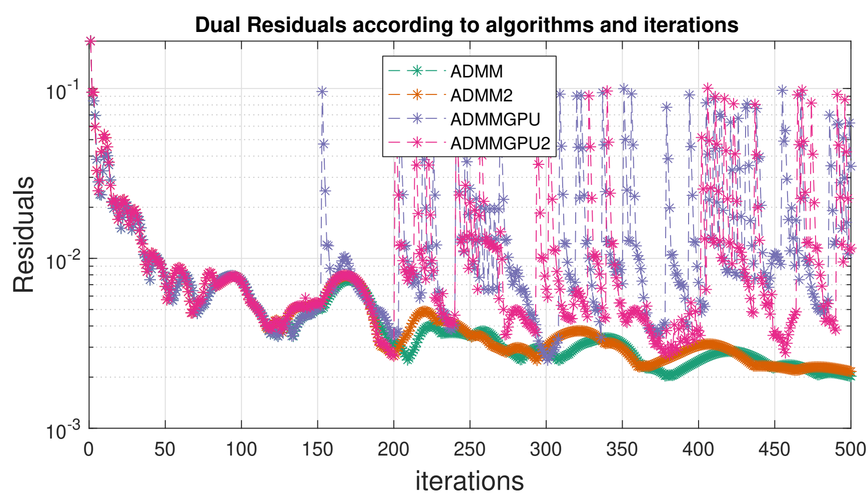


FIGURE 6.2 – Résidus dans le cas MatPower 69 bus (avec racines carrés en simple précision)

Bibliographie

État de l'art :

[1] RTE, Evolution de la production d'électricité, url : https://analysesetdonnees.rte-france.com/bilan-electrique-production (consulté le 20/02/2024)

[2] CRE, Présentation des réseaux électriques, url : https://www.cre.fr/Electricite/Reseaux-d-electricite/presentation-des-reseaux-d-electricite (consulté le 20/02/2024)

[3] Elec13-WordPress, l'électricité dans le monde, https://elec13.wordpress.com/2017/04/05/lelectricite-dans-le-monde-entier/ (consulté le 20/02/2024)

[4] RTE, La fréquence électrique, un indicateur d'équilibre du réseau, url : https://www.rte-france.com/riverains/la-frequence-electrique-un-indicateur-dequilibre-du-reseau (consulté le 20/02/2024)

[5] RTE, Mécanismes de marché, url : https://www.rte-france.com/chaque-seconde-courant-passe/concevoir-et-mettre-en-oeuvre-des-mecanismes-de-marche-innovants-pour-le-systeme-electrique (consulté le 20/02/2024)

[6] EDF, Mécanismes de capacité, url : https://www.edf.fr/entreprises/le-mag/le-mag-entreprises/decryptage-du-marche-de-l-energie/mieux-comprendre-le-mecanisme-de-capacite-en-3-questions-cles (consulté le 05/04/2024)

[7] Timothy Capper, Anna Gorbatcheva, Mustafa A. Mustafa, Mohamed Bahloul, Jan Marc Schwidtal, Ruzanna Chitchyan, Merlinda Andoni, Valentin Robu, Mehdi Montakhabi, Ian J. Scott, Christina Francis, Tanaka Mbavarira, Juan Manuel Espana, Lynne Kiesling, Peer-to-peer, community self-consumption, and transactive energy : A systematic literature review of local energy market models, Renewable and Sustainable Energy Reviews, Volume 162, 2022, 112403, ISSN 1364-0321, https://doi.org/10.1016/j.rser.2022.112403.

[8] H. Wang, C. E. Murillo-Sanchez, R. D. Zimmerman and R. J. Thomas, On Computational Issues of Market-Based Optimal Power Flow, IEEE Tran-

sactions on Power Systems, vol. 22, no. 3, pp. 1185-1193, Aug. 2007, doi : 10.1109/TPWRS.2007.901301.

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

[10] Bussar C, Stöcker P, Cai Z, Moraes Jr. L, Magnor D, Wiernes P, Van Bracht N, Moser A, Uwe Sauer D. 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, Volume 6, 2016, Pages 1-10, ISSN 2352-152X, https://doi.org/10.1016/j.est.2016.02.004.

[11] 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, ISSN 1364-0321, https://doi.org/10.1016/j.rser.2012.11.042.

[12] Saad Al-Sumaiti A, Ahmed M H, Salama M M. (2014) Smart Home Activities : A Literature Review, Electric Power Components and Systems, 42 :3-4, 294-305, doi : https://doi.org/10.1080/15325008.2013.832439

[13] 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 https://doi.org/10.1016/j.ijepes.2021.107639.,

[14] CRE, Introduction aux SmartGrid, url : https://www.smartgrids-cre.fr/introduction-aux-smart-grids (consulté le 06/03/2024)

[15] Tushar W, Saha T K, Yuen C, Smith D, Poor H V. Peer-to-peer trading in electricity networks : An overview. IEEE Transactions on Smart Grid, 11(4), 3185-3200. 2020

[16] Biskas P N, Bakirtzis A G . "A decentralized solution to the security constrained DC-OPF problem of multi-area power systems." IEEE Russia Power Tech. IEEE, 2005.

[17] Patari, N., Venkataramanan, V., Srivastava, A., Molzahn, D. K., Li, N., & Annaswamy, A. (2021). Distributed optimization in distribution systems : Use cases, limitations, and research needs. IEEE Transactions on Power Systems, 37(5), 3469-3481.

[18] Tushar W, Yuen C, Saha T K, Morstyn T, Chapman A C, Alam M J E, Poor H V . Peer-to-peer energy systems for connected communities : A review of recent advances and emerging challenges. Applied Energy. 2021

[19] Dong A, Baroche T, Le Goff Latimier R, Ben Ahmed, H. Convergence analysis of an asynchronous peer-to-peer market with communication delays. Sustainable Energy, Grids and Networks, 26, 100475. 2021

[20] Moret, F., Baroche, T., Sorin, E., & Pinson, P. (2018, June). Negotiation algorithms for peer-to-peer electricity markets : Computational properties. In 2018 power systems computation conference (PSCC) (pp. 1-7). IEEE.

[21] Dong A, Baroche T, Le Goff Latimier R, Ben Ahmed H. Asynchronous algorithm of an endogenous peer-to-peer electricity market. Madrid PowerTech. IEEE, 2021

[22] Tushar, W., Yuen, C., Mohsenian-Rad, H., Saha, T., Poor, H. V., & Wood, K. L. (2018). Transforming energy networks via peer-to-peer energy trading : The potential of game-theoretic approaches. IEEE Signal Processing Magazine, 35(4), 90-111.

[23] Sai, R. T. S., Mukherjee, A., Cecchi, V., & Kailas, A. (2012, March). Architecture exploration of a heterogeneous embedded processor for the smart grid. In 2012 Proceedings of IEEE Southeastcon (pp. 1-6). IEEE.

[24] Arenas-Martínez, M., Herrero-Lopez, S., Sanchez, A., Williams, J. R., Roth, P., Hofmann, P., & Zeier, A. A comparative study of data storage and processing architectures for the smart grid. IEEE International Conference on Smart Grid Communications (pp. 285-290). IEEE. 2010

[25] Baroche T, Le Goff Latimier R, Pinson P, Ben Ahmed H. Exogenous Cost Allocation in Peer-to-Peer Electricity Markets. IEEE Transactions on Power Systems, Institute of Electrical and Electronics Engineers, 2019, 34 (4), pp.2553 - 2564. ffhal-01964190f

[26] Chernova T, Gryazina E. Peer-to-peer market with network constraints, user preferences and network charges. International Journal of Electrical Power & Energy Systems, Volume 131, 2021, 106981, ISSN 0142-0615, `https://doi.org/10.1016/j.ijepes.2021.106981`.

[27] Romvary J J., et al. A proximal atomic coordination algorithm for distributed optimization. IEEE Transactions on Automatic Control 67.2 (2021) : 646-661.

[28] Ferro G, et al. A distributed-optimization-based architecture for management of interconnected energy hubs. IEEE Transactions on Control of Network Systems 9.4 (2022) : 1704-1716.

[29] Kargarian A, Mohammadi J, Guo J, Chakrabarti S, Barati M, Hug G, Baldick R. Toward distributed/decentralized DC optimal power flow implementation in future electric power systems. IEEE Transactions on Smart Grid, 9(4), 2574-2594. 2016

[30] Praça, I., Ramos, C., Vale, Z., & Cordeiro, M. (2003). MASCEM : a multiagent system that simulates competitive electricity markets. IEEE Intelligent Systems, 18(6), 54-60.

[31] Santos, G., Pinto, T., Praça, I., & Vale, Z. (2016). MASCEM : Optimizing the performance of a multi-agent system. Energy, 111, 513-524.

[32] Mikael Amelin. On Monte Carlo Simulation and Analysis of Electricity Markets, thèse de doct., 2004, url : https ://www.semanticscholar.org/paper/On-Monte-Carlo-Simulation-and-Analysis-of-Markets-Amelin/524cd7e75c14f165b3741 a0bd15b820c37e935f5.

[33] Wayes Tushar et al. Peer-to-Peer Trading in Electricity Networks : An Overview , in : IEEE Transactions on Smart Grid 11.4 (2020), p. 3185-3200, issn : 19493061, doi : 10.1109/TSG.2020.2969657, arXiv : 2001.06882.

[34] Wayes Tushar et al. A motivational game-theoretic approach for peer-to-peer energy trading in the smart grid . in : (2019), doi : 10.1016/j.apenergy.2019.03.111, url : https ://doi.org/10.1016/j.apenergy.2019.03.111

[35] Chenghua Zhang et al. Peer-to-Peer energy trading in a Microgrid , in : Applied Energy 220 (2018), p. 1-12, issn : 03062619, doi : 10.1016/J.APENERGY.2018.03.0 10, url : https ://doi.org/10.1016/j.apenergy.2018.03.010.

[36] Amrit Paudel et al., Peer-to-peer energy trading in a prosumer-based community microgrid : A game-theoretic model , in : IEEE Transactions on Industrial Electronics 66.8 (2019), p. 6087-6097, issn : 02780046, doi : 10.1109/TIE.2018.2874578.

[37] Morstyn, T., Teytelboym, A., & McCulloch, M. D. (2018). Bilateral contract networks for peer-to-peer energy trading. IEEE Transactions on Smart Grid, 10(2), 2026-2035.

[38] Sorin, E., Bobo, L., & Pinson, P. (2018). Consensus-based approach to peer-to-peer electricity markets with product differentiation. IEEE Transactions on Power Systems, 34(2), 994-1004.

[39] Baroche, T., Moret, F., Pinson, P. Prosumer markets : A unified formulation. PowerTech (pp. 1-6). IEEE. 2019

[40] Zhang, W., Kim, Y., & Kim, K. (2023). On solving unit commitment with alternating current optimal power flow on gpu. arXiv preprint arXiv :2310.13145.

[41] Keyhani, A., Abur, A., & Hao, S. (1989). Evaluation of power flow techniques for personal computers. IEEE transactions on power systems, 4(2), 817-826.

[42] Tinney, W. F., & Hart, C. E. (1967). Power flow solution by Newton's method. IEEE Transactions on Power Apparatus and systems, (11), 1449-1460.

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

[44] Li, X., & Li, F. (2014). GPU-based power flow analysis with Chebyshev preconditioner and conjugate gradient method. Electric Power Systems Research, 116, 87-93.

[45] Gopal, A., Niebur, D., & Venkatasubramanian, S. (2007, July). DC power flow based contingency analysis using graphics processing units. In 2007 IEEE Lausanne Power Tech (pp. 731-736). IEEE.

[46] Araújo, I., Tadaiesky, V., Cardoso, D., Fukuyama, Y., & Santana, Á. (2019). Simultaneous parallel power flow calculations using hybrid CPU-GPU approach. International Journal of Electrical Power & Energy Systems, 105, 229-236.

[47] Zhou, G., Bo, R., Chien, L., Zhang, X., Yang, S., & Su, D. (2017). GPU-accelerated algorithm for online probabilistic power flow. IEEE Transactions on Power Systems, 33(1), 1132-1135.

[48] Johnson, J., Vachranukunkiet, P., Tiwari, S., Nagvajara, P., & Nwankpa, C. (2005, August). Performance analysis of loadflow computation using FPGA. In Proc. of 15th Power Systems Computation Conference (pp. 22-26).

[49] Sooknanan, D. J., & Joshi, A. (2016, July). GPU computing using CUDA in the deployment of smart grids. In SAI Computing Conference (SAI) (pp. 1260-1266). IEEE.

[50] Foertsch, J., Johnson, J., & Nagvajara, P. (2005, October). Jacobi load flow accelerator using FPGA. In Proceedings of the 37th Annual North American Power Symposium, 2005. (pp. 448-454). IEEE.

[51] 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.

[52] 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.

[53] 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.

[54] 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.

[55] Frank, S., Steponavice, I., & Rebennack, S. (2012). Optimal power flow : A bibliographic survey I : Formulations and deterministic methods. Energy systems, 3, 221-258.

[56] Frank, S., Steponavice, I., & Rebennack, S. (2012). Optimal power flow : A bibliographic survey II : Non-deterministic and hybrid methods. Energy systems, 3, 259-289.

[57] Momoh, J. A., Adapa, R., & El-Hawary, M. E. (1999). A review of selected optimal power flow literature to 1993. I. Nonlinear and quadratic programming approaches. IEEE transactions on power systems, 14(1), 96-104.

[58] Momoh, J. A., El-Hawary, M. E., & Adapa, R. (1999). A review of selected optimal power flow literature to 1993. II. Newton, linear programming and interior point methods. IEEE transactions on power systems, 14(1), 105-111.

[59] Peng, Q., & Low, S. H. (2016). Distributed optimal power flow algorithm for radial networks, I : Balanced single phase case. IEEE Transactions on Smart Grid, 9(1), 111-121.

[60] Erseghe, T. (2014). Distributed optimal power flow using ADMM. IEEE transactions on power systems, 29(5), 2370-2380.

[61] Hug-Glanzmann, G., & Andersson, G. (2009). Decentralized optimal power flow control for overlapping areas in power systems. IEEE Transactions on Power Systems, 24(1), 327-336.

[62] . Lu, W., Liu, M., Lin, S., & Li, L. (2017). Fully decentralized optimal power flow of multi-area interconnected power systems based on distributed interior point method. IEEE Transactions on Power Systems, 33(1), 901-910.

[63] Molzahn, D. K., & Hiskens, I. A. (2019). A survey of relaxations and approximations of the power flow equations. Foundations and Trends® in Electric Energy Systems, 4(1-2), 1-221.

[64] , Kim, Y., & Kim, K. (2022, August). Accelerated computation and tracking of AC optimal power flow solutions using GPUs. In Workshop Proceedings of the 51st International Conference on Parallel Processing (pp. 1-8).

[65] Shin, S., Pacaud, F., & Anitescu, M. (2023). Accelerating optimal power flow with gpus : Simd abstraction of nonlinear programs and condensed-space interior-point methods. arXiv preprint arXiv :2307.16830.

[66] Geng, G., Jiang, Q., & Sun, Y. (2016). Parallel transient stability-constrained optimal power flow using GPU as coprocessor. IEEE Transactions on Smart Grid, 8(3), 1436-1445.

[67] Rakai, L., & Rosehart, W. (2014, January). GPU-accelerated solutions to optimal power flow problems. In 2014 47th Hawaii International Conference on System Sciences (pp. 2511-2516). IEEE.

[68] Roberge, V., Tarbouchi, M., & Okou, F. (2016). Optimal power flow based on parallel metaheuristics for graphics processing units. Electric Power Systems Research, 140, 344-353.

[69] Wei, H., Sasaki, H., Kubokawa, J., & Yokoyama, R. (1998). An interior point nonlinear programming for optimal power flow problems with a novel data structure. IEEE Transactions on Power Systems, 13(3), 870-877.

[70] Murach, M., Nagvajara, P., Johnson, J., & Nwankpa, C. (2005, October). Optimal power flow utilizing FPGA technology. In Proceedings of the 37th Annual North American Power Symposium, 2005. (pp. 97-101). IEEE.

[71] Youngdae Kim, François Pacaud, Kibaek Kim, and Mihai Anitescu. Leveraging GPU batching for scalable nonlinear programming through massive Lagrangian decomposition. arXiv preprint arXiv :2106.14995. 2021.

[72] Ryan H, Marqusee J. "Designing resilient decentralized energy systems : The importance of modeling extreme events and long-duration power outages." Iscience (2021) : 103630.

[73] R. D. Zimmerman, C. E. Murillo-Sánchez and R. J. Thomas, "MATPOWER : Steady-State Operations, Planning, and Analysis Tools for Power Systems Research and Education," in IEEE Transactions on Power Systems, vol. 26, no. 1, pp. 12-19, Feb. 2011, doi : 10.1109/TPWRS.2010.2051168.

[74] Pinson P, Baroche T, Moret F, Sousa T, Sorin E, You S. The emergence of consumer-centric electricity markets. Distribution & Utilization, 34(12), 27-31. 2017

[75] Martelli, M. Approche haut niveau pour l'accélération d'algorithmes pour des architectures hétérogènes CPU/GPU/FPGA. Application à la qualification des radars et des systèmes d'écoute électromagnétiqu. Calcul parallèle,distribué et partagé. Université Paris Saclay (COmUE), 2019

[76] S. Manavski and G. Valle, "CUDA Compatible GPU Cards as Efficient Hardware Accelerators for Smith-Waterman Sequence Alignment," BMC Bioinformatics, vol. 9, no. Suppl 2, p. S10, 2008.

[77] T. Preis, P. Virnau, W. Paul, and J. Schneider, "GPU Accelerated Monte Carlo Simulation of the 2D and 3D Ising Model," Journal of Computational Physics, vol. 228, no. 12, pp. 4468–4477, 2009.

[78] F. Xu and K. Mueller, "Real-Time 3D Computed Tomographic Reconstruction using Commodity Graphics Hardware," Physics in Medicine and Biology, vol. 52, pp. 3405–3419, 2007.

[79] Vincent Boulos. Adéquation Algorithme Architecture et modèle de programmation pour l'implémentation d'algorithmes de traitement du signal et de l'image sur cluster multi-GPU. Autre. Université de Grenoble, 2012. Français. ffNNT : 2012GRENT099ff. fftel-00876668

[80] Ginhac, D. (2008). Adéquation Algorithme architecture : Aspects logiciels, matériels et cognitifs (Doctoral dissertation, Université de Bourgogne).

[81] Dine, A. Localisation et cartographie simultanées par optimisation de graphe sur architectures hétérogènes pour l'embarqué (Doctoral dissertation, Université Paris Saclay (COmUE)).

[82] Imad El Bouazzaoui. Hardware Software Co-design of an Embedded RGB-D SLAM System. Robotics. Université Paris-Saclay, 2022. English.

[83] Yves SOREL. Adéquation Algorithme Architecture. Editorial. `https://www.google.com/url?sa=t&source=web&rct=j&opi=89978449&url=https://core.ac.uk/download/pdf/15487467.pdf&ved=2ahUKEwjKkZGN3q2HAxVjSaQEHcPWD4IQFnoECBIQAQ&usg=AOvVaw1NXOfdEvAnei5dVqGZQQ44` (Consulté le 17/07/2024)

[84] Lee, V. W., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A. D., ... & Dubey, P. (2010, June). Debunking the 100X GPU vs. CPU myth : an evaluation of throughput computing on CPU and GPU. In Proceedings of the 37th annual international symposium on Computer architecture (pp. 451-460).

[85] Memeti, S., Li, L., Pllana, S., Kołodziej, J., & Kessler, C. (2017, July). Benchmarking OpenCL, OpenACC, OpenMP, and CUDA : programming productivity, performance, and energy consumption. In Proceedings of the 2017 Workshop on Adaptive Resource Management and Scheduling for Cloud Computing (pp. 1-6).

[86] Alvarado F.L. . Computational complexity in power systems Power. IEEE Transactions on Power Apparatus and System, vol.95, no.4, pp. 1028- 1037, July1976doi : 10.1109/T-PAS.1976.32193

[87] Green Robert C, Wang Lingfeng, Alam Mansoor. Applications and trends of high performance computing for electric power systems : Focusing on smart grid. IEEE Transactions on Smart Grid. 2013

[88] Ullah, M. H., & Park, J. D.. Peer-to-peer energy trading in transactive markets considering physical network constraints. IEEE Transactions on Smart Grid. 2021

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

[90] Boyd, S., Parikh, N., Chu, E., Peleato, B., & Eckstein, J. (2011). Distributed optimization and statistical learning via the alternating direction method of multipliers. Foundations and Trends® in Machine learning, 3(1), 1-122.

[91] Houska, B., Frasch, J., & Diehl, M. (2016). An augmented Lagrangian based algorithm for distributed nonconvex optimization. SIAM Journal on Optimization, 26(2), 1101-1127

[92] Bagnara, R. (1995). A unified proof for the convergence of Jacobi and Gauss–Seidel methods. SIAM review, 37 (1), 93-97.

[93] Stellato B, Banjac G, Goulart P, Bemporad A, Boyd, S. OSQP : an operator splitting solver for quadratic programs. Mathematical Programming Computation `https://doi.org/10.1007/s12532-020-00179-2`

[94] Culler, D., Singh, J. P., & Gupta, A. (1999). Parallel computer architecture : a hardware/software approach. Gulf Professional Publishing.

[95] Cui, T., & Franchetti, F. (2012). Optimized parallel distribution load flow solver on commodity multi-core CPU. In 2012 IEEE Conference on High Performance Extreme Computing (pp. 1-6). IEEE.

[96] Jin, S., & Chassin, D. P. (2014, January). Thread Group Multithreading : Accelerating the Computation of an Agent-Based Power System Modeling and Simulation Tool–C GridLAB-D. In 2014 47th Hawaii International Conference on System Sciences (pp. 2536-2545). IEEE.

[97] Thurner, L., Scheidler, A., Schäfer, F., Menke, J. H., Dollichon, J., Meier, F., ... & Braun, M. (2018). pandapower—an open-source python tool for convenient modeling, analysis, and optimization of electric power systems. IEEE Transactions on Power Systems, 33(6), 6510-6521.

[98] Coffrin, C., Bent, R., Sundar, K., Ng, Y., & Lubin, M. (2018, June). Powermodels. jl : An open-source framework for exploring power flow formulations. In 2018 Power Systems Computation Conference (PSCC) (pp. 1-8). IEEE.

[99] Perkel, J. M. (2019). Julia : come for the syntax, stay for the speed. Nature, 572(7767), 141-142.

[100] Ramadhan, S., Hariadi, F. I., & Achmad, A. S. (2016, November). Development FPGA-based Phasor Measurement Unit (PMU) for smartgrid applications. In 2016 International Symposium on Electronics and Smart Devices (ISESD) (pp. 21-25). IEEE.

[101] Song, X., Wang, H., & Wang, L. (2014, April). FPGA implementation of a support vector machine based classification system and its potential application in smart grid. In 2014 11th International Conference on Information Technology : New Generations (pp. 397-402). IEEE.

[102] Miao, L., Wei, G., Fang, X., & Risheng, J. (2015, July). The strategy of the voltage control in smart grid based on modern control method and FPGA. In 2015 34th Chinese Control Conference (CCC) (pp. 8964-8968). IEEE.

[103] Ofenbeck G, Steinmann R, Caparros V, Spampinato D G, Püschel M. Applying the roofline model. 2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2014, pp. 76-85, doi : `https://doi.org/10.1109/ISPASS.2014.6844463`

[104] Konstantinidis E, Cotronis Y. A quantitative roofline model for GPU kernel performance estimation using micro-benchmarks and hardware metric profiling. Journal of Parallel and Distrind Computing. 2017. vol 107. p 37-56.

[105] Jensen T, Pinson P. RE-Europe, a large-scale dataset for modeling a highly renewable European electricity system. Sci Data 4, 170175 .2017. https://doi.org/10.1038/sdata.2017.175

[106] IEEE PES Distribution Systems Analysis Subcommittee Radial Test Feeders [Online], Available : http ://ewh.ieee.org/soc/pes/dsacom/testfeeders.html

[107] Schubiger M, Banjac G, and Lygeros J. GPU Acceleration of ADMM for Large-Scale Quadratic Programming

[108] 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).

[109] Harris M. NVIDIA Developer Technology. Optimizing Parallel Reduction in CUDA

Mes contributions :

[110] Thomas B, Le Goff Latimier R, A, Ben Ahmed H, Jodin G, El Ouardi Bouaziz Samir. Hardware-Software Codesign for Peer-to-Peer Energy Market Resolution. Available at SSRN : https://ssrn.com/abstract=4422414orhttp://dx.doi.org/10.2139/ssrn.4422414

[111] 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. International Conference on Electrical Sciences and Technologies in Maghreb (CISTEM), 2022

[112] Thomas B, El Ouardi A, Bouaziz Samir, Le Goff Latimier R, Ben Ahmed H. GPU Optimisation of an Endogenous Peer-to-Peer Market with Product Differentiation. PowerTech. IEEE, 2023

[113] Thomas B, El Ouardi A, Bouaziz Samir, Le Goff Latimier R, Ben Ahmed H. Adéquation Algorithme Architecture pour le calcul d'un AC-Power Flow. Symposium de Génie Electrique, 2023

[114] Thomas B, El Ouardi A, Bouaziz Samir, Le Goff Latimier R, Ben Ahmed H. A. Propriétés algorithmiques des simulations de Smartgrid sur CPU-GPU. JCGE, 2024

[115] Thomas B, El Ouardi A, Bouaziz Samir, Le Goff Latimier R, Ben Ahmed H. Acceleration of a decentralized radial AC-OPF on GPU. Innovative SmartGrid Technologies (ISGT). IEEE, 2024