Modélisation multi-échelles et calculs parallèles appliqués à la simulation de l'activité neuronale

Mathieu Bedez

École doctorale n° 269 : Mathématiques, Sciences de l'Information et de l'Ingénieur

Doctorat

THÈSE

pour obtenir le grade de docteur délivré par

l'Université de Haute-Alsace

Spécialité doctorale "Mathématiques Appliquées"

présentée et soutenue publiquement par

Mathieu BEDEZ

le 18 décembre 2015

Modélisation multi-échelles et calculs parallèles appliqués à la simulation de l'activité neuronale

Jury

M. Bernard Girau,	Professeur, Université Henri Poincaré Nancy	Président du jury
M. Kaber Sidi Mahmoud,	MCF-HDR, Université Pierre et Marie-Curie	Rapporteur
M. Nicolas Rougier,	HDR, INRIA Bordeaux-Sud Ouest	Rapporteur
M. Olivier Haeberlé,	Professeur, Université de Haute-Alsace	Directeur de thèse
M. Zakaria Belhachmi,	Professeur, Université de Haute-Alsace	Directeur de thèse
M. Serge Bischoff,	Professeur, Président de Rhenovia Pharma	Examinateur

Rhenovia Pharma, 20C Rue de Chemnitz, 68200 MULHOUSE
MIPS EA 2332, 12 rue des Frères Lumière, 68093 MULHOUSE
LMIA EA 3993, 4 rue des frères Lumière, 68093 MULHOUSE

THESE

Remerciements

Beaucoup de personnes ont été présentes et m'ont aidées durant ma thèse. J'espère n'oublier personne au cours de ces remerciements.

Tout d'abord, je tiens à remercier Serge Bischoff, président de Rhenovia Pharma pour la confiance qu'il m'a accordé en m'embauchant au sein de Rhenovia, mais surtout lorsqu'il m'a permis d'effectuer cette thèse. Je remercie également Michel Faupel, ancien vice-président de Rhenovia Pharma, pour son énergie, sa gentillesse et son inventivité. Mes remerciements vont également à Saliha Moussaoui qui m'a soutenu sans relâche lors des moments difficiles inhérents à la thèse. Mais je n'oublierai pas non plus toute l'équipe opérationnelle de Rhenovia Pharma qui a été le cœur de cette entreprise multidisciplinaire, dont la joie de vivre, l'abnégation et le soutien ont été très importants au cours de ces années, notamment dans ces derniers mois. Ainsi je remercie, Renaud Greget, Arnaud Legendre, Florent Laloue, Florence Keller, Merdan Sarmis et Nicolas Ambert.

Je tiens à remercier mes directeurs de thèse, Olivier Haeberlé et Zakaria Belachmi, pour leur disponibilité et leurs aides durant ces années. Ils m'ont été d'un aide précieuse dans l'encadrement de mon travail, mais également lorsque la situation économique de Rhenovia Pharma a mis en péril la fin de ma thèse. Sans leur présence, cette thèse n'aurait probablement pas aboutie.

Je remercie les membres du jury et plus précisément les rapporteurs, qui ont accepté de lire et de commenter mon travail.

Je remercie également mes amis dont la présence a été importante, notamment pour décompresser lors des phases de doutes et de labeurs.

Finalement, je tiens à remercier toute ma famille pour leurs sacrifices ainsi que la confiance qu'ils m'ont accordée. Tout cela m'a permis d'atteindre cet objectif, notamment dans les moments compliqués. Je leur en suis à jamais reconnaissant. Sans eux, rien de tout ça n'aurait été possible.

« La vie, c'est comme une bicyclette, il faut avancer pour ne pas perdre l'équilibre. »
Albert Einstein

Table des matières

Présentation de Rhenovia-Pharma

Rhenovia Pharma est une société de biotechnologie, fondée le 3 mai 2007 par une équipe internationale engagée dans la découverte de nouvelles solutions permettant de traiter les maladies du système nerveux, dont les plus connues sont la maladie d'Alzheimer et de Parkinson. La société a développé et mis en œuvre une technologie unique au monde permettant d'intégrer les données issues des neurosciences dans des plateformes de bio-simulation *In Silico*.

Les difficultés à répondre aux besoins médicaux dans le domaine du système nerveux, notamment pour la maladie d'Alzheimer, sont indiscutablement liées à l'extrême complexité de cette maladie, ainsi qu'aux aspects multifactoriels et dynamiques des processus physiopathologiques. C'est dans ce contexte que l'approche classique de découverte de médicaments utilisée par la plupart des entreprises pharmaceutiques est limitée pour résoudre les multiples facettes de ces maladies. Le processus de modélisation mathématique et intégratif permet la génération de modèles dynamiques intégrant des mécanismes détaillés. L'objectif principale de Rhenovia Pharma est d'utiliser cette technologie de rupture dans le domaine des neurosciences afin d'apporter de nouvelles contributions à la découverte de nouveaux médicaments ou d'associations de médicaments, permettant le traitement de pathologies du système nerveux.

Le travail mené par les équipes de Rhenovia Pharma consiste à développer une bibliothèque de modèles fonctionnels, aussi appelés modèles élémentaires, des réactions chimiques qui ont lieu dans les cellules nerveuses, et de les assembler afin d'obtenir un système décrivant de manière la plus réaliste les mécanismes de la propagation du signal électrique au sein d'une synapse, du neurone et de réseaux de neurones, et également au sein du cerveau tout entier. Les travaux menés jusqu'à maintenant ont permis de modéliser les mécanismes essentiels et de reconstituer le comportement des synapses glutamatergiques, GABAergiques et dopaminergiques. Rhenovia Pharma est aujourd'hui en mesure de définir les bases d'un modèle numérique décrivant la communication synaptique, la physiologie neuronale et le comportement de populations de neurones, pour mener des études fondamentales ou appliquées à la découverte de molécules, et ainsi permettre d'accélérer le processus de recherche et développement en pharmacologie.

L'idée directrice à la base de la création de Rhenovia Pharma est d'apporter des solutions inno-

vantes aux processus de recherche et développement de nouveaux produits pharmaceutiques, à l'aide d'une approche audacieuse qui transforme la complexité des systèmes biologiques en systèmes d'équations mathématiques.

L'objectif est d'augmenter la prédictibilité des tests effectués sur les animaux en phase clinique, et donc de réduire drastiquement le nombre d'animaux utilisés et par conséquent, diminuer le taux d'attrition des nouveaux candidats pharmacologiques en phase clinique. En effet, de nombreuses limitations chez les animaux, telles que la différence d'organisation du cerveau, l'absence de certains génotypes, les différences d'affinité et d'efficacité entre certains récepteurs chez le rat et l'humain et de possibles interférences pharmacodynamiques à l'issue de co-médication, expliquent en partie le taux d'échec élevé des expériences dans le processus de développement d'un nouveau médicament.

Le processus de découverte de nouveaux médicaments

Pour l'industrie du médicament, la capacité à mettre au point des molécules innovantes est le principal critère de réussite. Or, la recherche de nouveaux médicaments est un processus long et coûteux. La mise au point d'un nouveau médicament prend entre 15 à 20 ans et se décompose en trois étapes majeures : l'isolement de la molécule, l'étude préclinique et l'étude clinique (Figure 1).

a) Isolement de la molécule. Les laboratoires pharmaceutiques trouvent de nouvelles molécules à tester soit par la synthèse chimique (chimie combinatoire), soit par l'extraction à partir d'une source naturelle (le plus souvent végétale), soit par une méthode biotechnologique. Une fois ces nouvelles molécules obtenues, les laboratoires pharmaceutiques appliquent la méthode de recherche habituelle dite de criblage, ou « screening » en Anglais, qui consiste à tester toutes les molécules dont dispose le laboratoire sur toutes les cibles biologiques connues, l'objectif étant de repérer celles qui présentent des propriétés intéressantes, c'est-à-dire dénicher les principes actifs qui auront un effet curatif. Cette méthode du screening est une opération

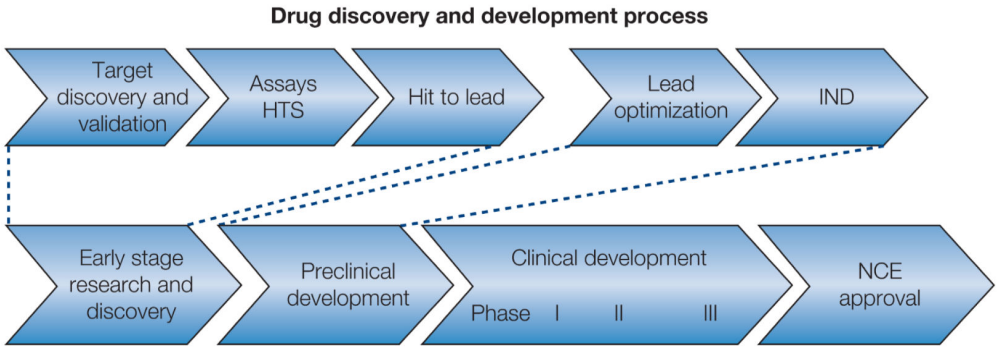

Figure 1 – Processus de recherche et de développement de nouveaux médicaments, d'après Brady *et al.* (2008).

de longue haleine, et son rendement est, comme on peut l'imaginer, faible. Ainsi, sur 100 000 molécules criblées, seules 100 molécules seront retenues pour les tests appelés « précliniques ». Cette phase de recherche est évaluée à une durée de deux à trois ans en moyenne.

b) Études précliniques. Il s'agit successivement d'identifier, de cribler, de sélectionner puis d'optimiser les quelques molécules qui présenteront le meilleur potentiel de développement aussi bien en terme d'efficacité qu'en terme de tolérance. Cette phase de recherche porte le nom d'études précliniques (Drug Discovery (DD)) et dure généralement deux à trois ans. À la fin de ce processus, la molécule sera admise à passer en étude chez l'homme. Les études précliniques sont conduites à l'aide de robots et de cultures de tissus et permettent ainsi de réduire considérablement l'utilisation d'animaux de laboratoire. À ce stade, il est vérifié que la molécule n'est pas cancérogène, mutagène ou tératogène et ne provoque pas d'autres effets toxiques inacceptables. Les procédés de fabrication industrielle sont mis au point à ce stade de développement. Cette phase s'effectue dans des unités de recherche (synthèse et extraction, screening et essais pharmacologiques, essais de toxicité et de sécurité, dosage, formulation et stabilité). Les études précliniques sont des passages obligés avant toute étape de test sur l'homme. En effet, à ce stade, on ne connaît ni le degré de tolérance du médicament, ni la façon dont le corps le transformera, ni les doses à préconiser, ni la voie d'admission (buccale, percutanée...) la mieux adaptée. Cette expérimentation a pour but d'acquérir des données d'efficacité et de toxicité de base, et, si les résultats n'annoncent pas une sécurité d'emploi fiable, d'abandonner son développement. Les tests précliniques sont de trois formes :

- La pharmacologie expérimentale : des essais d'efficacité sont réalisés sur des systèmes moléculaires inertes, sur des cellules et cultures et enfin sur l'animal. Le nouveau produit est identifié. Ces tests visent à définir le profil des composés sélectionnés lors du screening. Ils comprennent à la fois l'étude de pharmacodynamique (on teste le mode d'action de la molécule et la relation dose-effet) et l'étude pharmacocinétique (le devenir du produit dans l'organisme).
- La toxicologie : ces études évaluent les risques d'effets secondaires des futurs médicaments. Elles ont pour but de s'assurer de l'innocuité de la molécule et d'analyser ses effets toxiques sur l'organisme. Soumises aux dispositions de la directive européenne 75/318, elles prennent en compte la toxicité aiguë et chronique du produit. Elles sont réalisées sur deux espèces de mammifères : un rongeur et un non-rongeur. On s'intéresse à l'effet tératogène du produit : est-il susceptible d'engendrer des malformations sur l'embryon et le fœtus ? On analyse également son impact sur la reproduction ainsi que ses éventuels effets mutagènes : la molécule risque-t-elle de transformer le patrimoine génétique cellulaire de l'individu ? Enfin, on s'attarde sur l'aspect cancérogène du produit : risque-t-il de provoquer un cancer ?
- La pharmacocinétique et le métabolisme du médicament : ces études portent sur des propriétés pharmaceutiques de la molécule telles que l'absorption, le métabolisme, la distribution, l'élimination du « candidat-médicament » dans l'organisme. Mais elles permettent aussi de prouver les propriétés pharmacologiques.

Le développement préclinique s'étend généralement sur deux à trois années. Si les résultats

de ces études sont positifs, le médicament entre en phase d'essais cliniques sur l'homme. Les affaires du Stalinon, de la Thalidomide, du Distilbène et plus récemment du Mediator, médicaments dangereux, dont l'évaluation initiale et la surveillance n'avaient pas été assez strictes, ont fait prendre conscience des conséquences sanitaires possibles des carences ou des fautes des systèmes de santé publique. Des procédures de mise sur le marché, puis de pharmacovigilance ont ainsi été définies et mises en œuvre.

c) Développement et études cliniques. La phase de développement (drug development) correspond aux études cliniques chez l'homme. Elle dure huit ans en moyenne. Conformément aux accords d'Helsinski de 1975, les études cliniques ne peuvent être menées qu'avec le consentement éclairé et écrit des patients ou des volontaires sains. Les études dites de phase I permettent d'établir la dose thérapeutique maximale administrable chez l'homme. Ces études sont menées chez des volontaires sains. Elle permet d'évaluer les grandes lignes du profil de tolérance du produit (évaluer la sécurité d'emploi du produit, son devenir dans l'organisme, son seuil de tolérance ainsi que le déclenchement d'effets indésirables) et de son activité pharmacologique. Les études suivantes de phase II ont pour objectif de déterminer la dose thérapeutique et de démontrer l'effet thérapeutique attendu. Ces études sont aussi menées chez des malades qui ne répondent plus à aucun traitement (dans le cas des cancers par exemple). Il s'agit ici de définir la dose optimale, c'est-à-dire celle pour laquelle l'effet thérapeutique est le meilleur pour le moins d'effets secondaires. Les études de phase III incluent plusieurs milliers de patients pour vérifier statistiquement l'efficacité et la tolérance de la molécule étudiée en comparaison aux thérapeutiques de référence ou au placebo. Ceci est vérifié sur un grand groupe de malades. Les précautions d'emplois et risques d'interactions avec d'autres produits sont identifiés. Les essais peuvent couvrir plusieurs centaines à plusieurs milliers de patients. Ces essais sont souvent effectués dans plusieurs zones géographiques, afin de prendre en compte les caractéristiques de diverses populations. Les essais de phase IV ont lieu après que le médicament ait été mis sur le marché et permettent de tester son efficacité sur une population plus large, pour d'éventuelles extensions d'indications thérapeutiques. Parallèlement aux essais cliniques, le laboratoire étudie la phase d'industrialisation du candidat-médicament, sa forme pharmaceutique ainsi que son procédé de synthèse. Le laboratoire pharmaceutique réfléchit à la forme galénique du futur médicament : celle-ci doit être la mieux adaptée au traitement de la pathologie ciblée. Il opte pour un procédé de synthèse : la préparation des substances actives aura lieu soit par extraction à partir d'une matière naturelle, soit par synthèse chimique. Enfin, il doit se conformer aux BPF (bonnes pratiques de fabrication).

La figure 2 synthétise l'ensemble des processus nécessaires à l'élaboration et à la commercialisation d'un nouveau composé thérapeutique. Ce long processus prend entre 15 et 20 ans, ne laissant que 5 à 10 ans pour rentabiliser l'investissement de Recherche et Développement. De plus, il est extrêmement couteux dans la mesure où il faut répéter ce processus pour différentes molécules afin d'être sûr d'avoir une molécule qui arrive jusqu'à la phase de commercialisation.

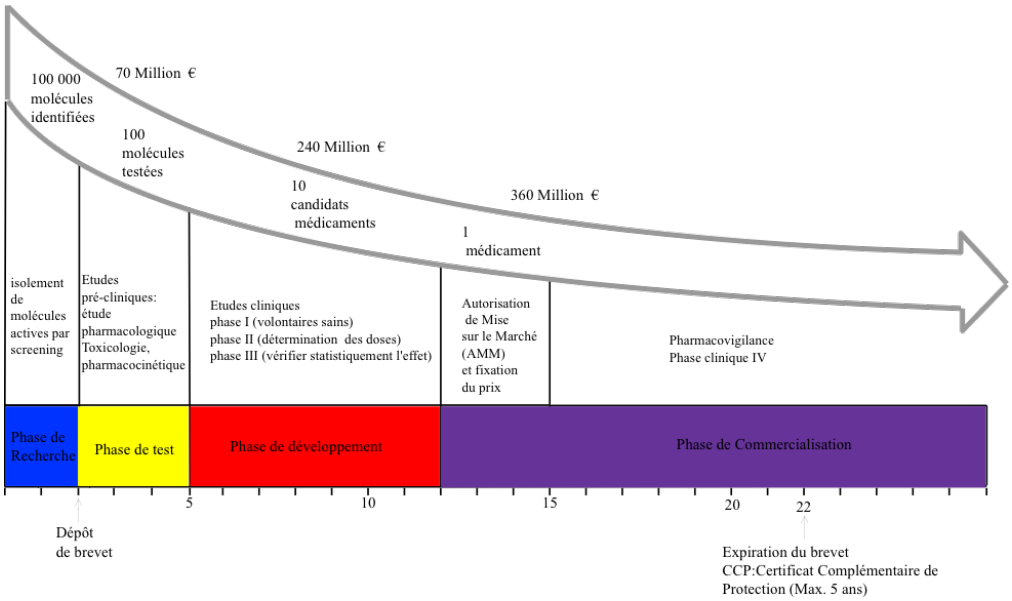

Figure 2 – Représentation schématique des différents stades nécessaires au développement d'un nouveau composé thérapeutique.

Des difficultés de l'innovation thérapeutique

Ce long processus prend entre 15 et 20 ans, ne laissant que 5 à 10 ans pour rentabiliser l'investissement de Recherche et Développement (R&D). Ce processus est aussi extrêmement risqué, et seule une petite fraction des molécules candidates obtiennent une autorisation de mise sur le marché. Ainsi, seulement 8% des molécules entrant en phase I sont finalement commercialisées (Munos, 2010; Grabowski, 2004; Grasela et Slusser, 2010; Kaitin, 2010; Robinson et Sethuraman, 2010). Ce taux d'attrition particulièrement élevé est dû principalement à des défaillances de la prédictibilité des modèles animaux d'une part et une compréhension incomplète de la physiologie et des pathologies du cerveau humain d'autre part. Le contexte économique actuel ne favorise pas ces aspects, et les médicaments doivent renouveler les preuves de leur efficacité, sous peine de ne pas faire l'objet de remboursement par l'assurance maladie.

Il y a un nombre grandissant d'évidences que la recherche et le développement de nouveaux candidats médicamenteux en général et tout particulièrement dans le domaine des maladies du SNC connaissent un taux de succès inadéquat par rapport aux moyens mis en œuvre Munos (2010). En effet, alors que le coût de R&D a doublé entre 1998 et 2008, le nombre de nouvelles molécules introduites sur le marché a diminué de plus de moitié, démontrant clairement l'efficacité limitée de la stratégie employée par les groupes pharmaceutiques (Figure 3). D'autre part, l'expiration rapide des brevets exposent leurs détenteurs à la commercialisation de médicaments génériques concurrents, et les obligent à trouver des solutions pour amortir leurs investissements. Il en résulte un désengagement d'une partie de l'industrie dans la

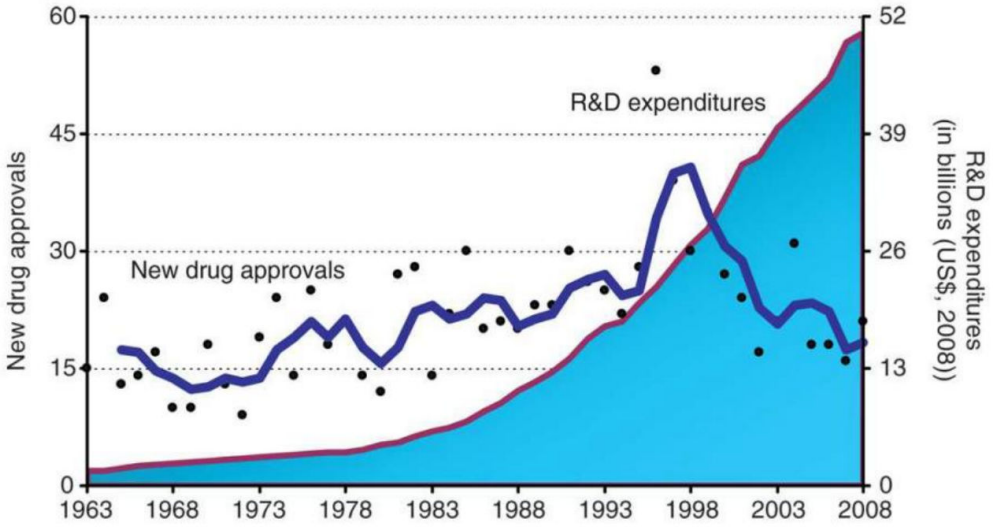

Figure 3 – Comparaison entre les dépenses R&D des groupes pharmaceutiques (courbe rouge) et le nombre de médicaments (points noirs) mis sur le marché entre 1993 et 2008, d'après Kaitin (2010).

recherche de molécules innovantes appliquées aux pathologies du système nerveux, et une diminution manifeste du nombre de médicaments mis sur le marché ces dernières années pour ces indications.

Comme le souligne par exemple (Brady *et al.*, 2008; Bennani, 2011), l'industrie pharmaceutique a un besoin urgent de mobiliser de nouvelles approche pour dynamiser son innovation. Les approches de pharmacologie quantitative (Allerheiligen, 2010), les techniques de modélisation et de simulation (Grasela et Slusser, 2010), et les nouvelles technologies (Robinson et Sethuraman, 2010) doivent être mises en œuvre tous azimuts afin de relever les nombreux défis scientifiques, industriels, économiques et sociaux que posent les maladies neurodégénératives (Brady *et al.*, 2008).

Introduction générale

La modélisation biologique, permettant de simuler au mieux l'activité de certains organes d'un être vivant, est une discipline relativement récente comparée aux modélisations physiques présentes dans la météorologie, l'aéronautique ou encore le nucléaire. Cette modélisation est plus complexe dans la mesure où il y a une dépendance entre les différentes structures ainsi qu'un caractère probabiliste provenant de la différence entre les individus.

Les neurosciences computationnelles ont permis de développer des outils mathématiques et informatiques permettant la création, puis la simulation de modèles représentant le comportement de certaines composantes de notre cerveau à l'échelle cellulaire. Ces derniers sont utiles dans la compréhension des interactions physiques et biochimiques entre les différents neurones, au lieu d'une reproduction fidèle des différentes fonctions cognitives comme dans les travaux sur l'intelligence artificielle. La construction de modèles décrivant le cerveau dans sa globalité, en utilisant une homogénéisation des données microscopiques est plus récent, car il faut prendre en compte la complexité géométrique des différentes structures constituant le cerveau. Il y a donc un long travail de reconstitution à effectuer pour parvenir à des simulations.

Ce travail de thèse a permis d'étudier différents modèles, décrivant l'activité électrique à l'échelle d'un neurone, puis à l'échelle d'un tissu neuronal. Lors de son activité, un champ électrique est créé à partir d'un potentiel d'action manifestant l'activité du neurone. Ce champ électrique se propage suivant les différentes directions spatiales, en fonction des caractéristiques des différents tissus constituant notre organe. L'activité électrique peut être mesurée à la surface du crâne grâce à un examen : l'électroencéphalogramme (EEG). L'utilité de cet examen est de pouvoir détecter différents dysfonctionnements au sein de cette activité électrique, découlant d'un problème neurologique tel que la maladie de Parkinson, ou encore l'épilepsie.

D'un point de vue mathématique, les différents modèles sont décrits à l'aide de systèmes d'équations différentielles ordinaires, et d'équations aux dérivées partielles. Le problème majeur de ces simulations vient du fait que le temps de résolution peut devenir très important, lorsque des précisions importantes sur les solutions sont requises sur les échelles temporelles mais également spatiales.

L'objet de cette étude est d'étudier les différents modèles décrivant l'activité électrique du cerveau, en utilisant des techniques innovantes de parallélisation des calculs, permettant ainsi de gagner du temps, tout en obtenant des résultats très précis. Quatre axes majeurs permettront de répondre à cette problématique : description des modèles, explication des outils de parallélisation, applications sur deux modèles macroscopiques.

Partie I

La première partie est consacrée à la compréhension générale du mode de fonctionnement du cerveau, ainsi que de son principal constituant : le neurone. Un neurone possède deux caractéristiques physiologiques : l'excitabilité, en d'autres termes, il est capable de répondre à une stimulation puis de convertir cette dernière en une impulsion nerveuse, et la conductivité, c'est-à-dire la capacité de transmettre une impulsion, qui va se propager d'un neurone à un autre. Il convient ainsi de comprendre les bases de cette cellule ainsi que son mode de fonctionnement en partie caractérisé par un potentiel d'action, manifestation de son activité électrique. Ce potentiel peut être déterminé à l'aide de descriptions purement mathématiques grâce à des systèmes d'équations différentielles ordinaires (EDO). Les différents modèles mathématiques seront ainsi présentés. J'aborderai ensuite la modélisation multi-échelles à l'aide de deux modèles décrivant l'activité électrique d'un tissu neuronal (et du cerveau dans sa globalité). A plus haute échelle, les neurones sont connectés et organisés en réseaux, permettant de délimiter un milieu intra-cellulaire et un milieu extra-cellulaire. Lorsque l'on monte encore d'une échelle, on doit prendre en compte les propriétés des tissus dans lequel le potentiel électrique va évoluer. Une description mathématique de ces modèles, à l'aide d'équations aux dérivées partielles (EDP) sera effectuée. Le modèle le plus complet se présente comme une EDP d'évolution de type réaction-diffusion couplée avec une EDP elliptique et un système d'EDO.

Partie II

La deuxième partie consiste à décrire les différentes méthodes de résolution servant à simuler les modèles présentés dans la première partie. En particulier, ces méthodes vont permettre d'intégrer les différents problèmes en utilisant l'aspect multi-coeurs des processeurs actuels, ainsi que les graphical processing units (GPU), de plus en plus utilisés en calcul scientifique. Je détaillerai un algorithme permettant de paralléliser un problème suivant l'échelle de temps : la méthode pararéel. Jusqu'ici, la technique pour décomposer le problème sur l'échelle temporelle n'a pas bénéficié du même effort que sur l'échelle spatiale, à cause du caractère séquentiel de cette partie du problème. La principale motivation de cette méthode est de pouvoir effectuer des simulations sur des intervalles de temps concrets, d'où le nom de la méthode : pararéel, mais également de pouvoir effectuer de très longues simulations, afin d'obtenir une solution de très bonne qualité. Plusieurs variantes de cet algorithme existent. Je donnerai ainsi un détail de ces différentes versions, puis j'effectuerai un choix sur l'un d'entre eux, pour la

suite de notre étude. Ensuite, je développerai les bases des méthodes de décomposition de domaines. Elles serviront notamment à paralléliser les problèmes sur l'échelle de l'espace, ce qui permettra de réduire le coût des calculs. Ces méthodes permettent de définir un problème global à partir de problèmes locaux plus petits. Le but est donc de pouvoir exécuter ces différents problèmes locaux en parallèle, afin d'accélérer la résolution du problème global. L'intérêt final est de pouvoir coupler ces deux méthodes afin de pouvoir obtenir des solutions de meilleures qualités en réduisant drastiquement le temps de calcul, mais également pouvoir définir des zones d'intérêts où une haute précision est recommandée.

Dans un deuxième temps, j'introduirai des aspects plus techniques sur les outils informatiques permettant de pouvoir implémenter de façon efficace les différentes méthodes présentées précédemment. La première technologie se nomme Message Passing Interface (MPI) permettant de tirer parti des multiples coeurs formant les processeurs actuels. Puis je poserai les bases nécessaires à la compréhension d'un développement sur GPU à l'aide de l'environnement CUDA.

Partie III

La troisième partie s'attache à utiliser toutes les techniques décrites précédemment, afin de simuler au mieux le modèle monodomaine, simplification du problème bidomaine décrivant la propagation du signal électrique au sein d'un tissu neuronal. Ce modèle est composé d'une EDP de type réaction-diffusion, ainsi que d'un système d'EDO.

Dans un premier temps, j'utiliserai une implémentation en Python de la méthode des différences finies afin d'intégrer l'EDP du modèle. A partir de cette formulation, j'appliquerai l'algorithme pararéel à l'aide de MPI sur ce modèle. Puis j'intégrerai le modèle monodomaine à l'aide de l'environnement CUDA, permettant de coupler les parallélisations sur les échelles de temps et spatiale, grâce à la parallélisation dynamique des nouveaux GPU. Ces différentes implémentations permettent de diminuer de façon importante les temps de simulation de ce modèle.

Dans un deuxième temps, la méthode des éléments finis sera utilisée, grâce à FreeFem++, afin de pouvoir simuler le modèle sur des géométries plus complexes. A partir de cette implémentation, j'appliquerai l'algorithme pararéel afin de diminuer le temps de calcul, et ainsi permettre des simulations sur des échelles de temps plus grandes. Puis, je détaillerai un algorithme prenant en compte la parallélisation en temps, mais également l'utilisation de la décomposition de domaine, permettant ainsi une méthode de résolution totalement parallèle du problème monodomaine. Enfin, j'appliquerai ces différentes techniques lors de la simulation de ce modèle sur une géométrie réaliste du cerveau, requérant une grande puissance de calcul.

Partie IV

La dernière partie de cette étude consiste à appliquer les méthodes, utilisées dans la partie III, lors de la simulation du modèle bidomaine. Il permet de décrire l'activité électrique d'un tissu neuronal de façon plus précise que le modèle monodomaine, en supposant l'existence des milieux intra et extra-cellulaire. Le modèle bidomaine est décrit grâce à une EDP de type réaction-diffusion, d'une EDP elliptique et d'un système d'EDO. Le problème à résoudre est ainsi beaucoup plus important puisque à chaque temps, il est indispensable de résoudre une équation elliptique. Ainsi, lorsque une haute précision en temps et en espace est souhaitée, il est nécessaire d'utiliser de nouvelles techniques de résolution permettant de diminuer le temps de calcul.

Je m'intéresserai, tout d'abord, à la résolution du modèle bidomaine à l'aide de la méthode des différences finies. Grâce à cette implémentation, j'appliquerai ensuite l'algorithme pararéel avec l'aide de la technologie MPI, parallélisant ainsi le problème suivant l'échelle de temps. Puis, une programmation GPU sera effectuée, afin de profiter pleinement de la parallélisation massive que permet cette technologie. A nouveau, une parallélisation de l'échelle spatiale sera couplée à la parallélisation de l'échelle de temps. Cette méthode permet une diminution importante du temps de calcul, tout en délivrant une solution précise qui demande énormément de temps lors d'une simulation séquentielle.

Dans un deuxième temps, j'utiliserai la méthode des éléments finis afin de pouvoir prendre en considération des géométries plus complexes. L'algorithme pararéel sera à nouveau utilisé, puis un couplage à l'aide de la décomposition de domaine permettra d'obtenir une résolution parallèle sur toutes les échelles. Cette technique permettra notamment de définir des zones d'intérêts sur la géométrie, sur lesquelles un maillage raffiné est souhaitable. Finalement cette méthode sera utilisée pour résoudre le problème bidomaine sur une géométrie complexe en trois dimensions du cerveau.

Finalement, ce travail va permettre l'application de méthodes de résolution afin de diminuer le temps nécessaire à la simulation de l'activité électrique neuronal, tout en permettant d'augmenter le niveau de précision des solutions obtenues. Le modèle décrivant cette activité permettra également à l'avenir, de pouvoir simuler de façon réaliste l'activité électrique cérébrale tel que l'électroencéphalogramme (EEG). Ce travail permet également d'introduire un peu plus l'utilisation des cartes graphiques (GPU) aux neurosciences computationnelles, permettant ainsi d'ouvrir encore le champ des possibilités à l'application de ces outils.

Partie I

De la biologie aux modèles

1 Du système nerveux au neurone

1.1 Le système nerveux central (CNS) : aperçu global

Le cerveau est certainement la structure biologique la plus complexe, mais également une source de fascination et d'inspiration pour l'homme, qui ne cesse de vouloir le reproduire sans y arriver, pour le moment. Nous allons présenter dans la suite un aperçu global de cette structure, afin d'aider le lecteur à mieux comprendre le domaine d'étude du sujet. Il est évident qu'il serait impossible de décrire dans ce manuscrit les différentes caractéristiques de la neurophysiologie humaine en détails. Il existe néanmoins plusieurs ouvrages de références vers lesquels le lecteur pourra se tourner, s'il souhaite approfondir ses connaissances, et assouvir sa curiosité (Hasboun, 2004; Purves et Augustine, 2001; Kandel, 2012).

Le système nerveux est le centre de contrôle de l'organisme, c'est lui qui permet de garantir l'ensemble des fonctions tels que : percevoir l'environnement, traiter puis stocker ces informations sensorielles, puis produire et délivrer une réponse adaptée (Purves et Augustine, 2001). Fonctionnellement, on peut le diviser en deux grandes parties : le système nerveux somatique, associé au contrôle volontaire et le système nerveux autonome (ou végétatif) qui régule les fonctions vitales.

A l'échelle microscopique, le système nerveux est composé de deux grands types cellulaires (voir Figure 1.1) :

1. **Les neurones**, qui sont des cellules excitables qui propagent l'information sous forme de signal électrique.
2. **Les cellules gliales**, qui sont divisées en plusieurs sous-groupes, parmi lesquels on trouve les astrocytes, les oligodendrocytes, les cellules de Schwann, les cellules microgliales et les cellules épendymaires.

Le système nerveux central est constitué de sept régions principales (voir Figure 1.2) : le télencéphale, le diencéphale, le mésencéphale, le cervelet, le pont, le bulbe rachidien (moelle allongée ou medulla oblon) et la moelle épinière. Chacune de ces régions possède une struc-

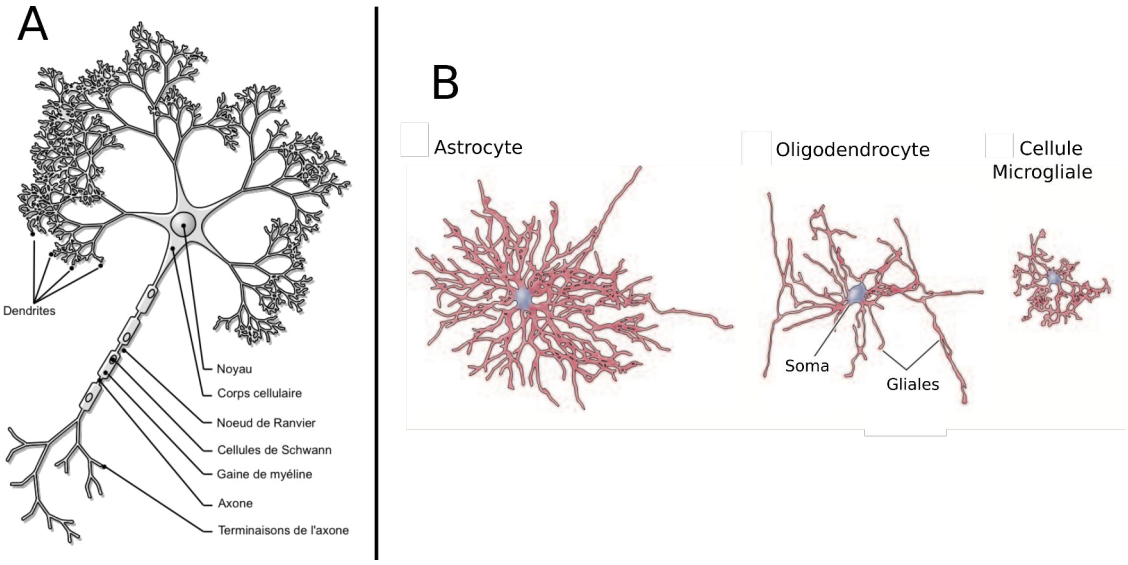

Figure 1.1 – A : Représentation schématique d'un neurone. **B** : Représentation schématique de différentes variétés de cellules gliales.

ture interne, et certaines d'entres elles sont formées de couches formant des replis sinueux (cortex cérébral, cervelet). Elles ont également leurs propres rôles, qui sont encore à l'heure actuelle soumis à discussion.

1. **Le télencéphale** est formé du cortex cérébral, la couche circonvoluée la plus externe, et de trois autres structures plus internes : les noyaux de base (striatum, pallidum, noyau sous-thalamique et substance noire), l'hippocampe et les noyaux amygdaliens. Les noyaux de la base participent au contrôle moteur, l'hippocampe assure l'encodage de la mémoire, et les noyaux amygdaliens coordonnent les réponses autonomes et endocrines associées aux états émotionnels.
2. **Le diencéphale** est composé de deux structures importantes : le thalamus, qui traite la majorité des informations voyageant entre le cortex et le reste du système nerveux central, et l'hypothalamus, qui régule les fonctions autonomes, endocrines et viscérales.
3. **Le mésencéphale** contrôle des fonctions sensorielles et motrices, comme le mouvement des yeux et la coordination des réflexes visuels et auditifs.
4. **Le cervelet** connecté au tronc cérébral (bulbe rachidien, pont et mésencéphale), module la précision des mouvements.
5. **Le pont** véhicule des informations sur le mouvement des hémisphères cérébraux vers le cervelet.
6. **Le bulbe rachidien** (ou moelle allongée) comprend différents centres de contrôle de fonctions autonomes, comme la digestion, la respiration, et le rythme cardiaque.
7. **La moelle épinière** recueille et traite les informations sensorielles, puis envoie les informations motrices vers les différents organes.

Ces différents mécanismes combinés permettent d'expliquer la mémoire, les mouvements, et possiblement la conscience.

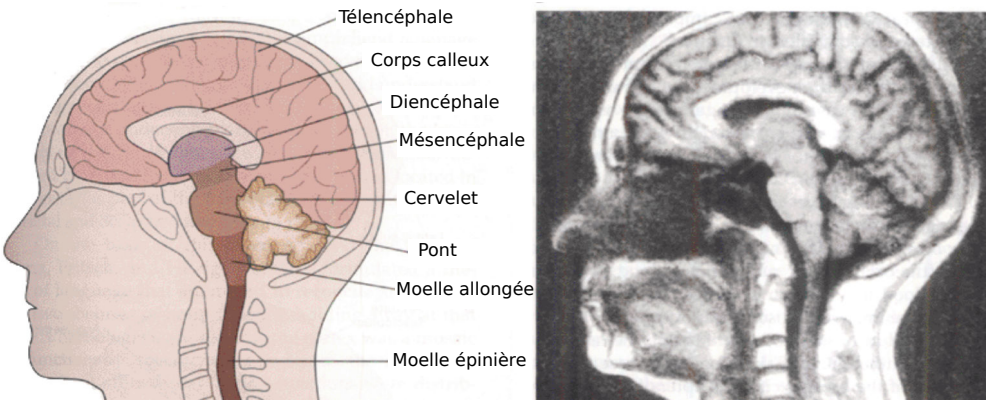

Figure 1.2 – Illustration des principales subdivisions anatomiques du système nerveux central d'après Kandel (2012).

Les hémisphères cérébraux sont les deux parties gauche et droite, symétriques du cerveau. Ces zones sont elles mêmes divisées en cinq lobes (voir Figure 1.3), séparés par des sillons, et communiquant entre eux par des commissures (faisceaux d'axones) et des corps calleux. Ces différents lobes sont souvent subdivisés en aires qui sont délimitées par des limites anatomiques ou fonctionnelles telles que les sillons, les fissures, ou encore le gyrus. En général, ces lobes sont associés aux fonctions suivantes :

1. **Le lobe frontal** est associé à la pensée rationnelle, à la résolution de problème, à la planification, à certains aspects du langage (aire de Broca), au mouvement (cortex moteur) et au comportement émotionnel.

2. **Le lobe occipital** est associé à la vision.

3. **Le lobe pariétal** permet la perception liée aux stimuli de température, de pression, de toucher et de douleur.

4. **Le lobe temporal** est impliqué dans la perception et la reconnaissance de stimuli auditifs, mais est également associé aux processus d'apprentissage et de mémoire.

5. **Le lobe insulaire** est certainement une des aires les moins connues. Actuellement, son rôle est surtout associé dans la dépendance, le dégoût et la conscience de soi.

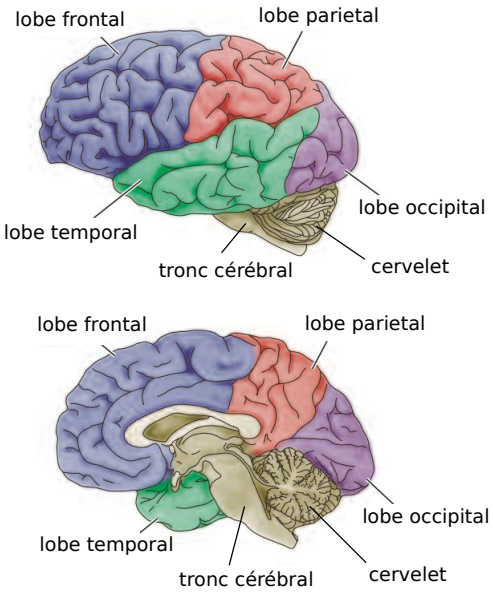

Figure 1.3 – Illustration des lobes des hémisphères cérébraux d'après Purves et Augustine (2001) (vue latérale de l'hémisphère gauche et vue d'une coupe).

1.2 Le neurone

1.2.1 Introduction

Le cerveau possède une forte concentration de neurones, estimée à environ 100 milliards, mais ce n'est pas la seule composante de notre organisme qui en possède, par exemple, l'intestin en possède environ 200 millions, selon de récentes estimations. Chaque neurone peut recevoir des stimuli d'environs 1 000 à 100 000 autres neurones voisins (Nunez et Srinivasan, 2006). Cette cellule est, en outre, à la base du fonctionnement de notre système nerveux.

Un neurone possède deux caractéristiques physiologiques : l'excitabilité, en d'autres termes, il est capable de répondre à une stimulation, puis de convertir cette dernière en une impulsion nerveuse, et la conductivité, c'est-à-dire la capacité de transmettre une impulsion.

Les neurones ont une grande diversité, d'où l'intérêt de les ranger en famille suivant leur morphologie, leur localisation dans le système nerveux, et leur fonction. Une représentation illustrant les différentes morphologies est visible Figure 1.4.

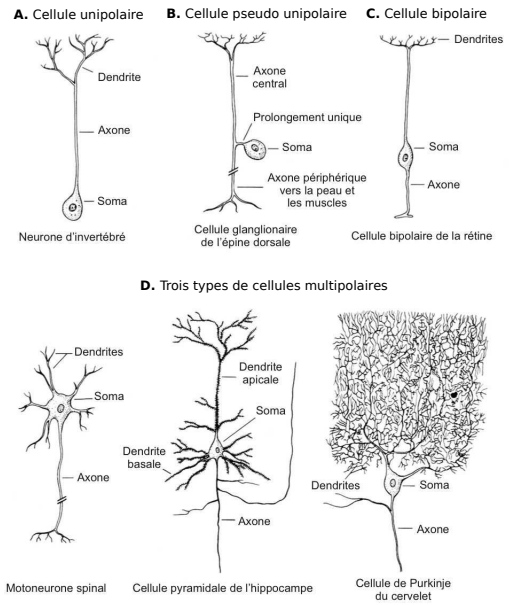

Figure 1.4 – Représentation schématique illustrant les différentes morphologies d'un neurone d'après Kandel (2012).

Néanmoins, parmi toutes les morphologies existantes, on retrouve toujours une base architecturale commune, composée : d'un corps cellulaire, aussi appelé soma ou péricaryon, d'un axone, ainsi que de(s) dendrite(s). La différenciation entre les morphologies se base sur la taille et la forme du soma et le nombre, la répartition et la longueur des dendrites. Dans la

suite, nous allons expliciter le rôle de chacun de ses constituants.

1.2.2 Le soma et la membrane plasmique

Le soma (ou péricaryon) peut avoir, en fonction du type de cellule, différentes formes. Son diamètre est en général de l'ordre de 20 μm avec une épaisseur de membrane d'environ 5 nm. Le corps cellulaire intègre les différents signaux provenant des dendrites, pour ensuite produire une réponse adaptée qui se propagera le long de l'axone. Il est à noter que ce corps cellulaire peut présenter à sa surface des synapses.

Le soma est constitué notamment d'une membrane plasmique séparant la partie intérieure, le cytoplasme, et la partie extérieure de la cellule. Cette membrane permet le contrôle des échanges entre la cellule et son milieu extérieur. Elle est composée d'une double couche de molécules, les phospholipides, une couche interne et une couche externe apposées symétriquement. Cette couche bilipidique est également traversée par différents types de protéines, ce qui permet d'assurer le flux de substances à travers la membrane. Chaque protéine est spécialisée dans le transport d'un élément bien précis. Les milieux intra et extra-cellulaire possèdent les mêmes espèces ioniques, mais avec des concentrations distinctes. Par exemple, le milieu extra-cellulaire est plus riche en ions sodium Na^+, mais au contraire est plus pauvre en ions potassium K^+ que le milieu intra-cellulaire (Figure 1.5). Ces différences de concentrations entraînent une différence de potentiel entre les deux milieux, que l'on appelle potentiel transmembranaire. Cette différence de potentiel est en général négative, bien qu'elle puisse être variable au cours du temps. On l'appelle potentiel de repos.

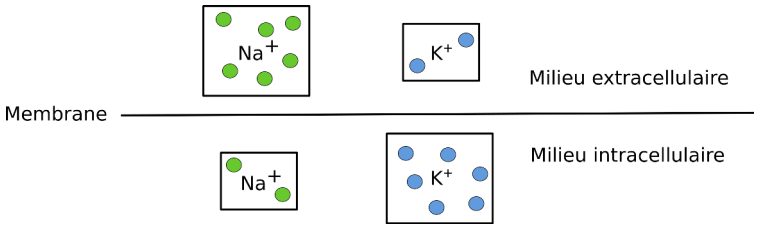

Figure 1.5 – Schéma de la membrane, montrant le différentiel de concentration entre deux espèces ioniques.

Bien entendu, il existe d'autres types d'ions, comme les ions calciques Ca^{2+}, dont les différences de concentrations sont souvent proportionnelles aux changements de potentiel du neurone, ce qui peut-être utilisé comme marqueur de l'activité électrique du neurone.

1.2.3 L'axone

L'axone est le prolongement long et cylindrique du soma. Son diamètre est inférieur à celui d'une dendrite, alors que sa longueur peut atteindre l'ordre du mètre chez l'homme. Sa

fonction principale est de propager le signal électrique du corps cellulaire vers les zones synaptiques. Ce signal est formé de potentiels d'action, dont la genèse se produit au sein du segment initial de l'axone. Il peut être entouré d'une gaine de myéline qui permet l'accélération de la propagation de ce signal, en améliorant au passage ses propriétés électriques. On estime qu'un tiers des axones est recouvert de myéline. Cette gaine est synthétisée par les cellules de Schwann au sein du système nerveux périphérique. On trouve des points, le long de l'axone, où cette gaine s'interrompt : les noeuds de Ranvier, qui permettent de régénérer le signal électrique.

1.2.4 Les dendrites

Les dendrites sont des ramifications provenant du corps cellulaire et permettant les connexions afférentes au neurone. Elles peuvent se diviser par dichotomie successives, ce qui leur confère une apparence arborescente. La communication entre les différents neurones est rendue possible grâce à ces dendrites. Elles transmettent les informations générées au niveau des synapses pour les transférer vers le soma. Par rapport à un axone, leurs contours sont plus irréguliers, et leur diamètre diminue lors de l'éloignement du soma. Contrairement à l'axone, les dendrites ne sont jamais recouvertes de myéline.

1.2.5 Les synapses

La synapse n'est pas à proprement parler un élément d'un neurone en particulier. Elle désigne plutôt la zone de contact fonctionnelle qui s'établit entre deux neurones, ou entre un neurone et une autre cellule (récepteurs sensoriels, cellules musculaires, etc.). C'est elle qui va traduire le potentiel d'action venant du neurone présynaptique, en signal dans la partie postsynaptique. Il a été estimé qu'environ 40 % de la surface membranaire peut être recouverte de synapses, dans certains types cellulaires, telles que les cellules pyramidales ou les cellules de Purkinje.

Les synapses sont scindées en deux grandes familles :

- **La synapse électrique** est la moins répandue. Le signal est transmit électriquement par l'intermédiaire d'une jonction communicante (ou gap-junction en anglais) (Figure 1.6 A)
- **La synapse chimique** se trouve en très grand nombre. Elle utilise des neurotransmetteurs pour diffuser l'information envoyée par la partie présynaptique. Ainsi, elle traduit le signal électrique présynaptique en message chimique se diffusant dans la synapse, pour finalement être à nouveau traduit en signal électrique dans la partie postsynaptique (Figure 1.6 B).

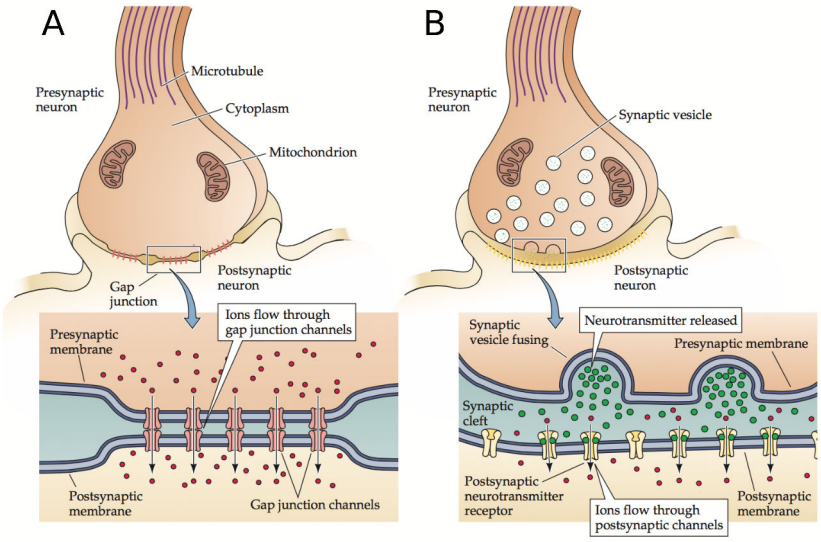

Figure 1.6 – A : Représentation schématique d'une synapse électrique. Les GAP junctions permettent aux ions de passer de la partie présynaptique au postsynaptique. Cette entrée d'ion dans la partie postsynaptique entraîne le déclenchement d'un potentiel d'action. **B** : Représentation schématique d'une synapse chimique. Contrairement à la synapse électrique, aucune continuité est assurée pour le passage des ions. Lors de l'arrivée d'un potentiel d'action, des neurotransmetteurs sont libérés pour aller se lier à des récepteurs, laissant ainsi passer les ions lors de leur ouverture.

1.2.6 Activité neuronale

L'électroencéphalographie (EEG) permet de mesurer l'activité électrique du cerveau, grâce à des électrodes placées à des endroits bien spécifiques du crâne. Cette technique permet de détecter et de quantifier les activités rythmiques provenant aussi bien d'un réseaux de neurones, que d'une structure du cerveau. L'activité des neurones étant oscillatoire, une classification a pu être établie suivant la fréquence des oscillations (Taleau 1.1).

Bandes de fréquences	Fréquences (Hz)
δ	1 - 4
θ	4 - 8
α	8 - 12
β	12 - 30
γ	30 - 80
γ rapides	80 - 200
γ ultrarapides	200 - 600

Tableau 1.1 – Classifications de l'activité oscillatoire des neurones.

Par exemple, les fonctions motrices sont générées par des oscillations ayant une bande de

fréquence δ, γ, β et γ. Une bande de fréquence γ révèle une activité de la perception et de l'attention, alors que des fréquences dans la bande β apparaissent lors de la concentration ou également de l'anxiété. Pour plus d'informations sur le lien entre les activités oscillatoires des neurones et les fonctions motrices, on invite le lecteur à consulter des références telles que Courtemanche *et al.* (2003) et Dostrovsky et Bergman (2004).

L'EEG permet notamment de détecter une activité oscillatoire anormale résultante de maladies neurologiques telles que la maladies de Parkinson, ou l'épilepsie. Cette dernière est certainement la pathologie la plus étudiée par l'EEG.

1.2.7 Conclusion

Nous venons de donner le détail des différents constituants d'un neurone, cellule à la base de notre système nerveux. Dans le chapitre suivant, nous allons tenter d'expliquer le fonctionnement de cette cellule si particulière. Ainsi, nous verrons les différentes étapes, ou physiologie, conduisant un neurone à une phase dite de dépolarisation, puis d'un retour progressif à son état d'origine.

1.3 Physiologie du neurone

1.3.1 Introduction

Les éléments : potentiel de repos et potentiel d'action ont été introduits dans le chapitre précédent, lorsque nous avons détaillé l'anatomie d'un neurone. Ces concepts sont très importants à comprendre, car ce sont sur ces caractéristiques que les dynamiques engendrées par les modèles mathématiques sont obtenues. Nous allons rapidement expliquer le fonctionnement physiologique d'un neurone, étape essentielle pour la suite.

1.3.2 Potentiel de repos

Le potentiel de repos représente le potentiel de la membrane lorsque celle-ci n'est soumise à aucune excitation. Autrement dit, le système constitué des milieux interne et externe de la cellule est à l'équilibre. Malgré cet équilibre, il existe une différence de potentiel résultant de l'inégalité de concentration des ions, de part et d'autre de la membrane. Cette différence de potentiel s'appelle le potentiel de repos, qui est en général de l'ordre de -65 mV.

Le transfert au niveau de la couche bilipidique s'effectue notamment grâce à un mécanisme de pompage qui repose sur les effets antagonistes de la pompe sodium-potassium. Les ions potassium sont majoritaires dans le milieu intra-cellulaire, alors que les ions sodium sont minoritaires dans le milieu externe de la cellule. L'ouverture des canaux potassium permet au gradient chimique du potassium de se dissiper. La membrane, au repos, est plus perméable aux ions potassium.

1.3.3 Potentiel d'action

Le potentiel d'action, encore appelé influx nerveux, correspond au signal électrique se propageant le long de l'axone, à la suite d'une excitation de la membrane cellulaire. Cette stimulation permet à la membrane de devenir extrêmement perméable aux ions sodium Na^+.

Le potentiel d'action se décompose en plusieurs étapes décrites ci-après et représentées Figure 1.7 :

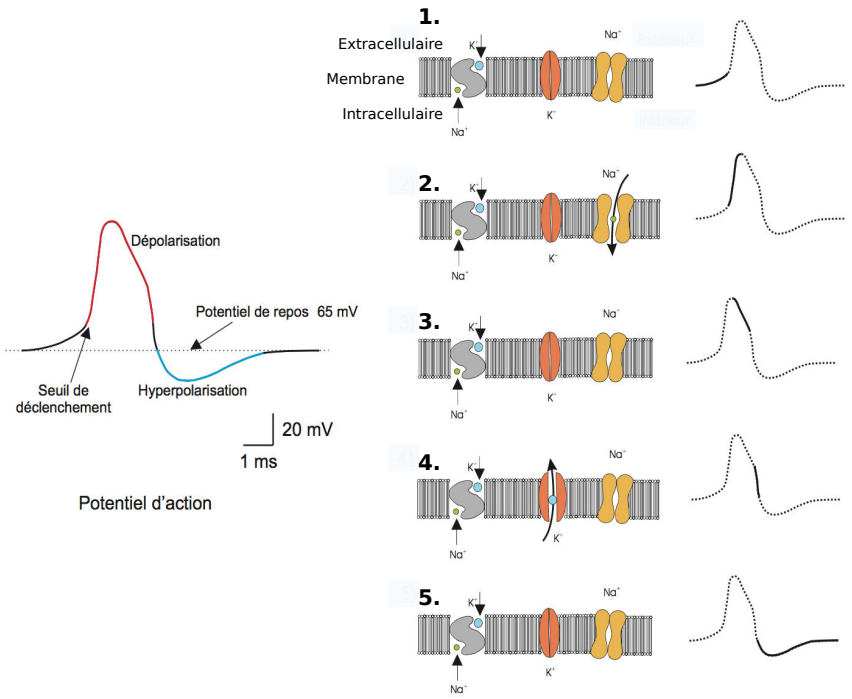

Figure 1.7 – Les différentes étapes de la création d'un potentiel d'action comme présentées dans Buhry (2010).

1. Dépolarisation de la membrane suite à une stimulation.
2. Si la stimulation continue, la dépolarisation se poursuit, jusqu'à atteindre le seuil de déclenchement du potentiel d'action. Cela entraîne une ouverture extrêmement rapide des canaux perméables aux ions sodium, et donc une dépolarisation encore plus rapide. Alors que la stimulation peut s'arrêter, le passage des ions sodium continue si le seuil a été franchi.
3. Lorsque la tension de membrane atteint la valeur de 58 mV, les canaux sodiques se ferment spontanément, ce qui entraîne l'arrêt de l'augmentation de la dépolarisation.
4. Un autre mécanisme intervient : l'ouverture des canaux potassiques, entraînant la repolarisation de la membrane cellulaire. Cette période est encore appelé « période réfractaire absolue », car aucun second potentiel d'action ne peut intervenir durant cette période, à cause de l'inactivation des canaux sodiques.
5. Une hyperpolarisation transitoire (ou période réfractaire relative) de la membrane est provoquée par le délai de fermeture des canaux potassiques. La membrane retrouvera son potentiel de repos après la fermeture des canaux potassiques.

Ce potentiel d'action est généré dans le soma, puis propagé vers les autres neurones à l'aide de l'axone, et ceci sans atténuation grâce à la gaine de myéline et aux noeuds de Ranvier.

1.3.4 Conclusion

Le fonctionnement d'un neurone est ainsi analysé grâce à la dynamique de son potentiel d'action. Cet aspect est très important, car les modèles mathématiques décrivant son activité ont comme variable, ce potentiel. Pour faire simple, c'est lui qui est la variable de nos problèmes, car c'est sa dynamique qui permet de définir l'état global d'un neurone, voir d'un réseau de neurones. Dans la suite, les différents modèles mathématiques de neurone vont être introduits, du plus simple au plus complexe, pouvant ainsi traiter différents problèmes, à diverses échelles d'espace.

2 Modèles mathématique des systèmes biologiques

2.1 Modèles biophysiques du neurone

2.1.1 Introduction

Maintenant que les bases biologiques du neurone, constituant de base du système nerveux central, ont été posées, nous allons dans cette partie décrire les modèles mathématiques permettant de modéliser l'activité électrique d'un neurone biologique. Bien que la contribution de processus thermodynamiques soit effectivement démontrée (Destexhe et Huguenard, 2000), les modèles actuels ne prennent pas en compte cette influence. Suivant la complexité du problème étudié, que cela soit dans l'étude d'un neurone ou d'un réseau de neurones, plusieurs choix de modèles sont disponibles. Bien entendu, plus le modèle décrira finement l'activité d'un neurone, plus le besoin en ressources de calculs devra être important et par conséquent, le temps de simulation sera bien plus long que le temps réel biologique. Il est donc évident que le modèle de neurone choisi lors de l'intégration d'un réseau de neurones ne sera pas le même lorsque l'on souhaitera étudié l'activité d'un neurone en particulier. Différents types de modèles de neurones ont été développés au cours du temps. On citera notamment les modèles biophysiques qui sont des modèles très détaillés permettant une modélisation précise des différents processus biologiques, lors de l'activité d'un neurone.

D'autres modèles, tels que les modèles connexionnistes ont également été développés, afin de résoudre des problèmes d'apprentissages automatiques (ou encore machine learning en anglais) à l'aide de modèles simplifiés. Au sein de ces modèles, on trouve les perceptrons multi-couches (Rosenblatt, 1958; Malsburg, 1986) ou encore les réseaux de Hopfield (MacKay, 2003). Ainsi ces modèles sont capables de s'adapter suivant le problème étudié, en modifiant les différents poids de connexions entre les différents neurones au cours de l'apprentissage. Ces modèles sont très utilisés pour la reconnaissance et la classification d'images, la reconnaissance de la parole, l'approximation de fonction (régression). Ces problèmes étant très éloignés des travaux présentés ici, ces modèles ne seront pas présentés en détail ici.

Les champs de neurones (Beurle, 1956; Wilson et Cowan, 1973; Taylor, 1999) font également

partis de cette famille de modèles permettant de faire une description de l'activité neuronale. Ce sont des macro-modèles décrivant, entre autre, l'évolution spatio-temporelle du taux de décharge moyen d'une population de neurones. Ils permettent de déterminer le champ neuronal représentant l'activité électrique d'une population de neurone à une localisation bien précise. Le force de connexion entre les neurones est prise en compte, à l'aide d'une fonction décrivant l'empreinte synaptique. Bien évidemment, cette dernière est l'objet de beaucoup d'attention, car c'est elle qui rythme l'activité de se réseau (Ben-Yishai *et al.*, 1995; Laing et Troy, 2003). Le modèle du champ de neurones le plus simple, peut être étendu afin de pouvoir décrire plusieurs populations de neurones, ou encore la neuromodulation. Par exemple, Faugeras *et al.* (2009) a donné différentes variantes de ces modèles, en prenant en compte les caractéristiques statistiques et stochastiques des neurones. Cette approche permet de combler le fossé entre l'échelle microscopique, où le neurone est considéré comme isolé et décrit grâce à un système d'équations différentielles stochastiques. A l'échelle mésoscopique, l'interaction entre les populations de neurones sont décrits par équations similaires. Ce pan de la modélisation bien que très intéressant ne rentre pas non plus dans le cadre de cette étude. En effet, il peut être intéressant d'étendre les modèles de neurones, comme celui de Hodgkin-Huxley, en prenant en compte de nouveaux courants ioniques décrits par des modèles synaptiques développés au sein de Rhenovia Pharma. Ainsi l'approche stochastique n'est pas abordé dans cette étude afin de se concentrer sur une observation moyenne de l'activité neuronale, évitant ainsi des temps de calculs trop importants induits par une modélisation probabiliste.

2.1.2 Modèle Integrate-and-Fire (IF)

Le modèle "Integrate-and-Fire" est la toute première description mathématique d'un neurone biologique. Il fut introduit par un médecin et physiologiste français, Louis Lapicque, en 1907, qui s'inspira d'un modèle de charge et décharge d'un condensateur à travers une résistance (Abbott, 1999), ce qui est traduit par l'équation suivante :

$$I(t) = C_m \frac{dV_m}{dt} \quad , \tag{2.1}$$

où C_m représente la capacité de la membrane, V_m le potentiel de la membrane et I le courant traversant la membrane. Le caractère simpliste de ce modèle ne permet pas l'apparition d'un potentiel d'action qui doit être introduit sous la forme d'un Dirac. Une impulsion est appliquée à chaque fois que le potentiel de membrane atteint un certain seuil, dont la valeur est le plus souvent fixé à V_s = -50 mV. Après chaque émission, la valeur du potentiel est ramenée à sa valeur de repos, à savoir -65 mV. Une période réfractaire peut-être ajoutée, ce qui limite notamment la fréquence d'apparition des potentiels d'actions, lorsque le neurone se trouve dans des conditions où l'apparition d'un nouveau potentiel d'action est possible.

2.1.3 Modèle Leaky-Integrate-and-Fire (LIF)

Un deuxième modèle faisant suite à "Integrate-and-Fire" fut proposé afin que le potentiel puisse retrouver progressivement sa valeur de repos. Un terme de fuite est ainsi ajouté, ce qui permet de décrire la diffusion d'ions à travers la membrane, lorsqu'un équilibre est atteint dans la cellule. L'équation (2.1) devient :

$$I(t) - \frac{V_m}{R_m} = C_m \frac{dV_m}{dt} \quad , \tag{2.2}$$

où R_m est la résistance de la membrane. Pour qu'il y ait l'apparition d'un potentiel d'action, le courant entrant doit dépasser le seuil défini par $I_s = V_s/R_s$. L'équation peut donc également s'écrire :

$$C_m \frac{dV_m}{dt} = I(t) - g_{fuite}(V_m - E_{fuite}) \quad , \tag{2.3}$$

où g_{fuite} est la constante représentant la conductance du courant de fuite, et E_{fuite} est potentiel de repos du courant de fuite. Etant donné que ces deux équations ont peu de caractéristiques à ajuster (le seuil V_s et la fréquence d'oscillation), ces dernières sont principalement utilisées lors de la simulation de très grands réseaux de neurones, comme des structures cérébrales où uniquement l'activité globale du système est importante.

2.1.4 Modèle de Izhikevitch

Le modèle de Izhikevitch est une extension du modèle Integrate-And-Fire (Izhikevich, 2004; Izhikevich *et al.*, 2004), sensé être plus réaliste. Pour définir son modèle, il utilise les méthodes de bifurcation pour pouvoir réduire la biophysique du modèle de Hodgkin-Huxley présenté dans la section suivante. Pour cela, il fait intervenir une expression quadratique, ainsi que le courant potassique à faible seuil. Ce modèle est décrit par le système d'équations différentielles suivant :

$$\frac{dv(t)}{dt} = 140 + 0.04v(t)^2 + 5v(t) - u(t) \quad , \tag{2.4}$$

$$\frac{du(t)}{dt} = a.(b.v(t) - u(t)) \quad , \tag{2.5}$$

avec des conditions initiales sur v et u, et la condition de mise à jour des variables, lorsque la valeur de la tension dépasse un certain seuil :

$$\text{si } v = 1 \text{ alors } v \leftarrow c,\ u \leftarrow u + d \quad ,$$

où v représente une tension, u un courant et a, b, c, d sont des constantes différentes suivant les caractéristiques du neurone étudié.

Bien que la résolution de ce modèle soit aussi simple qu'un modèle "Integrate-And-Fire", car ce système contient uniquement deux équations, il fournit des comportements plus réalistes que ce dernier. Comme pour les modèles précédents, il est surtout utilisé dans la simulation de très grands réseaux de neurones, ou des structures cérébrales comme celles impliquées dans la maladie de Parkinson.

2.1.5 Modèle de Hodgkin-Huxley

2.1.5.1 Equation globale

Le modèle d'Hodgkin-Huxley est certainement le modèle mathématique le plus connu (Hodgkin et Huxley, 1952a,b,c,d,e) permettant de décrire la genèse, puis la propagation d'un potentiel d'action, au sein d'un neurone. Ce dernier est formé par un ensemble d'équations différentielles ordinaires (voir section 3.1.2) décrivant les caractéristiques électrique des cellules excitables. A la différence des modèles précédents qui étaient plus abstraits, ce modèle utilise une description plus orienté « biophysique », à l'échelle des sous-éléments constituant la cellule. Les différents éléments constituant la membrane sont assimilés à des grandeurs électriques, comme des conductances, ou une capacité par exemple.

La bicouche lipidique est représentée par une capacité C_m. Les canaux ioniques sont représentés par des conductances électriques g_n (n représentant un ion spécifique) qui dépendent à la fois du temps, mais aussi du voltage. Le courant de fuite est également représenté grâce à une conductance linéaire, que l'on notera g_L.

Le potentiel de membrane est défini par V_m, et le courant traversant la couche bilipidique est décrit par l'équation suivante :

$$I_c = C_m \frac{dV_m}{dt} \quad , \tag{2.6}$$

et le courant traversant un canal ionique est donné par :

$$I_i = g_i(V_m - V_i) \quad , \tag{2.7}$$

où V_i est le potentiel de repos du canal ionique i. Finalement, pour une cellule ayant des canaux sodium et potassium, le courant total traversant la membrane est donné par :

$$I = C_m \frac{dV_m}{dt} + g_K(V_m - V_K) + g_{Na}(V_m - V_{Na}) + g_L(V_m - V_L) \quad , \tag{2.8}$$

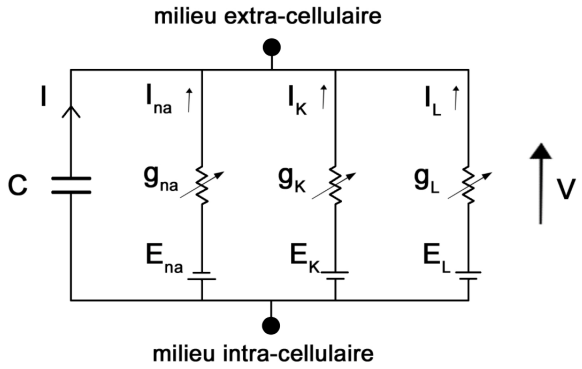

Figure 2.1 – Schéma électrique de la membrane.

où g_{Na} et g_K sont les conductances du sodium et du potassium par unité d'aire, V_{Na} et V_K sont les potentiels de repos du sodium et du potassium, V_L et g_L sont le potentiel de repos et la conductance du courant de fuite (Figure 2.1).

2.1.5.2 Canaux ioniques

Les propriétés des cellules excitables sont décrites par un ensemble de quatre équations différentielles ordinaires. On détermine les conductances intervenant dans l'équation du courant total en utilisant les relations suivantes :

$$g_K = \overline{g_K} n^4 (V_m - V_K) \quad , \tag{2.9}$$

$$g_{Na} = \overline{g_{Na}} m^3 h (V_m - V_{Na}) \quad , \tag{2.10}$$

où $\overline{g_K}$ et $\overline{g_{Na}}$ sont les conductances maximales du potassium et du sodium, et m, n, h sont des variables dépendantes du voltage V_m ainsi que du temps, sans dimension et dont les valeurs sont comprises entre 0 et 1. Elles sont caractérisées par l'équation générale suivante :

$$\frac{dp}{dt} = \alpha_p(V_m)(1-p) - \beta_p(V_m) p \quad , \tag{2.11}$$

où $p = \{m, n, h\}$ avec :

$$\alpha_n(V_m) = \frac{0.01(V_m-10)}{exp(\frac{V_m-10}{10}-1)} \quad \alpha_m(V_m) = \frac{0.1(V_m-25)}{exp(\frac{V_m-25}{10}-1)} \quad \alpha_h(V_m) = 0.07exp(\frac{V_m}{20})$$

$$\beta_n(V_m) = 0.125exp(\frac{V_m}{80}) \quad \beta_m(V_m) = 4exp(\frac{V_m}{18}) \quad \beta_h(V_m) = \frac{1}{exp(\frac{V_m-30}{10})+1}$$

Les canaux de fuite sont responsables de la perméabilité naturelle de la membrane aux ions, et la conductance est vue, dans ce modèle, comme une simple constante.

2.1.6 Modèle de FitzHugh-Nagumo

La complexité du modèle d'Hodgkin-Huxley a entraîné un certain nombre de variantes simplifiées, dont celle introduite par FitzHugh et Nagumo en 1961 (FitzHugh, 1955; Fitzhugh, 1961; Nagumo *et al.*, 1962). Ce modèle est basé sur la relaxation d'un oscillateur, permettant une simplification pouvant être employée dans la modélisation de très grands réseaux de neurones.

Les équations du modèle de FitzHugh-Nagumo sont les suivantes :

$$\frac{dV_m(t)}{dt} = V_m(t) - V_m^3(t) - w(t) - I(t) \quad , \tag{2.12}$$

$$\tau\frac{dw(t)}{dt} = V_m(t) - a - bw(t) \quad , \tag{2.13}$$

avec des conditions initiales sur les variables V_m et w. où w est une variable homogène à une tension, I est un courant de stimulation extérieur, τ, a et b sont des constantes caractéristiques du neurone étudié.

2.1.7 Modèle de Morris-Lecar

Le modèle de Morris et Lecar a été développé en 1981, dont une étude précise a été faite dans Tsumoto *et al.* (2006) pour décrire l'activité électrique du muscle des cirripèdes, puis comme un modèle réduit pour l'excitabilité du neurone. Ce modèle utilise un courant potassique I_K similaire à celui de Hodgkin-Huxley et un courant calcique supposé très rapide et donc modélisé comme instantané.

Les équations de ce modèle sont les suivantes :

$$C\frac{dV}{dt} = -I_{ion}(V,w) + I_{app} \tag{2.14}$$

$$= -(\overline{g_{Ca}} m_\infty(V)(V-\overline{V}_{Ca}) + \overline{g_K} w(V-\overline{V}_K) + \overline{g_L}(V-\overline{V}_L)) + I_{app} \tag{2.15}$$

$$\frac{dw}{dt} = \frac{\phi[w_\infty - w]}{\tau_w(V)} \quad . \tag{2.16}$$

La variable d'activation w est la fraction de canaux K^+ ouvert et fournit la lente rétroaction de la tension requise pour l'excitabilité de la cellule. Etant donné la faible complexité de ce modèle, il est surtout utilisé, comme la plupart des autres modèles présentés dans ce chapitre, dans la simulation de très grands réseaux de neurones.

2.1.8 Conclusion

Nous avons présenté différents modèles décrivant mathématiquement l'activité d'un neurone. Il existe deux types de modèles, ceux permettant de simuler un neurone avec détails comme le modèle de Hodgkin-Huxley, et ceux pouvant être utilisés dans les simulations de réseaux de neurones à cause de leur faible complexité, comme le modèle de Fitzhugh-Nagumo. Il est maintenant intéressant de se focaliser sur description à plus haute échelle du système nerveux. C'est dans cette optique que le chapitre suivant va introduire un modèle permettant de décrire de façon macroscopique l'activité électrique d'un tissu neuronal.

2.2 Modèle bidomaine d'un tissu neuronal

2.2.1 Introduction

Les modèles mathématiques d'un tissu neuronal peuvent fournir des perspectives intéressantes en ingénierie neuronale et en neurophysiologie, en couplant les modèles et des résultats obtenus expérimentalement. Cette approche est dirigée par les technologies thérapeutiques qui utilisent des stimulations électriques pour restaurer des fonctions d'un tissu endommagé (Dowling, 2005). Pour cela, il faut comprendre les différents mécanismes permettant la propagation de l'activité électrique du neurone. C'est dans cette optique que les modèles mathématiques de la stimulation électrique peuvent aider à prédire la localisation de l'activité. Dans la modélisation d'un tissu excitable, le modèle bidomaine est souvent employé, notamment dans le cas du coeur. Malgré un comportement différent de celui du coeur, il est envisageable d'utiliser ce modèle dans le but de décrire d'une façon macroscopique l'activité du cerveau. Je noterai quelques limites de ce modèle dans ce chapitre lors de son application dans le cas des neurosciences.

A l'échelle microscopique, le tissu se décompose en deux milieux bien distincts. Le premier est composé des cellules excitatrices : les neurones. Le second est constitué du reste, notamment du liquide interstitiel, ainsi que d'autres types de cellules, dont nous ne parlerons pas ici. En physiologie, ces deux milieux sont nommés : milieu intra-cellulaire et milieu extra-cellulaire. Bien entendu, par « milieu extra-cellulaire » cela ne signifie aucunement « en dehors des cellules », mais « à l'extérieur des cellules excitatrices ». Comme nous l'avons vu dans la section 1.2, la membrane cellulaire permet de séparer les deux milieux.

D'un point de vue microscopique, la propagation du potentiel d'action se fait grâce à la communication entre les différents neurones formant un petit réseau.

Le modèle bidomaine a été proposé pour la première fois par Schmitt (Schmitt, 1969) et formulé mathématiquement par Tung (Tung, 1978). Ce modèle permet une description continue de l'activité électrique, en couplant les milieux intra-cellulaire et extra-cellulaire. Ce couplage peut-être vu comme un flux de courant à travers la membrane d'une cellule modélisée en utilisant les modèles de neurone présentés dans le chapitre précédent. Il est à noter que l'équation du câble est une variante de ce modèle dans le cas où l'espace est à une dimension. Ce modèle a surtout été utilisé pour la modélisation de l'activité électrique du coeur (Geselowitz et Iii, 1983; Joakim Sundnes, 2001; Henriquez, 1993).

Nous allons tout d'abord décrire la construction du modèle bidomaine, en utilisant la très bonne présentation faite dans la thèse de (Pierre, 2005), en partant des échelles microscopique et macroscopique.

2.2.2 Echelle microscopique

Supposons un volume B représentant l'espace occupé par le cerveau. Ce volume est divisé en deux sous-espaces B_i et B_e, représentant respectivement le milieu intra-cellulaire et le milieu extra-cellulaire. D'un point de vue géométrique, ces deux ensembles sont vus comme des ouverts de tels sorte que :

$$\overline{B} = \overline{B_i} \cup \overline{B_e} \quad , \qquad B_i \cap B_e = \emptyset \quad ,$$

De plus, on suppose que la membrane cellulaire, notée Γ_m, séparant les deux milieux est définie de la façon suivante :

$$\Gamma_m = \partial B_i \cap \partial B_e = \partial B_i - \partial B \quad .$$

On introduit également le vecteur normal unitaire $\mathbf{n}_i$ à Γ_m sortant pour B_i.

Comme les échelles de temps en électrophysiologie sont de l'ordre de la microseconde, alors que celles en électromagnétisme sont bien plus petites, on se permet d'identifier les deux milieux extra et intra-cellulaire comme des conducteurs passifs ayant un état quasistatique. En utilisant cette hypothèse, on peut ainsi relier les potentiels électriques $\phi_{i,e}$ définis, respectivement dans les deux milieux, aux densités volumiques de courant $\mathbf{J}_{i,e}$ par la loi d'Ohm :

$$\mathbf{J}_i = -g_i \nabla \phi_i \quad \text{dans } B_i \quad , \tag{2.17}$$

$$\mathbf{J}_e = -g_e \nabla \phi_e \quad \text{dans } B_e \quad , \tag{2.18}$$

avec :

- les potentiels électriques : $\phi_{i,e} : B_{i,e} \to \mathbb{R}$,
- les densités volumiques de courant : $\mathbf{J}_{i,e} : B_{i,e} \to \mathbb{R}^3$,
- les tenseurs de conductivités : $g_{i,e} : B_{i,e} \to \mathbb{R}^+$.

En supposant qu'il n'y a aucune création de charges, on en déduit que les courants volumiques $\mathbf{J}_{i,e}$ sont de divergences nulles

$$\nabla \mathbf{J}_i = 0 \quad \text{dans } B_i \quad , \tag{2.19}$$

$$\nabla \mathbf{J}_e = 0 \quad \text{dans } B_e \quad . \tag{2.20}$$

En injectant (2.17) - (2.18) dans (2.19) - (2.20), on en déduit que les potentiels $\phi_{i,e}$ sont solutions d'une équation de Laplace :

$$\nabla \cdot (g_i \nabla \phi_i) = 0 \quad \text{dans } B_i \quad , \tag{2.21}$$

$$\nabla \cdot (g_e \nabla \phi_e) = 0 \quad \text{dans } B_e \quad . \tag{2.22}$$

On introduit la densité de surface de courant $I_m : \Gamma_m \to \mathbb{R}$ mesurée de B_i vers B_e :

$$I_m = \mathbf{J}_i \cdot \mathbf{n}_i = \mathbf{J}_e \cdot \mathbf{n}_e \quad , \tag{2.23}$$

ou encore en utilisant la notion de potentiel :

$$I_m = -g_i \nabla \phi_i \cdot \mathbf{n}_i = -g_e \nabla \phi_e \cdot \mathbf{n}_e \quad . \tag{2.24}$$

Introduisons le potentiel de membrane $V_m : \Gamma_m \to \mathbb{R}$:

$$V_m = \phi_i - \phi_e \quad . \tag{2.25}$$

Le comportement de la membrane Γ_m est à la fois résistif et capacitif. La caractéristique capacitive vient du fait que le membrane est composée d'une couche bilipidique isolant le milieu extra-cellulaire du milieu intra-cellulaire. Ce comportement est modélisé grâce à une capacité par unité de surface membranaire : C_m. La caractéristique résistive de la membrane provient du transport d'ions entre les milieux extra et intra-cellulaire, assuré par les protéines de la membrane.

On introduit également un courant ionique I_{ion} que l'on mesure de B_i vers B_e et traversant la membrane Γ_m. Ce courant ionique est déterminé à l'aide des modèles de neurones étudiés dans la section 2.1.

On peut émettre une remarque, en disant que cette description résistif/capacitif de la membrane cellulaire est une modélisation microscopique de Γ_m, car si on se trouve sur la couche bilipidique, alors la membrane a un caractère capacitif, et si on se trouve sur une protéine alors cette même membrane possède cette fois, un comportement résistif. Etant donné qu'une protéine est de très petite taille, et que leur distribution est uniformément dense sur la membrane Γ_m, on peut formuler un comportement moyen de la membrane à l'échelle de la cellule.

De plus I_m doit obéir à la relation suivante :

$$I_m = C_m \frac{\partial V_m}{\partial t} + I_{ion} = -g_i \nabla \phi_i \cdot \mathbf{n}_i = -g_e \nabla \phi_e \cdot \mathbf{n}_e \quad . \tag{2.26}$$

On peut noter que la première partie de cette équation est identique au modèle de Hodgkin-Huxley. Ici, on ne fait que rajouter à cette équation un caractère spatial au modèle pour prendre en compte une géométrie complexe, ainsi que les différentes conductivités du volume étudié. Au final, le modèle microscopique est un ensemble de deux équations aux dérivées partielles de Laplace (2.21) (2.22), dans les domaines B_i et B_e, avec lesquelles on couple la condition de transmission (2.26) sur Γ_m. Pour finir, on rajoute des conditions aux limites sur le bord du domaine de définition qui seront détaillées dans la section 2.2.5.

L'existence de solutions pour un tel type de système d'équations a été montrée dans Piero Colli Franzone (2002), pour des conditions aux limites de type Neumann (voir section 2.2.5).

2.2.3 Echelle macroscopique

A cette échelle, les milieux intra-cellulaire B_i et extra-cellulaire B_e sont supposés indifférentiables, on introduit donc un volume global comme étant la superposition des deux milieux

$$B = B_i = B_e$$

Cette simplification se traduit par la présence d'un nouveau paramètre : le taux de surface membranaire par unité de volume, noté A_m.

La superposition des deux domaines permet d'homogénéiser les équations (2.22) (2.23) (2.25) (2.26) en utilisant une analyse présentée dans Neu et Krassowska (1993). On suppose un volume $\mathbf{V}$ comportant un grand nombre de cellules, et on définit sur $\mathbf{V}$ les quantités homogénéisées $\tilde{I}_m$, $\tilde{I}_{ion}$, $\tilde{V}_m$ de la manière suivante :

$$\tilde{I}_m = \frac{1}{|\mathbf{V} \cap \Gamma_m|} \int_{\mathbf{V} \cap \Gamma_m} I_m ds \quad , \tag{2.27}$$

$$\tilde{I}_{ion} = \frac{1}{|\mathbf{V} \cap \Gamma_m|} \int_{\mathbf{V} \cap \Gamma_m} I_{ion} ds \quad , \tag{2.28}$$

$$\tilde{V}_m = \frac{1}{|\mathbf{V} \cap \Gamma_m|} \int_{\mathbf{V} \cap \Gamma_m} V_m ds \quad , \tag{2.29}$$

où $|\mathbf{V} \cap \Gamma_m|$ représente la surface de la membrane contenue dans $\mathbf{V}$. Toutes ces quantités

vérifient l'équation (2.26)

$$\tilde{I}_m = C_m \frac{\partial \tilde{V}_m}{\partial t} + \tilde{I}_{ion} \quad . \tag{2.30}$$

En ce qui concerne les densités volumiques de courant, on obtient les relations suivantes :

$$\frac{1}{\mathbf{V}} \int_{\partial \mathbf{V}} \mathbf{J}_e \cdot \mathbf{n} ds = \frac{1}{\mathbf{V}} \int_{\partial \mathbf{V} \cap \Gamma_m} \mathbf{J}_e \cdot \mathbf{n}_i ds = \frac{1}{\mathbf{V}} \int_{\partial \mathbf{V} \cap \Gamma_m} I_m ds = \frac{|\mathbf{V} \cap \Gamma_m|}{|\mathbf{V}|} \tilde{I}_m \quad , \tag{2.31}$$

où $|\mathbf{V}|$ est la mesure de $\mathbf{V}$, et $\mathbf{n}$ désigne la normale sortante de $\mathbf{V}$. On a de manière analogue :

$$\frac{1}{\mathbf{V}} \int_{\partial \mathbf{V}} \mathbf{J}_i \cdot \mathbf{n} ds = -\frac{|\mathbf{V} \cap \Gamma_m|}{|\mathbf{V}|} \tilde{I}_m \quad . \tag{2.32}$$

Les différentes quantités $\int_{\partial \mathbf{V}} \mathbf{J}_{i,e} \cdot \mathbf{n} ds / |\mathbf{V}|$ s'interprétent à travers la formule de divergence comme étant similaire à la divergence du courant volumique homogénéisé $\tilde{\mathbf{J}}_{i,e}$:

$$\nabla \cdot \tilde{\mathbf{J}}_{i,e} = \frac{1}{|\mathbf{V}|} \int_{\partial \mathbf{V}} \mathbf{J}_{i,e} \cdot \mathbf{n} ds \quad . \tag{2.33}$$

En insérant le taux moyen de surface membranaire par unité de volume A_m, on peut donner une version condensée des équations homogènes (en omettant les tildes des équations précédentes) :

$$\nabla \cdot (\mathbf{J}_i + \mathbf{J}_e) = 0 \quad \text{dans } B \quad , \tag{2.34}$$

$$A_m (C_m \frac{\partial V_m}{\partial t} + I_{ion}) = \nabla \cdot \mathbf{J}_e \quad \text{dans } B \quad . \tag{2.35}$$

On étend également les variables ϕ_i, ϕ_e, V_m représentant les potentiels, ainsi que le courant ionique membranaire I_{ion} à des fonctions définies sur tout B :

$$\phi_i,\ \phi_e,\ V_m : B \to \mathbb{R} \quad .$$
$$V_m = \phi_i - \phi_e \quad \text{dans } B$$

2.2.4 Anisotropie

A l'échelle macroscopique, la conductivité du cerveau est inhomogène et anisotropique. Autrement dit, la conductivité n'est pas la même suivant la direction longitudinale ou transversale.

Ces deux propriétés jouent un rôle important dans l'activation du tissu excitable. L'orientation du champ électrique par rapport à l'axe de la fibre affecte le seuil de stimulation et module les modes de propagation électrique.

L'anisotropie provient du fait que les axones sont parallèles les uns aux autres dans la matière blanche, et donc la conductivité longitudinale est plus grande que la conductivité transversale (Nicholson, 1965 ; Ranck and BeMent, 1965). L'inhomogénéité vient de la différence entre les différentes régions du tissu neuronal (c'est-à-dire la matière blanche et la matière grise), ce qui cause des différences entre les conductivités des milieux.

On introduit alors les tenseurs de conductivités M_i, M_e, dans les milieux intra-cellulaire et extra-cellulaire respectivement. Dans le cas 3D, ces tenseurs sont définis (dans une base orthonormée) de la manière suivante :

$$M_{i,e} = diag(c^l_{i,e}, c^t_{i,e}, c^t_{i,e})$$

où les constantes $c^l_{i,e}$ et $c^t_{i,e}$ désignent les conductivités électriques intra-cellulaire et extra-cellulaire dans la direction longitudinale ou transversale.

On peut donc réecrire les équations (2.34) (2.35) en termes de potentiels ainsi :

$$\nabla \cdot (M_i \nabla \phi_i + M_e \nabla \phi_e) = 0 \quad \text{dans } B \quad , \tag{2.36}$$

$$A_m(C_m \frac{\partial V_m}{\partial t} + I_{ion}) = -\nabla \cdot (M_e \nabla \phi_e) \quad \text{dans } B \quad , \tag{2.37}$$

avec la relation

$$V_m = \phi_i - \phi_e \quad \text{dans } B \quad , \tag{2.38}$$

ce qui devient en insérant (2.38) dans (2.36) :

$$\nabla \cdot ((M_i + M_e) \nabla \phi_e) = -\nabla \cdot (M_i \nabla V_m) \quad \text{dans } B \quad . \tag{2.39}$$

2.2.5 Conditions aux limites au bord du cerveau

Pour que le modèle soit complet, il faut encore définir les conditions aux limites au bord du cerveau. Ces conditions peuvent être classifiées comme des conditions de Dirichlet ou de Neumann.

Les conditions de Dirichlet sont spécifiées quand le potentiel électrique aux limites géométriques est connu. L'exemple le plus répandu est lorsque l'on spécifie la valeur du potentiel extra-cellulaire à une surface d'électrode.

Les conditions de Neumann sont utilisées dans le cas où les densités de courant sont définies aux frontières. Pour des frontières isolées, la densité de courant normal est nulle, ce qui est également appelé : condition aux limites à flux nul. Cela signifie qu'aucun courant ne circule à travers la frontière. Dans la littérature, nous trouvons essentiellement le modèle bidomaine couplé à des conditions aux limites de type Neumann (R. Szmurlo, 2007).

D'un point de vue purement mathématique, les conditions de Neumann appliquées au problème bidomaine se traduisent par :

$$(M_i \nabla \phi_i) \cdot \mathbf{n}_{\partial B} = 0 \quad \text{sur } \partial B \quad , \tag{2.40}$$

$$(M_e \nabla \phi_e) \cdot \mathbf{n}_{\partial B} = 0 \quad \text{sur } \partial B \quad , \tag{2.41}$$

où $\mathbf{n}_{\partial B}$ est la normale sortante au bord du cerveau ∂B.

Il faut noter qu'en plus de ces conditions de flux nul, on doit ajouter au moins une condition aux limites de Dirichlet à un endroit du domaine. Le plus souvent cette condition est imposée au potentiel extra-cellulaire ϕ_e en spécifiant un potentiel de masse.

Les conditions précédentes s'appliquent uniquement dans le cas où l'on suppose que le cerveau est totalement isolé, sans aucun contact avec le crâne. Or en réalité, les conditions aux limites sont des relations de couplage entre l'activité électrique du cerveau et sa continuité dans les différentes couches du crâne au travers de la frontière ∂B séparant le cerveau du crâne.

2.2.6 Relations de couplage

Les conditions aux limites détaillées dans la section précédente s'appliquent dans le cas où on considère le volume représentant le tissu neuronal comme complètement isolé. Dans la réalité, et encore plus si on considère le cerveau dans sa globalité, et non pas uniquement une fraction de celui-ci, les conditions aux limites au bord de ce volume seront physiquement différentes. La propagation du signal électrique se poursuivra, malgré le changement de milieu, en vérifiant une autre équation. Dans ce qui va suivre, nous allons présenter différents types de relations de couplage reliant le cerveau et le crâne, et qui semblent plausibles au vu des données physiologiques.

Le crâne est modélisé par un conducteur passif, dont la conductivité est variable selon les couches qui le compose, et potentiellement anisotrope. Pour simplifier l'écriture du problème,

on suppose dans ce qui va suivre que le crâne est constitué d'une seule couche, ce qui permet d'introduire le tenseur de conductivité M_H. De plus, on émet l'hypothèse que ce conducteur est dans un état quasistatique, ce qui entraîne que le courant volumique J_H dans le crâne est relié au potentiel du crâne ϕ_H par la loi d'Ohm :

$$J_H = -M_H \nabla \phi_H \quad \text{dans } H \quad . \tag{2.42}$$

Etant donné qu'il n'y a pas de création de charge, on en déduit que ce courant a une divergence nulle, ce qui entraîne que le potentiel ϕ_H est donné par une équation de Laplace dans le crâne (représenté par le domaine H)

$$\nabla \cdot (M_H \nabla \phi_H) = 0 \quad \text{dans } H \quad , \tag{2.43}$$

que l'on complète en rajoutant une condition de flux nul sur le bord extérieur du domaine représentant le crâne :

$$M_H \nabla \phi_H \cdot \mathbf{n}_{\partial H} = 0 \quad \text{sur } \partial H \quad , \tag{2.44}$$

où $\mathbf{n}_{\partial H}$ désigne la normale unitaire sortante sur ∂H.

On complète le modèle en y ajoutant des relations de couplage entre le potentiel extra-cellulaire défini dans B et le potentiel défini dans H, permettant la continuité de la propagation du signal électrique dans H :

$$\phi_e = \phi_H \qquad \text{sur } \partial B \quad , \tag{2.45}$$

$$M_H \nabla \phi_H \cdot \mathbf{n}_{\partial B} = M_e \nabla \phi . \mathbf{n}_{\partial B} \qquad \text{sur } \partial B \quad , \tag{2.46}$$

ce qui peut-être traduit par le fait que la liaison entre le domaine B et H se fait uniquement via le potentiel extra-cellulaire.

2.2.7 Autre relations de couplage

Une des difficultés dans la résolution du problème bidomaine en espace vient du fait que la relation de couplage (2.46) n'est pas compatible avec l'équation (2.39), dans le sens où il n'y a pas d'annulation des termes sur le bord ∂B dans une formulation faible de (2.37) et (2.39). On trouve une autre classe de relations de couplage dans la littérature :

$$\phi_e = \phi_H \qquad \text{sur } \partial B \quad , \tag{2.47}$$

$$(M_i \nabla \phi_i + M_e \nabla \phi_e) \cdot \mathbf{n}_{\partial B} = M_H \nabla \phi_H \cdot \mathbf{n}_{\partial B} \quad \text{sur } \partial B \quad , \tag{2.48}$$

$$M_i \nabla V_m \cdot \mathbf{n}_{\partial B} = 0 \quad \text{sur } \partial B \quad . \tag{2.49}$$

On peut s'apercevoir qu'en utilisant ces relations de couplage et la condition aux limites (2.43), le problème bidomaine en espace possède une formulation faible simple sur le domaine $\Omega = B \cup H \cup \partial B$.

$$\forall \eta \in H^1(\Omega) \; : \; \int_\Omega \nabla \eta \cdot \tilde{M} \nabla \Phi = \int_B (M_i \nabla V_m) \eta \quad , \tag{2.50}$$

où $\Phi \in \mathrm{V} = \{ \, v \in H^1(\Omega) : \int_\Omega v \, dx = 0 \, \}$ est l'inconnue représentant les potentiels sur les différents domaines :

$$\Phi = \begin{cases} \phi_e & \text{sur} \quad B \\ \phi_H & \text{sur} \quad H \end{cases} ,$$

et avec le tenseur $\tilde{M}$:

$$\tilde{M} = \begin{cases} M_i + M_e & \text{sur} \quad B \\ M_H & \text{sur} \quad H \end{cases} .$$

L'expression (2.48) peut-être vue comme une continuité du flux de courant à travers ∂B. Si on raisonne en termes de courants volumiques, cela correspond à une conservation du courant sur l'interface cerveau/crâne. Par contre, la justification physiologique de la relation (2.49) n'est pas évidente.

2.2.8 Résumé du modèle bidomaine

Maintenant que chaque élément permettant de construire le modèle est posé, nous allons effectuer un résumé global du modèle bidomaine. Ce modèle possède trois inconnues représentant des potentiels : ϕ_e, ϕ_i et V_m. Ces trois inconnues sont reliées par l'expression $V_m = \phi_i - \phi_e$. Notre choix s'est porté sur une formulation du problème bidomaine en fonction des grandeurs V_m et ϕ_e.

D'un point de vue purement mathématique, les modèles sont identiques peu importe le

choix des grandeurs (Szmurlo *et al.*, 2007), mais d'un point de vue physiologique, la meilleur formulation est celle comportant les trois inconnues, étant donné qu'une des conditions de flux nul dépend de ϕ_i.

Le problème à intégrer est donc constitué d'une équation d'évolution couplée à une équation elliptique :

$$-\nabla\cdot(M_i\nabla V_m)+A_m(C_m\partial_t V_m+I_{ion})=\nabla\cdot(M_i\nabla\phi_e)\qquad\text{dans } B\times[0;T]\quad, \tag{2.51}$$

$$M_i\nabla\phi_e\cdot n_{\partial B}=-M_i\nabla V_m\cdot n_{\partial B}\qquad\text{sur } \partial B\times[0;T]\quad, \tag{2.52}$$

$$M_i\nabla\phi_H\cdot n_{\partial H}=0\cdot n_{\partial H}\times[0;T]\quad, \tag{2.53}$$

avec une condition initiale sur V_m :

$$V_m(t=0)=V_m^0\qquad\text{sur } B\quad. \tag{2.54}$$

et la variable I_{ion} est déterminée grâce à un système d'EDO décrivant l'activité d'un neurone. Le potentiel extra-cellulaire initial $\phi_e(t=0)$ est défini grâce à l'équation elliptique et les conditions aux limites au bord du cerveau :

$$\phi_e=\phi_H\qquad\text{sur } \partial B\quad, \tag{2.55}$$

$$M_e\nabla\phi_e\cdot\mathbf{n}_{\partial B}=M_H\nabla\phi_H\cdot\mathbf{n}_{\partial B}\quad, \tag{2.56}$$

$$M_i\nabla\phi_e\cdot\mathbf{n}_{\partial B}=-M_i\nabla V_m\cdot\mathbf{n}_{\partial B}\quad. \tag{2.57}$$

2.2.9 Simplification : le modèle monodomaine

La plus grande difficulté du problème bidomaine, que ce soit mathématique ou numérique, réside dans le fait que l'opérateur différentiel $\nabla\cdot(M_i\nabla\phi_e)$ n'est pas donné explicitement à partir du potentiel de membrane V_m, mais uniquement en passant par le problème elliptique (2.52). Il faut donc résoudre à chaque pas de temps un problème elliptique afin de pouvoir mettre à jour V_m. Ce calcul peut devenir très coûteux en terme de ressources lorsque la discrétisation en espace du problème (ou le pas de temps choisi) devient très fine, sachant que si des méthodes

itératives sont utilisées, le temps de résolution peut grandir très rapidement. C'est pour cela qu'un nouveau modèle développé, permettant ainsi de simplifier le modèle existant en évitant de devoir résoudre un problème elliptique à chaque pas de temps.

Supposons maintenant que les rapports d'anisotropie entre les milieux intra et extra-cellulaires sont égaux, autrement dit : il existe une constante $\lambda > 0$ telle que :

$$\forall x \in B \ : M_i(x) = \lambda M_e(x) \quad . \tag{2.58}$$

On peut donc simplifier le problème en fusionnant les équations (2.51) (2.52) en une seule portant sur V_m uniquement.

Le problème elliptique (2.52) se réecrit :

$$(\lambda + 1)\nabla \cdot (M_e \nabla \phi_e) = -\nabla \cdot (M_i \nabla V_m) \qquad \text{dans } B \times [0;T] \quad , \tag{2.59}$$

d'où on en déduit la nouvelle expression de (2.51)

$$A_m(C_m \frac{\partial V_m}{\partial t} + I_{ion}) = \frac{1}{\lambda + 1}\nabla \cdot (M_i \nabla V_m) \qquad \text{dans } B \times [0;T] \quad . \tag{2.60}$$

Le modèle décrit par cette unique équation est appelé « modèle monodomaine ». On complète cette équation avec des conditions aux limites de type Neumann, afin de pouvoir déterminer le potentiel de membrane en tout point du cerveau. Il est à noter que cette formulation simplifiée du modèle bidomaine s'applique au cas du cerveau isolé. Si on souhaite avoir l'activité du modèle couplé, on est obligé de mettre à jour le potentiel extra-cellulaire ϕ_e, ce qui revient à résoudre le problème bidomaine.

2.2.10 Limites de cette approche

Ce modèle présente néanmoins des limites lors de son application sur un tissu neuronal, car même si celui ci possède des cellules excitatrices tout comme le coeur, son fonctionnement n'est pas identique. En effet, les cellules cardiaques s'excitent de proche en proche (Pierre, 2005), ce qui n'est pas toujours le cas des neurones. De plus, à haute échelle, les différentes zones du cerveau, que l'on peut séparer en 66 sous-régions anatomiques possèdent des connexions entre 998 régions d'intérêts ayant une taille moyenne de $1,5\ cm^2$, pouvant être représentées au sein d'un graphe, par des poids de connexion (Hagmann *et al.*, 2008). Ainsi, il existe des connexions privilégiées entre certaines régions d'intérêts qui ne sont pas morphologiquement proches. Cette caractéristique a été souvent montré au sein d'études (Sporns *et al.*, 2005; Achard et Bullmore, 2007; Bassett *et al.*, 2006) créant une cartographie du cerveau, mais également une meilleure compréhension entre la connectivité structurelle et la

connectivité fonctionnelle. Comme on peut le constater, le modèle bidomaine présenté dans ce chapitre, ne permet pas de prendre en compte cette complexité, entraînant une limite lors de l'utilisation sur tout le cerveau. Il est tout de même possible de définir des régions d'intérêts ayant des propriétés physiques différentes, comme cela est effectué au sein du coeur, qui pour rappel n'a pas les mêmes caractéristiques partout, mais la grande difficulté réside dans la traduction de la matrice des connexions déterminée grâce à la cartographie du cerveau.

Même si ce modèle à des limites lorsque l'on voudra l'appliquer au cerveau dans sa globalité, il peut être appliqué à des portions plus restreintes, comme lors de l'observation de l'activité d'un tissu neuronal. Notamment, il peut être utilisé afin de déterminer le Local Field Potential (LFP) d'un petit réseau de neurones. Ce champ est généré par le courant électrique provenant de plusieurs neurones voisins au sein d'un volume, en général assez petit, de tissu neuronal (Klas H Pettersen, 2010; Bédard *et al.*, 2004).

Il est aussi possible de l'utiliser afin de simuler la propagation électrique le long d'un nerf (R. Szmurlo, 2007; He, 2013). Dans R. Szmurlo (2007), le modèle bidomaine est appliqué pour la simulation lors de l'excitation du nerf vague. Dans cet exemple, on distingue deux domaines dont un représentant la fibre nerveuse (Ω_N) et l'autre le tissu entourant cette fibre (Ω_T). Puis deux électrodes sont modélisées afin d'exciter le nerf (Figure 2.2).

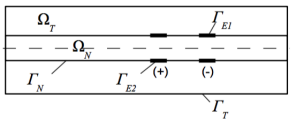

Figure 2.2 – Sous-domaines du modèle du nerf vague d'après R. Szmurlo (2007).

Lors de la simulation, le nerf a été excité à l'aide d'un pulse de 1 ms. Un front de propagation du potentiel d'action a pu être observé, et ce résultat a pu être confronté avec des données expérimentales. R. Szmurlo (2007) en déduit que les valeurs obtenues grâce à la simulation sont très proches des valeurs expérimentales.

Ainsi le champ d'applications de ce modèle, à des problèmes de neuroscience n'est pas nul.

2.2.11 Conclusion

Le modèle bidomaine a été construit dans le but de décrire l'activité électrique d'un tissu neuronal, et, par extension, le cerveau dans sa globalité. Il est composé de deux équations aux dérivées partielles et d'une équation différentielle ordinaire, ce qui fait de lui un modèle lourd à simuler sur de grandes échelles de temps, surtout, si en fonction du problème posé, une géométrie complexe est requise. Une première manière de contourner cette limite est d'utiliser un modèle simplifié dit modèle monodomaine. L'équation aux dérivées partielles elliptique disparaît pour laisser place à une seule équation selon le potentiel de membrane,

car c'est bien cette étape qui est la plus coûteuse en terme de temps de calculs (Pierre, 2005). Il faut bien entendu toujours rajouter l'équation différentielle ordinaire pour mettre à jour la variable représentant le courant ionique. Cette dernière est choisie de manière à ce que la solution soit la plus réaliste possible, tout en ayant une complexité moindre. Comme on l'a vu, le calcul du courant ionique est déterminé grâce à un modèle de neurone, en général du type Fitzhugh-Nagumo, servant dans la plupart des cas à la simulation des réseaux de neurones, contrairement à un modèle de type Hodgkin-Huxley, décrivant parfaitement les dynamiques du potentiel de membrane, mais demandant plus de ressources lors de sa simulation.

Afin d'intégrer les systèmes d'équations de manière la plus efficace possible, c'est-à-dire en obtenant une solution de très bonne qualité en un minimum de temps de calcul, ils existent diverses solutions. Dans la partie suivante, je vais présenter différentes méthodes, utilisant l'aspect multi-coeur des ordinateurs actuels, ainsi que leurs ressources graphiques, faisant appel à des GPU (Graphical Process Unit). Le but final est de pouvoir appliquer ces diverses méthodes aux modèles monodomaine, puis bidomaine, afin d'effectuer les simulations sur des géométries complexes ou/et sur de longues durées.

Partie II

Méthodes numériques de parallélisation et matériels

3 Principe de la parallélisation

3.1 Parallélisation en temps

3.1.1 Introduction

Suite à l'émergence des ordinateurs multi-processeurs puis des processeurs de cartes graphiques, une nouvelle manière de résoudre les problèmes est apparue : la parallélisation. De nouvelles techniques de parallélisation ont été constamment adaptées pour tenir compte de ces nouvelles architectures. Le but de ces méthodes est de pouvoir résoudre des problèmes en temps réel. Parmi ces méthodes, il existe les décompositions de domaine, qui sont des techniques efficaces pour le découpage d'un problème d'équation aux dérivées partielles (voir section 3.2). Jusqu'ici, la technique pour décomposer le problème au niveau de l'échelle temporelle n'a pas bénéficié du même effort que pour l'échelle spatiale, à cause du caractère séquentiel de cette partie du problème. Avant de discuter des différents algorithmes permettant une parallélisation d'un problème suivant l'échelle de temps, nous allons rapidement présenter les méthodes de résolution des problèmes d'évolution, pouvant être décrites par des équations différentielles ordinaires (EDO).

3.1.2 Intégration des équations différentielles ordinaires

Les équations différentielles ordinaires (EDO) permettent notamment de définir des modèles décrivant des phénomènes physiques ou biologiques. Un exemple concret de l'utilisation des EDO est la modélisation de l'activité électrique d'un neurone, comme nous avons pu le voir dans la partie 1. Bien évidemment, effectuer une résolution analytique de ces problèmes est impossible dans la majorité des cas. Comme une solution continue ne peut-être obtenue, on cherche à déterminer une solution dite discrète, autrement dit trouver les valeurs de la fonction solution en des dates bien précises. Supposons que l'on souhaite résoudre le

problème général suivant :

$$\frac{du}{dt} = f(u(t), t) \text{ avec } t \in [0; T] \quad . \tag{3.1}$$

On découpe l'intervalle $[0; T]$ en N points $t_i = i\Delta t$ avec $i \in 0, 1, ..., N-1$ et Δt un pas de temps. Pour obtenir les solutions à ces problèmes, on fait appel à des méthodes de résolution numérique, qui se chargent d'approcher la solution par une fonction affine par morceaux sur chaque intervalle $[t_i; t_{i+1}]$. Il existe deux types de méthodes utilisant des pas de temps différents : les méthodes à pas fixe, où celui-ci ne varie pas en fonction du temps, et les méthodes multi-pas, dont le pas varie en fonction de l'erreur commise sur le calcul de la solution. Pour des raisons pratiques liées à l'utilisation des algorithmes de parallélisation, je ne détaillerai pas ici les méthodes multi-pas, qui ont été l'objet de nombreuses études (Hairer, 1993; Hairer et Wanner, 1996), et notamment au sein de Rhenovia, avec les thèses de (Greget, 2011; Sarmis, 2013).

La méthode la plus ancienne, mais également la plus simple, est sans conteste la méthode d'Euler. Notons u_n la valeur approchée de $u(t_n)$. En utilisant un développement limité de u_{n+1} à l'ordre 1, on arrive à l'expression suivante :

$$u(t_{n+1}) = u(t_n) + \Delta t \, \frac{du}{dt}(t_n) + O(\Delta t) \quad , \tag{3.2}$$

qui devient, en utilisant les approximations u_n de $u(t_n)$:

$$u_{n+1} \approx u_n + \Delta t \, \frac{du_n}{dt} + O(\Delta t) \quad . \tag{3.3}$$

On en déduit donc la méthode d'Euler :

$$\frac{u_{n+1} - u_n}{\Delta t} = f(u_n, t_n) \quad , \tag{3.4}$$

qui peut-être écrite sous sa forme plus conventionnelle :

$$u_{n+1} = u_n + \Delta t \, f(u_n, t_n) \quad . \tag{3.5}$$

Cette méthode est notamment appelée Euler explicite, car elle utilise la valeur de la solution au temps t_n pour déterminer sa valeur au temps suivant t_{n+1}. En utilisant un autre

développement limité d'ordre 1 pour écrire u_{n+1} :

$$u_{n+1} = u_n + \Delta t \, \frac{du_{n+1}}{dt} + O(\Delta t) \quad , \tag{3.6}$$

on peut en déduire la méthode d'Euler dite implicite :

$$\frac{u_{n+1} - u_n}{\Delta t} = f(u_{n+1}, t_{n+1}) \quad , \tag{3.7}$$

ou encore sous sa forme plus conventionnelle :

$$u_{n+1} = u_n + \Delta t \, f(u_{n+1}, t_{n+1}) \quad . \tag{3.8}$$

Par rapport à la méthode dite explicite, le terme u_{n+1} apparaît dans les termes de l'équation, ce qui contraint à utiliser des méthodes de résolution numérique spécifiques dans le cas où la fonction f est non-linéaire. Cette méthode a cependant le mérite d'être bien plus stable, ce qui permet d'utiliser des pas de temps plus grands lors de la résolution.

Afin d'améliorer la méthode d'Euler explicite, de nouvelles méthodes, de type Runge-Kutta, ont été développées afin de permettre une meilleure stabilité lors de l'utilisation de pas de temps assez grands, mais également, en gardant la simplicité des méthodes explicites. Contrairement à la méthode d'Euler, les méthodes de Runge-Kutta utilisent des points intermédiaires $t_n < t_{n,i} < ... < t_{n+1}$ avec $i \in]0;1[$ et $t_{n,i} = t_n + c_i \Delta t$.

Ces points intermédiaires sont associés à des poids :

$$p_{n,i} = f(t_{n,i}, u_{n,i}) \quad . \tag{3.9}$$

Comme u est solution de l'équation différentielle définit précédemment, on peut donc écrire l'égalité suivante :

$$u(t_{n,i}) = u(t_n) + \int_{t_n}^{t_{n,i}} f(s, u(s)) ds \quad . \tag{3.10}$$

En effectuant un changement de variable $s = t_n + k\Delta t$, on obtient :

$$u(t_{n,i}) = u(t_n) + \Delta t \int_0^{c_i} f(t_n + k\Delta t, u(t_n + k\Delta)) dk \quad . \tag{3.11}$$

De même :

$$u(t_{n+1}) = u(t_n) + \Delta t \int_0^1 f(t_n + k\Delta t, u(t_n + k\Delta))dk \quad . \tag{3.12}$$

On se donne pour chaque $i = 1, 2, ..., q$ des nombres $(a_{ij})_{1 \leq j < i}$ et $(b_j)_{1 \leq j < q}$ tels que :

$$\int_0^{c_i} \phi(k)dk = \sum_{1 \leq j < i} a_{ij}\phi(c_j) \quad , \tag{3.13}$$

$$\int_0^1 \phi(k)dk = \sum_{1 \leq j < q} b_j\phi(c_j) \quad . \tag{3.14}$$

On suppose que les conditions suivantes sont toujours vérifiées :

$$c_i = \sum_{1 \leq j < i} a_{ij} \ \forall 1 \leq i \leq q \quad , \tag{3.15}$$

$$\sum_{1 \leq j < q} b_j = 1 \quad . \tag{3.16}$$

En insérant ces dernières formules dans (3.11) (3.12), on obtient alors :

$$u(t_{n,i}) \approx u(t_n) + \Delta t \sum_{1 \leq j < i} a_{ij} f(t_{n,j}, u(t_{n,j})) \quad , \tag{3.17}$$

$$u(t_{n,i}) \approx u(t_n) + \Delta t \sum_{1 \leq j < q} b_j f(t_{n,j}, u(t_{n,j})) \quad . \tag{3.18}$$

La méthode de Runge-Kutta d'ordre q correspondante est la suivante :

$$\begin{cases} t_{n,i} = t_n + c_i \Delta t \\ u_{n,i} = u_n + \Delta t \sum_{1 \leq j < i} a_{ij} p_{n,j} \\ p_{n,i} = f(t_{n,i}, u_{n,i}) \\ t_{n+1} = t_n + h_n \\ u_{n+1} = u_n + \Delta t \sum_{1 \leq j < k} b_j p_{n,j}. \end{cases}$$

Le plus souvent, on résume la méthode en référençant les différents poids de quadrature au sein d'un tableau nommé « tableau de Butcher » (Tableau 3.1).

c_1					
c_2	$a_{2,1}$				
c_3	$a_{3,1}$	$a_{3,2}$			
$\vdots$	$\vdots$	$\vdots$	$\ddots$		
c_q	$a_{q,1}$	$a_{q,2}$	$\cdots$	$a_{q,q-1}$	
	b_1	b_2	$\cdots$	b_{q-1}	b_q

Tableau 3.1 – Tableau de Butcher (Butcher, 1996) pour la méthode de Runge-Kutta.

Il est trivial de dire que la méthode de Runge-Kutta d'ordre 1 est équivalente à la méthode d'Euler. La méthode de Runge-Kutta d'ordre 4 est certainement la plus connue et la plus utilisée. L'écriture du schéma de résolution d'un problème écrit sous la forme (3.1) est donnée par l'équation :

$$u_{n+1} = u_n + \frac{\Delta t}{6}(k_1 + 2k_2 + 2k_3 + k_4) \quad , \tag{3.19}$$

où

$$k_1 = f(t_n, u_n) \quad ,$$

$$k_2 = f(t_n + \frac{\Delta t}{2}, u_n + \frac{\Delta t}{2} k_1) \quad ,$$

$$k_3 = f(t_n + \frac{\Delta t}{2}, u_n + \frac{\Delta t}{2} k_2) \quad ,$$

$$k_4 = f(t_n + \Delta t, u_n + \Delta t\ k_2) \quad .$$

On notera, que la méthode de Runge-Kutta d'ordre 4 commet une erreur à chaque étape de l'ordre de Δt^5, alors que celle accumulée est de l'ordre de Δt^4.

Il existe aussi des formulations implicites de ces méthodes de Runge-Kutta, de plus les formules dites backward differentiation formulas (BDF) sont également très répandues (Hairer et Wanner, 1996). Les méthodes présentées sont les plus utilisés lorsqu'il s'agit des méthodes dites explicites. Néanmoins, il existe des méthodes également très répandues lorsqu'il est nécessaire de résoudre des problèmes d'EDP dans la direction temporelle. Parmi ces méthodes, on trouve notamment le θ-schéma qui permet de retrouver la méthode de Crank-Nicolson lorsque $\theta = 1/2$, ainsi que la méthode d'Euler implicite lorsque $\theta = 1$. La méthode de Crank-Nicolson permet de réduire l'erreur sur la dérivée temporelle en $O(\Delta t^2)$ au lieu de $O(\Delta t)$ ce

qui en fait une méthode plus stable que les méthodes d'Euler. Elle consiste à estimer la valeur de u au temps t_{n+1} à l'aide de l'expression (3.20) :

$$u_{n+1} = u_n + \frac{\Delta t}{2} \left(f(u_n, t_n) + f(u_{n+1}, t_{n+1}) \right) \quad . \tag{3.20}$$

On trouve dans la littérature (com, 2006; Sundnes *et al.*, 2005; Krishnamoorthi *et al.*, 2013) des méthodes réduisant le modèle monodomaine sous la forme d'une EDP linéaire et d'une ODE non-linéaire. Afin de résoudre ce système d'équations de manière efficace, une méthode de Crank-Nicolson est utilisée. Cette réduction a également pu être effectuée sur le modèle bidomaine (com, 2006; Torabi Ziaratgahi *et al.*, 2014), où la méthode de Crank-Nicolson a été également utilisée afin de garantir une meilleure stabilité. Cette opération de réduction sur le modèle bidomaine est très similaire à celui observé pour le monodomaine. Le système original d'EDP et d'ODE non linéaires sont réduits à des ODE non linéaires et à des EDP linéaires, qui sont ensuite résolues de manière séquentielle. Bien évidemment, la différence avec la résolution du problème monodomaine est la présence d'une EDP supplémentaire qu'il convient d'intégrer. Les différents documents de la littérature montre qu'il y a effectivement convergence de cette méthode lors de son application aux modèles monodomaine et bidomaine. Pour cela, les erreurs L_2 sont déterminées entre les solutions obtenues à l'aide de pas de temps et de maillages différents, et une solution de référence calculée à l'aide de la formulation radiale du problème. Les limites de cette approche vient du fait qu'une solution numérique a été utilisée en tant que solution de référence, avec des géométries idéalisées, et on ne peut pas supposer que les propriétés de convergence de ces méthodes soient toujours vraies sur des expériences plus réalistes. Or, il paraît difficilement réalisable de se passer de cette technique, notamment lorsqu'il s'agit du modèle bidomaine, car pour obtenir une solution analytique à ce problème, il convient d'utiliser un modèle simplifié, entraînant également une limite forte sur les tests de convergence.

Une deuxième méthode nommée SDIRK2, également très répandue, a été utilisée dans les travaux de Torabi Ziaratgahi *et al.* (2014). Un des problèmes soulevé dans cette étude est que la méthode de découpage introduite dans com (2006) est stable pour des pas de temps Δt grand lorsque le paramètre θ du θ-schéma vaut 1, mais il existe des restrictions sur Δt lorsque le schéma de Crank-Nicolson est utilisé, afin de réduire les oscillations non-physiques. C'est pour cette raison que Torabi Ziaratgahi *et al.* (2014) a décidé d'utiliser une méthode L-stable, permettant de supprimer les oscillations non-physiques apparues avec la méthode précédente. Les méthodes SDIRK font parties des méthodes L-stables les plus simples. En particulier, la méthode SDIRK du second ordre définie par le tableau de Butcher 3.2 est utilisée.

γ	γ	0
1	$1-\gamma$	γ
	$1-\gamma$	γ

Tableau 3.2 – Tableau de Butcher pour la méthode de SDIRK2.

où $\gamma = \frac{2-\sqrt{2}}{2}$.

Pour un problème générale de la forme de (3.1), la méthode SDIRK détermine la solution u au temps t_{n+1} grâce à la valeur de u au temps t_n, en effectuant les itérations (3.21) et (3.22) :

$$U^{(1)} - \Delta t \gamma f(U^{(1)}, t^{n,\gamma}) = u^n \quad , \tag{3.21}$$

$$u_{n+1} - \Delta t \gamma f(u_{n+1}, t_{n+1}) = u_n + \Delta t(1-\gamma) f(U^{(1)}, t^{n,\gamma}) \quad , \tag{3.22}$$

où $t^{n,\gamma} = t^n + \gamma \Delta t$. En général l'utilisation de cette méthode est un peu plus compliqué, mais ceci est facilité par le fait que le système discrétisant le modèle bidomaine est linéaire. De plus la formulation matricielle du problème est identique au cours de toute la simulation, pour un pas de temps Δt donné, il n'y a donc pas de reconstruction nécessaire à chaque itération. Une analyse de la stabilité a été effectuée, afin de comparer les différences de comportement entre les méthodes de Crank-Nicolson et SDRIK2, permettant de dire que les facteurs d'amplification ont des valeurs absolues inférieures à 1 pour tout $\Delta t > 0$, ce qui signifie que les deux méthodes sont linéairement inconditionnellement stable. C'est deux méthodes tendent à supprimer les oscillations non-physiques qui ont pu être observées lors de simulations effectuées avec des maillages, des modèles de cellules, des conditions initiales différentes (Whiteley, 2006). L'étude de Torabi Ziaratgahi *et al.* (2014) a permis de mettre en évidence différents scénarios montrant ces oscillations. Il a été notamment démontré que la méthode de Crank-Nicolson est moins stable que la méthode SDIRK2 a pas de temps Δt et paramètres de modèles identiques. En général, ces oscillations apparaissent lors de la phase de dépolarisation de la membrane, car une forte perturbation du potentiel de membrane a lieu. Il confirme également que la méthode Euler implicite est inconditionnellement stable, puisque aucune oscillations n'intervient.

Ainsi plusieurs de méthodes existent pour résoudre les problèmes monodomaine et bidomaine, et beaucoup d'études ont été faites afin d'améliorer la stabilité de ces méthodes, tout en évitant de devoir sacrifier le temps de résolution en prenant des pas de temps Δt plus petits. La méthode qui va être présentée dans mon travail de thèse ne se base pourtant pas sur ces méthodes de Crank-Nicolson ou SDIRK. J'ai décidé de prendre les méthodes dites explicites, censées être moins stables, puisqu'elles ont besoin d'un pas de temps plus petit pour éviter les phénomènes d'oscillations non-physiques, dont nous avons discuté plut tôt. Le but est de parallèliser ces méthodes, afin de diminuer le différentiel de temps de calcul engendré par l'utilisation de méthodes explicites. L'interêt est de pouvoir utiliser des pas de temps plus fins, afin d'augmenter la précision des résultats des problèmes monodomaine et bidomaine, sans augmenter de façon drastique le temps de résolution. Dans ce qui va suivre, je vais introduire des techniques de parallélisations permettant de remplir cet objectif.

3.1.3 La méthode pararéel

La parallélisation des équations différentielles par une décomposition de l'intervalle de temps fut pour la première fois proposée par Lions (Lions, 2001), faisant suite aux nombreux efforts antérieurs pour développer des méthodes multi-grilles en espace-temps (Bastian *et al.*, 1991; Hackbusch, 2013). En effet, l'algorithme pararéel peut-être réécrit comme une méthode multi-grille en temps à deux niveaux (Gander et Vandewalle, 2007). La principale motivation de cette méthode est de pouvoir effectuer des simulations sur des intervalles de temps réalistes, d'où le nom de la méthode : pararéel, mais également de pouvoir effectuer de très longues simulations, afin d'obtenir une solution de très bonne qualité. Une analyse détaillée de la convergence de cet algorithme peut être trouvée dans (Gander et Vandewalle, 2007). Dans la suite, nous allons présenter la méthode comme elle a été décrite dans sa première version, puis nous détaillerons deux autres versions de cette méthode présentée dans (Aubanel, 2011) comme étant plus efficaces que la version originale.

3.1.3.1 Description générale

Considérons le problème d'évolution général aux dérivées partielles suivant :

$$\frac{\partial u}{\partial t} + Au = f \text{ avec } t \in [0, T] \quad , \tag{3.23}$$

avec comme condition initiale :

$$u(t = 0) = u_0 \quad , \tag{3.24}$$

et des conditions aux limites que nous n'expliciterons pas ici.

L'algorithme pararéel est défini comme utilisant deux opérateurs de propagation. On introduit l'opérateur $G(t_2, t_1, u_1)$ qui effectue une approximation grossière de $u(t_2)$ de la solution de (3.23) avec la condition initiale $u(t_1) = u_1$, alors que l'opérateur $F(t_2, t_1, u_1)$ permet de déterminer une approximation plus précise de $u(t_2)$.

On découpe uniformément l'intervalle $[0, T]$ en sous-intervalles $[T_n, T_{n+1}]$, puis on résout en parallèle à l'aide de l'opérateur fin F, sur chacun des sous-intervalles, l'équation :

$$\frac{\partial u_n}{\partial t} + Au_n = f_n \text{ où } f_n = f|_{[T_n, T_{n+1}]} \quad , \tag{3.25}$$

avec comme condition initiale :

$$u_n(t = T_n) = \lambda_n \quad , \tag{3.26}$$

où les λ_n sont des fonctions pour $n = 0, ..., N-1$ ($\lambda_0 = u_0$) et vérifiant les mêmes conditions aux limites que u.

L'algorithme commence avec une approximation initiale U_n^0, avec $n = 0, 1, ..., N$ au temps $t_0, t_1, ..., t_N$ donnés, grâce à un calcul en séquentiel de $U_{n+1}^0 = G(t_{n+1}, t_n, U_n^0)$, avec $U_0^0 = u_0$, puis on effectue pour chaque itération $k = 0, 1, 2, ...$ la correction suivante, après avoir effectué des simulations, en parallèle, avec le propagateur fin F :

$$U_{n+1}^{k+1} = G(t_{n+1}, t_n, U_n^{k+1}) + F(t_{n+1}, t_n, U_n^k) - G(t_{n+1}, t_n, U_n^k) \quad . \tag{3.27}$$

On remarquera que quand $k \to \infty$, l'algorithme pararéel (3.27) entraînera la convergence de la série définie par les valeurs de U_n qui satisfont $U_{n+1} = F(t_{n+1}, t_n, U_n)$. Autrement dit, l'approximation aux temps t_n s'achèvera lorsque la précision atteindra celle de l'opérateur F. L'algorithme peut être interprété plus naturellement comme une méthode de correction différée classique (Skeel, 1982).

L'algorithme s'arrête lorsque la condition d'arrêt suivante est respectée :

$$||U_{n+1}^{k+1} - U_{n+1}^k||_\infty < \epsilon \quad , \tag{3.28}$$

où ϵ est la tolérance permise sur l'erreur.

La figure 3.1 présente une version schématisée de la méthode pararéel dite classique ou standard.

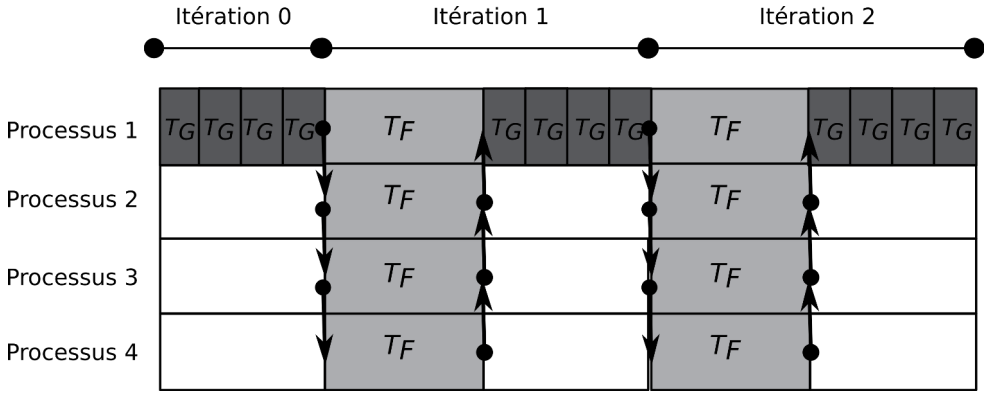

Figure 3.1 – Schéma de l'algorithme parareal classique.

3.1.3.2 Pseudo-code

On présente l'algorithme pararéel, schématisé par la figure 3.1, sous forme d'un pseudo-code (Algorithme 1).

Algorithme 1 : Pseudo-code de la méthode pararéel standard.

Initialisation $\lambda_0^0 \leftarrow u_0$
pour $i = 0,..,N-1$ **faire**
    $\lambda_{i+1}^0 \leftarrow G(\lambda_i^0)$
Résoudre en parallèle pour $i = 0,..,N-1$ les problèmes fins $F(\lambda_i^0)$
$k \leftarrow 0$
tant que *Vrai* **faire**
    $\lambda_0^{k+1} \leftarrow \lambda_0^k$
    pour $i = 0,..,N-1$ **faire**
        Résoudre $G(\lambda_i^{k+1})$
        $\lambda_{i+1}^{k+1} \leftarrow G(\lambda_i^{k+1}) + F(\lambda_i^k) - G(\lambda_i^k)$
    si *convergence* **alors**
        STOP
    Résoudre en parallèle pour $i = 0,..,N-1$ les problèmes fins $F(\lambda_i^{k+1})$
    $k \leftarrow k+1$

3.1.3.3 Interprétation algébrique

La résolution du problème discret (3.23) grâce à l'opérateur fin F peut s'écrire ainsi : pour tout n, trouver $\lambda_n = F(t_n, t_{n-1}, U_{n-1})$ ou encore en utilisant une formulation matricielle, résoudre :

$$\begin{pmatrix} I & 0 & 0 & \cdots & 0 \\ -F & I & 0 & \cdots & 0 \\ 0 & -F & I & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \\ 0 & \cdots & 0 & -F & I \end{pmatrix} \begin{pmatrix} \lambda_0 \\ \lambda_1 \\ \lambda_2 \\ \vdots \\ \lambda_{N-1} \end{pmatrix} = \begin{pmatrix} u_0 \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \quad , \tag{3.29}$$

que l'on peut synthétiser de la façon suivante :

$$M\Lambda = F \quad . \tag{3.30}$$

La nature séquentielle du schéma fin apparaît clairement ici étant donné qu'une inversion standard de ce système triangulaire inférieur implique $\mathcal{O}(N)$ propagations du système :

$$\lambda_n = F(T_{n-1}, T_n, \lambda_{n-1}) \quad . \tag{3.31}$$

Afin d'accélérer la procédure de résolution de ce système, via la méthode pararéel, on définit une suite Λ^k qui converge toujours vers la solution exacte de (3.30) lorsque $k \to \infty$. Le choix de l'opérateur grossier permet de définir la matrice :

$$\tilde{M} = \begin{pmatrix} I & 0 & 0 & \cdots & 0 \\ -G & I & 0 & \cdots & 0 \\ 0 & -G & I & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \\ 0 & \cdots & 0 & -G & I \end{pmatrix} \quad , \tag{3.32}$$

puis de proposer la formulation algébrique de l'algorithme pararéel :

$$\Lambda^{k+1} = \Lambda^k + \tilde{M}^{-1} Res^k \quad , \tag{3.33}$$

où le résidu Res^k est défini par $Res^k = F - M\Lambda^k$. La convergence de l'algorithme pararéel est assurée au moins lorsque la matrice A du problème (3.23) est linéaire définie positive, ce qui conduit à la conclusion suivante : dans une certaine mesure, la matrice $\tilde{M}^{-1}$ peut être considérée comme un préconditionneur de M, dans le sens où la matrice $\tilde{M}^{-1}M$ est proche de l'identité.

3.1.3.4 Stabilité

L'analyse de stabilité (Hairer, 1993) des méthodes standards de résolution des EDO permet de constater que le domaine de stabilité est beaucoup plus restreint, lorsqu'il s'agit d'une méthode d'Euler explicite, alors qu'une méthode de Runge-Kutta d'ordre 4 permet de prendre des pas de temps plus importants, puisque son domaine de stabilité est plus étendue. En ce qui concerne la méthode d'Euler implicite, tous les choix de pas de temps sont possibles. Dans ce contexte, on en déduit que l'opérateur grossier doit nécessairement être une méthode de résolution ayant un domaine de stabilité le plus important possible comme c'est le cas pour Runge-Kutta d'ordre 4 ou encore Euler implicite. Au contraire, l'opérateur fin ne prenant que des pas de temps relativement petits, une méthode d'Euler explicite est suffisant.

Maintenant, nous allons donner certains détails de l'analyse de stabilité de l'algorithme pararéel, dont les détails se trouvent dans les études de Staff et Rønquist (2005); Charbel Farhat (2003). Supposons le système d'équations différentielles autonomes suivant :

$$u' = Au \quad , \quad u(0) = u_0 \quad , \tag{3.34}$$

où A peut-être écrit sous une forme factorisée spectrale, c'est-à-dire $A = VDV^{-1}$, où D la matrice diagonale contenant les valeurs propres de A, et V la matrice contenant les vecteurs

propres de A associés.

On suppose que différentes méthodes peuvent être utilisées pour les opérateurs grossier et fin. Lors de chaque itération grossière de taille ΔT, on réalise des résolutions fines à l'aide d'un pas de temps δt. La première étape afin de déterminer la stabilité de la méthode pararéel consiste à réécrire l'étape de prédiction-correction (3.27) sous la forme :

$$\lambda_n^k = VH(n,k,r(D\delta t),R(D\Delta T))V^{-1}\lambda_0 \quad , \tag{3.35}$$

où n est le nombre de sous-intervalles en temps, k le nombre d'itération de l'algorithme pararéel, H la fonction de stabilité pour la méthode pararéel, r la fonction de stabilité de l'opérateur fin, R la fonction de stabilité de l'opérateur grossier. Dans un premier temps, afin de simplifier le problème, appliquons l'étape de prédiction-correction (3.27) au problème :

$$u' = \mu u \quad , \quad u(0) = u_0 \quad , \mu > 0 \quad . \tag{3.36}$$

On obtient :

$$\lambda_n^k = \bar{r}(\mu\delta t)\lambda_{n-1}^{k-1} + R(\mu\Delta T)\lambda_{n-1}^k - R(\mu\Delta T)\lambda_{n-1}^{k-1} \quad , \tag{3.37}$$

où $\bar{r}(\mu\delta t) = r(\mu\delta t)^s$ est la fonction de stabilité de l'opérateur fin après $s = \frac{\Delta T}{\delta t}$ pas fins de taille δt et $R(\mu\Delta T)$ est la fonction de stabilité de l'opérateur grossier. Afin de simplifier l'écriture, on note $\bar{r} = \bar{r}(\mu\delta t)$ et $R = R(\mu\Delta T)$. On arrangeant (3.37), on obtient :

$$\lambda_n^k = R\lambda_{n-1}^k + (\bar{r} - R)\lambda_{n-1}^{k-1} = R\lambda_{n-1}^k + S\lambda_{n-1}^{k-1} \quad . \tag{3.38}$$

En effectuant les récursions nécessaires, on peut reconnaître un arbre de Pascal, permettant de réécrire (3.38) de la manière suivante :

$$\lambda_n^k = \left(\sum_{i=0}^{n} \binom{n}{i} (\bar{r} - R)^i R^{n-1}\right)\lambda_0 \quad , \tag{3.39}$$

où on peut identifier la fonction de stabilité H :

$$H = \sum_{i=0}^{n} \binom{n}{i} (\bar{r} - R)^i R^{n-1} \quad . \tag{3.40}$$

On peut donc étendre cela au système d'équations (3.34), afin d'obtenir :

$$\lambda_n^k = VH(n,k,r,R)V^{-1}\lambda_0 \quad . \tag{3.41}$$

Au final, la stabilité est atteinte si :

$$\sup_{1\leq n\leq N}\sup_{1\leq k\leq N}|H(n,k,r,R)| \leq 1 \quad , \quad \forall \mu_i, \quad i=1,...,M. \tag{3.42}$$

3.1.3.5 Complexité et performance

Indépendamment des schémas utilisés lors de l'application de la méthode pararéel, la complexité est proportionnelle au nombre de pas de temps requis, entre T_0 et T_N. La complexité est donc égale à $\frac{T_N-T_0}{\delta t} = \frac{N\Delta T}{\delta t}$ fois le le coût élémentaire pour effectuer une étape, notée C_F, qui dépend du solveur utilisé.

On peut effectuer tous les calculs impliquant le solveur fin en parallèle, du moment que l'on a N processeurs. Ainsi, la complexité sur chaque processeur est réduit à $\frac{\Delta t}{\delta t}$ d'où une complexité N fois inférieure à l'implémentation séquentielle de la méthode.

Notons maintenant C_G le coût pour effectuer le solveur grossier, et K le nombre d'itérations nécessaire pour arriver à la convergence de la méthode pararéel (K résolutions pour le solveur grossier et $K-1$ pour le solveur fin). Alors la complexité cumulée de cet algorithme est :

$$KNC_G + (K-1)N\frac{\Delta T}{\delta t}C_F \quad , \tag{3.43}$$

mais la complexité locale, sur chaque processeur, est :

$$KNC_G + (K-1)\frac{\Delta T}{\delta t}C_F \quad , \tag{3.44}$$

qui devrait être comparée avec

$$\frac{T_N - T_0}{\delta t}C_F = N\frac{\Delta t}{\delta t}C_F \quad . \tag{3.45}$$

En assumant que le coût du solveur grossier est relativement négligeable par rapport au coût du solveur fin, on en déduit que l'efficacité du parallélisme, liée à la méthode pararéel, est

égale à :

$$\frac{N}{K-1} \quad . \tag{3.46}$$

L'efficacité totale, autrement dit lorsque le temps de simulation est divisé par un facteur proche du nombre de processeurs utilisé lors de la résolution, est seulement obtenue si le coût du solveur grossier C_G est négligeable par rapport au coût du solveur fin C_F, et si l'algorithme converge avec $K = 2$. Pour les problèmes complexes, le premier point est facile à atteindre, mais pour le deuxième point, cela n'est pas évident. Il est donc à noter que, si le nombre d'itérations nécessaires pour atteindre la convergence est très proche du nombre de processeurs utilisés, l'efficacité tombera à son minimum, et on en conclura que la méthode pararéel ne doit pas être utilisée.

Même si l'efficacité semble être faible, nous pouvons noter un certain nombre de points. Une des premières motivations de cet algorithme est son application à des problèmes où la discrétisation spatiale par décomposition de domaines semble limitée. Une deuxième motivation est la manière dont les processus vont se partager ou encore collecter les différentes tâches, afin d'optimiser au mieux l'efficacité.

Bien entendu, il peut devenir intéressant de coupler les deux parallélisations, si le problème s'y prête. Nous aurons l'occasion de revenir sur ce point plus tard.

3.1.4 Une première variante : l'algorithme gestionnaire/travailleurs

3.1.4.1 Description générale

Comme son nom l'indique, il convient ici de définir les différentes tâches dédiées au gestionnaire et aux travailleurs, ainsi que la manière dont les portions séquentielles et parallèles de l'algorithme vont se chevaucher. A l'itération initiale, le gestionnaire effectue une résolution grossière du problème, en utilisant le pas de temps ΔT, puis envoie l'initialisation à chaque travailleur, dès que celui-ci est disponible. A l'itération suivante, les travailleurs ayant effectués correctement leur résolution fine envoient leurs résultats au gestionnaire. Puis, ce dernier se charge d'appliquer une nouvelle résolution grossière, suivie d'une étape de correction à chaque solution obtenue à la fin des sous intervalles, puis envoie les corrections aux travailleurs. Lors des itérations suivantes, le rôle du gestionnaire consiste à effectuer l'intégration grossière, suivi de la correction et de tester si la convergence a bien eu lieu, ou si une nouvelle itération est nécessaire. Il est possible que la convergence soit atteinte pour un travailleur, sur un sous-intervalle bien précis. Mais pour que la convergence soit acceptée par le gestionnaire, il est obligatoire qu'il y ait convergence sur tous les sous-intervalles précédents.

En faisant une analyse de cet algorithme, on peut s'apercevoir que les résolutions grossière et fine se chevauchent. Ceci est la conséquence du fait que le gestionnaire envoie la solution

initiale de chaque sous-intervalle au travailleur associé, dès que celle-ci est déterminée. On note également que cette version de l'algorithme pararéel traite les itérations $k = 0$, $k = 1$, $k \geq 2$ différemment, car le gestionnaire a différentes tâches suivant le cas dans lequel l'algorithme se trouve.

A l'itération initiale, le gestionnaire effectue l'intégration grossière sur l'intervalle de temps global. Lors de la première itération, le gestionnaire envoie le résultat de sa résolution fine vers le premier travailleur avant de pouvoir effectuer la résolution fine. Pour les itérations suivantes, le gestionnaire attend les résultats provenant des différents travailleurs.

3.1.4.2 Performance

Nous allons brièvement parler des performances théoriques que l'on peut espérer en utilisant cet algorithme. Pour estimer le temps d'exécution de l'algorithme, on néglige les temps de communications et corrections. On obtient pour $k = 1$:

$$t_{(k=1)} = T_c + \frac{T_f}{N} + (1 - \frac{1}{N}) T_c \quad , \tag{3.47}$$

et pour $k \geq 2$:

$$t_{(k\geq 2)} = T_c + k\frac{T_f}{N} + (1 - \frac{2}{N}) T_c \quad , \tag{3.48}$$

où T_c et T_f représentent, respectivement, les temps mis par les solveurs grossier et fin sur l'intervalle de temps global $[0, T]$. Ces différents temps permettent de déduire les gains de temps théoriques, lors de l'utilisation de cet algorithme. Pour $k = 1$:

$$\Psi_{(k=1)} = \frac{N}{(2N-1)r+1} \quad , \tag{3.49}$$

et pour $k \geq 2$:

$$\Psi_{(k\geq 2)} = \frac{N}{2(N-1)r+k} \quad , \tag{3.50}$$

avec r le rapport entre le temps mis par le solveur grossier G et le solveur fin F sur le même intervalle de temps.

3.1.4.3 Pseudo-code

On présente l'algorithme pararéel dans sa version gestionnaire/travailleurs à l'aide du pseudo code suivant. L'algorithme 2 permet de définir le travail attribué à chaque travailleur.

Algorithme 2 : Pseudo-code des processus i (i=1, 2, ..., N-1) - Travailleurs.

Réception de $\tilde{u}_i^0$ venant du processus 0
// Itération 1
$\hat{u}_{i+1}^0 \leftarrow F(\tilde{u}_i^0)$
Envoie de $\tilde{u}_{i+1}^0$ au processus 0 et réception de u_i^1 venant du processus 0
// Itération 2 et supérieure
pour $k = 2, .., N$ **faire**
    $\hat{u}_{i+1}^{k-1} \leftarrow F(u_i^{k-1})$
    Envoie de $\hat{u}_{i+1}^{k-1}$ au processus 0
    Réception de $converge$ venant du processus 0
    si *convergence* **alors**
        STOP
    sinon
        Réception de u_i^k venant du processus 0

L'algorithme 3 permet, quand à lui, de décrire le rôle du gestionnaire.

Algorithme 3 : Pseudo-code du processus gestionnaire.

Initialisation $\tilde{u}_0^0 \leftarrow u_0$
// Itération 0
pour $i = 1,..,N-1$ **faire**
    $\tilde{u}_i^0 \leftarrow G(\tilde{u}_{i-1}^0)$
    Envoie de $\tilde{u}_i^0$ au processus i
$\tilde{u}_N^0 \leftarrow G(\tilde{u}_{N-1}^0)$
// Itération 1
$\hat{u}_1^0 \leftarrow F(u_0^0)$
$u_1^1 \leftarrow \hat{u}_1^0$
pour $i = 1,..,N-1$ **faire**
    Réception de $\hat{u}_{i+1}^0$ venant du processus i
    Envoie de u_i^1 au processus i
    $\tilde{u}_{i+1}^1 \leftarrow G(\tilde{u}_i^1)$
    $u_{i+1}^1 \leftarrow \tilde{u}_{i+1}^1 + \hat{u}_{i+1}^0 - \tilde{u}_{i+1}^0$
// Itérations supérieure à 2
$s \leftarrow 2$
pour $k = 2,..,N$ **faire**
    Réception de $\hat{u}_s^{k-1}$ venant du processus $s-1$
    $converge \leftarrow TRUE$
    Envoie de u_k^s au processus s
    pour $i = s,..,N-1$ **faire**
        $\tilde{u}_{i+1}^k \leftarrow G(\tilde{u}_i^k)$
        $u_{i+1}^k \leftarrow \tilde{u}_{i+1}^k + \hat{u}_{i+1}^{k-1} - \tilde{u}_{i+1}^{k-1}$
        si $converge$ **alors**
            $converge \leftarrow |u_{i+1}^k - u_{i+1}^{k-1}| < \epsilon$
            si $!converge$ **alors**
                $s \leftarrow i+1$
        Envoie de $converge$ au processus i
        si $!converge$ **alors**
            Envoie de u_i^k au processus i

3.1.5 Une deuxième variante : l'algorithme pararéel distribué

Nous allons maintenant nous intéresser à une variante de cet algorithme, présentée par (Aubanel, 2011), qui permet de mieux diviser la charge de travail entre les différents processeurs, et ainsi améliorer encore l'efficacité de la méthode pararéel.

3.1.5.1 Description

Pour optimiser au mieux la répartition des tâches entre les différents processeurs, (Aubanel, 2011) a proposé un algorithme distribuant la résolution grossière sur tous les processeurs.
A l'itération 0, le processeur 0 effectue la première étape de la simulation grossière $\tilde{U}_1^0 \leftarrow G(\tilde{U}_0^0)$ et envoie le résultat au processeur 1, qui continue la résolution grossière, puis envoie le résultat au processeur 2, ainsi de suite.
A chaque itération $k > 0$, chaque processeur (id) actif effectue une intégration fine $\hat{U}_{id+1}^{k-1} \leftarrow F(U_{id}^{k-1})$, reçoit l'état initial U_{id}^k, effectue la simulation grossière $\tilde{U}_{id+1}^k \leftarrow G(\tilde{U}_{id}^k)$, suivie de l'étape de correction $U_{id+1}^k \leftarrow \tilde{U}_{id+1}^k + \hat{U}_{id+1}^{k-1} - \tilde{U}_{id+1}^{k-1}$. Ce résultat est ensuite envoyé au processeur $(id+1)$, si $(id+1) < N$.

A chaque itération k, si le processeur $(id-1)$ a convergé, alors le processeur (id) effectue un test de convergence après avoir déterminé la solution de sa propagation grossière. S'il n'y a pas convergence, alors il le fera dans la prochaine itération, à partir d'un état initial plus précis. Sinon son travail est terminé.

Cet algorithme est schématisé dans la figure 3.2, en montrant les 3 premières itérations, sur 4 processeurs.

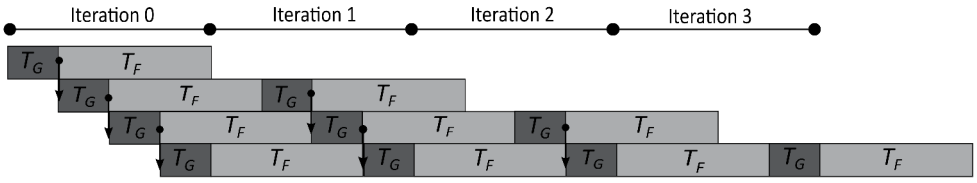

Figure 3.2 – Représentation schématique de l'algorithme pararéel distribué.

3.1.5.2 Performance

En reprenant les mêmes notations que dans la section 3.1.4.2, le temps d'exécution de cet algorithme est déterminer de la manière suivante :

$$t_{dist} = T_c + \frac{k}{N}(T_f + T_c) \quad . \tag{3.51}$$

Dans ce cas, le speedup est donné par

$$\Psi = \frac{N}{Nr + k(1+r)} \quad . \tag{3.52}$$

Théoriquement, les performances de cet algorithme sont meilleures que ceux obtenues avec les autres variantes. La distribution équitable des différentes charges permet de mieux équili-

brer chaque processus, pour au final, diminuer le temps de calcul.

3.1.5.3 Pseudo-Code

L'algorithme pararéel dans sa version distribuée est décrite grâce au pseudo code de l'algorithme 4 :

Algorithme 4 : Pseudo-code de la méthode pararéel distribuée.

$convergeNext \leftarrow$ FALSE
si *id==0* **alors**
  $\widetilde{U}_0^0 \leftarrow u_0$
  $\widetilde{U}_1^0 \leftarrow G(\widetilde{U}_0^0)$
  Envoie $\widetilde{U}_1^0$ au processeur 1
  $\hat{U}_1^0 \leftarrow F(\widetilde{U}_0^0)$
  $U_1^1 \leftarrow \hat{U}_1^0$
  $converge \leftarrow$ TRUE
  Envoie de $converge$ et U_1^1 au processeur 1
sinon
  Réception de $\widetilde{U}_{id}^0$ du processeur $(id-1)$
  $\widetilde{U}_{id+1}^0 \leftarrow G(\widetilde{U}_{id}^0)$
  si *id != N-1* **alors**
    Envoie de $\widetilde{U}_{id+1}^0$
$U_{id}^0 \leftarrow \widetilde{U}_{id}^0$
pour $k = 0, .., N$ **faire**
  $\hat{U}_{k-1}^{id+1} \leftarrow F(\widetilde{U}_{id}^{k-1})$
  si $convergeNext$ **alors**
    $converge \leftarrow$ TRUE
    $U_{id+1}^k \leftarrow \hat{U}_{id+1}^{k-1}$
    Envoie $converge$ et U_{id+1}^k au processeur $(id+1)$
    EXIT
  Réception de $converge$ et U_{id}^k du processeur $(id-1)$
  $\widetilde{U}_{id+1}^k \leftarrow G(U_{id}^k)$
  $U_{id+1}^k \leftarrow \widetilde{U}_{id+1}^k + \hat{U}_{id+1}^{k-1} - \widetilde{U}_{id+1}^{k-1}$
  si $(converge\ ET |U_{id+1}^k - U_{id+1}^{k-1}| \geq \epsilon)$ **alors**
    $converge \leftarrow$ FALSE
    $convergeNext \leftarrow$ TRUE
  si *id != N-1* **alors**
    Envoie de $converge$ et U_{id+1}^k au processeur $(id+1)$
  si $converge$ **alors**
    EXIT

3.1.6 Choix de l'algorithme

Dans l'étude faite par (Aubanel, 2011), divers tests ont été menés sur les trois algorithmes présentés, notamment en effectuant une prédiction sur les efficacités des différents algorithmes. On entend par efficacité, le rapport entre le speed-up Ψ et le nombre de processus N utilisés lors du lancement du programme. De cette étude, il est ressorti que l'algorithme le plus efficace présenté ici, en théorie, est la version distribuée, avec une efficacité supérieure de 45 % à 67 % à l'algorithme d'origine. En appliquant les différentes méthodes à un même problème, la résolution de l'équation de la chaleur, les résultats obtenus ont conforté les résultats théoriques.

De plus, l'avantage de cet algorithme distribué ne s'arrête pas à cette seule efficacité. En divisant toutes les tâches de façon équitable, il est largement portable sur GPU (sections 4.2 et 4.4), en plus d'un développement avec MPI (section 4.1). Les autres variantes sont bien plus difficilement portables sur cette technologie, car il convient de différencier deux parties dans le programme : la résolution grossière et les résolutions fines. Nous verrons dans la suite pourquoi il est nécessaire que les tâches du programme soient équitablement réparties entre tous les processus.

3.1.7 Conclusion

j'ai, dans ce chapitre, rapidement expliqué comment procéder pour intégrer des équations différentielles ordinaires. Puis, j'ai abordé le principe de parallélisation suivant l'échelle de temps, une décomposition du domaine peu connue et utilisée : la méthode pararéel. Mais, on ne veut pas s'arrêter en si bon chemin. Certains problèmes sont définis également dans l'espace, grâce à des équations aux dérivées partielles. Il peut donc devenir intéressant de paralléliser aussi cet espace là, et pourquoi pas, coupler les deux parallélisations, afin de pouvoir intégrer des équations sur des géométries fines, et cela durant de longues périodes.

3.2 Parallélisation en espace

3.2.1 Introduction

Dans cette partie qui va suivre, une rapide introduction aux équations aux dérivées partielles (EDP) est effectuée. A l'instar des EDO, les EDP sont utilisées pour modéliser mathématiquement certains phénomènes physiques, ce qui les rend incontournables. On les retrouve notamment en mécanique des fluides, en théorie de l'électromagnétisme, dans la simulation aéronautique, dans les prévisions météorologiques, mais également dans les finances. Dans la plupart des cas, il est quasiment impossible de déterminer une solution analytique à ces problèmes. On fait donc appel à certaines méthodes afin de calculer une solution discrète au problème. Pour notre étude, je vais me restreindre à méthode des différences finies, ainsi qu'à la méthodes des éléments finis, que je vais très brièvement présenter ici. Je reviendrai sur les discrétisations de chaque problème dans la partie 3. Pour un aperçu plus complet, j'invite le lecteur à se tourner vers des cours très complets (Lucquin, 1998; Olver, 2014; Manet, 2015; Borsuk, 2010).

Ensuite, je présenterai les bases des méthodes de décomposition de domaines. Elles serviront notamment à paralléliser le problème sur l'échelle de l'espace, ce qui permettra de réduire le coût des calculs. Ces méthodes permettent de définir un problème global à partir de problèmes locaux plus petits. Le but est donc de pouvoir exécuter ces différents problèmes locaux en parallèle, afin d'accélérer la résolution du problème global. Elles permettent également d'avoir une plus grande souplesse dans la discrétisation du problème. Il est donc possible d'utiliser une taille de maillage des ordres d'éléments finis différents, ou des modèles ayant des physiques différentes dans chacun des sous-domaines. Ces différentes méthodes sont, le plus souvent, regroupées en deux grandes familles : les méthodes avec recouvrement, et les méthodes sans recouvrement.

Ce chapitre n'a pas pour but de détailler toutes les méthodes de résolution des équations aux dérivées partielles et de décomposition de domaine existantes. Néanmoins, pour le lecteur, il existe un certain nombre de documents pouvant satisfaire sa curiosité (Smith *et al.*, 2004; Chan et Mathew, 1994; Tallec, 1994). Les méthodes présentées ici permettront de mieux comprendre les moyens mis en œuvre dans la suite, pour la résolution de nos différents problèmes.

3.2.2 Résolution des équations aux dérivées partielles

3.2.2.1 Approximation par des différences finies

Une approximation par des différences finies permet de déterminer la valeur d'une fonction u solution au problème d'EDP à une localisation x_0, grâce aux valeurs se trouvant à des localisations assez proches de x_0.

L'approximation la plus simple est donnée par le calcul de la dérivée, qui est une approxima-

tion de degré 1 :

$$u'(x) \approx \frac{u(x+h) - u(x)}{h} \quad . \tag{3.53}$$

En effet, si u est différentiable en x, alors $u'(x)$ est la limite, quand $h \to 0$ du quotient ci-dessus. Plus h sera petit, meilleure sera l'approximation de la dérivée. Il est également possible que le pas h soit positif ou négatif, auquel cas, l'approximation sera respectivement explicite ou implicite.

On suppose maintenant que la fonction u est deux fois différentiable, dans ce cas, on peut écrire la formule de Taylor du premier ordre :

$$u(x+h) = u(x) + u'(x)h + \frac{1}{2}u''(\xi)h^2 \quad . \tag{3.54}$$

En arrangeant l'équation 3.54, on obtient :

$$\frac{u(x+h) - u(x)}{h} - u'(x) = \frac{1}{2}u''(\xi)h \quad . \tag{3.55}$$

Alors l'erreur commise par l'approximation du premier ordre 3.53 peut être bornée par un multiple du pas de temps h

$$\left| \frac{u(x+h) - u(x)}{h} - u'(x) \right| \leq C|h| \quad , \tag{3.56}$$

où $C = max\left(\frac{1}{2}|u''(\xi)|\right)$. L'approximation est effectivement de premier ordre, car l'erreur est proportionnelle à h.

De façon similaire, il est possible d'utiliser des approximations d'ordres supérieurs, notamment si on veut approcher la dérivée seconde u'', qui peut intervenir dans l'EDP. Pour cela, on effectue des développements de Taylor d'ordre plus élevés. Dans le cas des différences finies, le maillage est vu comme un ensemble de points, que l'on appelle également nœuds. Cet ensemble de points permet de quadriller le domaine de définition de la fonction u. Le calcul des valeurs à chacun de ces nœuds permet de déterminer la solution discrète. Evidemment, des nœuds se trouvent sur le bord du domaine, ce qui permet d'imposer des conditions aux limites, permettant de décrire la dynamique de la fonction u sur ces bords. L'écart entre chaque nœud voisin est décrit par le pas du maillage, qui va dépendre de la direction, même si en général, le même pas est utilisé dans toutes les directions.

Dans le cas des différences finies, pour faciliter la discrétisation des différents opérateurs, on situe les nœuds du maillage sur une grille, dont les directions sont parallèles aux axes de

référence. Cela signifie que les mailles sont forcément de forme rectangulaire, ce qui entraîne l'utilisation d'une géométrie assez simple pour définir le domaine.

Je reviendrai plus en détails sur la méthode des différences finies dans la partie 3, qui seront sur les différents modèles. Je détaillerai bien évidemment le schéma de discrétisation pour chaque modèle, permettant de le résoudre numériquement.

3.2.2.2 Approximation par des éléments finis

Le but de cette section n'est pas de faire un cours détaillé, ni de re-démontrer certains aspects de la méthode des éléments finis. Pour cela, le lecteur peut trouver des ouvrages très complets sur ce sujet, comme Lucquin (1998); Ciarlet (2007). Je vais expliquer les principes de cette méthode, ainsi que ses différences avec la méthode présentée dans la section 3.2.2.1. Ensuite, je présenterai rapidement l'outil FreeFem, permettant de résoudre des EDP avec cette méthode.

Principes généraux

La méthodes des éléments finis consiste à trouver une solution fiable, de manière discrète, à un problème posé. Comme pour la méthodes des différences finies, le domaine de définition du problème est scindé en plusieurs morceaux, que l'on appellera mailles, de telle sorte que cet espace soit polygonal par morceaux.

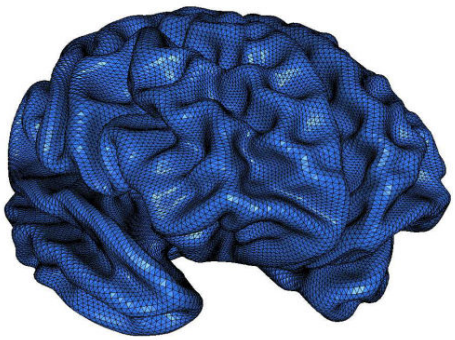

Figure 3.3 – Maillage du domaine représentant un cerveau (Lancaster *et al.*, 2007).

Cette méthode, contrairement à celle des différences finies, permet l'utilisation de mailles tétraédriques. Cela entraîne un meilleur découpage des géométries complexes, contrairement à un maillage régulier, comme on a pu le voir dans la méthode précédente. Le plus souvent, la

forme des mailles est triangulaire, mais rien empêche d'utiliser des formes de mailles plus exotiques. On trouvera, par exemple Figure 3.3, le maillage d'un domaine représentant le cerveau. Le maillage d'un tel domaine serait compliqué dans le cas des différences finies. Il n'est pas non plus nécessaire d'utiliser un maillage régulier, c'est-à-dire identique sur tout l'espace. Il est possible de resserrer le maillage à certains endroits d'intérêt, afin d'obtenir une solution plus fine, soit en utilisant des tailles de mailles de plus en plus petites, ou encore, en prenant des degrés plus élevés de polynômes, comme par exemple de degré 2 pour des côtés ou arêtes ayant une courbure.

Le maillage permet également de définir les éléments finis, qui sont les données d'un domaine géométrique, d'un espace de fonctions (polynomiales, en général) et d'un ensemble de formes linéaires sur cet espace, également appelées degrés de liberté.

Après obtention de ces éléments finis, on écrit la formulation variationnelle (ou formulation faible) du problème, que l'on cherche à résoudre sur un domaine Ω, ce qui revient à un problème du type :

$$\text{Trouver } u \in V \text{ tel que } a(u,v) = l(v), \quad \forall v \in V \quad ,$$

où V est un espace de Hilbert, a une forme bilinéaire et l une forme linéaire. On va ainsi chercher à déterminer une approximation de u, en utilisant le maillage de Ω, avec lequel on va définir un espace V_h de dimension finie N_h. Le problème discret sera donc le suivant :

$$\text{Trouver } u_h \in V_h \text{ tel que } a(u_h,v_h) = l(v_h), \quad \forall v_h \in V_h \quad ,$$

Ensuite, en utilisant une base $(\phi_1,...,\phi_{N_h})$ de V_h, on décompose u_h sous la forme d'une combinaison linéaire de cette dernière :

$$u_h = \sum_{i=1}^{N_h} \lambda_i \phi_i \quad . \tag{3.57}$$

En injectant, la relation 3.57 dans le problème discret, et en appliquant les propriétés de linéarité de a, on obtient :

$$\text{Trouver } \lambda_1,...,\lambda_{N_h} \text{ tels que } \sum_{i=1}^{N_h} \lambda_i a(\phi_i,v_h) = l(v_h), \quad \forall v_h \in V_h \quad .$$

Maintenant, prenons $v_h = \phi_j$ une fonction de la base de V_h, alors on a :

$$\text{Trouver } \lambda_1,...,\lambda_{N_h} \text{ tels que } \sum_{i=1}^{N_h} \lambda_i a(\phi_i,\phi_j) = l(\phi_j), \quad \forall j = 1,...,N_h \quad ,$$

ce qui est équivalent à résoudre le système linéaire suivant :

$$\begin{pmatrix} a(\phi_1,\phi_1) & \dots & a(\phi_{N_h},\phi_1) \\ \vdots & & \vdots \\ a(\phi_1,\phi_{N_h}) & \dots & a(\phi_{N_h},\phi_{N_h}) \end{pmatrix} \begin{pmatrix} \lambda_1 \\ \vdots \\ \lambda_{N_h} \end{pmatrix} = \begin{pmatrix} l(\phi_1) \\ \vdots \\ l(\phi_{N_h}) \end{pmatrix} \quad , \tag{3.58}$$

ou encore, en simplifiant l'écriture :

$$A\Lambda = L \quad . \tag{3.59}$$

Pour éviter que la résolution de ce système soit trop lourde, on cherche au maximum à rendre la matrice A creuse. Pour cela, il est conseillé de prendre une base $(\phi)_i$ dont les fonctions seront nulles le plus souvent, hormis sur quelques mailles. Cela aura pour conséquence d'annuler la plupart des termes $a(\phi_i,\phi_j)$, car correspondant à des fonctions ϕ_i et ϕ_j de supports disjoints. De plus, on réorganisera la matrice A de telle façon qu'elle soit une matrice à bande, dont la largeur sera la plus petite possible. Dans ce cas, on peut tirer avantage de la structure particulière de la matrice pour utiliser moins de mémoire. Il est inutile de stocker des zéros, on se restreint donc à mettre en mémoire uniquement les bandes de la matrice. On utilise alors soit un stockage profil, soit un stockage morse, ou encore un stockage bande, ne nécessitant que le stockage de deux ou trois tableaux à une dimension suivant le type de stockage. Ce type de stockage s'applique également au système obtenu par la méthode des différences finies, comme nous le verrons dans la partie III.

Une des principales difficultés réside dans le calcul des intégrales, afin de déterminer les termes $a(\phi_i,\phi_j)$ et $l(\phi_j)$. C'est pour cela que l'on prend des polynômes comme base, car les calculs des intégrales sont soit immédiats, ou alors en utilisant des formules de quadrature. De plus, la vitesse de convergence est directement liée au degré des polynômes.

FreeFem

Pour résoudre un problème d'EDP numériquement à l'aide des éléments finis, on peut développer son propre code en différents langages tels que C/C++, Python ou Fortran. Mais il existe également un freeware FreeFem++ (Hecht, 2012), développé au Laboratoire Jacques-Louis Lions de l'Université Pierre et Marie Curie an 1987 sous le nom de MacFem, et que l'on retrouve sur les principaux systèmes d'exploitation comme Windows, Mac OS X, ou Unix. Ce logiciel est caractérisé par un langage de script, hérité du C++ à partir de 1992.

Il permet de créer des maillages ainsi que de résoudre des équations aux dérivées partielles, en utilisant des bibliothèques de solveurs linéaires tels que : LU, Cholesky, gradient conjugué, GMRES, etc. Il est également possible de l'interfacer avec d'autres logiciels comme Medit ou Gnuplot. En plus de résoudre des EDP, il est possible de créer des maillages, de les sauver dans

un fichier, puis de les lire. On peut aussi adapter le maillage suivant les points d'intérêts de la solution.

Pour tout-autre information, le lecteur est invité à consulter la documentation officielle de FreeFem, très bien détaillée, où l'on peut trouver différents exemples d'utilisations.

3.2.3 Décompositions de domaine

Supposons un ensemble Ω connexe, ainsi que le problème défini par l'équation aux dérivées partielles suivant :

$$\left\{ \begin{array}{rcll} \mathscr{L}u & = & f & \text{dans} \quad \Omega \\ u & = & 0 & \text{sur} \quad \partial\Omega \quad . \end{array} \right. \tag{3.60}$$

Pour pouvoir appliquer les différentes méthodes de décomposition de domaine, on partitionne le domaine Ω en n (avec $n \geq 2$) sous-domaines Ω_i tels que :

$$\Omega = \bigcup_{i=1}^{n} \Omega_i \quad . \tag{3.61}$$

A partir de cette partition, ces méthodes sont capables de résoudre les différents problèmes locaux, plus petits, sur les sous-domaines Ω_i, permettant ainsi de déterminer la solution globale, sur le domaine Ω. Dans ce qui va suivre, nous allons voir les différentes méthodes liées au type de partitionnement choisi : avec ou sans recouvrement.

3.2.3.1 Méthodes de décomposition de domaine avec recouvrement

Lorsque la décomposition du domaine est avec recouvrement, alors chaque sous-domaine recouvre une partie des sous-domaines voisins (voir 3.4). Autrement dit, les intersections entre les différents voisins Ω_i et Ω_j ne sont pas obligatoirement vides.

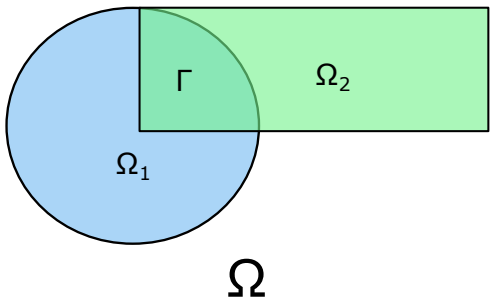

Figure 3.4 – Partitionnement du domaine Ω en deux sous-domaines Ω_1, Ω_2, avec recouvrement.

Pour résoudre ce type de problème, on peut utiliser des méthodes dites de Schwarz, introduite en 1870. Ce sont des méthodes itératives, qui effectuent une résolution alternée de l'équation différentielle dans chacun des sous-domaines, en ajoutant des conditions aux limites de Dirichlet, afin de prendre en compte les sous-domaines voisins. En résolvant chacun des problèmes, on obtient de nouvelles valeurs pour les bords, puis on réitère le processus jusqu'à atteindre le critère de convergence ϵ. Deux versions de cette méthode existent, la première consiste à prendre comme conditions de Dirichlet, les valeurs données par le domaine voisin à Ω_i, à l'itération précédente. Cette dernière est appelée méthode de Schwarz additive (voir Algorithme 5). Comme les résolutions dans chacun des deux domaines sont indépendantes, cette méthode est facilement parallélisable, est un rapprochement peut être fait avec la méthode de Jacobi.

Algorithme 5 : Pseudo-code de la méthode de Schwarz additive.

1 **tant que** $||u_1^{n+1} - u_1^n|| > \epsilon$ **faire**

2 Résoudre :

$$\begin{cases} \mathscr{L} u_1^{n+1} = f & \text{dans} \quad \Omega_1 \\ u_1^{n+1} = 0 & \text{sur} \quad \partial\Omega_1 \cap \partial\Omega \\ u_1^{n+1} = u_2^n & \text{sur} \quad \Gamma = \partial\Omega_1 \cap \partial\Omega_2 - \partial\Omega \end{cases}$$

3 Résoudre :

$$\begin{cases} \mathscr{L} u_2^{n+1} = f & \text{dans} \quad \Omega_2 \\ u_2^{n+1} = 0 & \text{sur} \quad \partial\Omega_2 \cap \partial\Omega \\ u_2^{n+1} = u_1^n & \text{sur} \quad \Gamma \end{cases}$$

La deuxième version de la méthode, appelée Schwarz multiplicatif, prend comme conditions de Dirichlet, les dernières valeurs d'interface calculées par le sous-domaine voisin (voir Algorithme 6). A nouveau, on continue le processus jusqu'à atteindre la convergence, quantifiée par la valeur ϵ.

Algorithme 6 : Pseudo-code de la méthode de Schwarz multiplicative.

1 **tant que** $||u_1^{n+1} - u_1^n|| > \epsilon$ **faire**

2 Résoudre :

$$\begin{cases} \mathscr{L} u_1^{n+1} = f & \text{dans} \quad \Omega_1 \\ u_1^{n+1} = 0 & \text{sur} \quad \partial\Omega_1 \cap \partial\Omega \\ u_1^{n+1} = u_2^n & \text{sur} \quad \Gamma = \partial\Omega_1 \cap \partial\Omega_2 - \partial\Omega \end{cases}$$

3 Résoudre :

$$\begin{cases} \mathscr{L} u_2^{n+1} = f & \text{dans} \quad \Omega_2 \\ u_2^{n+1} = 0 & \text{sur} \quad \partial\Omega_2 \cap \partial\Omega \\ u_2^{n+1} = u_1^{n+1} & \text{sur} \quad \Gamma \end{cases}$$

La convergence de la méthode additive peut être améliorée, en généralisant les conditions de

transfert entre les différents sous-domaines. Les conditions aux limites de Dirichlet sur les interfaces, généralement écrites de la manière suivantes : $u_i^{n+1} = u_j^n$, sont remplacées par des conditions de Robin, ce qui permet de déterminer l'algorithme 7.

Algorithme 7 : Pseudo-code de la méthode de Schwarz additive avec conditions aux limites de Robin.

1 **tant que** $||u_1^{n+1} - u_1^n|| > \epsilon$ **faire**

2 Résoudre :

$$\begin{cases} \mathscr{L} u_1^{n+1} = f & \text{dans } \Omega_1 \\ u_1^{n+1} = 0 & \text{sur } \partial\Omega_1 \cap \partial\Omega \\ \dfrac{\partial u_1^{n+1}}{\partial n_1} + \alpha u_1^{n+1} = \dfrac{\partial u_2^n}{\partial n_1} + \alpha u_2^n & \text{sur } \Gamma = \partial\Omega_1 \cap \partial\Omega_2 - \partial\Omega \end{cases}$$

3 Résoudre :

$$\begin{cases} \mathscr{L} u_2^{n+1} = f & \text{dans } \Omega_2 \\ u_2^{n+1} = 0 & \text{sur } \partial\Omega_2 \cap \partial\Omega \\ \dfrac{\partial u_2^{n+1}}{\partial n_2} + \alpha u_2^{n+1} = \dfrac{\partial u_1^n}{\partial n_2} + \alpha u_1^n & \text{sur } \Gamma \end{cases}$$

Lorsque $\alpha > 0$, la méthode ne demande plus qu'un partitionnement avec recouvrement pour atteindre la convergence, qui se fait plus rapidement que les méthodes standards, en prenant des bonnes de valeurs de α, qui se déterminent le plus souvent de façon empirique.

3.2.3.2 Méthodes de décomposition de domaine sans recouvrement

Maintenant, nous partitionnons notre domaine en sous-domaines, de telle sorte que l'intersection entre les voisins se limite à l'interface (voir Figure 3.5).

$$\forall 1 \leq i, j \leq n, \quad \Omega_i \cap \Omega_j = \partial\Omega_i \cap \partial\Omega_j \tag{3.62}$$

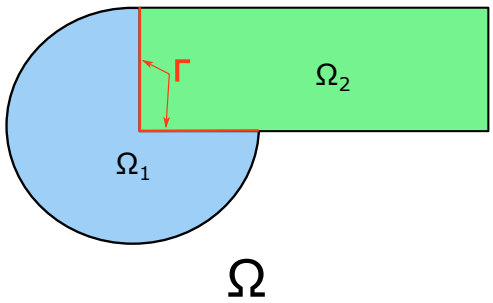

Figure 3.5 – Partitionnement du domaine Ω en deux sous-domaines Ω_1, Ω_2 , sans recouvrement.

Les méthodes utilisées pour résoudre les problèmes dans cette configuration, sont basées sur le complément de Schur. Elles cherchent à déterminer une valeur sur l'interface, en rajoutant des conditions d'égalité sur la solution, ainsi que sur les dérivées normales aux interfaces. Comme pour les méthodes de Schwarz, il existe deux méthodes permettant de résoudre ce type de problèmes : les méthodes de Schur primale et duale.

Méthode de Schur primale

La méthode consiste à résoudre le problème suivant 3.63 dans chacun des sous-domaines (ici, on se restreint à deux sous-domaines, donc $i \in \{1,2\}$). Le but est de déterminer la valeur de u_Γ sur l'interface, vérifiant la relation : $\dfrac{\partial u_1}{\partial n_1} = -\dfrac{\partial u_2}{\partial n_2}$.

$$\begin{cases} \mathscr{L} u_i = f & \text{dans} \quad \Omega_i \\ u_i = 0 & \text{sur} \quad \partial\Omega_i \cap \partial\Omega \\ u_i = u_\Gamma & \text{sur} \quad \Gamma \end{cases} . \tag{3.63}$$

La méthode permet de réarranger le système en regroupant les degrés de liberté liés au sous-domaine Ω_1, ensuite au sous-domaine Ω_2, et enfin à l'interface Γ. Ce tri permet au final, d'arriver au système matriciel ordonné suivant :

$$\begin{pmatrix} A_{11} & 0 & A_{1\Gamma} \\ 0 & A_{22} & A_{2\Gamma} \\ A_{\Gamma 1} & A_{\Gamma 2} & A_{\Gamma\Gamma} \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_\Gamma \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ f_\Gamma \end{pmatrix} . \tag{3.64}$$

Les inconnues u_i avec $i \in \{1,2\}$ sont les inconnues intérieures au domaine i et u_Γ sont les inconnues interface. Les matrices A_{ii} avec $i \in \{1,2\}$ représentent la discrétisation de l'opérateur linéaire $\mathscr{L}$ pour les inconnues intérieures du domaine i. Les matrices $A_{i\Gamma}$ et $A_{\Gamma i}$ avec $i \in \{1,2\}$ représentent, respectivement, l'influence du bord sur l'intérieur et l'inverse de cette dernière. Le second membre définie par f_i est la contribution pour les inconnues intérieures, alors que f_Γ est la contribution pour les inconnues interfaces.

Si les matrices A_{11} et A_{22} sont inversibles, on peut se ramener à un système linéaire à résoudre sur u_Γ

$$S u_\Gamma = b \quad \text{avec} \quad \begin{cases} S = A_{\Gamma\Gamma} - A_{\Gamma 1} A_{11}^{-1} A_{1\Gamma} - A_{\Gamma 2} A_{22}^{-1} A_{2\Gamma} \\ b = f_\Gamma - A_{\Gamma 1} A_{11}^{-1} f_1 - A_{\Gamma 2} A_{22}^{-1} f_2 \end{cases} . \tag{3.65}$$

Une méthode itérative préconditionnée permet de résoudre ce système, puis on en déduit la

solution sur chaque sous-domaine Ω_i, en utilisant la relation suivante :

$$u_i = A_{ii}^{-1}(f_i - A_{i\Gamma}u_\Gamma) \quad . \tag{3.66}$$

Méthode de Schur duale

Cette fois, on résout dans chaque sous-domaine un problème couplé à des conditions aux limites de type Neumann (voir le problème 3.67). Dans cette exemple, on se restreint toujours à prendre deux sous-domaines, c'est-à-dire $i \in \{1,2\}$.

$$\left\{\begin{array}{rcll} \mathscr{L}u_i & = & f & \text{dans } \Omega_i \\ u_i & = & 0 & \text{sur } \partial\Omega_i \cap \partial\Omega \\ \dfrac{\partial u_i}{\partial n_i} & = & (-1)^i\lambda & \text{sur } \Gamma \end{array}\right. \tag{3.67}$$

Pour déterminer λ, on utilise la condition définie sur l'interface Γ : $u_1 = u_2$. De plus, comme les normales sortantes des deux sous-domaines sont de signe contraire sur l'interface, on doit rajouter le terme $(-1)^i$. Ici, chaque sous-domaine numérote les noeuds de l'interface, qui n'est plus commune aux sous-domaines. C'est pour cela que la continuité de la solution $u_1 = u_2$ sur l'interface est considérée comme une contrainte.

Contrairement à la méthode primale, on décompose les éléments $A_{\Gamma\Gamma}$ et f_Γ de telle sorte que chaque sous-domaines apporte sa contribution. On peut donc écrire que $A_{\Gamma\Gamma} = A^1_{\Gamma\Gamma} + A^2_{\Gamma\Gamma}$ et $f_\Gamma = f^1_\Gamma + f^2_\Gamma$. Ainsi, le système à résoudre est le suivant :

$$\begin{pmatrix} A_{11} & A_{1\Gamma} & 0 \\ A_{\Gamma 1} & A^1_{\Gamma\Gamma} + A^2_{\Gamma\Gamma} & A_{2\Gamma} \\ 0 & A_{\Gamma 2} & A_{22} \end{pmatrix} \begin{pmatrix} u_1 \\ u_\Gamma \\ u_2 \end{pmatrix} = \begin{pmatrix} f_1 \\ f^1_\Gamma + f^2_\Gamma \\ f_2 \end{pmatrix} \quad . \tag{3.68}$$

Ensuite, nous dupliquons les degrés de liberté au niveau de l'interface en posant $u^1_\Gamma = u^2_\Gamma = u_\Gamma$, puis on répercute cela sur les contributions de chaque sous-domaine. Le système s'écrit alors :

$$\begin{pmatrix} A_{11} & A_{1\Gamma} & 0 & 0 \\ A_{\Gamma 1} & A^1_{\Gamma\Gamma} & 0 & 0 \\ 0 & 0 & A^2_{\Gamma\Gamma} & A_{2\Gamma} \\ 0 & 0 & A_{\Gamma 2} & A_{22} \end{pmatrix} \begin{pmatrix} u_1 \\ u^1_\Gamma \\ u^2_\Gamma \\ u_2 \end{pmatrix} = \begin{pmatrix} f_1 \\ f^1_\Gamma + f^2_\Gamma - A^2_{\Gamma\Gamma}u_\Gamma - A_{\Gamma 2}u_2 \\ f^1_\Gamma + f^2_\Gamma - A^1_{\Gamma\Gamma}u_\Gamma - A_{\Gamma 1}u_1 \\ f_2 \end{pmatrix} \quad . \tag{3.69}$$

De plus, on définit le vecteur λ sur l'interface par :

$$\begin{aligned} \lambda &= -(f_{\Gamma^2} - A^2_{\Gamma\Gamma} u_\Gamma - A_{\Gamma 2} u_2) \\ &= f^1_\Gamma - A^1_{\Gamma\Gamma} u_\Gamma - A_{\Gamma 1} u_1 \quad . \end{aligned} \tag{3.70}$$

On peut ainsi, isoler chaque sous-système, en scindant le système 3.68 en sous-systèmes caractérisant les sous-domaines. La résolution du système global revient à intégrer un problème quasiment identique au problème global dans chaque sous-domaine Ω_i :

$$A_i U_i = F_i - C_i \lambda \quad , \tag{3.71}$$

où les différentes matrices sont définies de la manière suivante :

$$A_i = \begin{pmatrix} A_{ii} & A_{i\Gamma} \\ A_{\Gamma i} & A^i_{\Gamma\Gamma} \end{pmatrix}, \quad F_i = \begin{pmatrix} f_i \\ f^i_\Gamma \end{pmatrix}, \quad U_i = \begin{pmatrix} u_i \\ u^i_\Gamma \end{pmatrix}, \quad C_i = \begin{pmatrix} 0 \\ (-1)^i Id \end{pmatrix} \quad .$$

Id représente la matrice identité. De plus, la relation de continuité entre les solutions d'interfaces $u^1_\Gamma = u^2_\Gamma$ s'écrit :

$$C_1^T U_1 + C_2^T U_2 = 0 \quad . \tag{3.72}$$

Ceci conduit à la résolution d'un problème de type point-selle :

$$\begin{pmatrix} A_1 & 0 & C_1 \\ 0 & A_2 & C_2 \\ C_1^T & C_2^T & 0 \end{pmatrix} \begin{pmatrix} U_1 \\ U_2 \\ \lambda \end{pmatrix} = \begin{pmatrix} F_1 \\ F_2 \\ 0 \end{pmatrix} \quad . \tag{3.73}$$

3.2.4 Conclusion

Dans ce chapitre, j'ai introduit deux des principales méthodes de résolution des EDP, puis nous avons vu d'une façon globale, les méthodes permettant d'intégrer un problème d'EDP à l'aide de la décomposition de domaine. Ces décompositions permettent de paralléliser les calculs, en découpant l'espace en au moins deux parties. Chaque sous-partie doit résoudre son sous-problème, puis les valeurs aux interfaces des sous-domaines sont échangées afin que la continuité soit respectée. Comme pour les méthodes introduites dans le chapitre 3.1, nous avons besoin de technologies permettant de tirer pleinement partie de l'aspect multi-coeurs des ordinateurs actuels. Dans les chapitres suivants, le mode de fonctionnement de ces différentes technologies est présenté.

4 Technologies parallèles en informatique

4.1 Introduction au Message Passing Interface (MPI)

4.1.1 Introduction

Afin de de pouvoir tirer parti de l'aspect parallèle de ces différents algorithmes, il est obligatoire de passer par des technologies existantes. Plusieurs outils sont mis à notre disposition, pour pouvoir exploiter des multiprocesseurs ou des réseaux d'ordinateurs, en les faisant communiquer. L'avantage est de diviser les différents calculs, habituellement réalisés séquentiellement, pour diminuer le temps de résolution. Un de ces outils, que nous allons présenter brièvement dans ce chapitre, est MPI (Message Passing Interface), conçu par plusieurs organisations en 1994, pour sa première version, alors que sa troisième version est en cours. C'est une norme définissant une bibliothèque de fonctions, que l'on peut utiliser avec différents langages, tels que C, Fortran, ou encore Python. Un autre outil permettant de réaliser cette tache est OpenMP, qui est également une bibliothèque multi-plateforme et utilisable avec de nombreux langages tels que C, C++ et Fortran. Contrairement à MPI, OpenMP repose sur le modèle de mémoire partagée et est optimisé pour un usage des processeurs multi-coeurs. Or lorsque l'on souhaite utiliser une répartition des calculs sur plusieurs machines permettant ainsi une parallélisation massive, MPI est bien plus recommandé, même si le temps de développement d'un programme MPI est plus complexe qu'un programme OpenMP. Sachant que le but de mon travail est d'effectuer des simulations sur le cluster de calculs de Rhenovia, l'usage de MPI est recommandé par rapport à celui de OpenMP. Ainsi, ce chapitre va permettre d'introduire MPI. Le but n'est pas de faire une description exhaustive de cette technologie, mais uniquement une introduction rapide, permettant une meilleure compréhension de ce qui va suivre. Le lecteur intéressé par plus de détails, est renvoyé à différents ouvrages de référence et documentations officielles (Saha *et al.*, 2006; Snir, 1998; Quinn, 2003; Pacheco, 1996).

4.1.2 Définitions

Lors de la programmation d'un code en séquentiel, celui ci est exécuté par un seul processus sur un processeur physique de l'ordinateur. Les opérations sont ainsi effectuées les unes après les autres, en utilisant des variables et des constantes qui sont allouées dans la mémoire dédiée au processus. Ce modèle est présenté Figure 4.1.

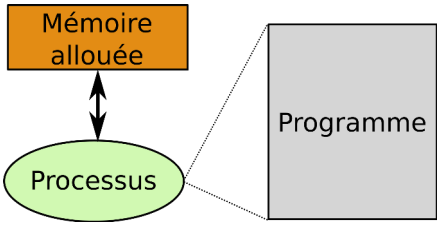

Figure 4.1 – Modèle simplifié de programmation séquentiel.

Pour une programmation sur le modèle MPI, il y a échange de messages entre différents processus, permettant à chacun d'entre eux d'exécuter une partie différente du code (Figure 4.2). Les processus ont accès à une mémoire allouée individuelle, où toutes les variables du programme sont privées. Pour faire transiter les données entre deux ou plusieurs processus, une partie du programme doit effectuer un appel à des sous-programmes bien particuliers.

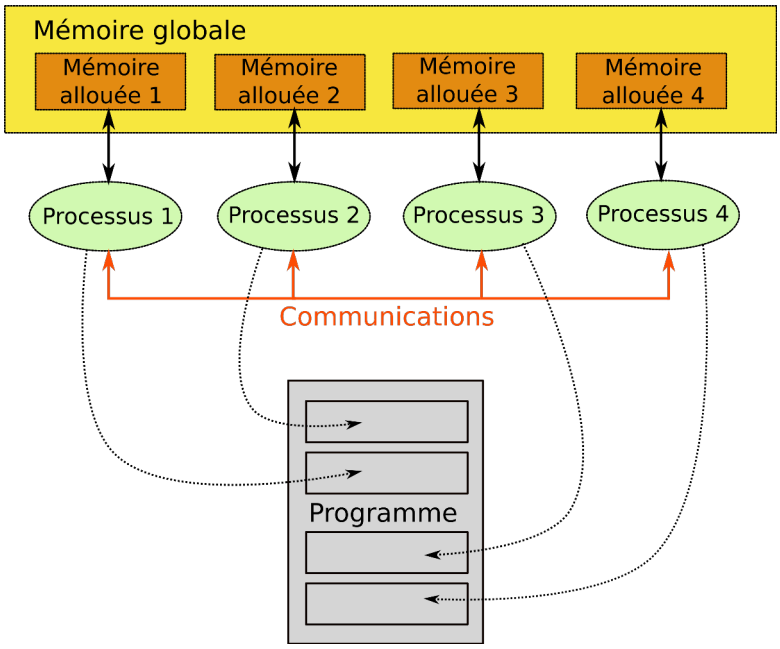

Figure 4.2 – Modèle de programmation MPI.

La technique employée pour effectuer ce parallélisme est le SPMD (Single Program Multiple

Data), qui permet d'exécuter un même programme sur tous les processus. C'est certainement la technique de programmation la plus utilisée, et supportée par toutes les machines.

4.1.3 Principes

Si N processus sont utilisés lors de l'exécution du programme, alors N copies de l'exécutable sont créées avec des identifiants de processus allant de $0, 1, ..., N-1$, et elles sont opérées de façon asynchrones, pour ensuite se synchroniser lors de l'échange des messages.

La communication entre les différents processus s'effectue en envoyant des messages entre eux. Un processus peut envoyer un message à un ou plusieurs autres processus, qui devront se charger de réceptionner les données. Il est donc obligatoire, dans ce cas, de distinguer la source des receveurs.

Les données qui transitent entre les différents processus peuvent être de nature différente, telles que des variables scalaires, des tableaux, des objets, etc. Mais en plus de ces données, un message envoyé doit contenir plusieurs autres informations :

- l'identifiant caractérisant le processus émetteur,
- le type de la donnée,
- la longueur de la donnée,
- l'identifiant désignant le(s) processus récepteur(s).

Si le programme est bien codé, toutes les opérations s'achèvent à la fin, sans qu'il y ait un blocage au niveau des communications, c'est-à-dire un processus qui attend un message, alors qu'il n'y a plus rien à attendre. Bien évidemment, il ne suffit pas de porter un code séquentiel en MPI, un travail d'adaptation doit être fait, mais également d'optimisation. On doit chercher à minimiser les communications entre les différents processus, qui peuvent vite devenir coûteuses et au final ralentir considérablement le temps d'exécution du programme.

4.1.4 Environnement

MPI est seulement une norme définissant des fonctions, permettant d'exploiter au mieux le caractère multiprocesseur des ordinateurs. Il existe plusieurs implémentation en C de cette norme, comme OpenMPI, MPICH, mais également mpi4py en Python, qui est un interfaçage de la bibliothèque MPI avec Cython.

mpi4py, utilisé pour la résolution de nos problèmes, permet une certaine souplesse dans le développement du code, par rapport au langage C. On utilise les avantages du Python, que ça soit au niveau du développement, mais aussi des différentes bibliothèques existantes pour le calcul matriciel.

Lors du développement d'un programme MPI en C, il est nécessaire d'inclure le fichier `"mpi.h"` dans l'en-tête. Il faut ensuite initialiser l'environnement, en utilisant `MPI_INIT()`,

et au contraire pour le désactiver, on fait appel à la fonction `MPI_FINALIZE()`. Si on utilise mpi4py, l'import du module suffit à initialiser l'environnement.

Pour que MPI puisse effectuer des opérations, il doit faire appel à des communicateurs, dont le plus utilisé est celui par défaut : `MPI_COMM_WORLD`, qui comprend tous les processus actifs. On peut se représenter (voir Figure 4.3) cela, en imaginant un grand ensemble contenant des objets étant assimilés aux processus. Grâce à ce communicateur, il est possible de déterminer le nombre de processus qu'il gère, en utilisant `MPI_COMM_SIZE()`, mais aussi le rang d'un processus (son identifiant), compris entre 0 et `MPI_COMM_SIZE()`-1, avec `MPI_COMM_RANK()`. Un exemple d'utilisation de ces fonctions est donné avec le code 4.1 écrit en Python.

Code 4.1 – Exemple d'utilisation de l'environnement mpi4py.

```
# Initialisation de l'environnement MPI
from mpi4py import MPI

# Récupération du communicateur
comm = MPI.COMM_WORLD

# Nombre de processus
size = comm.Get_size()

# Rang du processus
rank = comm.Get_rank()

# Print
print "Processus ",rank, " sur ",size
```

4.1.5 Communication point à point

La communication point à point se met en place entre deux processus identifiés par leurs rangs respectifs. L'un des processus joue le rôle de l'émetteur (ou encore source), et l'autre celui de récepteur (ou destinataire). Il y a un échange d'informations entre ces deux processus, dont la nature va varier, en fonction de la nature des données à communiquer. Comme il a été dit dans la section 4.1.3, le message envoyé est contenu dans une enveloppe contenant plusieurs autres informations, dont certaines sont indispensables pour une communication correcte. On rappelle brièvement les différents constituants de cette enveloppe :

- le rang du processus source,
- le rang du processus récepteur,
- l'étiquette du message, aussi appelé tag,
- le communicateur définissant le groupe de processus ainsi que le contexte de communication.

Le tag reste néanmoins optionnel, mais fortement recommandé lorsque beaucoup de communications sont échangées entre deux processus. Ce paramètre permet de faire un tri entre les

différentes données, notamment lorsque des communications non-bloquantes sont utilisées. En ce qui concerne les données échangées, elles sont typées, autrement dit, elles peuvent être des entiers, des réels, ou d'autres objets.

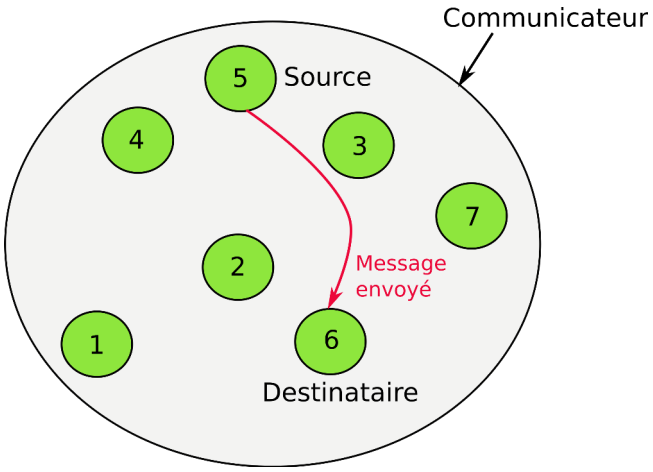

Figure 4.3 – Représentation d'un communicateur avec un envoi de message entre les processus 5 et 6.

Pour l'envoi d'un message, le processus source doit faire appel à la fonction `MPI_Send`, en indiquant le type, la taille des données, et le rang du processus récepteur. Dans le même temps, le processus destinataire doit faire appel à `MPI_Recv`, avec le type et la taille des données identiques à ceux données par le processus émetteur. De plus ce processus doit indiquer le rang du processus source. C'est la présence des ces deux fonctions au sein des différents processus, qui permet d'établir une bonne communication entre eux, et ainsi permettre une bonne réception du message par le destinataire. On peut trouver dans le code 4.2,un exemple d'utilisation écrit en Python à l'aide du module `mpi4py`.

Code 4.2 – Utilisation des communications avec mpi4py.

```
# Initialisation de l'environnement MPI
from mpi4py import MPI

# Définition du comunicateur
comm = MPI.COMM_WORLD
rank = comm.Get_rank()

# Envoi d'un objet "data" du processus 0 au processus 1
if rank == 0:
    data = {'a': 25.5, 'b': 16}
    comm.send(data, dest=1, tag=21)
    print 'Message envoyé : ',data
elif rank == 1:
    data = comm.recv(source=0, tag=21)
    print 'Message recu : ',data
```

Un point important est que l'utilisation de ces fonctions peut entraîner des communications bloquantes. Cela signifie que le programme associé au processus destinataire continuera uniquement lorsque ce dernier aura bien reçu le message qui lui est destiné, et que l'espace mémoire des données sera disponible. Finalement, le programme global pourra se retrouver totalement bloqué.

Si on veut éviter les communications bloquantes, il existe des communications dites non-bloquantes. Elles permettent au programme de continuer son exécution, en laissant en arrière plan les transferts se réaliser. Dans ce cas, on initialise la réception et l'envoi des données, ce qui aura pour but de retourner immédiatement une valeur de "status", vue comme une référence à cette communication. Pendant ce temps, il est possible de lancer d'autres fonctions de calcul, ce qui est impossible, avec les communications bloquantes.

4.1.6 Communications collectives

Certaines communications sont susceptibles d'être collective, autrement dit un processus peut avoir besoin de communiquer certaines de ses données à tous les autres processus. A partir de là, il devient forcément redondant de devoir faire un envoi séparé à tous processus, qui eux-même devront intégrer une réception associée à cette communication. C'est dans ce but que MPI contient un ensemble de fonctions, permettant d'établir certains types de communications collectives, tout en minimisant le code à écrire. Nous allons dans cette section, faire un rapide aperçu des fonctions existantes, et présenter les plus populaires.

Une communication collective concerne toujours tous les processus du communicateur. Lorsqu'une communication collective est utilisée, la gestion des identifiants permet de distinguer les processus (les rangs), de manière totalement transparente et à la charge du système. Le développeur n'a donc pas à se soucier, ici, de les écrire explicitement. De plus, ces communications ne peuvent pas interférer avec les communications point-à-point.

Il y a trois types de fonction, permettant d'assurer les communication collectives :

1. Les synchronisations globales, effectuées avec des barrières : `MPI_BARRIER()`. Tous les processus doivent atteindre cette barrière pour qu'ils puissent continuer.

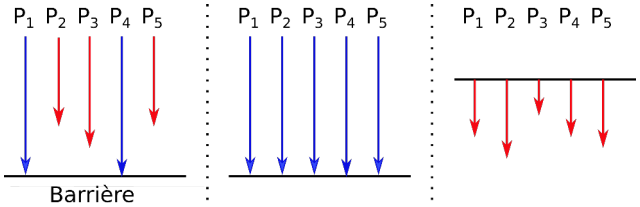

Figure 4.4 – Représentation d'une barrière MPI

2. Les transferts de données :

   – `MPI_BCAST()` consiste à envoyer la même information à tous les processus.

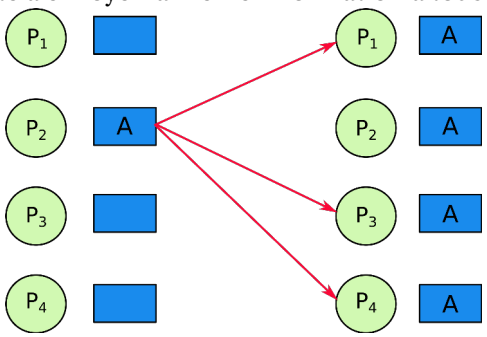

Figure 4.5 – Représentation du mode de fonctionnement de la fonction MPI_BCAST().

   – `MPI_SCATTER()` permet de fractionner une donnée importante en sous-ensembles, et de les répartir entre les processus (utilisé dans la distribution des tableaux, par exemple).

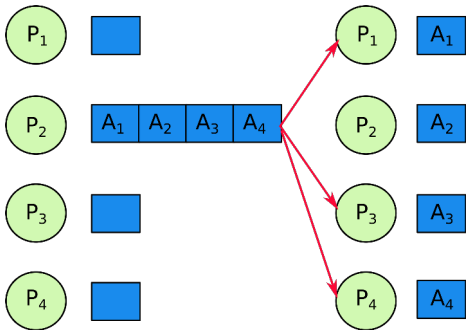

Figure 4.6 – Représentation du mode de fonctionnement de la fonction MPI_SCATTER().

   – `MPI_GATHER()` effectue le travail inverse de la fonction précédente. Chaque processus dispose d'un morceau du tableau, et l'un d'entre eux va reformer la donnée globale.

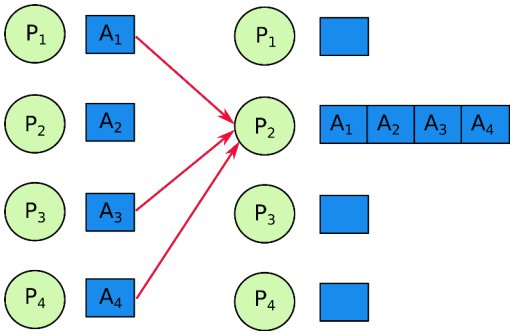

Figure 4.7 – Représentation du mode de fonctionnement de la fonction MPI_GATHER().

   – `MPI_ALLGATHER()` est un mixage des fonctions `MPI_GATHER()` et `MPI_BCAST()`.

3. Gestion des communications, et opérations sur les données : `MPI_REDUCE()`, `MPI_ALLREDUCE()`, `MPI_SUM()`, etc.

Le manuel de référence de MPI décrit en détail toutes ces fonctions, en donnant quelques exemples d'utilisation.

4.1.7 Conclusion

Dans ce chapitre, nous avons vu les bases du fonctionnement de la bibliothèque MPI, permettant de développer des applications parallèles sur des systèmes multiprocesseurs, ou encore des clusters de calcul. Bien évidemment, nous n'avons fait qu'effleurer toutes les possibilités offertes par cette bibliothèque. Vu son mode de fonctionnement, il est possible de découper son programme en sous-parties, qui peuvent être traitées indépendamment les unes des autres, sur des processus différents. Pour pouvoir paralléliser sur de nombreux processus, on se trouve dans l'obligation de quasiment posséder un cluster de calcul, possédant des noeuds, et pouvant communiquer entre eux. L'intérêt est d'appliquer, cette technologie, aux différents algorithmes parallèles présentés dans les sections 3.1 et 3.2. Mais, MPI n'est pas la seule technologie permettant de paralléliser un programme. Avec l'essor du calcul GPU, il est maintenant possible de paralléliser massivement les calculs, avec un minimum de matériel. Dans les chapitres suivants, nous allons décrire cette nouvelle technologie apportant une nouvelle vision dans le développement des programmes massivement parallèles, qui est bien différente de celle utilisée ici.

4.2 Architecture des Graphical Processing Units (GPU)

4.2.1 Introduction

Dans le but d'augmenter la précision des solutions, tout en diminuant le temps de calcul lors de la résolution d'un problème, on peut se tourner vers l'architecture GPU (Graphical Processing Unit) qui permet de faire du parallélisme massif. Les GPU sont des circuits qui ont été initialement conçus pour l'affichage graphique (notamment 2D puis 3D dans le cas des jeux vidéos). Ces circuits ont connu un essor fulgurant, avec une évolution vers des circuits dédiés au calcul hautement parallèle. Suivant le GPU utilisé, on est capable de traiter plusieurs centaines, voire plusieurs milliers de réels en simple ou double précision. On qualifie en général la programmation sur GPU de many-core, et qui correspond à du SIMD (Single Instruction Multiple Data), contrairement au SPMD (Single Program Multiple Data) vu dans la section 4.1. Comme le montre la Figure 4.8 , à la fin de l'année 2013, la puissance de calcul d'un CPU était de 500 GFLOP/s, alors que celle d'un GPU était de 5,5 TFLOP/s, soit un facteur 100.

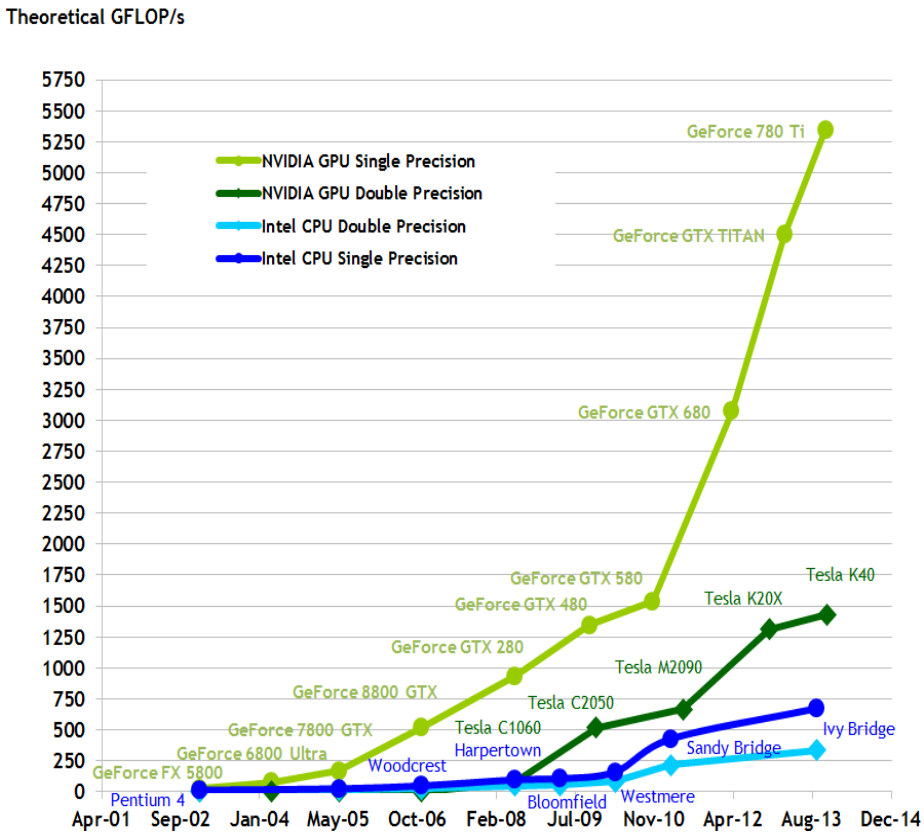

Figure 4.8 – Evolution des performances en GFlops des CPU et GPU, d'après (NVIDIA, 2015).

L'architecture GPU est totalement différente à celle du CPU (Central Processing Unit), ce qui se traduit par certains changements dans la manière d'écrire un programme permettant de résoudre un problème. Il ne suffit pas de transposer un code développé sur CPU, puis de le compiler sur GPU afin d'obtenir les bonnes solutions du problème. Afin d'aider le programmeur à développer son code, la société NVIDIA a introduit l'environnement Compute Unified Device Architecture (CUDA) en 2007 (voir section 4.4). A l'heure actuelle, CUDA est l'environnement de développement des applications sur GPU le plus utilisé, même si des alternatives existent, tel que OpenCL (Open Computing Language) géré par le groupe Khronos. OpenCL a l'avantage d'être multi-plateforme et portable. Plusieurs travaux (Karimi *et al.*, 2010) ont étudié les différences de performances entre les deux technologies, concluant que CUDA prenait l'avantage sur OpenCL.

Afin de mieux comprendre la manière de développer sur GPU, l'architecture GPU et ses principales différences avec le CPU sont détaillées dans la suite. Ensuite, je présenterai rapidement la méthodologie pour développer des applications sur GPU, pour finalement appliquer ces différents points à notre problème. Bien entendue, nous invitons le lecture intéressé par ce sujet est invité à se tourner vers des ouvrages plus complets (Kirk et Hwu, 2010; Tsuchiyama *et al.*, 2010).

4.2.2 Architecture des GPU

Sur une vue d'ensemble du couple CPU/GPU, comme représenté Figure 4.9, on remarque que le CPU utilise le GPU comme un coprocesseur permettant d'effectuer des calculs adaptés à la parallélisation massive. Le GPU comme le CPU sont multi-coeurs et suivent une hiérarchie de mémoires. Nous reviendrons plus en détail sur cette hiérarchie de la mémoire qui est cruciale dans le développement sur GPU, dans la section 4.2.3

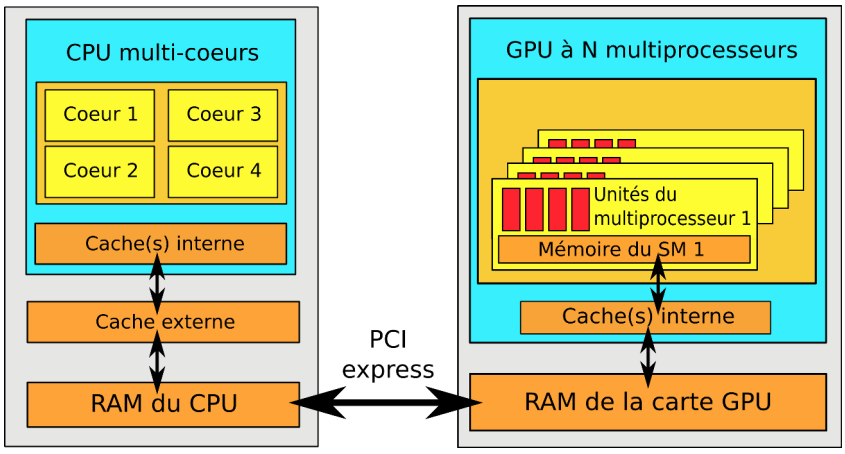

Figure 4.9 – Vue d'ensemble d'un couple CPU/GPU.

L'architecture des GPU est en constante évolution, pouvant apporter à chaque génération de profonds changements, alors que certaines autres caractéristiques évoluent très peu au cours du temps. En général, les unités constituant un GPU sont beaucoup plus simples que celles d'un CPU traditionnel. En 2012, les GPU les plus performants était constitués d'environ 500 cœurs, tandis que le meilleur CPU devait se contenter de 8 cœurs. Trois ans après, les meilleurs CPU doivent toujours se contenter d'en moyenne 8 cœurs, alors que les GPU ont vu leurs nombres de cœurs multiplié par dix. Dans les faits, les coeurs sont regroupés en grappes, appelées multi-processeurs (SM), de 32 cœurs (pour l'architecture Fermi) et jusqu'à 192 cœurs (pour l'architecture : Kepler). Il est très important de garder en mémoire ce nombre maximum de cœurs par SM, car nous verrons plus tard que ces coeurs doivent effectuer la même tâche pour obtenir une performance maximale.

Sur l'architecture Fermi, chacun de ces SM est capable d'effectuer 32 opérations de flottants ou d'entiers par coup d'horloge sur des nombres de 32 bits, ou alors 16 opérations sur des nombres de 64 bits. Depuis l'apparition de l'architecture Kepler, le SMX a prit la relève du SM, et au passage abandonne le système de double cadencement de Fermi, ce qui permet notamment de multiplier par deux la fréquence des unités de calculs.

Une autre différence majeure entre CPU et GPU, que l'on peut également mentionner est la latence de la mémoire. Au niveau du CPU, les tâches ti sont exécutées l'une après l'autre, avec un petit temps de latence entre chaque traitement, afin de fournir les données au processus. Comme ces temps de latence sont plus long sur GPU, il convient d'effectuer beaucoup de tâches, pour obtenir un haut débit. Ces tâches sont également appelées "threads" dans le langage CUDA. Ceci permet d'optimiser le temps d'exécution, en masquant le temps d'attente des données par le calcul des autres tâches. La Figure 4.10 illustre la latence faible d'un CPU et le haut débit d'un GPU.

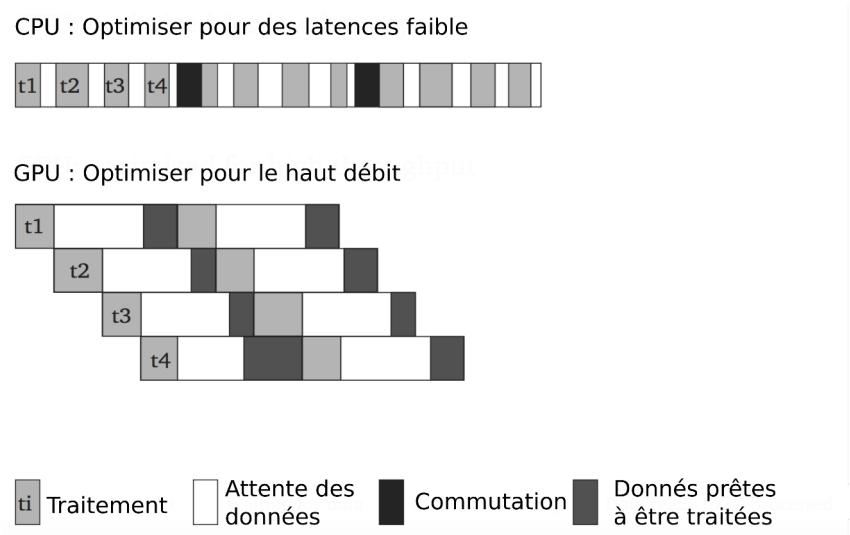

Figure 4.10 – Comparaison entre la latence faible d'un CPU et le haut débit d'un GPU.

4.2.3 Hiérarchie de la mémoire

La hiérarchie de la mémoire des GPU est très différente de celle des CPU. La mémoire des GPU est constituée des registres, de la mémoire locale, de la mémoire partagée, de la mémoire cache et de la mémoire globale.

Chaque thread peut accéder à son propre registre, dont l'accès est très rapide. Il est donc recommandé de faire appel à cette mémoire le plus souvent possible. Comme pour les registres, chaque thread peut accéder à une mémoire locale, qui est en général beaucoup plus lente que celles des registres. Suivant l'architecture, la mémoire locale peut-être une partie de la mémoire globale, ou de la mémoire cache L1. Lors de la compilation, la mémoire locale est automatiquement utilisée lorsque tous les registres sont déjà utilisés. Il est donc conseillé d'optimiser l'utilisation des registres.

La mémoire partagée est commune à un ensemble de threads formant ce qu'on appelle : un bloc. Cette mémoire permet de stocker et d'échanger temporairement des données afin d'augmenter encore les performances de calcul. La mémoire partagée bénéficie à beaucoup de problèmes, mais elle n'est pas toujours utilisable. Certains algorithmes tirent partie de cette mémoire, alors que d'autres tirent partie d'une hiérarchie de cache comme utilisée par les CPU. Il est possible que des conflits d'accès interviennent, entraînant une sérialisation des accès mémoires.

La mémoire globale est, en terme de capacité, la plus importante. Un facteur limitant de cette mémoire est la bande passante limitée, pouvant diminuer les performances de calculs. Comme pour la mémoire partagée, on peut avoir affaire à des conflits d'accès, si des threads d'une même grappe veulent faire appel à des accès mémoires non-voisines.

La Figure 4.11 schématise la hiérarchie des différentes mémoires dans un GPU.

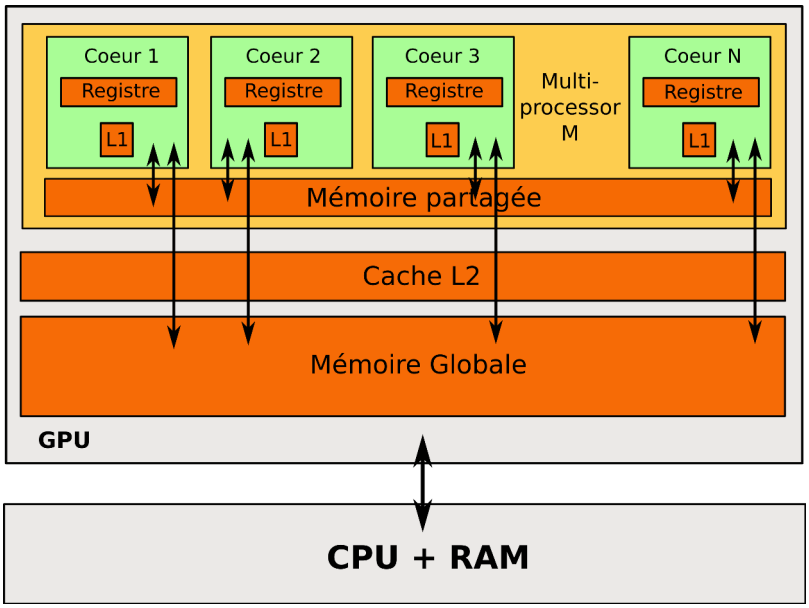

Figure 4.11 – Hiérarchie des différentes mémoires d'un GPU.

4.3 Conclusion

L'architecture GPU (Graphical Processing Unit) a été brièvement présentée, en montrant les différences avec l'organisation au sein d'un CPU (Central Processing Unit). L'aspect multi-parallèle de cet outil a été abordé, qui permet grâce à une bonne programmation de diminuer le temps de calcul de certains problèmes facilement parallélisables. Dans le chapitre qui va suivre, je vais présenter l'environnement CUDA, permettant de développer des programmes pouvant être déployés sur GPU.

4.4 La technologie Compute Unified Device Architecture (CUDA)

4.4.1 Introduction

Après une présentation des GPU d'un point de vue architectural, nous allons introduire l'environnement CUDA. Cela permettra de développer sur GPU des programmes, pouvant résoudre un problème de manière massivement parallèle. L'effet de cette programmation sera d'entraîner une diminution importante du temps de calcul. CUDA est un environnement propriétaire de NVIDIA, il est donc techniquement utilisable uniquement sur des cartes graphiques provenant du fondeur NVIDIA. En parallèle, il existe un autre environnement nommé OpenCL donc la marque est déposée par Apple et utilisé sous licence le groupe Khronos. Son mode de fonctionnement est proche de celui de CUDA, à ceci près que le vocabulaire diffère d'un environnement à l'autre. Contrairement à CUDA, cet environnement permet de faire du développement sur des GPU aussi bien de la marque NVIDIA que ATI. Une différence importante provient des mises à jour qui sont effectuées tous les 6 à 8 mois pour CUDA, alors que les mises à jour d'OpenCL s'effectuent tous les deux ans environ. De plus CUDA à l'avantage d'avoir énormément de bibliothèques dédiées pour le calcul comme cuBLAS, cuLAPACK, cuSPARSE, cuSOLVER, etc., mais également des bibliothèques diverses permettant d'accélérer la résolution des problèmes de divers domaines tels que : les réseaux de neurones artificiels, processus audio et vidéo, traitement d'images. Dans le cadre de cette thèse au sein de Rhenovia, il a été décidé de se tourner vers l'environnement CUDA, car des partenariats ont pu être établis afin d'avoir accès à des GPU de NVIDIA. De plus, il était envisagé de faire appel au méso-centre de calcul de Strasbourg, possédant une certification NVIDIA. Le but de ce chapitre n'est pas d'apprendre au lecteur à coder en CUDA, mais de mieux comprendre son fonctionnement (Sanders et Kandrot, 2010, 2011; Couturier, 2013; Kirk et Hwu, 2010; NVIDIA, 2015). Les notions principales à la compréhension des méthodes développées plus tard, pour les différentes équations vues dans la première partie, sont introduites.

4.4.2 Structure et bases d'un code CUDA

Un programme CUDA consiste en plusieurs étapes exécutées sur le CPU, également appelé host, et sur un GPU, nommé device. En général, la partie du programme comportant peu ou pas de parallélisme sera lancée sur le host, tandis que la partie parallélisée sera exécutée sur le device. En résumé, un programme CUDA intègre toutes ces parties, qui sont ainsi unifiées au sein d'un même code. C'est le compilateur NVIDIA C, aussi appelé nvcc, qui s'occupe de scinder les deux parties du programme. Je vais montrer les différents concepts permettant de comprendre et de pouvoir développer un programme en CUDA.

4.4.2.1 Les qualificatifs

Les qualificatifs permettent de distinguer les parties du code qui doivent être exécutées sur le device ou sur le host. Il existe des qualificatifs sur les fonctions mais également sur les variables. Les qualificatifs associés aux fonctions sont les suivants :

- **device** : permet d'appeler et d'exécuter une fonction sur le GPU,
- **host** : permet d'appeler et d'exécuter une fonction sur le CPU,
- **global** : permet d'appeler une fonction sur le CPU pour l'exécuter sur le GPU.

Les qualificatifs associés aux variables sont :

- **device** : permet de définir une variable dans la mémoire globale du GPU. Elle existera durant toute la vie de l'application, et elle sera accessible depuis le CPU et le GPU, notamment pour le rapatriement des résultats sur le CPU,
- **constant** : définit une variable dans la mémoire constante du CPU. Cette variable sera disponible durant toute la vie de l'application. Elle est écrite par le code CPU et lue par le code GPU,
- **shared** : définit une variable sur la mémoire partagée d'un multiprocesseur, qui sera disponible durant toute la vie du bloc de threads. Cette variable sera uniquement disponible via le code GPU.

4.4.2.2 Les kernels

Un kernel est toujours défini avec le qualificatif device, en spécifiant le nombre de threads que va exécuter ce noyau. Pour faire appel au kernel, on utilise la syntaxe <<<...>>>. Chaque thread exécutant ce kernel est identifié grâce à un identifiant unique, accessible au sein du kernel, en utilisant la variable : `threadId`.

Afin d'illustrer la définition ainsi que l'appel d'un kernel, on présente le code suivant qui permet d'additionner deux tableaux a et b de taille N, et de stocker le résultat dans un tableau c.

Code 4.3 – Exemple CUDA de la somme de deux tableaux.

```
1  // Définition du kernel
2  __global__ void AddVector(float* a, float* b, float* c) {
3      int i = threadIdx.x;
4      c[i] = a[i] + b[i];
5  }
6
7
8  int main() {
9      ... // Appel du kernel avec N threads
10     AddVector<<<1, N>>>(a, b, c);
11     ...
12 }
```

Dans cet exemple, chaque thread effectue une addition entre deux éléments du tableau. Là où un code CPU devrait se charger de faire l'addition des N éléments en utilisant une boucle, ici chaque thread effectue une opération unique en parallèle des autres.

4.4.2.3 Hiérarchie des threads

Nous avons vu dans la section précédente, qu'un thread est défini par un identifiant unique : `threadIdx`. Cet identifiant est un vecteur comportant trois composantes. Lors de l'exécution d'un kernel, on spécifie la dimension du bloc de threads, entraînant la création d'un indice permettant de différencier les threads. Bien entendu, le choix de la dimension est étroitement lié à la structure du problème.

Les coordonnées d'un thread au sein d'un bloc et son indice sont liés. Dans le cas où le bloc de thread est à une dimension, l'indice est défini par une coordonnée unique sur l'axe x.

Si le bloc de thread est à deux dimensions de taille (N_x, N_y) et l'identifiant d'un thread est défini par le couple (x, y), alors l'indice du thread au sein du bloc est : $x + yN_x$.

Enfin, si le bloc de threads est de dimension trois avec comme taille (N_x, N_y, N_z) et l'identifiant d'un thread de ce bloc est (x, y, z), alors son indice est le suivant : $x + yN_x + zN_xN_y$.

Il y a bien entendu une limite au nombre de threads par bloc, car tous les threads appartenant à ce bloc partagent la même mémoire, et les mêmes ressources. Sur les GPU actuels, chaque bloc peut contenir au maximum 1024 threads.

Les blocs sont organisés d'une façon similaire au threads. Ils peuvent être regroupés dans des grilles de une, deux ou trois dimensions. Comme pour la taille des blocs, la taille des grilles est déterminée en fonction des caractéristiques du problème traité. Un bloc au sein d'une grille est identifié par un identifiant unique `blockIdx` qui peut être un entier si la grille est à une dimension, où un vecteur à plusieurs composantes, dans le cas ou la grille est de dimension deux ou trois. La taille d'un bloc est accessible au sein du kernel, en utilisant la variable `blockDim`. Au final, une grille de blocs constitués de threads peut être illustrée par la Figure 4.12.

Reprenons l'exemple de notre programme CUDA (Code 4.3), en l'appliquant à la somme de deux matrices de taille $N \times N$. On stocke la matrice à deux dimensions dans un tableau à une dimension. On accède donc à un élément de la matrice grâce à la variable `idx`. Dans cet exemple, on définit des blocs de taille $(16, 16) = 256$ threads, au sein d'une grille dont la taille est déterminée en fonction de N.

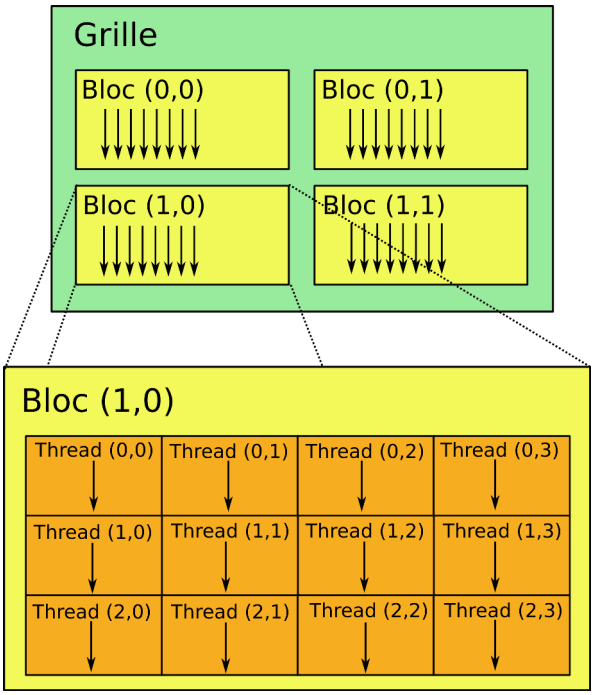

Figure 4.12 – Représentation d'une grille de blocs contenant des threads.

Les threads appartenant au même bloc peuvent partager des données à travers la mémoire partagée. De plus, on peut synchronizer leurs exécutions pour coordonner l'accès à la mémoire. Pour cela, on fait appel à la fonction `__syncthreads()` qui fonctionne comme une barrière au sein d'un bloc.

Code 4.4 – Exemple CUDA de la somme de deux tableaux

```
1  // Définition du kernel
2  __global__ void AddVector(float* a, float* b, float* c) {
3
4      int i = threadIdx.x;
5      int j = threadIdx.y;
6
7      int idx = i + j*blockDim.x
8
9      c[idx] = a[idx] + b[idx];
10 }
11
12
13 int main() {
14     ...
15     // Appel du kernel
16     dim3 threadsPerBlock(16, 16);
17     dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
18
19     AddVector<<<numBlocks, threadsPerBlock>>>(a, b, c);
```

```
20      ...
21  }
```

4.4.3 Streams CUDA

Les streams CUDA sont des séquences d'opérations exécutées dans l'ordre ou en concurrence, les unes par rapport aux autres. L'utilisation d'un stream se fait grâce aux lignes de codes présentées ci-dessous

Code 4.5 – Exemple basique de la création et de l'utilisation d'un stream CUDA.

```
1  // Déclaration d'un stream
2  cudaStream_t myStream;
3
4  // Initialisation d'un stream
5  error = cudaStreamCreate(&myStream);
6
7  // Utilisation d'un kernel avec un stream
8  kernel<<<dimGrid, dimBlock, myStream>>>(...);
9  cudaMemcpyAsync(..., myStream);
10
11 // Destruction du stream
12 error = cudaStreamDestroy(myStream);
```

Par défaut, les grilles lancées sont exécutées de façon totalement synchrone, au sein du stream par défaut (0 ou null). Autrement dit, la grille suivante commence uniquement quand la précédente a terminé son exécution (Code 4.6).

Code 4.6 – Lancement des opérations dans le stream par défaut 0.

```
1  cudaMalloc(&dev1,size);
2  double *host1 = (double*)malloc(&host,size);
3  ...
4
5  // Les opérations suivantes sont totalement synchrones
6  cudaMemcpy(dev1,host1,size,cudaMemcpyHostToDevice);
7  kernel2<<<grid,block,0>>>(...,dev2,...);
8  kernel3<<<grid,block,0>>>(...,dev3,...);
9  cudaMemcpy(host4,dev4,size,cudaMemcpyDeviceToHost);
10
11 ...
```

Si aucun stream n'est spécifié lors du lancement d'un kernel, c'est le stream par défaut qui est utilisé automatiquement. Il n'est donc pas nécessaire de spécifier le stream lors de la déclaration du kernel, comme cela a été fait dans le code 4.6. Toutes les opérations CUDA

sont synchrones, mais lorsque certaines opérations sont lancées sur le CPU, celles-ci sont asynchrones avec les kernels. Il est alors possible qu'il y ait chevauchement entre les différentes opérations. Reprenons notre code 4.6 en lui ajoutant un certain nombre de méthodes lancées sur le CPU.

Code 4.7 – Les kernels GPU sont asynchrones avec le CPU.

```
cudaMalloc(&dev1,size);
double *host1 = (double*)malloc(&host,size);
...

// Les opérations suivantes sont totalement synchrones
cudaMemcpy(dev1,host1,size,cudaMemcpyHostToDevice);
kernel2<<<grid,block,0>>>(...,dev2,...);
CPU_methodes();                              // Méthodes qui peuvent potentiellement se
                                             // chevaucher avec l'exécution du kernel 2
kernel3<<<grid,block,0>>>(...,dev3,...);
cudaMemcpy(host4,dev4,size,cudaMemcpyDeviceToHost);

...
```

Cette particularité est donc à prendre en compte lors de l'implémentation de programmes sur GPU. Il est possible que plus de concurrence soit souhaitée, autrement dit, que l'on puisse lancer plusieurs grilles, pour qu'elles s'exécutent en même temps. Cela permet notamment de réduire encore un peu plus le temps d'exécution du programme. On complète le code 4.7 pour prendre en compte différents streams et ainsi obtenir plus de concurrence.

Code 4.8 – Utilisation de différents streams.

```
// Création et initialisation des streams
cudaStream_t stream1, stream2, stream3, stream4;
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);
...

cudaMalloc(&dev1,size);
cudaMallocHost(&host1,size);
...

// On fait appel à des copies asynchrones
// Cette fois, toutes les opérations suivantes peuvent se chevaucher
cudaMemcpyAsync(dev1,host1,size,cudaMemcpyHostToDevice,stream1);
kernel2<<<grid,block,stream2>>>(...,dev2,...);
CPU_methodes();
kernel3<<<grid,block,stream3>>>(...,dev3,...);
cudaMemcpyAsync(host4,dev4,size,cudaMemcpyDeviceToHost,stream4);

...
```

Le code 4.8 permet d'avoir des opérations totalement asynchrones, et donc pleinement concurrentes. Par contre, il est à noter que les données utilisées par des opérations concurrentes sont totalement indépendantes.

Il peut-être obligatoire d'attendre que toutes les opérations sur le GPU soit effectuées, pour que les méthodes du CPU soit lancées. Dans ce cas, on bloque le CPU, en utilisant la commande `cudaDeviceSynchronize()` . Il est également possible de bloquer le CPU tant que toutes les opérations d'un stream particulier soient terminées. Pour cela, on utilise la syntaxe `cudaStreamSynchronize(id_stream)`.

Nous allons maintenant voir rapidement, qu'il est possible d'utiliser d'autres objets CUDA, lors d'une synchronisation.

4.4.4 Evénements

Pour effectuer des synchronisations plus précises entre les différents streams et le CPU, on peut utiliser des événements (Events) CUDA. Les événements peuvent être vus comme des balises qui vont être placées en différents endroits du programme, permettant ainsi d'effectuer des opérations suites à ces balises. Un exemple canonique de l'utilisation d'un événement est présenté dans le code 4.9. Ici, les événements `event1` et `event2` sont utilisés pour déterminer le temps écoulé entre le début et la fin des opérations effectuées dans `stream1`.

Code 4.9 – Utilisation standard d'un événement.

```
// 1. Déclaration des Events
cudaEvent_t event1;
cudaEvent_t event2;
```

```
4
5  // 2. Initialisation des Events
6  cudaCreateEvent(&event1);
7  cudaCreateEvent(&event2);
8
9  // 3. Enregistrement du premier Event dans un stream
10 cudaEventRecord(event1, stream1);
11
12 // 4. Opérations sur le GPU
13 GPU_methodes();
14
15 // 5. Enregistrement du second Event dans un stream
16 cudaEventRecord(event2, stream1);
17
18 // 6. Synchronisation
19 cudaEventSynchronize(event2);
20
21 // 7. Temps écoulé entre les deux enregistrements
22 float milliseconds = 0.;
23 cudaEventElapsedTime(&milliseconds,event1, event2);
24
25 // 8. Destruction des Events
26 cudaEventDestroy(&event1);
27 cudaEventDestroy(&event2);
```

En utilisant la fonction `cudaStreamWaitEvent(stream,event)`, on attend que `event` soit atteint dans `stream` pour continuer les autres opérations. Le code 4.10 donne un rapide aperçu de l'utilisation des événements lors d'une synchronisation.

Code 4.10 – Utilisation standard d'un événement.

```
1  // 1. Création de l'event
2  cudaEvent_t event;
3  cudaCreateEvent(&event);
4
5  // 2. Copie des données du CPU vers le GPU
6  cudaMemcpyAsync(device_in, host_in, size, cudaMemcpyHostToDevice, stream1);
7
8  // 3. Enregistrement de l'event
9  cudaEventRecord(event, stream1);
10
11 // 4. Copie des résultats précédents (on suppose que d'autres étapes ce sont déroulées)
12 cudaMemcpyAsync(host_out, device_out, size, cudaMemcpyDeviceToHost, stream2);
13
14 // 5. On attent l'event dans le stream 1
15 cudaStreamWaitEvent(stream2, event);
16
17 kernel<<< ..., stream2 >>>(d_in,d_out);
18
19 // 6. Autres méthodes lancées sur le CPU
20 CPU_méthodes_asynchrones();
```

Maintenant que les streams et les événements ont été correctement introduits, nous allons parler d'un concept important pour la suite de nos travaux.

4.4.5 Parallélismes dynamiques

Jusqu'à l'architecture Fermi, seul le CPU pouvait générer des opérations sur le GPU. L'arrivée de l'architecture Kepler a permis d'introduire un nouveau concept de parallélisme dynamique, permettant au GPU de pouvoir auto-générer des opérations.

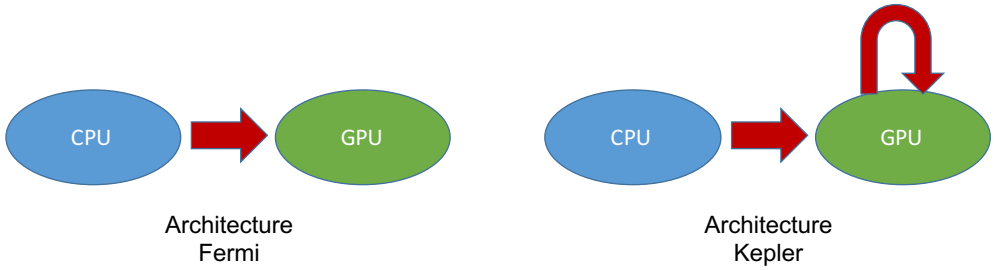

Figure 4.13 – Représentation de la différence de fonctionnement entre les architectures Fermi et Kepler.

Comme vu précédemment, une grille est constituée de blocs formés de threads. Grâce à l'architecture Kepler, il est maintenant possible d'effectuer une programmation, en utilisant le parallélisme dynamique d'une grille parent qui va pouvoir lancer des grilles filles. Ces grilles filles héritent de certains attributs et aussi de certaines limites de la grille parent, tel que la mémoire cache L1.

Potentiellement, chaque thread de la grille parent peut lancer un nouveau kernel, ce qui signifie que si la grille parent possède N_B blocs avec N_T threads, alors la grille pourra effectuer $N_B \times N_T$ kernels. Il est possible de spécifier à un seul thread de la grille parent de lancer un nouveau kernel. Un exemple de code utilisant le parallélisme dynamique est présenté ci-dessous

Code 4.11 – Exemple utilisant le parallélisme dynamique.

```
1
2  // Fonction utilisée par la grille fille et exécutée sur le GPU
3  __global__ void child_launch(int *data){
4      data[threadIdx.x] = data[threadIdx.x]+1;
5  }
6
7  // Fonction utilisée par la grille parent et exécutée sur le GPU
8  __global__ void parent_launch(int *data){
9      data[threadIdx.x] = threadIdx.x;
10     __syncthreads();
11     if(threadIdx.x == 0){
12         // Sur le thread d'ID 0, on lance la grille fille
```

```
13          child_launch<<< 1, 256 >>>(data);
14      }
15  }
16
17
18  // Fonction exécutée sur le CPU
19  void host_launch(int *data){
20      // Lancement de la grille parent
21      parent_launch<<< 1, 256 >>>(data);
22  }
```

Lorsqu'on utilise le parallélisme dynamique, les grilles lancées sont entièrement imbriquées. Autrement dit, la grille parent ne se terminera pas tant que la grille fille n'est pas terminée, et cela même si aucune synchronisation explicite n'est utilisée. Dans la Figure 4.14, on a représenté schématiquement cette imbrication, afin de mieux comprendre le fonctionnement de ce parallélisme.

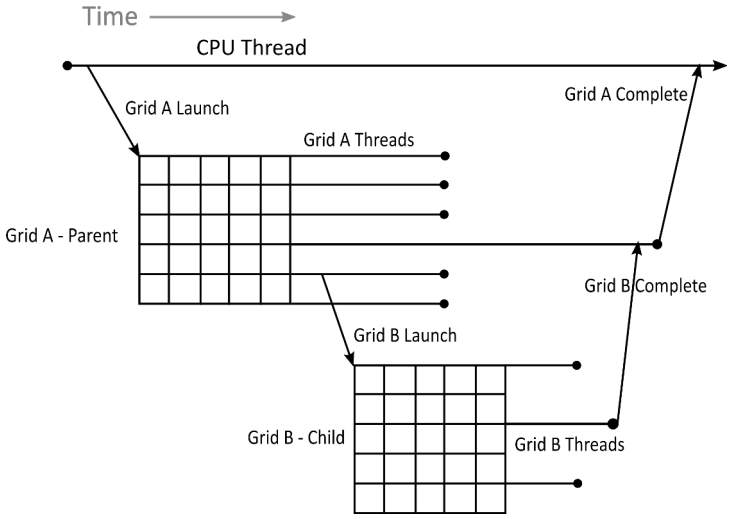

Figure 4.14 – Représentation de l'imbrication des grilles parent et fille.

Bien évidemment, si la grille parent a besoin des résultats de la grille fille à un moment bien précis, il faut s'assurer que cette dernière a bien achevé son exécution. Pour cela, on est obligé de réaliser une synchronisation explicite grâce à la fonction `cudaDeviceSynchronize()`. Cette fonction permet d'attendre que toutes les grilles lancées précédemment par le bloc de threads soient terminées. Notons que l'utilisation de cette fonction est coûteuse, car elle va mettre en pause le bloc en cours d'exécution.

La mémoire possède une certaine cohérence entre les différentes grilles. Cela signifie que si la grille-parent écrit dans la mémoire, puis lance une grille fille alors cette dernière pourra accéder à la valeur. De façon similaire, si la grille-fille écrit dans un emplacement de la mémoire, alors si la grille parent effectue une synchronisation, alors elle aura accès à cette zone de la

mémoire. La même remarque peut être faite si plusieurs grilles filles sont lancées séquentiellement. Si chaque grille effectue des écritures dans la mémoire, alors les grilles exécutées à la suite auront un accès à ces données, sans qu'aucune synchronisation ne soit faite entre chaque lancement.

Malgré cela, il faut faire attention lors de l'utilisation des différentes mémoires. Par exemple, toutes les variables définies dans la grille parent ne sont pas forcément utilisables comme argument lors du lancement d'une grille fille. Le tableau 4.1 résume les limitations liées aux pointeurs que l'on peut utiliser ou non pour l'exécution d'une grille fille.

Peut-être passé	Ne peut pas être passé
- Mémoire globale - Mémoire hôte - Mémoire constante	- Mémoire partagée (variables partagées également) - Mémoire locale

Tableau 4.1 – Limitations des pointeurs qui peuvent-être passés ou non lors du lancement d'une grille fille

Il faut savoir que le code contenant une erreur de déclaration entrainera un problème lors de la compilation. Il est néanmoins possible de contourner cette erreur, en passant par une structure qui contiendra le pointeur concerné. Mais il vaut mieux éviter cette solution, car cela peut conduire à une corruption des données et à des erreurs pratiquement indétectables, car CUDA n'identifiera pas ces erreurs lors de la compilation.

Comme pour les grilles lancées depuis le CPU, celles exécutées depuis un bloc de thread sont séquentielles. Comme vu dans la section 4.4.3, il est possible d'utiliser des streams au sein d'une grille parent, lors du lancement des grilles filles, permettant ainsi d'obtenir plus de concurrence. Le mode de fonctionnement des streams dans ce cas précis ne diffère pas de ceux utilisés dans la section 4.4.3. On remarquera que les streams par défaut, créés au sein de blocs différents, sont considérés comme différents. Malgré cette différence, une exécution simultanée des streams par défaut n'est pourtant pas garantie, comme on pourrait s'y attendre. Elle sert surtout à une meilleure gestion des ressources du GPU. Pour avoir une véritable concurrence, le programmeur est donc obligé de faire un effort dans le développement de son code, en utilisant les streams de façon optimale. De plus, une fois qu'un stream a été créé, il peut être utilisé par n'importe quel thread appartenant au même bloc. Il ne pourra plus être utilisé, lorsque le bloc a terminé son exécution ou par tout autre bloc. Finalement, la destruction d'un stream sur le GPU se fait de manière équivalente à ceux déclarés sur le CPU (voir la section 4.4.3).

De manière identique aux streams, les événements peuvent être utilisés sur GPU, en utilisant les mêmes fonctions décrites dans la section 4.4.4

4.4.6 Conclusion

Dans ce chapitre, nous avons introduit les bases ainsi que certains concepts de CUDA. Cela permettra donc de développer nos programmes sur GPU. Le but de cette présentation est d'utiliser, dans la suite, toutes ces méthodes et outils, pour pouvoir les appliquer à la résolution de notre problème monodomaine et bidomaine, permettant de modéliser l'activité électrique dans un tissu neuronal.

Partie III

Simulations et applications des méthodes de parallélisation au modèle monodomaine

5 Simulations des modèles de neurone

5.1 Introduction

Avant de résoudre le problème décrit par le modèle monodomaine, je vais m'intéresser à l'intégration de certains modèles de neurones, afin de mieux analyser leurs différences, ainsi que leurs caractéristiques. Comme nous l'avons vu dans la section 2.1, ces différents modèles sont décrits à l'aide d'une ou plusieurs EDO. Il convient ainsi d'utiliser les méthodes vues dans la section 3.1 pour pouvoir les résoudre de façon numérique, d'abord de façon séquentiel puis de développer la méthode de résolution pararéel avec les technologies MPI (section 4.1), et CUDA (section 4.4). L'objectif de ce chapitre est de donner un premier aperçu des gains de temps obtenus grâce à ces différentes méthodes. Je m'intéresserai à deux modèles en particulier, un modèle complexe : Hodgkin-Huxley, puis un modèle simplifié : Fitzhugh-Nagumo.

Le développement et les différents tests effectués dans ce chapitre ont été intégralement implémentés en Python, pour la partie séquentielle et MPI.

5.2 Première application : le modèle de Hodgkin-Huxley

Ce modèle est composé de quatre EDO modélisant les courants ioniques des deux principaux ions responsables de l'influx nerveux : les ions sodiques (Na^+) et potassiques (K^+). Ces équations, données dans la section 2.1, sont les suivantes :

$$\begin{cases} C_m \frac{\partial V_m}{\partial t} = m^3 h g_{Na}(E_{Na} - V_m) + n^4 g_K(E_K - V_m) + g_L(E_L - V_m) + I \\ \frac{\partial n}{\partial t} = \alpha_n(1-n) - \beta_n n \\ \frac{\partial m}{\partial t} = \alpha_m(1-m) - \beta_m m \\ \frac{\partial h}{\partial t} = \alpha_h(1-h) - \beta_h h \end{cases} \tag{5.1}$$

où $g_{Na}, g_K, g_L \in \mathbb{R}^+$ sont les conductances, I est le courant total, V_m est le potentiel de mem-

brane, $E_{Na}, E_K, E_L \in \mathbb{R}$ sont les potentiels d'équilibre des ions. Les paramètres I, g_{Na}, g_K, g_L, E_{Na}, E_K, E_L sont des constantes. Dans la littérature (Guckenheimer et Labouriau, 1993), on peut trouver les valeurs suivantes :

$$\begin{array}{ccccc} E_{Na} = 115\ mV & ; & E_K = -12\ mV & ; & E_L = 10{,}6\ mV \\ g_{Na} = 120\ mS/cm^2 & ; & g_K = -36\ mS/cm^2 & ; & g_L = 0{,}3\ mS/cm^2 \end{array}$$

La complexité de ce modèle rend son analyse mathématique difficile, de plus ce système d'équations est non-linéaire et ne peut donc pas être résolu analytiquement. On trouve néanmoins plusieurs études sur l'analyse mathématique de ce modèle, notamment les bifurcations (Guckenheimer et Labouriau, 1993; Hassard, 1978; Hassard et Shiau, 1996; Fukai *et al.*, 2000). Si le courant I est un paramètre de bifurcation, alors le modèle suit une bifurcation de Hopf.

5.2.1 Résolution séquentielle

Pour résoudre numériquement le système d'équations (5.1), j'utilise une discrétisation en temps suivant la méthode de résolution choisie : Euler explicite, et RK4 (section 3.1). Il est bien évidemment possible d'utiliser d'autres schémas de résolution, tels que les méthodes BDF ou les schémas à pas de temps variables. Des travaux portant sur les schémas à pas variables ont été effectués au sein de Rhenovia grâce au travail de Sarmis (2013).

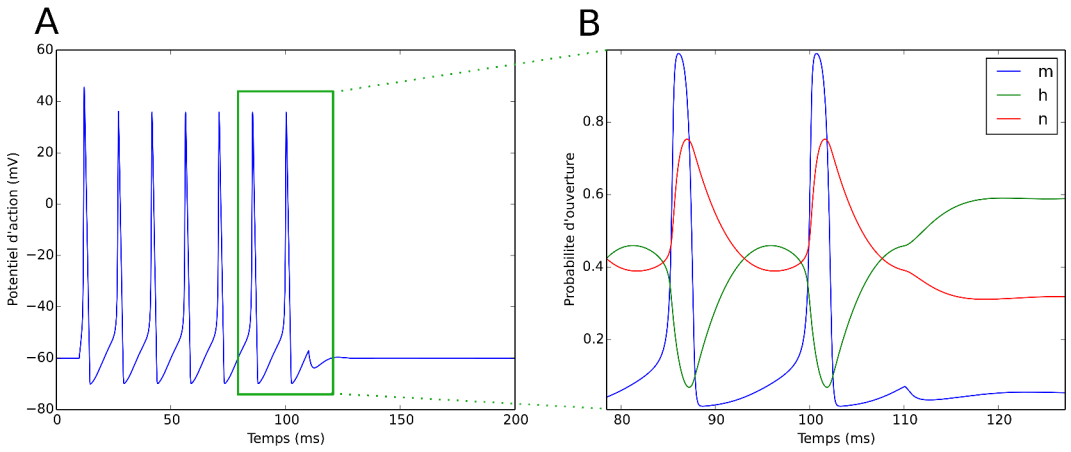

Figure 5.1 – Représentation graphique des fonctions intervenant dans le modèle de Hodgkin-Huxley : **A** : potentiel d'action V_m et **B** : les variables d'activation m, n et d'inactivation h zoomées sur une portion du temps.

La méthode d'Euler explicite étant moins précise que la méthode RK4, il est nécessaire de prendre un pas de temps plus petit, afin que la résolution se déroule sans problème. Nous simulons le modèle de Hodgkin-Huxley, en prenant les valeurs des paramètres données précédemment, et en utilisant un pas de temps très fin $\delta T = 10^{-4}\ ms$ pour les deux méthodes.

Dans le cadre de cette simulation , on soumet le neurone modélisé à un courant externe de 10 μA durant une période de 100 ms.

Les résultats, présentés dans la Figure 5.1, permettent d'observer les différentes phases de la création d'un potentiel d'action, décrites dans la section 1.3. Ce système d'équations ayant des dynamiques très rapides, venant des exponentielles très nombreuses dans le calcul des fonctions α et β (section 2.1), le pas de temps nécessaire à sa résolution ne peut pas être trop grand, notamment si le schéma d'Euler explicite est utilisé. Bien évidemment, avec un pas de temps identique, la solution numérique obtenue grâce à la méthode RK4 sera une meilleure approximation de la solution analytique. Dans ce cas, on constate une erreur quadratique de l'ordre de $0,23$ entre les deux méthodes de résolution. Malgré cela, la méthode d'Euler reste plus rapide à exécuter, car il faut environ 312 s pour intégrer ce système avec un pas de temps $\Delta t = 10^{-4}$ ms, alors qu'il faut 1465 s avec la méthode RK4, en utilisant un nœud du Cluster de Rhenovia cadencé à $2,5$ GHz. La différence entre ces deux temps est logique, étant donné que la méthode RK4 a besoin de faire le calcul de quatre étapes. Ces remarques sont à garder à l'esprit, car elles seront essentielles dans ce qui va suivre.

On constate que la simulation séquentielle d'un modèle tel que celui de Hodgkin-Huxley devient vite coûteux en terme de temps de calcul, lorsque l'on souhaite une grande précision pour les résultats, ou lorsque de très longues simulations doivent être exécutées. Il devient dans ce cas compliqué de concilier les deux critères : rapidité d'exécution et haute précision. Il convient donc de résoudre le problème, en utilisant les méthodes et techniques décrites dans la partie 2, permettant ainsi d'atteindre nos exigences.

5.2.2 Résolution avec la méthode pararéel en MPI

Afin de garder notre niveau d'exigence sur la qualité du résultat, tout en diminuant le temps lors de la simulation, je vais appliquer la méthode décrite dans la section 3.1, en utilisant la technologie MPI, vue dans la section 4.1. Peu de solutions ont été apportées, afin de résoudre un système d'ODE en parallèle, notamment à cause du caractère séquentiel du problème (la solution au temps t_n a besoin de la valeur t_{n-1} pour être déterminée). Une des solutions était de paralléliser les étapes des méthodes de Runge-Kutta (Liu *et al.*, 2011), mais elle présente certaines limites, car la parallélisation dépend étroitement de l'ordre de la méthode, limitant ainsi la force du parallélisme. Une deuxième solution est d'utiliser la méthode pararéel décrite section 3.1, notamment sa version distribuée, car de toutes les variantes présentées dans ce chapitre, elle est la plus efficace.

Comme nous l'avons vu, cette méthode a besoin de deux solveurs effectuant pour l'un, une intégration grossière du problème, et pour l'autre, une intégration fine de ce même problème. Chacun de ces solveurs est défini par une méthode de résolution d'EDO, comme par exemple Euler explicite, ou encore RK4. L'idée est de prendre une méthode avec un ordre élevé, par exemple RK4, pour la résolution grossière permettant ainsi de commettre une erreur moins importante sur le résultat envoyé au processus suivant, et une méthode d'ordre moins élevé,

comme Euler explicite, pour l'intégration fine afin que le temps de calcul lors de cette étape soit le moins important possible.

Pour résumer, deux choses permettent de différencier vraiment ces deux solveurs. La première concerne la méthode utilisée à proprement parlé, un ordre élevé sur le solveur grossier, et un ordre moins élevé sur le solveur fin. La deuxième, forcément lié à la première, concerne le pas de temps. On se permet d'utiliser un pas de temps plus important lors de l'intégration grossière, et un pas de temps plus fin pour la résolution fine. Tout cela, au final, dans le but d'accélérer la résolution des EDO, en ayant la même précision que la résolution en séquentielle.

Dans cette version de l'algorithme, toutes les tâches sont équitablement réparties entre les différents processus sur lesquels le modèle est simulé, permettant encore d'accélérer le processus.

Dans le développement du programme parallèle permettant la simulation du modèle de Hodgkin-Huxley à l'aide du MPI, il a fallu utiliser les différentes fonctions disponibles, déjà présentées ultérieurement (section 4.1), afin de pouvoir faire communiquer les différents processus entre eux. Chaque processus communique exclusivement avec ses plus proches voisins, c'est-à-dire : le processus P communique avec les processus $P-1$ et $P+1$ (Figure 5.2), puisqu'il reçoit des résultats du précédent, pour finalement envoyer les siens au suivant. Dans le cas du dernier processus, celui-ci ne renvoit rien, hormis son résultat final au premier processus, lors d'un `Broadcast()` final. En plus des différents résultats (c'est-à-dire, les valeurs des variables Vm, m, h et n pour le modèle de Hodgkin-Huxley, à la fin de chaque sous-intervalle) que chaque processus doit communiquer au suivant, un état de convergence doit également être envoyé, afin de faire savoir au processus P, si son prédécesseur a bien convergé ou non.

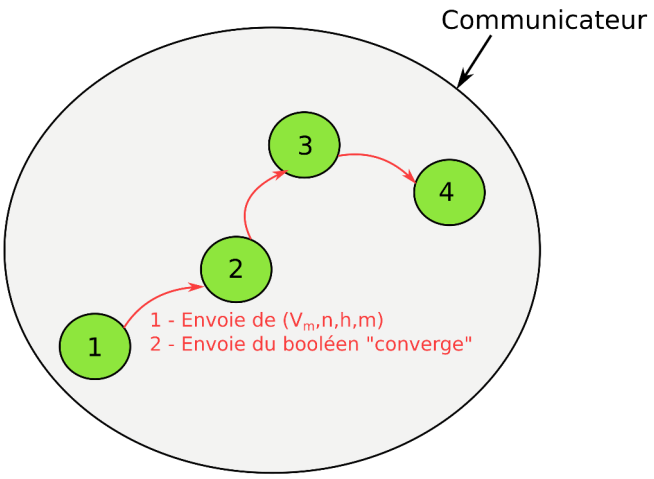

Figure 5.2 – Représentation schématique de la communication entre quatre processus, lors de l'exécution de programme parallèle.

Le choix des communications est également importante. Ici j'ai utilisé les communications dites bloquantes, car il faut que chaque processus P ait récupéré les résultats obtenus par le processus $P-1$, pour qu'il puisse à son tour accomplir sa tâche. On fait donc appel aux fonctions `send()` et `recv()`, comme on peut le constater dans la portion de code 5.1.

Code 5.1 – Portion du programme pararéel permettant la simulation du modèle de Hodgkin-Huxley.

```
from mpi4py import MPI

# Définition du communicateur et de ses caractéristiques
comm = MPI.COMM_WORLD
size = MPI.COMM_WORLD.Get_size()
rank = MPI.COMM_WORLD.Get_rank()
name = MPI.Get_processor_name()

# Processus 0
if(rank == 0):

    # Résolution Grossière du Processus 0
    UG = CoarseResolRK4(f,Utild[:,0,0],TG,DT,IappG)
    Utild[:,1,0] = UG[:,-1]

    # Envoi des résultats au Processus suivant
    comm.send(Utild[:,1,0],dest=1,tag=1)

    # Résolution Fine du Processus 0
    UF = FineResolEuler(f,Utild[:,0,0],TF,dt,IappF)
    Uchap[:,1,0] = UF[:,-1]
    U[:,1,1] = Uchap[:,1,0]

    # Initialization du booléen "converge"
    converge = True

    # Envoi des différents résultats au Processus suivant
    comm.send(converge,dest=1,tag=2)
    comm.send(U[:,1,1],dest=1,tag=3)

# Processus P
else:
    # Réception des résultats venant du Processus P-1 précédent
    Utild[:,rank,0] = comm.recv(source=rank-1,tag=1)

    # Résolution Grossière du Processus P
    UG = CoarseResolRK4(f,Utild[:,rank,0],TG,DT,IappG)
    Utild[:,rank+1,0] = UG[:,-1]

    # Si le Processus n'est pas le dernier, alors envoie des
    # résultats au Processus P+1 suivant
    if(rank != (size-1)):
        comm.send(Utild[:,rank+1,0],dest=rank+1,tag=1)
```

Pour l'implémentation de cette méthode de résolution, j'ai pris le schéma RK4 comme solveur grossier, et le schéma d'Euler explicite pour le solveur fin. Pour rappel, l'intervalle de temps est ensuite découpé en fonction du nombre de processus sur lesquels le programme est exécuté. Puis, chaque processus se charge d'intégrer grossièrement, sur un sous-intervalle, le problème avec un pas de temps grossier Δt. Suite à cette première approximation, la résolution fine avec la méthode d'Euler explicite est effectuée avec un pas de temps $\delta t << \Delta t$.

Pour pouvoir au mieux comparer les résultats obtenus par les méthodes séquentielle et parallèle, on utilise les pas de temps suivants :

$$\Delta t = 3,125 \times 10^{-3} \quad ; \quad \delta t = 1,0 \times 10^{-4} \quad ,$$

On utilise le même protocole de simulation introduit dans la section 5.2.1, à savoir : un temps simulé de 200 ms, avec un courant externe I de 10 μA appliqué sur une période de 100 ms. L'intérêt ici est de déterminer l'erreur entre les deux solutions obtenues séquentiellement et à l'aide de la méthode pararéel, puis de déterminer le nombre d'itérations nécessaires afin de constater une convergence des deux solutions, et finalement en déduire le gain de temps représenté par l'accélération. On définit l'accélération comme étant le rapport entre le temps de résolution séquentielle et le temps de résolution parallèle.

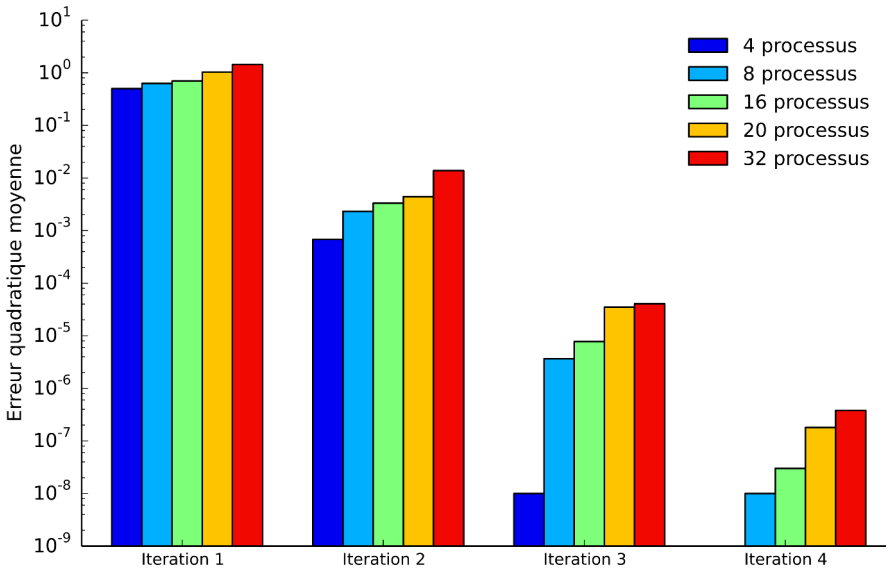

Figure 5.3 – Erreurs quadratiques moyennes pour chaque itération en fonction du nombre de processus MPI choisi, lors de la simulation du modèle de Hodgkin-Huxley.

Les résultats de ces tests sont présentés dans la Figure 5.3. On constate que l'erreur atteint une

tolérance acceptable de l'ordre de 10^{-6} à partir de la quatrième itération, hormis lorsqu'on utilise quatre processus, où ce critère est atteint à partir de la troisième itération. On note que lors de l'utilisation de quatre processus, le nombre maximum d'itérations possible est de trois, d'où ce résultat. On constate également une diminution de l'erreur au fur et à mesure des différentes itérations. La présence d'une augmentation des erreurs, lorsque le nombre de processus augmente peut-être expliquée par le fait que les erreurs commises sur les solutions envoyées mettent plus de temps à être corrigées, car il y a un effet de propagation de l'erreur. Ce qui entraîne un petit délai en plus, dans le cas où un grand nombre de processus est utilisé.

Il est intéressant, maintenant, de comparer les différents gains de temps obtenus, en fonction de l'itération. Ces résultats sont présentés dans la Figure 5.4.

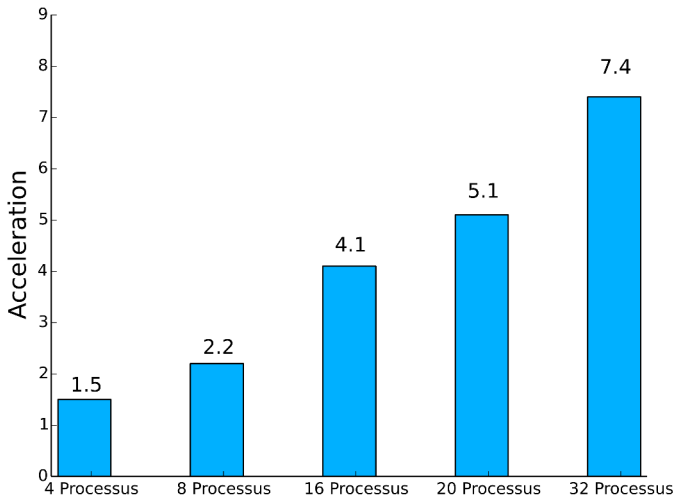

Figure 5.4 – Accélérations entre les temps de résolution séquentielle et parallèle (MPI) du modèle de Hodgkin-Huxley, en fonction de l'itération.

On constate ainsi que, pour atteindre notre critère de tolérance, l'accélération augmente logiquement en fonction du nombre de processus. Il devient donc intéressant d'utiliser cette méthode de résolution, car le gain est évident lorsque l'on veut une haute précision sur nos solutions, mais aussi si on souhaite effectuer de très longues simulations.

J'ai développé cet algorithme à l'aide de la technologie MPI. J'ai aussi testé ce même algorithme, mais en le développant sur GPU, à l'aide de l'environnement CUDA, introduit dans la section 4.4.

5.2.3 Résolution avec la méthode pararéel en CUDA

L'utilité d'une répartition équitable des charges de travail va prendre tout son sens maintenant. Contrairement à MPI, où chaque processus peut effectuer des tâches totalement indépendantes et différentes, sur GPU l'approche est totalement différente. Cet aspect a déjà été évoqué dans les sections 4.2 et 4.4. Lorsqu'une grille contenant des blocs de threads est exécutée, tous ces threads réalisent la même tâche avec des données initiales différentes. C'est pour cela que l'algorithme pararéel dans sa version distribuée est le plus adapté au calcul GPU, car aucune distinction entre travailleurs et manager n'est à faire.

L'autre différence avec le MPI vient du fait qu'il n'y a aucune communication explicite entre les différents threads d'une grille, car aucune fonction `send()` ou `recv()` n'existe en CUDA. Pour récupérer les données d'un thread spécifique, on utilise la mémoire partagée. Si les deux threads appartiennent au même bloc, on fait appel à la mémoire globale, même si celle-ci est plus coûteuse en temps d'accès. Pour ces tests, je me suis restreins à l'utilisation d'un seul bloc de threads permettant ainsi l'utilisation de la mémoire partagée, tout en veillant à ce qu'il n'y ai pas de conflits au moment de la lecture.

Toutes les fonctions utiles pour la bonne résolution des équations doivent être définies sur le GPU à l'aide du qualificatif `__device__`, notamment les différents solveurs RK4 et Euler explicite, ou encore la fonction `Stim()` appliquant le courant externe I au modèle. La fonction principale appelée par le CPU sur le GPU, permettant le lancement de la grille utilise le qualificatif `__global__` (voir code 5.2)

Code 5.2 – En-tête de certaines méthodes.

```
// Bibliothèques du système
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <cmath>

// CUDA runtime
#include <cuda_runtime.h>

__device__ double Stim(int i,double dtf,double coeff, Param p){
    ...
}

__device__ void Coarse_RK4(double **UG,int idx,Param p){
    ...
}

__global__ void parareal(double *un,double *unG_Final,Param p){
    ...
}
```

On définit ensuite un tableau dans la mémoire partagée, contenant les valeurs de la solution à la fin d'un sous-intervalle, ainsi qu'un deuxième tableau, permettant d'échanger les statuts de la convergence sur chacun de ces sous-intervalles. Comme présenté dans la section 4.4, chaque thread au sein de la grille est caractérisé par un identifiant, permettant ainsi de trouver la bonne valeur au sein des différents tableaux de la mémoire partagée.

Une des difficultés, par rapport à l'implémentation MPI, est de rendre certaines portions du code bloquantes. Il est obligatoire, à certains moments uniquement, que le P-ième thread ait reçu les résultats calculés par le thread $P-1$ (par exemple, lorsque la solution grossière initiale est déterminée). Contrairement au MPI, où il est possible de définir des communications explicites bloquantes, la programmation CUDA ne permet pas de faire cela aussi simplement. Pour contourner ce problème, il faut boucler sur le nombre total de threads, puis faire un test pour savoir si le compteur de la boucle correspond à l'identifiant du thread courant. Si c'est le cas, alors le thread effectue son travail, et les autres attendent. Si non, le thread attend que le thread concerné par cet indice ait terminé sa tâche (voir le code 5.3). Pour réussir cette phase d'attente, il est obligatoire de mettre une barrière de synchronisation (`__syncthreads()`). Le seul inconvénient à cette fonction est qu'elle ne synchronise que des threads appartenant au même bloc.

Code 5.3 – Action bloquante : Le thread P attend $P-1$.

```
for(int i=0;i<BLOCK_SIZE;i++){

    if(i == idx){

    // Résolution grossière
    ...

    break;
    }
    __syncthreads();
}

// On continue l'algorithme en effectuant les différentes itérations
// de l'algorithme
...
```

Pour éviter que le thread venant de terminer son action continue d'attendre dans la boucle, je le libère à l'aide d'un `break`. Sinon, l'avantage de la parallélisation disparaît clairement, si tout le bloc doit attendre la fin de cette opération pour continuer.

Avant le lancement du kernel par le CPU, il est obligatoire d'initialiser les différentes variables qui seront utilisées durant l'exécution du programme. Pour cela, j'alloue l'espace mémoire des différents tableaux sur le CPU, puis je les initialise. Ensuite, j'alloue la mémoire requise, sur le GPU, et je copie les données du CPU vers les GPU (voir la section 4.4).

Enfin, je lance le kernel en appelant la fonction ayant le qualificatif `__global__`, en lui spécifiant la taille de la grille (code 5.4), grâce à des variables de type `dim3`. Une fois l'exécution terminée, il ne faut pas oublier de vider la mémoire allouée en début de programme.

Code 5.4 – Allocation mémoire et lancement du kernel.

```
#define BLOCK_SIZE 1000

// Allocation de la mémoire sur le CPU
unsigned int size_U0 = 4;
unsigned int mem_size_U0 = sizeof(double) * size_U0;
double *h_U0 = (double *)malloc(mem_size_U0);

// Initialisation de U0 avec les conditions initiales
h_U0[0] = -60.0;
h_U0[1] = 0.0529324852572;
h_U0[2] = 0.596120753508;
h_U0[3] = 0.317676914061;

// Allocation de la mémoire sur le GPU
double *d_U0, *d_UF_Final;

cudaMalloc((void **) &d_U0, mem_size_U0);
cudaMalloc((void **) &d_UF_Final, mem_size_UF);

// Copie des données du CPU vers le GPU
cudaMemcpy(d_U0, h_U0, mem_size_U0, cudaMemcpyHostToDevice);

// Définition des tailles des blocs et de la grille
dim3 threads(BLOCK_SIZE, 1);
dim3 grid(1, 1);

// Execution du kernel
parareal<<< grid, threads >>>(d_U0,d_UF_Final,p);

// Copie des résultats du GPU vers le CPU
cudaMemcpy(h_UF_Final, d_UF_Final, mem_size_UF, cudaMemcpyDeviceToHost);

// Libération de la mémoire
free(h_U0);
free(h_UF_Final);
cudaFree(d_U0);
cudaFree(d_UF_Final);

cudaDeviceReset();
```

D'un point de vue technique, tous les paramètres de simulations tels que les pas de temps, le temp initial et le temps final, les données de stimulation, sont regroupés au sein d'une structure `Param` (code 5.5). Cela évite de perdre des informations en cours de programme, et permet de réduire la lourdeur du code, mais également d'avoir un code plus organisé.

Il est possible d'envoyer une structure directement sur le GPU, lors de l'appel au kernel, en le spécifiant comme argument de cette fonction (code 5.4), après l'avoir créé et initialisée.

Code 5.5 – Définition de la structure Param.

```
// Structure Param contenant les paramètres de la simulations

typedef struct Param{
    // Temps initial et final de la simulation
    double Ti;
    double Tf;

    // Pas de temps grossier et fin;
    double dtg;
    double dtf;

    // Tolérance de la méthode pararéel sur l'erreur
    double EPS_ERR;

    // Paramètres de la stimulation : Amplitude, temps de départ,
    // temps d'arrêt
    double I_APP;
    double T_STIM_START;
    double T_STIM_END;

} Param;

// Initialisation
Param p;

p.Ti = 0.;
p.Tf = 200.;
p.dtg = 3.125e-3;
p.dtf = 1.e-4;
p.EPS_ERR = 1.e-5;
p.I_APP = 0.4;
p.T_STIM_START = 10.;
p.T_STIM_END = 110.;
```

Maintenant que la méthode d'implémentation a été explicitée, j'ai les résultats obtenus avec le GPU Tesla K20C de NVIDIA. Pour cela, j'ai exécuté la résolution des équations sur plusieurs tailles de grilles. Comme je l'ai dit précédemment, toutes ces grilles contiennent un seul bloc. La taille de ce bloc va différer à chaque test, comme le nombre de processus lors des tests avec MPI. Il faut savoir que l'architecture Kepler permet de définir des blocs pouvant contenir jusqu'à 1024 threads. Un nombre important au regard des 32 processus lancés pour le MPI, et qui avait requis 2 nœuds du cluster de Rhenovia.

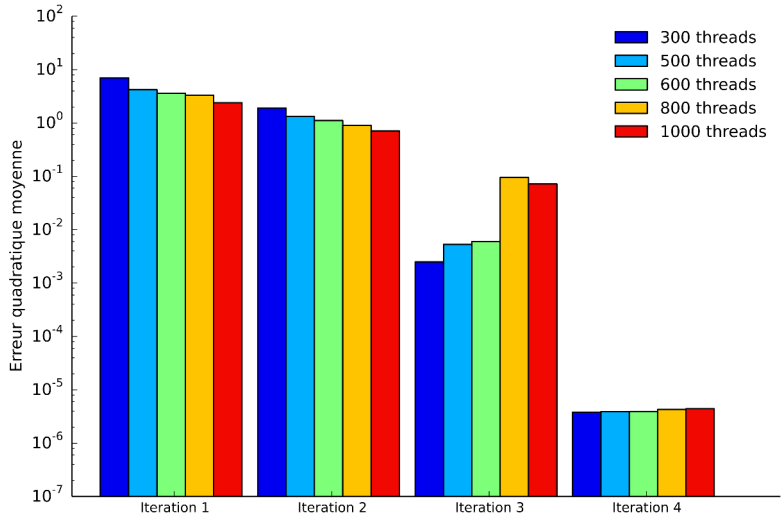

Figure 5.5 – Erreurs quadratiques moyennes pour chaque itération en fonction du nombre de threads CUDA choisi, lors de la simulation du modèle de Hodgkin-Huxley.

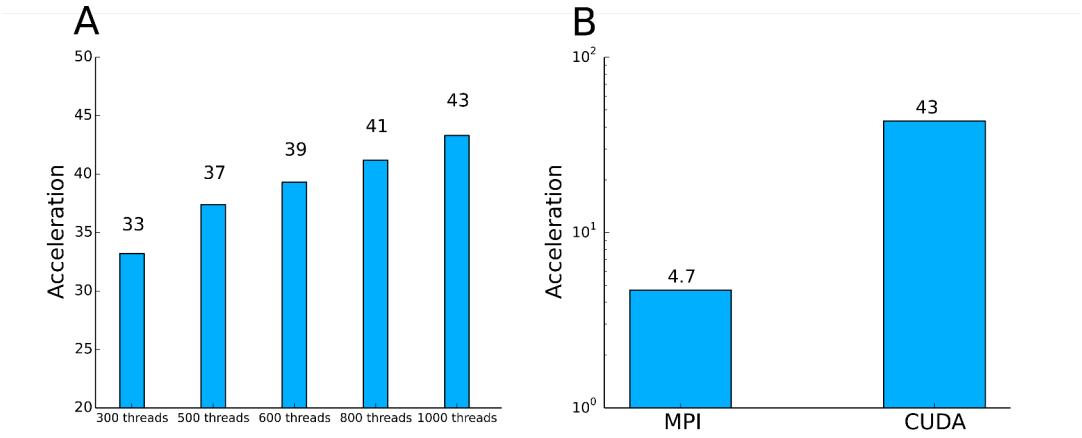

Figure 5.6 – A : Speed-up entre les temps de résolution séquentielle et parallèle (CUDA) du modèle de Hodgkin-Huxley, en fonction de l'itération. **B** : Comparaison entre les meilleurs speed-up obtenus en MPI et CUDA

Les résultats présentés dans la Figure 5.5 montrent que l'erreur commise, entre la solution séquentielle et le solution déterminée à l'aide de l'algorithme pararéel, diminue lorsque le nombre d'itérations augmente. Comme pour la solution MPI, la tolérance est atteinte au cours de la quatrième itération dans tous les cas présentés ici, entraînant l'arrêt immédiat de l'algorithme. Les résultats observés concordent avec ceux obtenus plus tôt, lorsque la technologie MPI était utilisée. La Figure 5.6 présentent les différents speed-up obtenus, suivant

la taille de la grille, et la confrontation avec les speed-up réalisés à l'aide de MPI.

On constate un gain de temps très important lors du passage à CUDA, alors qu'une faible ressource du GPU a été utilisée, contrairement à celle mobilisée en MPI. Il y a donc bien un avantage matériel, comme économique à utiliser cette technologie, même lors de la résolution de problèmes n'étant pas facilement parallélisables, comme un système d'EDO à résoudre.

Il est donc maintenant intéressant de valider ces résultats, lors de la simulation d'un deuxième modèle de neurone, plus simple que le modèle de Hodgkin-Huxley : le modèle de Fitzhugh-Nagumo.

5.3 Deuxième application : le modèle de Fitzhugh-Nagumo

Ce modèle a déjà été introduit dans la section 2.1. Pour rappel, ce modèle est une réduction à deux équations du modèle de Hodgkin-Huxley à partir d'observations expérimentales donnant ainsi lieu à diverses approximations. Les équations de ce modèle sont les suivantes :

$$\begin{cases} \frac{\partial V_m(t)}{\partial t} = V_m(t) - V_m^3(t) - w(t) - I(t) \\ \tau \frac{\partial w(t)}{\partial t} = V_m(t) - a - bw(t) \end{cases} \tag{5.2}$$

où τ, a et b sont des constantes, I le courant externe appliqué. En général, on prend les valeurs de paramètres données par Fitzhugh (1961) :

$$\tau = 13 \quad ; \quad a = 0,7 \quad ; \quad b = 0,8$$

5.3.1 Résolution séquentielle

Lors de la simulation du modèle de Hodgkin-Huxley, j'ai utilisé deux schémas de discrétisation en temps : Euler explicite et RK4. Ces deux méthodes sont à nouveau utilisées pour l'intégration du système d'équations (5.2). A nouveau, j'utilise un pas de temps $\delta\ T\ =\ 10^{-4}\ ms$ pour les deux schémas de résolution. Puis, j'applique le protocole de simulation suivant : je soumet le neurone modélisé à un courant externe durant une période de 100 ms. Les résultats obtenus (Figure 5.7) sont bornés entre -2 et 2, car on obtient des valeurs normalisées. Il convient de les re-dimensionner, si on veut retrouver l'ordre de grandeur du modèle de Hodgkin-Huxley. Cela entraîne que les potentiels d'actions sont plus longs à se déclencher, et également que leur durée est plus importante. Tout ceci peut être modifié en utilisant d'autres valeurs pour les paramètres τ, a et b.

Comparativement à la simulation du modèle de Hodgkin-Huxley, le temps d'intégration est beaucoup moins important, car il n'y a que deux équations à résoudre dans ce système. De plus, aucune fonction exponentielle supplémentaire n'est à mettre à jour ici, entraînant une réduction du temps de calcul supplémentaire. La résolution à l'aide de la méthode d'Euler

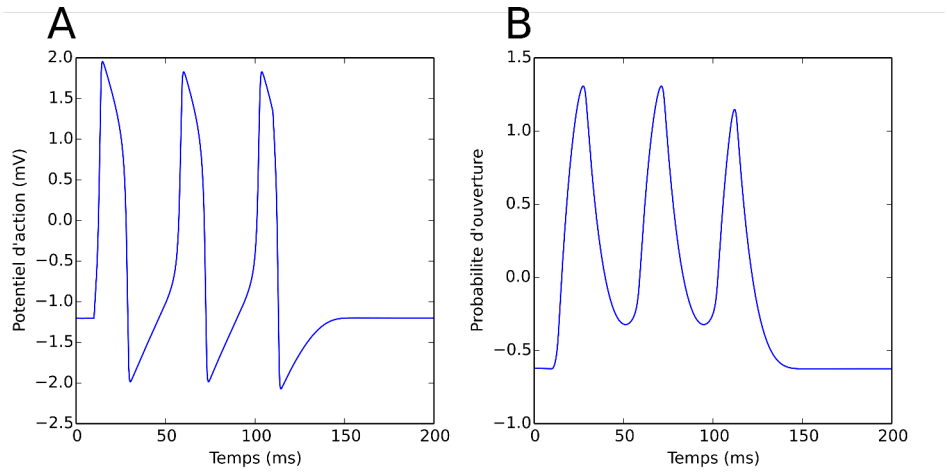

Figure 5.7 – Représentation graphique des fonctions intervenant dans le modèle de Fitzhugh-Nagumo : **A** : potentiel d'action V_m et **B** : la variables w.

est effectuée en environ 60 s, alors qu'il faut 220 s avec RK4, en ayant une erreur quadratique moyenne entre les deux solutions de l'ordre de $2,5 \times 10^{-3}$. Une nouvelle fois, la dynamique plus lente de ce modèle se répercute sur la valeur de l'erreur entre les deux schémas, car dans l'application au modèle Hodgkin-Huxley, l'erreur quadratique entre les deux solutions étaient bien plus grande.

On constate, néanmoins, que le temps de calcul reste important pour un modèle aussi simple, lorsque l'on souhaite, soit une grande précision, soit simuler durant une longue période. Je vais maintenant appliquer la méthode développée dans la section 5.2.

5.3.2 Résolution avec la méthode pararéel en MPI

Afin de résoudre plus rapidement le système d'équations décrivant le modèle de Fitzhugh-Nagumo (5.2), la méthode décrite et développé dans la section 5.2.2 est utilisée maintenant. Peu de choses changent par rapport à la version décrite précédemment. J'ai écrit la méthode de manière à ce qu'elle soit exploitable pour n'importe quel problème d'EDO, en ayant juste à changer les paramètres de l'algorithme, tels que : les pas de temps $(\Delta t\,;\delta t)$, le critère d'arrêt, les conditions initiales du problème, la fonction f contenant le système d'équation sous la forme d'un tableau `res` (voir le code 5.6) :

$$\begin{cases} res_0 = V_m(t) - V_m^3(t) - w(t) - I(t) \\ res_1 = \frac{1}{\tau}(V_m(t) - a - bw(t)) \end{cases} \tag{5.3}$$

La variable `Iapp` n'est pas obligatoire, permettant ainsi d'utiliser toutes les méthodes pour tous les problèmes d'EDO. Ainsi, cette bibliothèque de fonctions permet à d'autres utilisateurs d'utiliser cette méthode, sans le moindre développement fastidieux de leur part.

Code 5.6 – Définition de la fonction f contenant les équations du système.

```
# f permet de construire un tableau
# contenant dy/dt
def f(t,y,Iapp=None):

    res = np.zeros(2,dtype='float')
    res[0] = y[0] - y[0]**3/3. - y[1] + Iapp
    res[1] = (1./tau)*(y[0] + aa - bb*y[1])

    return res
```

Afin de simuler le modèle de Fitzhugh-Nagumo, on utilise à nouveau les pas de temps suivants :

$$\Delta t = 3,125 \times 10^{-3} \quad ; \quad \delta t = 1,0 \times 10^{-4} \quad ,$$

Le protocole de simulation détaillé dans la section 5.3.1 est à nouveau utilisé, afin de comparer au mieux les solutions séquentielle et parallèle.

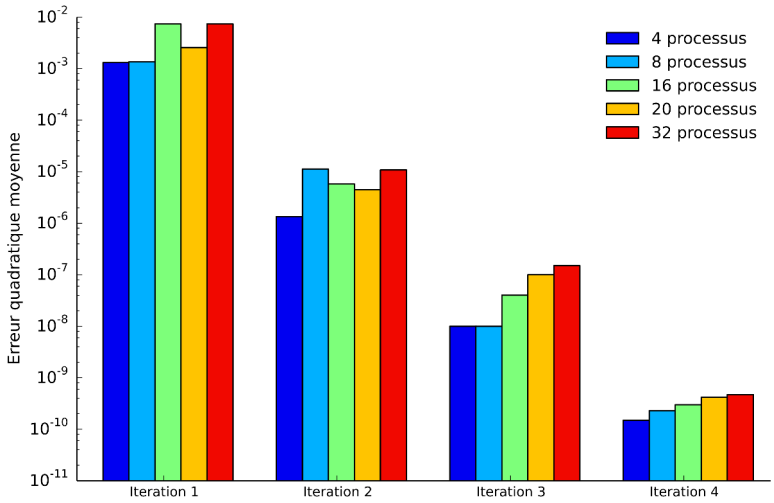

Figure 5.8 – Erreurs quadratiques moyennes pour chaque itération en fonction du nombre de processus MPI choisi, lors de la simulation du modèle de Fitzhugh-Nagumo.

La Figure 5.8 présente les résultats concernant l'erreur commise à chaque itération. On constate, à nouveau, que l'erreur à tendance à baisser au fil des itérations, pour au final atteindre rapidement la convergence aux alentours de la troisième itération, suivant la tolérance souhaitée. On observe également une fluctuation des erreurs, en fonction du nombre de processus lancés. Les résultats observés lors de la simulation du modèle précédent sont

donc confirmés, avec une vitesse de convergence accrue. Ceci s'explique par la complexité inférieure du modèle.

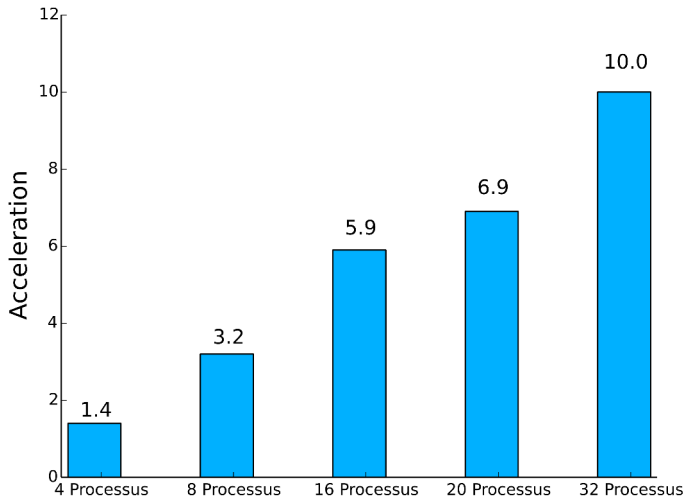

Figure 5.9 – Accélérations entre les temps de résolution séquentielle et parallèle (MPI) du modèle de Fitzhugh-Nagumo, en fonction de l'itération.

En plus des erreurs, je détermine le gain de temps observé, en calculant l'accélération, présenté Figure 5.9. Comme prévu, l'accélération augmente lorsque le nombre de processus augmente, entraînant un gain de temps non négligeable, puisque l'accélération de la résolution est d'un facteur 10 lorsque 32 processus sont utilisés. Bien évidemment, théoriquement avec 32 processus on peut arriver au mieux à un speed-up de 32, mais cette valeur théorique n'est, en pratique, jamais atteinte. Néanmoins, le résultat est plus que satisfaisant lorsque l'on souhaite obtenir une solution de grande qualité, ou encore simuler sur de longues périodes.

Les résultats en MPI du modèle de Hodgkin-Huxley ont été confortés par l'application au modèle de Fitzhugh-Nagumo. Il est évident que je vais également tenter de valider les résultats obtenus en CUDA, en l'appliquant à ce même modèle.

5.3.3 Résolution avec la méthode pararéel en CUDA

Comme pour la méthode développée avec MPI, j'applique celle programmée avec CUDA pour la tester sur le modèle de Fitzhugh-Nagumo. J'ai effectué le même travail de généralisation détaillé dans la section précédente, pour permettre l'intégration de n'importe quel problème d'EDO. Nous avons déjà vu la structure `Param` regroupant les différents paramètres de simulation. Les solveurs RK4 et Euler explicite, permettant d'effectuer les résolutions grossière et fine, prennent en paramètre une fonction `f` faisant le calcul et stockant les calculs des dérivées dans un tableau `res`, de la même manière qu'en MPI (voir le code 5.7)

Code 5.7 – Définition de la fonction f contenant les équations du système en CUDA.

```
__device__ void f(int i,int idx,double *UG,double *res,Param p){

    res[0] = UG[0] - pow(UG[0],3)/3. - UG[1] + Stim(idx*Nb + i-1,dtf,coeff,p);
    res[1] = (1./13.)*(UG[0] + 0.7 - 0.8*UG[1]);

}
```

L'utilisateur doit uniquement définir les conditions initiales du problème, en plus des différents paramètres et de cette fonction `f`, pour que l'intégration fonctionne. Le reste est totalement transparent pour lui. Il lui incombe néanmoins de définir lui même les fonctions, ou paramètres supplémentaires dont il aurait besoin, comme par exemple, la fonction `Stim(...)` , appliquant un courant externe au modèle, présente dans le code 5.7.

Afin de réaliser mes différentes simulations, je prends le protocole de simulation détaillé dans la section précédente. Les résultats de ces simulations sont synthétisés dans la Figure 5.10. On remarque une diminution de l'erreur, identique aux précédents tests, lorsque l'itération augmente. Une nouvelle fois, une erreur acceptable de l'ordre de 10^{-5} est atteinte lors des itérations trois ou quatre, montrant une convergence rapide. La fluctuation des erreurs observée lors des tests MPI semble plus mesurée ici, entrainant une erreur quasi-identique à la quatrième itération.

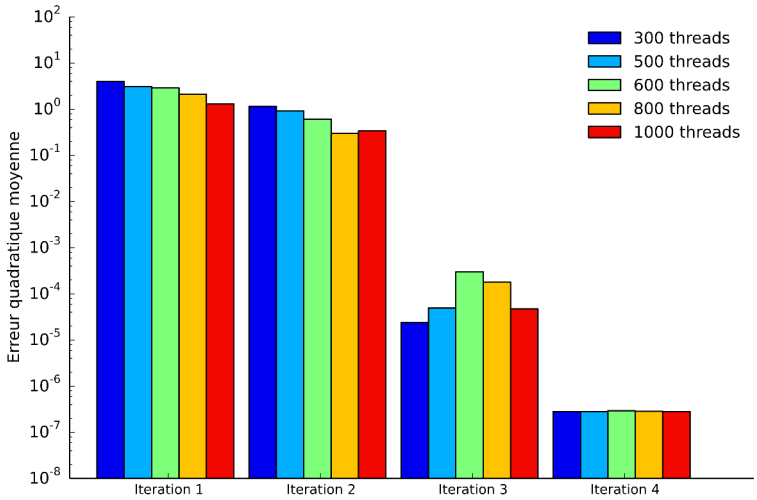

Figure 5.10 – Erreurs quadratiques moyennes pour chaque itération en fonction du nombre de threads CUDA choisi, lors de la simulation du modèle de Fitzhugh-Nagumo.

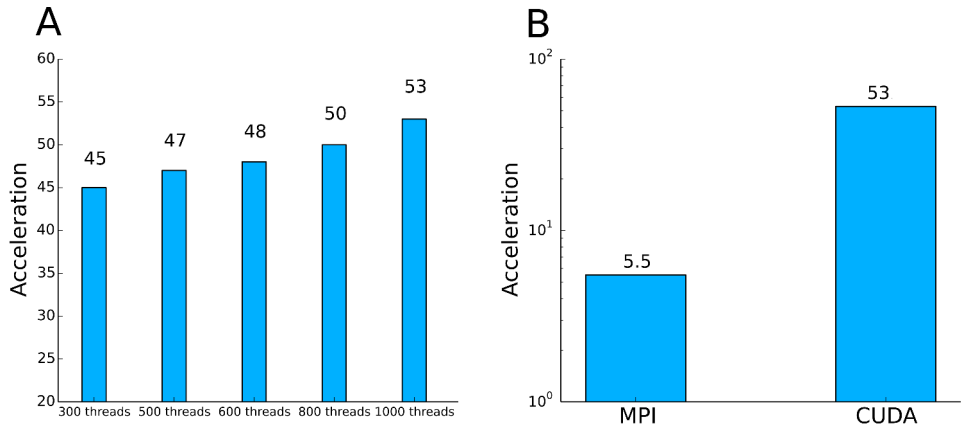

Figure 5.11 – A : Accélérations entre les temps de résolution séquentielle et parallèle (CUDA) du modèle de Fitzhugh-Nagumo, en fonction de l'itération. **B** : Comparaison entre les meilleures accélérations obtenus en MPI et CUDA.

On détermine à nouveau les accélérations afin de valider les gains de temps enregistrés lors de la simulation du modèle de Hodgkin-Huxley (Figure 5.11). On observe à nouveau un gain important lors de l'utilisation de l'algorithme pararéel en CUDA. Le maximum est dix fois supérieur au meilleur gain obtenu avec MPI (Figure 5.11 **B**). Encore une fois, il est à rappeler que dans le cadre d'une simulation sur GPU, une seule carte graphique est utilisée, alors que lors de l'application MPI, deux nœuds complets du cluster de Rhenovia sont employés sur les huit disponibles. Cela amène à des temps de résolution de l'ordre de la seconde, là où il faut quelques minutes en séquentiel.

5.4 Conclusion

Dans ce chapitre, j'ai détaillé le développement de l'algorithme pararéel grâce à deux technologies : MPI et CUDA. Ce développement a amené à la construction d'une petite bibliothèque de calcul, pouvant être facilement adopté à n'importe quel problème d'EDO. Son mode de fonctionnement a été expliqué, puis appliqué à la simulation de deux modèles de neurone. Il a permis de constater le gain évident apporté par cette méthode. Il est à noter que l'avantage du pararéel ne se limite pas à une diminution du temps de résolution, mais qu'il peut aussi être utilisé afin de prendre en compte deux dynamiques au sein d'un même modèle. Un modèle n'a pas forcément le même comportement suivant l'échelle de temps sur lequel on le regarde, d'où l'utilité d'appliquer des paramètres différents suivants que l'on se trouve sur l'échelle de temps grossière ou fine.

6 Simulations du modèle monodomaine à l'aide des différences finies

6.1 Introduction

Ce chapitre va traiter de la simulation, à l'aide de la méthode des différences finies, du modèle monodomaine introduit dans la section 2.2.9. Ce modèle est une version simplifiée du modèle bidomaine et s'énonce comme une équation de réaction-diffusion sur le potentiel de membrane V_m couplé avec un système d'EDO :

$$\begin{cases} A_m\left(C_m\dfrac{\partial V_m}{\partial t} + f(V_m, w)\right) = \dfrac{1}{\lambda+1}\nabla\cdot(M_i\nabla V_m) \\ \dfrac{\partial w}{\partial t} = g(V_m, w) \end{cases} \tag{6.1}$$

Le système g d'EDO est basé sur les modèles de neurones vus dans la section 2.1. Bien que ce modèle soit simplifié, il présente toujours une difficulté du modèle bidomaine : l'existence d'une dynamique lente et d'une dynamique rapide, dépendant de la conductance C_m. La variable V_m est donc rapide par rapport à w, entraînant des modifications soudaines, comme on a déjà pu le voir avec le modèle de Hodgkin-Huxley. Ces différentes dynamiques entraînent des instabilités numériques (Pierre, 2005). Il est ainsi nécessaire de faire appel à des résolutions assez fines, afin d'obtenir des résultats stables.

Après une étude de ce modèle, nous développerons les méthodes utilisées dans le chapitre 5 afin de diminuer le temps de simulation de ce modèle, tout en ayant une exigence certaine sur la qualité des résultats. Ces méthodes ont déjà prouvé leur efficacité sur les systèmes d'EDO, que nous retrouvons en partie dans le modèle monodomaine. Contrairement au chapitre précédent, la dimension spatiale est à prendre ici en considération.

Ce développement amènera, en plus, à l'écriture de méthodes en Python puis en CUDA permettant d'effectuer des calcul fonctionnant sur le même principe que celle présentée dans le chapitre 5.

6.2 Simulations en dimension un

On considère le modèle monodomaine sur $\mathbb{R}$ décrit par les équations :

$$\begin{cases} A_m\left(C_m\dfrac{\partial V_m}{\partial t}+f(V_m,w)\right)=\dfrac{1}{\lambda+1}m_i\Delta V_m \\ \dfrac{\partial w}{\partial t}=g(V_m,w) \quad , \end{cases} \tag{6.2}$$

où $m_i \in \mathbb{R}$ est la conductivité du milieu intra-cellulaire.

Pour la simulation de ce modèle, on considère le système d'équations (6.2) sur le segment $[0;1]$ pour des fonctions f et g de type Fitzhugh-Nagumo décrites par les équations :

$$\begin{cases} f(V_m,w)=V_m-V_m^3-w-I(t) \\ g(V_m,w)=V_m-a-bw \quad . \end{cases} \tag{6.3}$$

On complète ces équations par des conditions aux limites de type Neumann homogènes en 0 et en 1 :

$$\frac{\partial V_m}{\partial x}(0)=0 \qquad , \qquad \frac{\partial V_m}{\partial x}(1)=0 \quad . \tag{6.4}$$

Les solutions à ce type de problème ont été largement étudiées, notamment dans la thèse de Pierre (2005), mais également dans diverses autres études (Hastings, 1976; Smoller, 1994; Fife et McLeod, 1977).

Supposons un maillage du segment $[0,1]$ en N sous-segments de taille identique $\Delta x = 1/N$. On définit $V_{m,j}^n$ l'approximation de $V_m(x_j,t_n)$ où $x_j = j\Delta x$, $t_n = n\Delta t$.

La discrétisation du système (6.4) s'écrit donc :

$$\begin{cases} A_m C_m\dfrac{V_{m,j}^{n+1}-V_{m,j}^n}{\Delta t}=\dfrac{d}{\Delta x^2}AV_m^n-I_{ion}(V_{m,j}^n,w_j^n) \\ \dfrac{w_j^{n+1}-w_j^n}{\Delta t}=g(V_{m,j}^n,w_j^n) \\ d=\dfrac{m_i}{\lambda+1} \quad , \end{cases} \tag{6.5}$$

où $(V_m^n)_{n\geq 0}$ est une suite finie de fonctions dont les valeurs sont dans $L^2([0,1])$. De plus, la discrétisation de l'opérateur Laplacien est donnée par la matrice tridiagonale A suivante :

$$A = \begin{pmatrix} -1 & 1 & & & \\ 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -2 & 1 \\ & & & 1 & -1 \end{pmatrix} .$$

Le schéma présenté est de type explicite, mais il est également possible d'utiliser le schéma RK4 afin de se rapprocher de l'utilisation des méthodes décrites dans le chapitre 5.

6.2.1 Etude de convergence

Il semble important d'expliquer pourquoi le schéma semi-implicite n'est pas retenu ici. En se basant sur les résultats de la thèse de Pierre (2005), il a été montré que le choix du pas de temps Δt est dicté par des conditions de stabilité pour les schémas explicites, mais également implicites. Contrairement à l'équation de la chaleur, où un schéma d'Euler implicite est inconditionnellement stable, la présence d'un terme de réaction, ici, entraîne une contrainte importante sur le pas de temps Δt. Au final, pour des maillages assez fins, la contrainte de stabilité sur Δt sera équivalente pour les deux schémas. De plus, lors d'une résolution à l'aide d'un schéma implicite, il est nécessaire de résoudre un système linéaire, autrement dit une inversion de matrice est indispensable. L'utilisation d'un tel schéma devient intéressante lorsque l'on souhaite recourir à des techniques de pas adaptatifs, car la contrainte de stabilité imposée au schéma implicite s'applique lorsque nous sommes en présence de zones raides, comme les fronts de dépolarisation et de repolarisation. Finalement, dans le cas d'un pas de temps fixe, le temps de résolution sera moindre pour un schéma explicite (Figure 6.1).

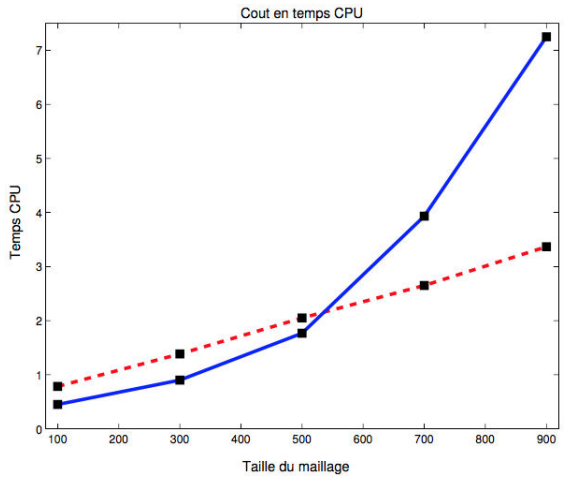

Figure 6.1 – Coût en temps CPU comparés entre un schéma explicite (traits pleins) et un schéma semi-implicite (pointillés) d'après Pierre (2005).

Le temps nécessaire pour la résolution suppose d'utiliser un schéma explicite, mais une autre comparaison doit être faite à précision égale. Ce point a été discuté également dans la thèse de Pierre (2005), où il a été montré que le schéma explicite était le plus précis (Figure 6.2). Suite, à ces différents points j'ai décidé, pour le travail présenté dans ce document, d'intégrer les équations du modèle à l'aide de la méthode explicite, conjointement à la méthode RK4 pour la mise à jour de la variable w lors de l'implémentation parallèle.

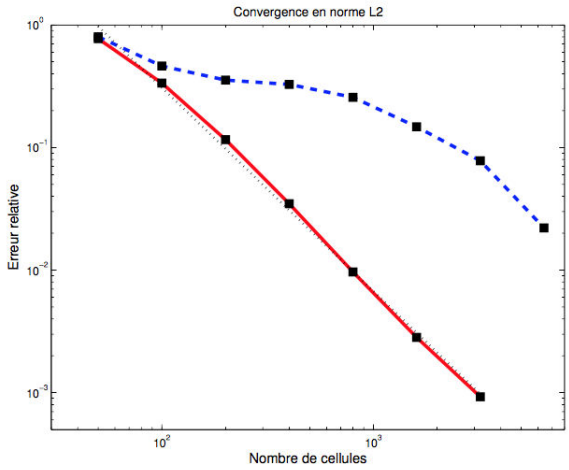

Figure 6.2 – Erreur relative en norme L^2, pour un schéma explicite (traits pleins) et un schéma semi-implicite (pointillés) ; la droite en pointillé est de pente -1.66 (d'après Pierre (2005)).

6.2.2 Resolution séquentielle

Je m'attache maintenant à effectuer la résolution du problème (6.2) à l'aide d'un schéma explicite. Le développement est fait en Python, permettant ainsi une utilisation des différentes bibliothèques mathématiques disponibles, et une grande souplesse d'écriture. Je soumet le modèle monodomaine à un protocole de simulation définissant plusieurs critères, notamment le temps de simulation, les pas de temps et d'espace, les paramètres de la stimulation. En ce qui concerne le courant externe appliqué (stimulation), j'excite une portion du segment $[0;1]$ durant une période (Figure 6.3).

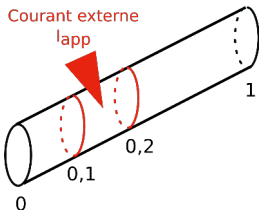

Figure 6.3 – Représentation schématique du segment $[0;1]$ où la portion $[0,1;0,2]$ est soumis à un courant externe I_{app}.

On récapitule dans le tableau 6.1, les différents paramètres du modèles, ainsi que leurs valeurs.

Paramètres	Valeurs
T_{final}	$100\ ms$
Δt	$1,0 \times 10^{-4}\ ms$
Δx	$1,0 \times 10^{-3}$
A_m	$2000\ cm^{-1}$
C_m	$1\ \mu F.cm^{-1}$
λ	$0,1$
m_i	$1\ mS.cm^{-1}$
$[T_i; T_f]_{stim}$	$[10; 100]$
$[x_i; x_f]_{stim}$	$[0,1; 0,2]$
I_{app}	$0,4\ \mu A$

Tableau 6.1 – Valeurs des paramètres de simulation du modèle monodomaine en dimension un.

A partir des ces paramètres, un objet Python `SimParam` est créé grâce à la classe 6.1. Des valeurs par défaut sont définies, avec les valeurs présentées dans le tableau 6.1. Cela permet à l'utilisateur de modifier rapidement certains éléments de la simulation.

Code 6.1 – Une partie de la classe SimParam définissant les paramètres de simulation.

```
1  class SimParam():
2      def __init__(self,Tf=100.,dt=1.e-4,Nx=1000,lamb=0.1,mi=1.,T_stim_start = 10.,
         T_stim_end=100.,I_app=0.4):
3          self.Tf = Tf
4          self.dt = dt
5          self.Nx = Nx
6          self.dx = 1./Nx
7          self.lamb = lamb
8          self.mi = mi
9          self.T_stim_start = T_stim_start
10         self.T_stim_end = T_stim_end
11         self.I_app = I_app
12
13     # Autres méthodes de la classe SimParam
14     ...
```

Ensuite l'utilisateur peut effectuer la simulation du modèle monodomaine, en utilisant la fonction `MonodomainSimulation1D(...)` (voir code 6.2), en spécifiant l'objet contenant les paramètres de Simulation, ainsi que le schéma de résolution choisi. Malgré les observations faites dans la section 6.2.1, le choix de la méthode parmi l'explicite, l'implicite, et RK4 est donné à l'utilisateur.

Code 6.2 – Fonction lançant la simulation du modèle monodomaine 1D.

```
# method = "Explicit", "Implicit", "RK4"
def MonodomainSimulation1D(param,method="Explicit"):
    if param is None:
        sys.exit("Erreur ! Aucun paramètre de simulation défini !")
    # Exécution de la simulation
    ...
```

La Figure 6.4 présente les résultats de la simulation effectuée avec les paramètres du tableau 6.1. Pour les obtenir, la simulation enregistre le signal en trois positions distinctes au cours du temps. Dans un second temps (Figure 6.5), j'enregistre le signal en chaque point du segment [0; 1] à trois dates différentes.

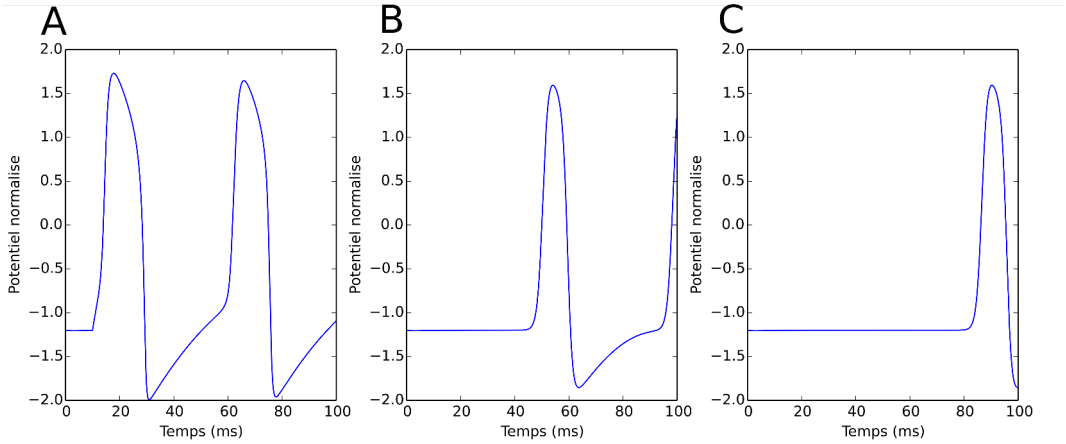

Figure 6.4 – Représentation graphique du potentiel de membrane, suite à la simulation du modèle monodomaine en une dimension en différentes positions : **A** : x = 0,2 ; **B** : x = 0,4 et **C** : x = 0,6.

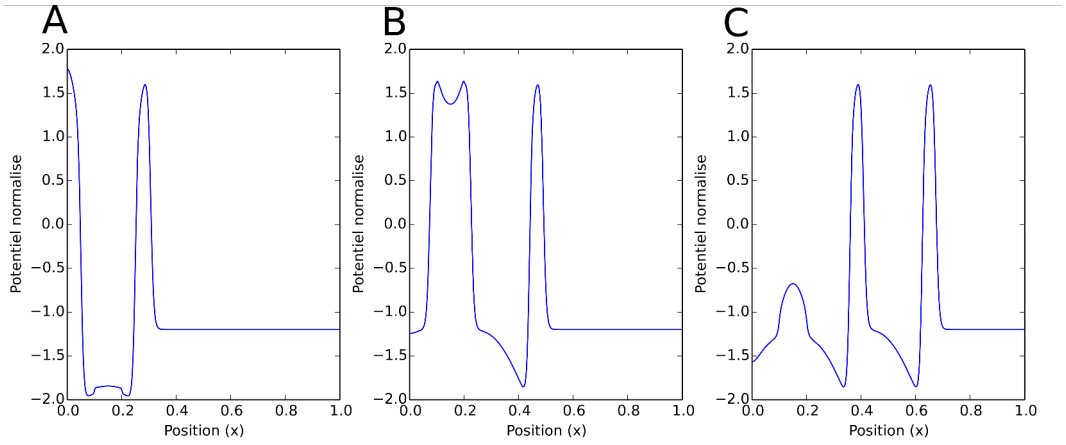

Figure 6.5 – Représentation graphique du potentiel de membrane en chaque position du segment [0; 1] à des dates différentes : **A** : t = 35 ms ; **B** : x = 65 ms et **C** : x = 100 ms.

On observe ainsi le déplacement des potentiels d'action le long du segment. Une faible perte d'amplitude peut être aperçue lorsque l'on quitte la zone $[0,1;0,2]$, où le potentiel est généré, mais la forme générale ainsi que les valeurs du modèle de Fitzhugh-Nagumo sont bien retrouvées. Cette perte de signal est due au paramètre de conductance. Bien évidemment, le nombre de potentiels d'action créés durant cette simulation dépend des paramètres utilisés pour le calcul de la variable w, mais également lors de la détermination de la fonction f servant au calcul du courant ionique dans le modèle monodomaine. La vitesse de propagation dépend quand à elle des paramètres A_m, C_m et de la conductance intra-cellulaire m_i.

6.3 Simulations en dimension deux

Maintenant, je vais effectuer la simulation du modèle monodomaine en dimension deux. Il est décrit sur $\mathbb{R}^2$ grâce au système d'équations (6.1) où M_i est un tenseur de conductivité représenté par :

$$M_i = \begin{pmatrix} c_x & 0 \\ 0 & c_y \end{pmatrix} \quad ,$$

où c_x et c_y représentent respectivement la conductivité selon l'axe x et l'axe y.

Cette simulation va s'effectuer sur le carré $\Omega = [0;1]^2$, en utilisant à nouveau des fonctions f et g de type Fitzhugh-Nagumo. Le système d'équation (6.1) est complété par une condition de type Neumann homogène :

$$M_i \nabla V_m \cdot \mathbf{n}_{\partial\Omega} = 0 \quad . \tag{6.6}$$

Pour la résolution de ce problème, supposons deux entiers N_x et N_y égaux, représentant le nombre de nœuds dans chaque direction x et y. Ce choix permet d'obtenir un maillage uniforme de l'espace de définition avec des carrés de taille identique $\Delta x \times \Delta y$, avec $\Delta x = 1/N_x$ et $\Delta_y = 1/N_y$. Chaque maille est identifiée à l'aide de ses coordonnées : $(x_i; y_j) = (i\Delta x; j\Delta y)$. Attention, l'indice i ici ne correspond pas à l'indice i de la matrice M_i, qui dans ce cas signifie « intra-cellulaire ». On approche la solution $V_m(x_i, y_j, t_n)$ par ses valeurs en chacun de ces nœuds $V^n_{m,i,j}$. Afin de simplifier les notations, nous modifions, dans le schéma numérique, la notation V_m par u. Ansi, on a : $u^n_{i,j} = V^n_{m,i,j}$. Cela évitera au lecteur de se perdre dans les trop nombreux indices.

Le point sensible dans cette discrétisation intervient dans le calcul de : $\nabla \cdot (M_i \nabla V_m)$, en considérant les conditions de Neumann. On a :

$$\nabla \cdot (M_i \nabla u) = c_x \frac{\partial^2 u}{\partial x^2} + c_y \frac{\partial^2 u}{\partial y^2} \quad , \tag{6.7}$$

qui devient, en approchant par des différences finies :

$$c_x \frac{\partial^2 u}{\partial x^2} + c_y \frac{\partial^2 u}{\partial y^2} = c_x \frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{\Delta x^2} + c_y \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{\Delta y^2} \quad . \tag{6.8}$$

De plus, on suppose que $\Delta_x = \Delta_y = h$, alors la relation (6.8) devient :

$$c_x \frac{\partial^2 u}{\partial x^2} + c_y \frac{\partial^2 u}{\partial y^2} = \frac{c_x(u_{i-1,j} - 2u_{i,j} + u_{i+1,j}) + c_y(u_{i,j-1} - 2u_{i,j} + u_{i,j+1})}{h^2} \quad . \tag{6.9}$$

En arrangeant, on a :

$$c_x \frac{\partial^2 u}{\partial x^2} + c_y \frac{\partial^2 u}{\partial y^2} = \frac{-2(c_x + c_y)u_{i,j} + c_x(u_{i-1,j} + u_{i+1,j}) + c_y(u_{i,j-1} + u_{i,j+1})}{h^2} \quad . \tag{6.10}$$

D'un point de vue matriciel, on définit le vecteur solution U de la manière suivante :

$$U = \left(\begin{array}{c} u_{0,0} \\ u_{1,0} \\ \vdots \\ u_{N_x-1,0} \\ \hline u_{0,1} \\ \vdots \\ u_{N_x-1,1} \\ \hline \vdots \\ \hline u_{0,N_y-1} \\ \vdots \\ u_{N_x-1,N_y-1} \end{array}\right) \quad .$$

On peut écrire le système global $\nabla \cdot (M_i \nabla u)$ comme un problème linéaire de la forme AU, avec les différences finies, grâce aux matrices $F, D \in \mathbb{R}^{N_x \times N_x}$ suivante :

$$F = \begin{pmatrix} -2(c_x + c_y) & c_x & 0 & 0 & \dots & 0 \\ c_x & -2(c_x + c_y) & c_x & 0 & \dots & 0 \\ & \ddots & \ddots & \ddots & & \\ 0 & \dots & 0 & 0 & c_x & -2(c_x + c_y) \end{pmatrix}$$

$$D = \begin{pmatrix} c_y & 0 & 0 & 0 & \dots & 0 \\ 0 & c_y & 0 & 0 & \dots & 0 \\ & \ddots & \ddots & \ddots & & \\ 0 & \dots & 0 & 0 & 0 & c_y \end{pmatrix} ,$$

et à la matrice globale $A \in \mathbb{R}^{N_x^2 \times N_x^2}$:

$$A = \begin{pmatrix} F & D & 0 & 0 & \dots & 0 \\ D & F & D & 0 & \dots & 0 \\ & \ddots & \ddots & \ddots & & \\ 0 & \dots & 0 & 0 & D & F \end{pmatrix} .$$

Il reste néanmoins à intégrer les conditions de Neumann au sein de ces matrices. La matrice globale A devient donc :

$$A = \begin{pmatrix} E & D & 0 & 0 & \dots & 0 \\ D & F & D & 0 & \dots & 0 \\ & \ddots & \ddots & \ddots & & \\ 0 & \dots & 0 & D & F & D \\ 0 & \dots & 0 & 0 & D & E \end{pmatrix} ,$$

avec les matrices E, F et $D \in \mathbb{R}^{N_x \times N_x}$ suivantes :

$$E = \begin{pmatrix} -(c_x + c_y) & c_x & 0 & 0 & \dots & 0 \\ c_x & -(2c_x + c_y) & c_x & 0 & \dots & 0 \\ & \ddots & \ddots & \ddots & & \\ 0 & \dots & 0 & c_x & -(2c_x + c_y) & c_x \\ 0 & \dots & 0 & 0 & c_x & -(c_x + c_y) \end{pmatrix}$$

$$F = \begin{pmatrix} -(c_x + 2c_y) & c_x & 0 & 0 & \dots & 0 \\ c_x & -2(c_x + c_y) & c_x & 0 & \dots & 0 \\ & \ddots & \ddots & \ddots & & \\ 0 & \dots & 0 & c_x & -2(c_x + c_y) & c_x \\ 0 & \dots & 0 & 0 & c_x & -(c_x + 2c_y) \end{pmatrix}$$

$$D = \begin{pmatrix} c_y & 0 & 0 & 0 & \dots & 0 \\ 0 & c_y & 0 & 0 & \dots & 0 \\ & \ddots & \ddots & \ddots & & \\ 0 & \dots & 0 & 0 & 0 & c_y \end{pmatrix} .$$

Une fois la discrétisation du terme $\nabla \cdot (M_i \nabla V_m)$ effectuée, il ne reste plus qu'à définir l'écriture du schéma de résolution du système (6.1). On note U^n le vecteur U au temps t_n, et W^n le vecteur contenant les valeurs de $w_{i,j}^n$ sur le même modèle que U. On obtient alors :

$$\begin{cases} U^{n+1} = U^n + \dfrac{1}{A_m C_m (1+\lambda)} A U^n - \dfrac{1}{C_m} f(U^n, W^n) \\ W^{n+1} = g(U^n, W^n) \quad . \end{cases} \tag{6.11}$$

On utilise à nouveau un schéma explicite pour la simulation du modèle en deux dimensions, dont les modalités sont détaillées dans la section suivante.

6.3.1 Résolution séquentielle

Afin de simuler le modèle monodomaine en deux dimensions, je développe le simulateur en Python, afin de profiter de la facilité d'écriture des calculs matriciels. Un protocole de simulation, contenant les différents paramètres de stimulation, le pas de temps, la taille du maillage, etc. est défini. Ce dernier permettra à l'utilisateur de pouvoir exécuter rapidement une simulation avec ses propres paramètres. On retrouve ainsi une grosse partie des paramètres se trouvant dans la classe `SimParam` de la section 6.2.2, en plus des paramètres inhérents à la dimension deux.

Ainsi la classe `SimParam` devient ici (code 6.3) :

Code 6.3 – Classe SimParam pour la simulation 2D.

```
class SimParam():
    def __init__(self, Tf=100., dt=1.e-4, N = 100, lamb=0.1, cx=1., cy=1., cur=None):
        self.Tf = Tf
            self.dt = dt
                self.Nx = N
        self.Ny = N
            self.dx = 1./Nx
        self.dy = 1./Ny
        self.lamb = lamb
        self.cx = cx
        self.cy = cy
        self.T_stim_start = T_stim_start
        self.T_stim_end = T_stim_end
        self.I_app = I_app

```

```
16          if(cur is None):
17              self.cur = CurrentApplied()
18          else:
19              self.cur = cur
20
21          # Autres méthodes de la classe SimParam
22          ...
```

Cette classe fait appel à un nouvel objet `CurrentApplied` (Code 6.4) permettant de définir tous les paramètres indispensables à l'application d'un courant externe, afin d'exciter le modèle. Cette stimulation doit être définie sur une portion carré du domaine $[x_i; x_f] \times [y_i; y_f]$ durant une certaine période (Figure 6.6). Ainsi un objet `CurrentApplied` doit-être initialisé avant la création d'un objet `SimParam`, sinon les valeurs définies par défaut seront prises.

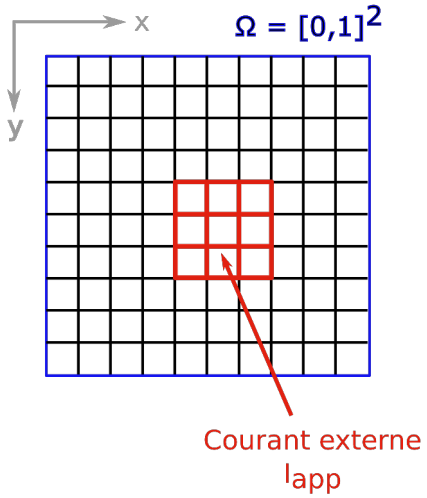

Figure 6.6 – Représentation schématique du domaine de définition à deux dimension $\Omega = [0;1]^2$ soumis à un courant externe I_{app}. Chaque carré représente une maille de la dicrétisation.

Code 6.4 – Classe CurrentApplied pour la simulation 2D.

```
1  class CurrentApplied():
2      def __init__(self, xi=0.4, xf=0.6, yi=0.4, yf=0.6, T_stim_start=10., T_stim_end
         =100., I_app=0.4):
3          self.xi = xi
4          self.xf = xf
5          self.yi = yi
6          self.yf = yf
7          self.T_stim_start = T_stim_start
8          self.T_stim_end = T_stim_end
9          self.I_app = I_app
10
11     # Autres méthodes
12     ...
```

Pour nos différents tests, nous prenons les paramètres par défaut (Table 6.2).

Paramètres	Valeurs
T_{final}	100 ms
Δt	$1,0 \times 10^{-4}$ ms
Δx ; Δy	$5,0 \times 10^{-2}$; $5,0 \times 10^{-2}$
A_m	2000 cm^{-1}
C_m	1 $\mu F.cm^{-1}$
λ	0,1
c_x	1 $mS.cm^{-1}$
c_y	1 $mS.cm^{-1}$
$[T_i; T_f]_{stim}$	[10;100]
$[x_i; x_f]_{stim} \times [y_i; y_f]_{stim}$	$[0,4;0,6] \times [0,4;0,6]$
I_{app}	0,4 μA

Tableau 6.2 – Valeurs des paramètres de simulation du modèle monodomaine en dimension deux.

Le but de ce test est d'obtenir une solution ayant une excellente précision dans la direction temporelle. Pour ce faire, le pas de temps Δt est fixé à $1,0 \times 10^{-4}$. Les résultats obtenus sont représentés dans la Figure 6.7, montrant que l'amplitude du potentiel d'action, au même titre que la simulation en une dimension, perd de son intensité lorsque l'on s'éloigne de son lieu de genèse, généralement à cause des valeurs de conductance. On observe bien évidemment la propagation du signal électrique, puis un retour à la valeur de repos après son passage.

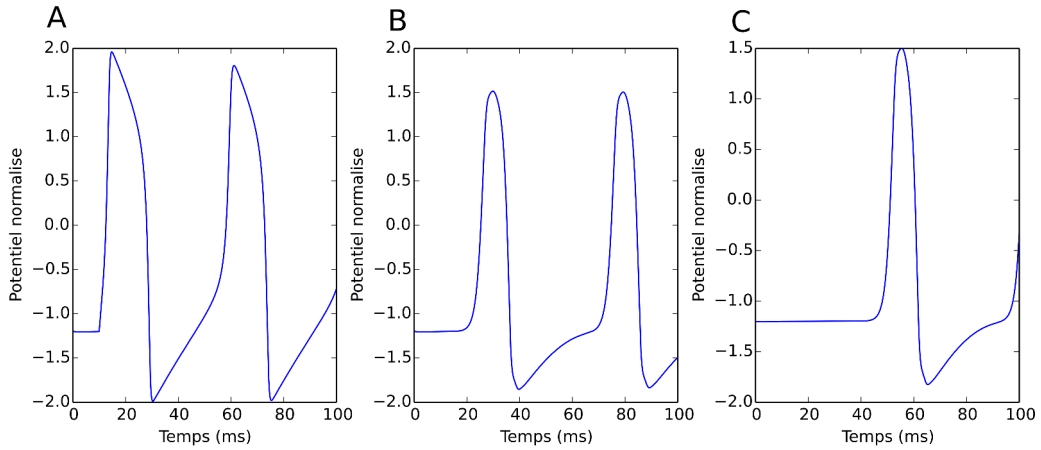

Figure 6.7 – Représentation graphique du potentiel de membrane, suite à la simulation du modèle monodomaine en deux dimensions en différentes positions : **A** : (x ; y) = (0,5 ; 0,5) ; **B** : (x ; y) = (0,5 ; 0,7) et **C** : (x ; y) = (0,5 ; 0,9).

Hélas, le temps de résolution devient important, puisqu'il faut environ 2500 s soit l'équivalent

de 41 min pour la simulation avec cet ensemble de paramètres. Afin de diminuer le temps de calcul,on peut adapter la méthode de résolution parallèle utilisée dans le chapitre 5, puis la tester afin de voir si les résultats sont reproduits, et si un gain de temps est effectivement observé.

6.3.2 Simulations avec la méthode pararéel en MPI

La même manière d'opérer vue dans le chapitre 5 a été appliquée. Pour pouvoir utiliser au mieux l'algorithme pararéel distribué, on doit pouvoir simuler le modèle sur deux échelles de temps différentes : une fine et une grossière. Pour cela, on utilise, à nouveau, les deux schémas de résolution : Euler explicite et RK4, ce qui se traduit par l'implémentation de deux méthodes différentes, pouvant être appelées à tout moment lors de l'application de l'algorithme pararéel. En plus de ces deux intégrateurs, j'ai aussi rajouté une méthode : Euler implicite.

Les différentes classes `SimParam` et `CurrentApplied`, introduites dans la section précédente, sont utilisées afin de condenser au mieux toutes les données du problème. On rajoute néanmoins un nouveau paramètre, déjà-vu dans le chapitre 5, représentant la tolérance souhaitée pour l'arrêt de l'algorithme pararéel.

Il faut ensuite construire la matrice A. Un processus MPI est en charge de cette tâche. Puis lorsque cette construction est finie, il la partage avec tous les autres processus, afin qu'ils puissent l'utiliser lors de la résolution sur leur intervalle de temps $[t_n; t_{n+1}]$. Cette construction dépendra bien évidemment des paramètres entrés par l'utilisateur, tels que les conductances, la taille du maillage, etc.

Une des difficultés, dépendant de la taille du maillage, vient du fait que chaque valeur du maillage, au temps final du sous-intervalle $[t_n; t_{n+1}]$ doit-être communiquée. Cela entraîne une communication importante entre les différents processus, pouvant entraîner l'interruption du programme. Il faut savoir que les fonctions MPI : `Send()` et `Recv()` permettent d'envoyer et de recevoir des messages, jusqu'à une certaine limite de taille, qui est le plus souvent donnée par la constante `INT_MAX` correspondant à 2×10^9 éléments au sein d'un tableau. Cela semble important, mais il est à prendre en considération que plusieurs valeurs sont à sauvegarder tels que V_m et w. Si les fonctions f et g du système (6.1) suivent d'autres modèles, comme celui de Hodgkin-Huxley, quatre variables sont à envoyer. Ce point peut avoir une incidence, notamment lorsque tous les résultats finaux doivent être récupérés. A chaque pas de temps de taille 10^{-4}, on récupère toutes les valeurs du maillage. Dans ce cas, si le maillage est trop important, il faut évidemment faire certaines concessions au moment de la sauvegarde des résultats, en ne sauvant que ce qui a une réelle importance aux yeux de l'utilisateur.

Suite à la définition des deux objets de paramètres, l'utilisateur doit faire appel à la fonction `simulateMonodomain2DWP(...)` (Code 6.5) afin d'effectuer la simulation du modèle monodomaine en 2D avec la méthode pararéel. Elle prend en argument un objet `SimParam` ainsi que le nombre de processus requis et la tolérance pour l'arrêt de la méthode, dont les valeurs

par défaut sont respectivement 2 et 1×10^{-4}. Elle renvoie une liste contenant les temps ainsi que les résultats associés.

Code 6.5 – Exemple d'exécution d'une simulation du modèle monodomaine en parallèle.

```
# Définitions des paramètres pour le courant externe appliqué
# Les paramètres T_stim_end et I_app prennent les valeurs par défaut
current = CurrentApplied(xi=0.3,xf=0.5,yi=0.3,yf=0.5,T_stim_start=20.)

# Définitions des paramètres de simulation
# Les paramètres Tf, lamb, cx, cy prennent les valeurs par défaut
simParam = SimParam(dt=1.e-3,N=200,cur=current)

# Lancement de la simulation en parallèle
# En retour la fonction renvoye les valeurs de la solution aux différents pas de temps
[time, resultats] = simulateMonodomain2DWP(current,NbProc=32,tol=1.e-5)
```

On effectue des simulations pour différents nombres de processus, puis on compare les résultats à différentes itérations, en déterminant l'erreur quadratique moyenne. On relève également l'accélération entre les temps de résolution en parallèle et en séquentiel, afin de constater le gain de temps gagné.

D'après la figure 6.8, on constate que l'erreur baisse effectivement au cours des différentes itérations, avant d'arriver à une erreur de l'ordre de 10^{-6} en moyenne vers la sixième itération. On constate néanmoins une fluctuation des erreurs, comme on avait déjà pu le voir chapitre 5. Ceci est très probablement dû à la propagation de l'erreur, plus longue à résorber lors de l'utilisation de nombreux processus.

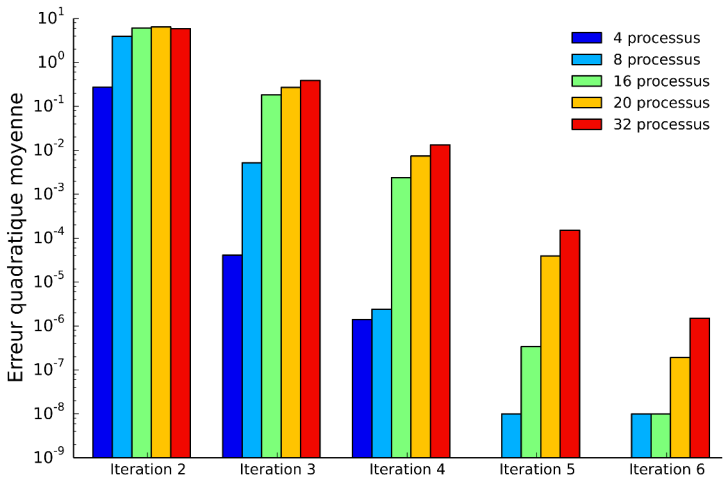

Figure 6.8 – Comparaison pour différents nombres de processus, de l'erreur quadratique moyenne à chaque itération, lors de l'utilisation de la méthode pararéel en MPI.

La figure 6.9 présente les différents accélérations obtenus lors de la résolution en parallèle du système (6.1). On constate une accélération des intégrations lorsque le nombre de processus augmente. Même si l'accélération reste assez faible, par rapport au gain théorique maximal, elle est non-négligeable, puisque avec 32 processus, on arrive à un speed-up d'environ 4,3. Il est également plus faible que le meilleur résultat obtenu chapitre 5, notamment parce que la convergence est ici atteinte en deux itérations supplémentaires.

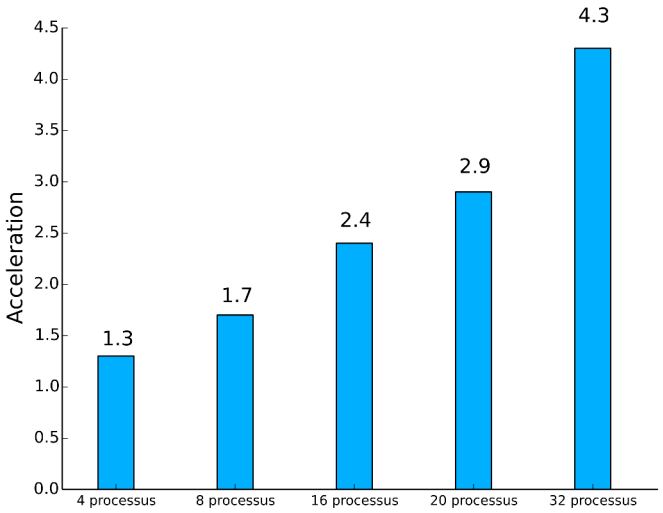

Figure 6.9 – Accélérations obtenues à l'aide de MPI, pour chaque nombre de processus

Malgré des résultats encourageant, qui permettent notamment d'obtenir des solutions précises dans la direction temporelle, le gain n'est pas aussi important que prévu. De plus, nous voudrions pouvoir obtenir une précision aussi fine dans la direction spatiale également, et ceci en même temps. Je vais me tourner une nouvelle fois vers la technologie GPU (section 4.2) associé à l'environnement CUDA (section 4.4).

6.3.3 Simulations avec la méthode pararéel en CUDA

Dans cette section, je présente la méthode pararéel sur GPU, déjà développée dans le chapitre 5 au problème d'EDO, au modèle monodomaine. Mais, il faut maintenant aller plus loin. En effet, il est souhaitable d'obtenir une solution dont la précision sur l'échelle de temps est excellente. Or, cette contrainte peut également s'appliquer à la dimension spatiale. C'est dans ce but, que je vais tirer partie de la notion, déjà introduite dans la section 4.4 : parallélisme dynamique.

L'intérêt est de lancer une grille CUDA effectuant la simulation grâce au pararéel, comme lors de l'application aux modèles de neurone. Chaque thread intégrant le sous-problème associé à

chaque sous-intervalle, exécutera à son tour une grille calculant les nouvelles valeurs de V_m et w sur chaque maille (Figure 6.10).

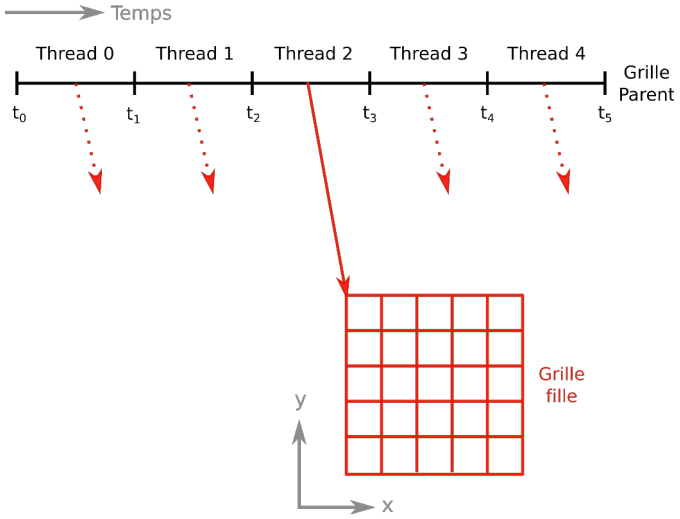

Figure 6.10 – Principe de la parallèlisation massive sur GPU.

L'avantage de CUDA est de pouvoir paralléliser massivement un problème. Après avoir discrétisé l'espace de définition, on définit une grille CUDA ayant la même taille que le maillage, puis on affecte à chaque thread de cette grille, une maille bien précise. Dans ce cas, on ne passe plus par la forme matricielle découlant de la discrétisation de $\nabla \cdot (M_i \nabla V_m)$, mais uniquement par l'expression :

$$c_x \frac{\partial^2 u}{\partial x^2} + c_y \frac{\partial^2 u}{\partial y^2} = c_x \frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{\Delta x^2} + c_y \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{\Delta y^2} \quad . \tag{6.12}$$

Autrement dit, le thread ayant comme coordonnées $(i; j)$ au sein de la grille, a à sa charge la valeur de $u_{i,j}$. A partir de ces coordonnées, le thread courant fait appel aux valeurs $u_{i\pm1,j\pm1}$ des ses threads voisins, afin de mettre à jour $u_{i,j}$. Les conditions aux limites doivent aussi être prises en compte, en distinguant les différents cas où les mailles sont concernées. Tout est basé sur le jeu des coordonnées du thread dans la grille, déterminées grâce aux différentes informations fournies, déjà introduites auparavant (sections 4.4 et 5) telles que, le nombre de blocs, la dimension d'un blocs, etc. On passe par cette formulation explicite, pour éviter de devoir copier la matrice A du CPU vers le GPU, une opération qui peut devenir très vite coûteuse en temps, lorsque le problème est important.

Nous avons déjà vu dans la section précédente, qu'il est obligatoire de faire communiquer toutes les valeurs à la fin des sous-intervalles, au sein du maillage. Cela entraîne un grand nombre de valeurs à enregistrer en mémoire, notamment si le maillage est très grand.

Il est donc pratiquement impossible de faire appel à la mémoire partagée du bloc effectuant la méthode parareél.

Fondamentalement, la manière de procéder pour la parallélisation en temps est identique à celle utilisée dans le chapitre 5. Mais contrairement aux cas précédents, on ne peut pas lancer la méthode pararéel sur une grille contenant un bloc de 1000 threads, par exemple. Il faut savoir que lorsque l'on lance une grille dynamique, toutes les grilles s'exécutent séquentiellement. On perd ainsi toute concurrence entre elles, et au final toute intérêt de simuler le modèle en parallèlisant totalement les calculs. Pour pouvoir réintroduire de la concurrence, il est obligatoire d'utiliser des "streams" (voir section 4.4). Chaque action lancée dans des streams différents sera effectuée en totale concurrence. Un problème se pose néanmoins, il y a un nombre limite au nombre de streams que l'on peut exécuter en concurrence. Ce nombre était encore de 8 lorsque les GPU étaient basés sur l'architecture Fermi, alors qu'il est de 32 depuis l'architecture Kepler, sur laquelle est basée notre Tesla K20C. Ce nombre augmentera encore lorsque les Tesla utiliseront l'architecture Maxwell.

A partir de cette information, je définis un tableau de streams (Code 6.6), puis chaque grille fille effectuant le calcul en espace sera lancée au sein de l'un d'eux. Afin d'avoir le maximum de concurrence, la grille parent doit contenir le même nombre de threads que le nombre de streams souhaité. Autrement dit, si on veut le maximum de concurrence, on doit définir 32 streams, ce qui entraine l'utilisation d'une grille parent contenant un bloc de 32 threads.

Code 6.6 – Définition d'un tableau de streams CUDA.

```
# BLOCK_SIZE 32

// Allocation d'un tableau de BLOC_SIZE streams
cudaStream_t *streams = (cudaStream_t *) malloc(BLOCK_SIZE * sizeof(cudaStream_t));

// Création de chaque stream
for (int i = 0 ; i < BLOCK_SIZE ; i++){
    cudaStreamCreateWithFlags(&(streams[i]), cudaStreamNonBlocking);
}
```

Le code 6.7 montre l'utilisation des grilles filles au sein de la grille parent. Chaque itération en temps entraîne l'exécution de la grille fille mettant à jour les valeurs des variables de potentiel V_m et de recouvrement w sur chaque maille. Même si dans cette exemple, la concurrence ne paraît pas évidente, elle servira dans les étapes d'après puisque elle permettra à la résolution fine de démarrer, une fois l'intégration grossière terminée, alors qu'il aurait fallu attendre que tous les threads de la grille parent terminent leur résolution grossière avant que les autres puissent continuer. Sans l'utilisation des streams, le pararéel n'aurait rien apporté dans ce cas.

Code 6.7 – Aperçu de l'utilisation des streams au sein de la méthode pararéel.

```
__global__ void parareal(double *U, double *W, double *Utild, double *Wtild, double *
  Uchap, double *Wchap, double *UF, double *WF, Param p){

    // Exécution avant
    ...


    // Allocation du tableau de streams
    cudaStream_t *streams = (cudaStream_t *) malloc(BLOCK_SIZE * sizeof(cudaStream_t));

    // Création de chaque stream
    for (int i = 0 ; i < BLOCK_SIZE ; i++){
        cudaStreamCreateWithFlags(&(streams[i]), cudaStreamNonBlocking);
    }

    // Dimension des grilles filles
    dim3 block(THREAD_NUM_X,THREAD_NUM_Y, 1);
    dim3 grid(BLOCK_NUM_X,BLOCK_NUM_Y, 1);

    // Résolution permettant l'initialisation de la méthode pararéel
    for(int i=0;i<BLOCK_SIZE;i++){

        if(i == idx){

            // Copie de la condition initiale
            cp_next_data<<<grid, block,0,streams[idx]>>>(Utild,Wtild,U,W,i);

            // Résolution grossière au sein du stream[idx]
            for(int j=0;j<Nb1;j++){
                coarseRK4_iteration<<<grid, block,0,streams[idx]>>>(Utild,Wtild,p,idx,j)
                  ;
            }

            // Copie de la solution finale
            cp_data<<<grid, block,0,streams[idx]>>>(U,W,Utild,Wtild,i+1);

            cudaDeviceSynchronize();
        }
        __syncthreads();

    }

    // Exécution après
    ...
```

A nouveau, l'utilisateur peut utiliser la structure `Param` afin de définir tous les paramètres de simulation et de stimulation du modèle, permettant une grande flexibilité. Suite à l'implémentation de cette méthode, j'ai effectué divers tests, en utilisant un nombre de streams différents. La figure 6.11 présente les erreurs quadratiques moyennes entre la solution parallèle

et séquentielle, lors de différentes itérations. On constate bien une diminution de l'erreur lorsque le nombre d'itérations augmentent.

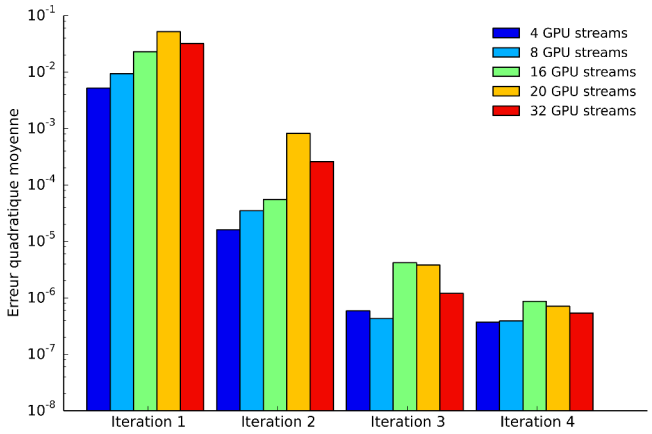

Figure 6.11 – Comparaison pour différents nombres de processus, de l'erreur quadratique moyenne à chaque itération, lors de l'utilisation de la méthode pararéel en CUDA.

Les speed-up sont référencés dans la figure 6.12. On observe un gain très important en couplant les deux parallélismes. Chaque thread a très peu d'opérations à effectuer, entraînant une accélération très importante de 328 lorsqu'on utilise le nombre limite de streams utilisables en concurrence.

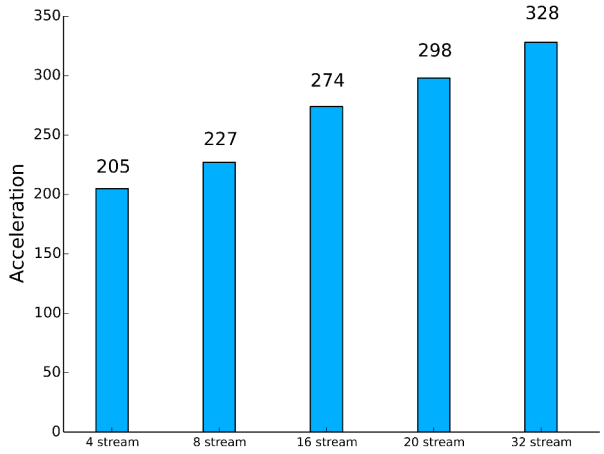

Figure 6.12 – Accélération obtenu à l'aide de MPI, pour chaque nombre de processus.

6.4 Simulations en dimension trois

Je m'attache maintenant à réaliser la simulation du modèle monodomaine en dimension trois, toujours modélisé grâce aux équations (6.1) où M_i est le tenseur de conductivité en trois dimensions décrit grâce à la matrice suivante :

$$M_i = \begin{pmatrix} c_x & 0 & 0 \\ 0 & c_y & 0 \\ 0 & 0 & c_z \end{pmatrix} \quad ,$$

où c_x,c_y et c_z sont respectivement les conductivités dans les trois directions parallèles aux axes x, y et z.

On définit le domaine de résolution comme étant le cube unitaire $[0;1]^3$. Les mêmes conditions aux limites de Neumann présentées dans la section 2.2.5 sont utilisées.

Pour résoudre le problème en dimension trois, on étend la discrétisation à une troisième dimension, entraînant l'introduction d'un nouveau paramètre N_z définissant le nombre de nœuds dans la direction z, en plus de N_x et N_y. On suppose de plus que les nombresN_x, N_y et N_z sont égaux, permettant d'obtenir un maillage uniforme du cube unitaire. Ce maillage est ainsi constitué de cube ayant un volume de $\Delta x \times \Delta y \times \Delta z$ avec $\Delta x = \Delta y = \Delta z = 1/N_x$. Chacune de ces mailles est caractérisées par des coordonnées dans l'espace : $(x_i; y_j; z_k) = (i\Delta_x; j\Delta_y; k\Delta_z)$. Grâce à ce système de coordonnées, on souhaite déterminer l'approximation $V^n_{m,i,j,k}$ de la solution $V_m(x_i; y_j; z_k; t_n)$. Pour une question de commodité, on substitue à nouveau la notation V_m par u.

On note U^n le vecteur contenant toutes les approximations $u^n_{i,j,k}$. Au final, ce vecteur est de taille $N_x \times N_y \times N_z$. On discrétise ensuite, grâce aux différences finies, l'opérateur suivant :

$$\nabla \cdot (M_i \nabla u) = c_x \frac{\partial^2 u}{\partial x^2} + c_y \frac{\partial^2 u}{\partial y^2} + c_z \frac{\partial^2 u}{\partial z^2} \quad . \tag{6.13}$$

On obtient ainsi à nouveau un système linéaire AU, où la matrice globale $A \in \mathbb{R}^{N_x^3}$, a la particularité d'être creuse. Nous ne détaillons pas les calculs fastidieux de la matrice, ainsi que l'intégration des conditions aux limites au sein de celle-ci. Le lecteur intéressé trouvera le détail de la matrice globale A dans l'annexe 1. Comme précédemment, une méthode explicite est utilisée. Ainsi, en notant U^n le vecteur contenant les valeurs de $u^n_{i,j,k}$ et W^n le vecteur contenant les valeurs de $w^n_{i,j,k}$ on obtient le système discret suivant, identique à celui en dimension deux :

$$\begin{cases} U^{n+1} = U^n + \dfrac{1}{A_m C_m (1+\lambda)} AU^n - \dfrac{1}{C_m} f(U^n, W^n) \\ W^{n+1} = g(U^n, W^n) \quad . \end{cases} \tag{6.14}$$

6.4.1 Résolution séquentielle

Un nouveau protocole de simulation est défini afin de prendre en compte la troisième dimension, notamment pour la définition de la conductance c_z et lors de la création de la zone de stimulation, représentée ici par un cube. Le lecteur est renvoyé au code 6.3 et 6.4 dont les utilisations sont quasi-identiques ici. Afin d'effectuer différents tests, qui vont notamment servir à la comparaison avec les solutions obtenues dans les sections suivantes, nous utilisons le protocole de simulation définit dans le tableau 6.3.

Pour résoudre ce problème, la construction de la matrice globale A peut devenir très vite coûteuse en terme de temps CPU, mais également en espace mémoire. Il est donc nécessaire d'éviter tout stockage superflu, entraînant une baisse significative des performances. C'est pour cela que l'aspect creux de la matrice est exploité, en utilisant des matrices dites sparse. En principe, seuls les éléments non-nuls sont sauvegardés au sein d'un tableau à une dimension, alors que les coordonnées de ces éléments sont stockées dans deux autres tableaux à une dimension. Cela permet notamment de diminuer le temps de résolution, en utilisant des fonctions optimisées pour ce genre de stockage, mais également en diminuant la ressource mémoire utilisée.

Paramètres	Valeurs
T_{final}	$100\ ms$
Δt	$1,0 \times 10^{-3}\ ms$
$\Delta x\ ;\ \Delta y\ ;\ \Delta z$	$1,0 \times 10^{-1}\ ;\ 1,0 \times 10^{-1}\ ;\ 1,0 \times 10^{-1}$
A_m	$2000\ cm^{-1}$
C_m	$1\ \mu F.cm^{-1}$
λ	$0,1$
c_x	$1\ mS.cm^{-1}$
c_y	$1\ mS.cm^{-1}$
c_z	$1\ mS.cm^{-1}$
$[T_i; T_f]_{stim}$	$[10;100]$
$[x_i; x_f]_{stim} \times [y_i; y_f]_{stim} \times [z_i; z_f]_{stim}$	$[0,4;0,6] \times [0,4;0,6] \times [0,4;0,6]$
I_{app}	$0,4\ \mu A$

Tableau 6.3 – Valeurs des paramètres de simulation du modèle monodomaine en dimension trois.

La Figure 6.13 présente des résultats obtenus en exécutant cette simulation, confirmant certaines observations faites lors de l'intégration du problème en dimension deux, à savoir une amplitude qui s'atténue lorsque le potentiel d'action quitte sa région de création. Mais les valeurs obtenues sont cohérentes avec celles du modèle de Fitzhugh-Nagumo (FitzHugh, 1955; Fitzhugh, 1961; Nagumo *et al.*, 1962).

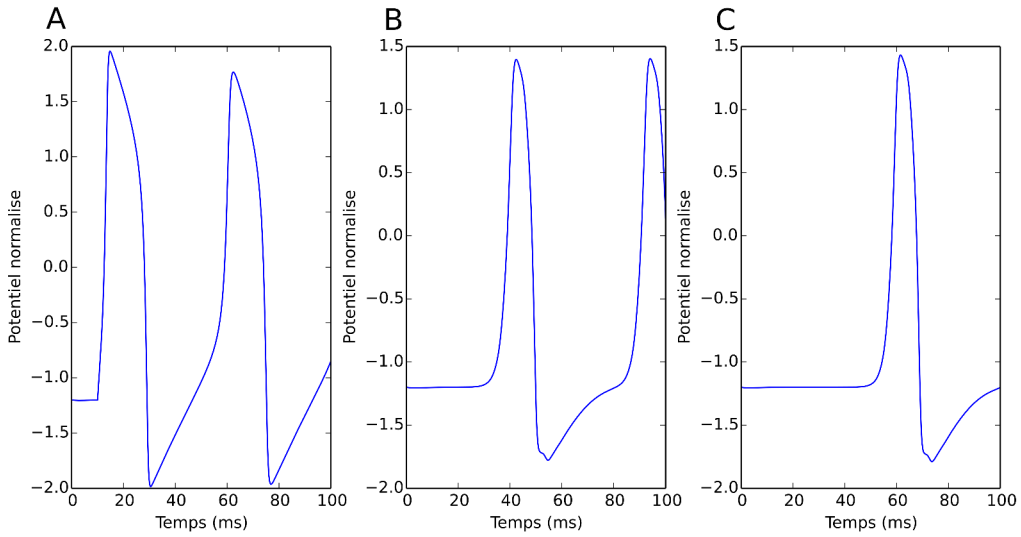

Figure 6.13 – Représentation graphique du potentiel de membrane, suite à la simulation du modèle monodomaine en trois dimensions en différentes positions : **A** : (x ; y ; z) = (0,5 ; 0,5 ; 0,5) ; **B** : (x ; y ; z) = (0,5 ; 0,5 ; 0,75) et **C** : (x ; y ; z) = (0,5 ; 0,5 ; 0,875).

Mais, comme on peut s'en douter, le temps de résolution devient très important, surtout lorsque l'on souhaite une solution ayant une très bonne précision temporelle. Il est donc nécessaire de faire des sacrifices sur la résolution spatiale, pour éviter que le temps CPU explose. Cela n'est pas sans conséquence, car la qualité du signal simulé sera forcément détériorée. Pour éviter d'avoir à faire des concessions sur la précision, je vais appliquer les méthodes introduites dans les sections précédentes.

6.4.2 Simulations avec la méthode pararéel en MPI

J'utilise ce qui a déjà été vu dans la section 6.3.2, en l'appliquant au cas en trois dimensions. Peu de choses sont à modifier, hormis la matrice globale qui est différente. Un processus se charge de sa construction, avant de la partager avec tous les autres. Je réutilise également le stockage creux de la matrice afin de réduire les coûts lors de sa communication. Encore une fois, il est à noter qu'il faut faire attention à ne pas dépasser la limite lors de l'utilisation des fonctions `Send()` et `Recv()`.

Les différentes classes définies précédemment afin de prendre en compte les différents paramètres sont quasiment identiques. Elles permettent de spécifier en plus les paramètres dépendants de la troisième dimension spatiale, comme la conductivité par exemple. J'utilise le même protocole de simulation donné pour la résolution séquentielle, afin de pouvoir comparer au mieux les résultats des différentes méthodes. Les différentes simulations sont exécutées sur un nombre croissant de processus, puis je détermine l'erreur commise entre les solutions

séquentielle et parallèle suivant l'itération d'arrêt de l'algorithme pararéel. On constate que tous les résultats observés dans les sections précédentes sont à nouveau confortés par ceux obtenus ici. On constate une baisse continue des erreurs lorsque l'itération augmente, avant d'atteindre une tolérance acceptable aux alentours de la quatrième itération (Figure 6.14). Encore une fois la méthode converge rapidement, indépendamment du nombre de processus.

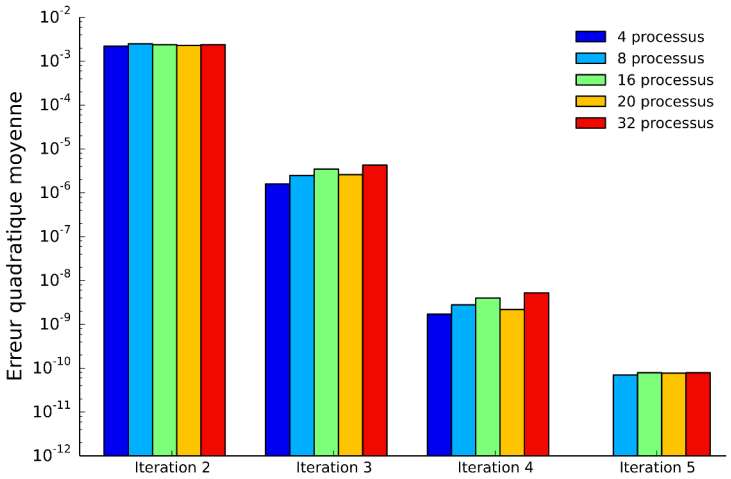

Figure 6.14 – Comparaison pour différents nombres de processus, de l'erreur quadratique moyenne à chaque itération, lors de l'utilisation de la méthode pararéel en CUDA.

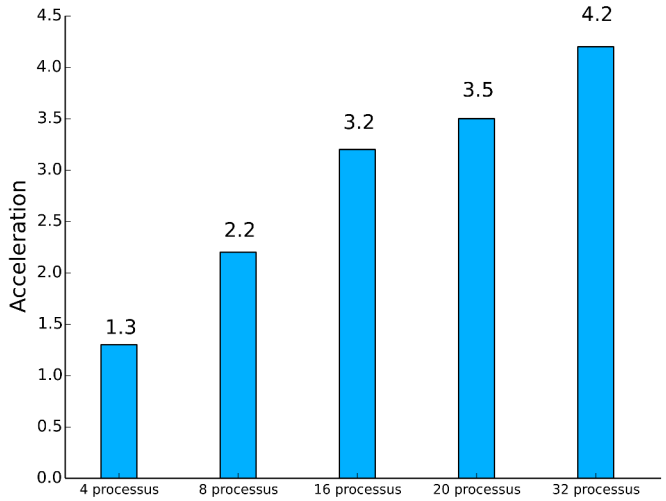

Figure 6.15 – Accélérations obtenues à l'aide de MPI, pour chaque nombre de processus.

On relève également les accélérations permettant de constater du gain de temps obtenu, lorsqu'on utilise l'algorithme pararéel (Figure 6.15). On observe une accélération croissante lorsque le nombre de processus augmente, confirmant notamment celles atteintes dans la section 6.3.2. Malgré des accélérations pratiques loin des valeurs théoriques (accélération de 32 théorique pour 32 processus, par exemple), le gain est non-négligeable, et permet de pouvoir obtenir de meilleurs scores s'il est possible de lancer la simulation sur un nombre plus important de processus.

6.4.3 Simulations avec la méthode pararéel en CUDA

Comme en dimension deux, obtenir une solution précise dans la direction temporelle est important, mais il est encore plus intéressant d'avoir une solution précise également dans le domaine spatial. Pour cela, j'utilise à nouveau la méthode développée en CUDA dans la section 6.3.3.

Mais si la méthode globale est identique à celle exposée dans la section 6.3.3, le solveur permettant de mettre à jour la partie spatiale est à modifier totalement, car il faut prendre en compte cette troisième dimension intervenant dans la discrétisation en différences finies de l'opérateur $\nabla \cdot (M_i \nabla V_m)$. Les conditions de Neumann doivent être également intégrées, en distinguant les mailles où elles s'appliquent. Ensuite, chaque grille fille doit être définie de sorte qu'elle possède la même taille que le maillage de l'espace. En général, on suppose que la grille fille contient des blocs de threads à trois dimensions. La limite à la taille des grilles filles et des grilles parents, est fixée à $(2147483647; 65535; 65535)$ avec des tailles maximales de blocs $(1024; 1024; 64)$ pour le GPU Tesla K20C. Théoriquement, si le maillage dépasse cette taille de grille, il faudra dans ce cas supposer que chaque thread s'occupe de plusieurs mailles.

On utilise le protocole définit dans les sections précédentes, afin de pouvoir comparer les solutions entre-elles. Les résultats sont synthétisés dans la Figure 6.16, confirmant les tests déjà effectués. A chaque itération, l'erreur quadratique moyenne baisse indépendamment du nombre de streams utilisés.

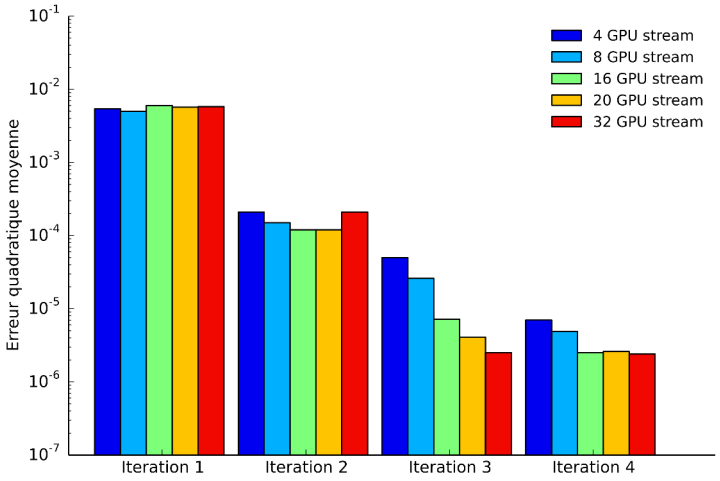

Figure 6.16 – Comparaison pour différents nombres de processus, de l'erreur quadratique moyenne à chaque itération, lors de l'utilisation de la méthode pararéel en CUDA.

En comparant les accélérations (Figure 6.17) obtenus, on observe un gain, encore une fois, très important. Il n'atteint pas les valeurs des simulations en deux dimensions, pour la simple raison qu'on utilise, en séquentiel, un stockage creux des matrices, ce qui ralentit l'accélération. Néanmoins, le gain est très intéressant.

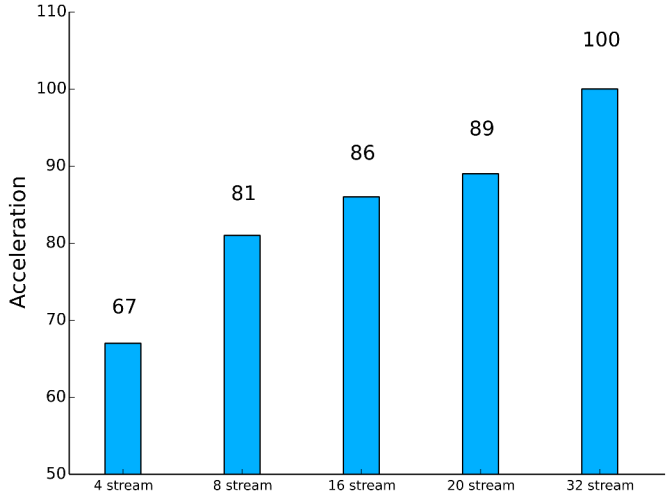

Figure 6.17 – Accélérations obtenues à l'aide de CUDA, pour chaque nombre de processus.

6.5 Conclusion

Dans ce chapitre, la méthode des différences finies a été appliquée afin de simuler le modèle monodomaine. La dimension un a été utilisé afin de mieux comprendre le modèle d'un point de vue convergence, stabilité et ainsi déterminer les meilleurs méthodes pour sa simulation. Puis, j'ai appliqué ces méthodes au modèle sur des dimensions supérieures. Afin de pouvoir obtenir des solutions de meilleurs qualités, ou également pour pouvoir simuler sur de plus longues périodes, la méthode pararéel a été utilisée avec un développement en MPI, puis en CUDA, où j'ai profité de l'aspect multi-thread du GPU pour développer une méthode massivement parallèle, couplant la parallélisation en temps et en espace. Cette résolution totalement parallèle permet de diminuer considérablement le temps de simulation de ce modèle, puisque que celle ci se déroule dans des temps proches de la seconde, là où il faut plusieurs minutes, avec la résolution séquentielle voir uniquement pararéel.

En utilisant ces méthodes, des géométries plus complexes peuvent être simulées, qui seront présentées dans le chapitre suivant.

7 Simulations du modèle monodomaine à l'aide des éléments finis

7.1 Introduction

Dans ce chapitre, la méthode des éléments finis va être appliquée à la simulation du modèle monodomaine. Cette méthode permettra ainsi de pouvoir définir le problème sur un ensemble de géométries plus complexes, que ce que permet la méthode des différences finies. La convergence lors de la résolution en dimension deux sera étudiée, puis les différents algorithmes de parallélisation vu dans les sections 3.1 et 3.2 seront appliqués.

Les différents programmes sont implémentés à l'aide du software FreeFem++ présenté brièvement dans la section 3.2.2.2.

7.2 Simulations en dimension deux

On considère le problème monodomaine décrit par les équations du modèle monodomaine sur le disque unité noté Ω :

$$\begin{cases} A_m\left(C_m\dfrac{\partial V_m}{\partial t}+f(V_m,w)\right)=\dfrac{1}{\lambda+1}\nabla\cdot(M_i\nabla V_m) \\ \dfrac{\partial w}{\partial t}=g(V_m,w) \end{cases} \qquad \text{sur } \Omega\times[0;T] \quad , \tag{7.1}$$

où M_i est un tenseur de conductivités dans $\mathbb{R}^{2\times 2}$ avec des conditions initiales et des conditions aux limites de types Neumann homogène :

$$M_i\nabla V_m\cdot\mathbf{n}_{\partial\Omega}=0 \qquad \text{sur } \partial\Omega\times[0;T] \quad . \tag{7.2}$$

On considère une discrétisation du problème basé sur un maillage triangulaire du disque unité. D'après ce que a été vu dans le cas des différences finies, un schéma explicite sera utilisé

afin de résoudre ce problème. L'inconnue du problème monodomaine est $V_m^p \in H^1(\Omega)$ où Ω est l'espace délimité par un cercle de rayon 1. De plus V_m^p est l'approximation de $V_m(t^p)$. Sur ce domaine, on peut écrire la formulation faible suivante :

$$\frac{1}{\lambda+1}\int_\Omega M_i \nabla V_m^p \nabla \eta + \int_\Omega A_m C_m \partial_t V_m^p \eta - \int_\Omega A_m I_{ion}(V_m^{p-1}, w^{p-1})\eta = 0 \quad , \tag{7.3}$$

avec $\eta \in H^1(\Omega)$. Ce problème admet une solution unique.

7.2.1 Résolution séquentielle

La triangularisation du domaine de définition est faite automatiquement en FreeFem. Pour cela, il suffit de définir l'espace formé par un cercle à l'aide du mot clé `border`, qui utilise une fonction paramétrique. Puis on utilise la fonction `buildmesh` créant ainsi une triangularisation de l'espace (Figure 7.1), en fonction de la taille du découpage voulue par l'utilisateur.

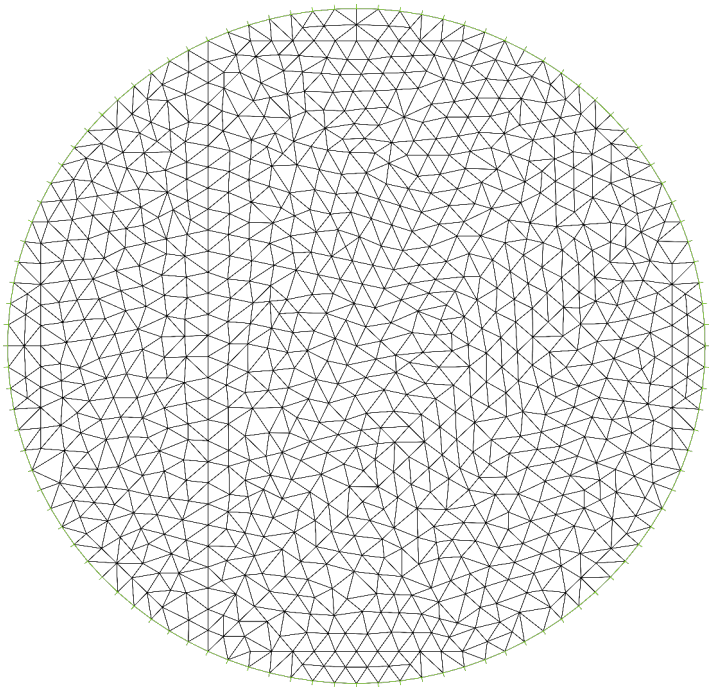

Figure 7.1 – Exemple de maillage du cercle unité sous FreeFem.

A nouveau, on prend les fonctions f et g de type Fitzhugh-Nagumo, puis on applique un protocole de simulation au modèle (Table 7.1). Le modèle est stimulé à l'aide d'un courant extérieur, appliqué sur un disque de rayon 0.1 pendant une période de 10 ms. On prend comme tenseur de conductivité M_i, la matrice identité de taille 2×2, dont les composantes diagonales sont respectivement c_x et c_y.

Paramètres	Valeurs
T_{final}	100 ms
Δt	0,7 ms
λ	0,1
A_m	2000 cm^{-1}
C_m	1 $\mu F.cm^{-1}$
c_x	1 $mS.cm-1$
c_y	1 $mS.cm-1$
$[T_i;T_f]_{stim}$	[10;20]
Rayon de la stimulation	0,1
I_{app}	0,4 μA

Tableau 7.1 – Valeurs des paramètres de simulation du modèle monodomaine en dimension un.

Lors de la définition du problème, il est possible d'utiliser différents types de solveurs intégrés directement dans FreeFem, tels que : UMFPACK, GMRES, méthode du gradient conjugué. Nous présentons un petit comparatif des temps de simulation lors de l'utilisation de ces différents solveurs (Table 7.2).

Méthodes	Temps CPU
UMFPACK	347 s
GMRES	204 s
CG	170 s

Tableau 7.2 – Temps CPU des différentes méthodes intégrées dans FreeFem.

La méthode du gradient conjugué s'avère la plus rapide et sera donc utilisée pour la suite. En ce qui concerne la méthode de résolution en séquentielle, j'ai décidé d'utiliser une méthode semi-implicite afin de résoudre le problème monodomaine, dont l'efficacité a été montré dans la section 3.1.2. Il faut savoir que toutes les méthode décrites dans cette précédente section s'appliquent également très bien lorsque on applique la méthode des éléments finis. Cette méthode sera à nouveau utilisé lors de l'application de l'algorithme pararéel, en tant qu'opérateur grossier, tandis que le schéma explicite effectuera la tache de l'opérateur fin. La simulation séquentielle est effectué sur un noeud du cluster de Rhenovia, dont le CPU est un Intel Xeon E5-2600 composé de 8 coeurs cadencés à 2,5 GHz et associé à 32 Go de RAM. Le même noeud a déjà été utilisé afin d'effectuer la simulation séquentielle du modèle monodomaine avec les différences finies. Ce même noeud réalisera les simulations séquentielles du modèle bidomaine. En plus des différents solveurs offerts par FreeFem pour résoudre un problème séquentiellement, j'ai effectué quelques tests sur les solveurs parallèles offerts par ce logiciel tels que : MUMPS (MUltifrontal Massively Parallel Solver), SuperLU, Paxtix (basé sur des décompositions LU incomplètes ILU(k)). D'autres solveurs itératifs, cette fois, sont égale-

ment disponibles : HIPS (Hierarchical Iterative Parallel Solver, très efficace pour de très grands systèmes linéaires) et HYPRE. Ces solveurs demandent d'écrire explicitement les problèmes sous une version purement matricielle, afin d'être totalement optimales. J'ai effectué quelques développement afin de tester ces différents solveurs, permettant de conclure que pour ce type de problème, la méthode MUMPS donne le meilleur rapport entre vitesse de résolution et erreur entre solutions séquentielle et parallèle. En général, avec le solveur MUMPS, on obtient une solution 6 fois plus rapidement que lors d'une résolution séquentielle lorsque 32 processus sont utilisés. De plus, l'erreur entre les deux solutions est de l'ordre de 10^{-5}. Il serait, bien entendu, intéressant de coupler les solveurs parallèles proposés par FreeFem avec la méthode que nous allons décrire dans la suite, mais le plus gros problème lors de l'utilisation de ces solveurs est qu'ils fonctionnent comme des boîtes noires. Il est impossible de savoir de quelle manière comment les processus communiquent entre-eux. Il est ainsi relativement compliqué, à l'heure actuelle, de les utiliser conjointement à d'autres méthodes parallèles. Dans la suite, nous omettrons donc ces solveurs parallèles.

Avant d'étudier la convergence, nous présentons les résultats déterminés en utilisant une suite de maillages de plus en plus fins (Figure 7.2). On constate que l'utilisation de maillages différents entraîne un dynamique différente lors de la propagation du potentiel d'action. De plus, on constate que la forme de cette propagation est difforme lorsque l'on a recours à un maillage grossier, alors qu'il devrait avoir une forme circulaire car les conductances sont les mêmes suivant les différents axes x et y. Pour valider cette observation, j'ai effectué une étude de convergence permettant ainsi de valider les résultats.

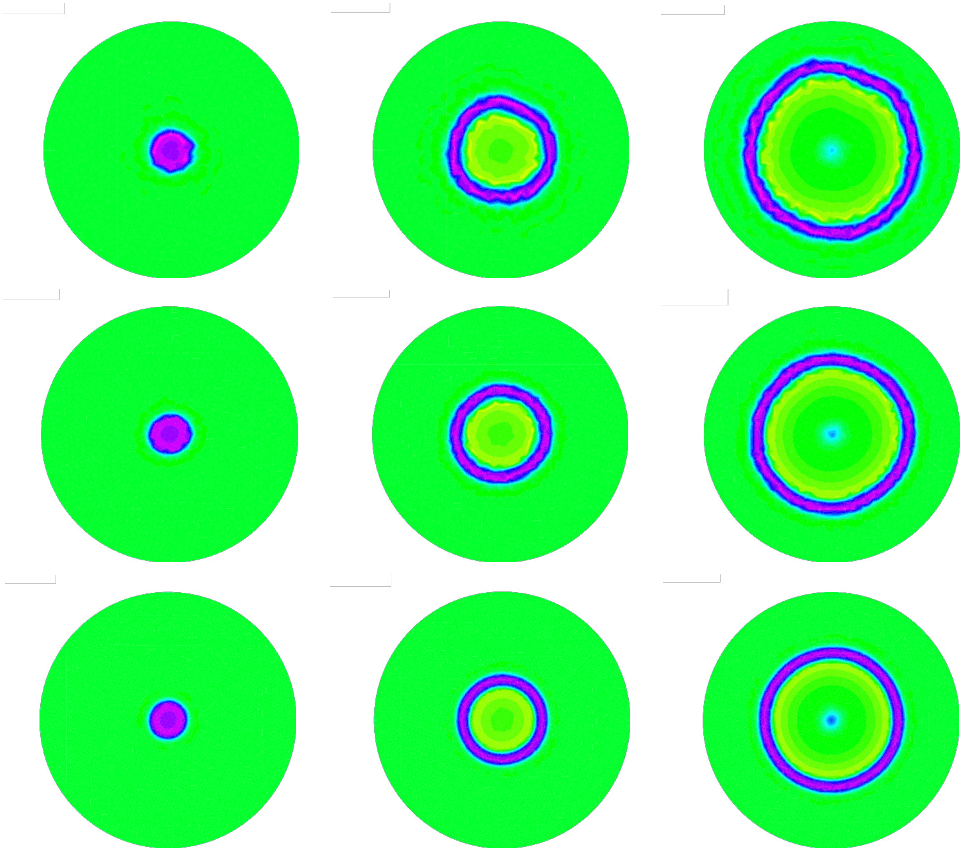

Figure 7.2 – Propagation du potentiel d'action au cours du temps (de gauche à droite) obtenus sur des maillages de 1766 triangles (en haut), de 2978 triangles (au milieu) et de 7064 triangles (en bas).

7.2.2 Convergence

En général, on détermine une solution de référence, qui est le plus souvent analytique et on étudie la différence entre celle-ci et les solutions calculées. Vu la difficulté à déterminer une solution analytique au problème, on utilise ici une méthode indirecte de validation des résultats.

On fixe le pas de temps et on étudie la convergence du schéma en utilisant une suite de maillages dont la taille est croissante. La solution de ce problème doit être un front d'onde qui se propage dans toutes les directions à la même vitesse compte tenu de la conductivité du milieu. Théoriquement, ce front d'onde doit s'approcher de la forme d'un cercle lorsqu'on affine le maillage, comme le montre ces solutions prises au même instant, pour des maillages de plus en plus fins (Figure 7.2).

Pour étudier la convergence, on enregistre la solution des différentes simulations en un certain nombre de points se trouvant sur un cercle de rayon 0.5, et ceci pour différentes tailles de maillages.

On étudie plusieurs caractéristiques des résultats obtenus, notamment la moyenne $\bar{\tau}$ des temps précis τ où le maximum de chaque enregistrement est atteint, la différence $\Delta\tau$ entre les temps du premier et le dernier maximum enregistré, l'écart-type σ_τ de ces différents temps. On analyse également les données spatiales enregistrées, en déterminant la déviation spatiale Δr de la solution circulaire, ainsi que la résolution Δx du maillage.

La déviation spatiale Δr est obtenue en utilisant la relation :

$$\Delta r = \hat{v}\Delta\tau \quad ,$$

avec la constante de vitesse de propagation :

$$\hat{v} = r/\bar{\tau} \quad ,$$

où r est la distance de déplacement du front d'onde. Le résolution Δx est déterminée par l'expression :

$$\Delta x = \sqrt{\frac{\text{Aire du domaine}}{\text{Nombre de triangles}}} \quad .$$

On présente ci-après les résultats obtenus lors de l'analyse des différents critères (Tableau 7.3).

Nombre d'élèments	$\bar{\tau}$	$\Delta\tau$	σ_τ	Δr	Δx
446	53,25	67,2	14,27	0,631	0,042
1 766	54,13	4,2	0,909	0,039	0,021
3 924	60,27	4,9	1,256	0,041	0,014
7 064	64,17	2,1	0,455	0,016	0,010
15 508	66,78	0,7	0,343	0,005	0,007
43 312	68,9	0,7	0,347	0,005	0,004
85 306	69,98	0,7	0,125	0,005	0,003
141 282	70	0	0	0	0,0023
207 856	70	0	0	0	0,0019

Tableau 7.3 – Analyse des résultats du modèle monodomaine 2D couplé avec le modèle Fitzhugh-Nagumo.

On trace également les enregistrements du potentiel de membrane obtenus pour les différentes tailles de maillage. On constate, avec les résultats obtenus et analysés, qu'il y a effectivement convergence de la solution. Plus le maillage est raffiné, plus les points d'enregistrements se superposent, ce qui correspond parfaitement avec l'hypothèse émise au début de cette

partie. On observe, à nouveau, que l'utilisation de tailles différentes de maillage entraîne une vitesse de propagation du potentiel d'action différente.

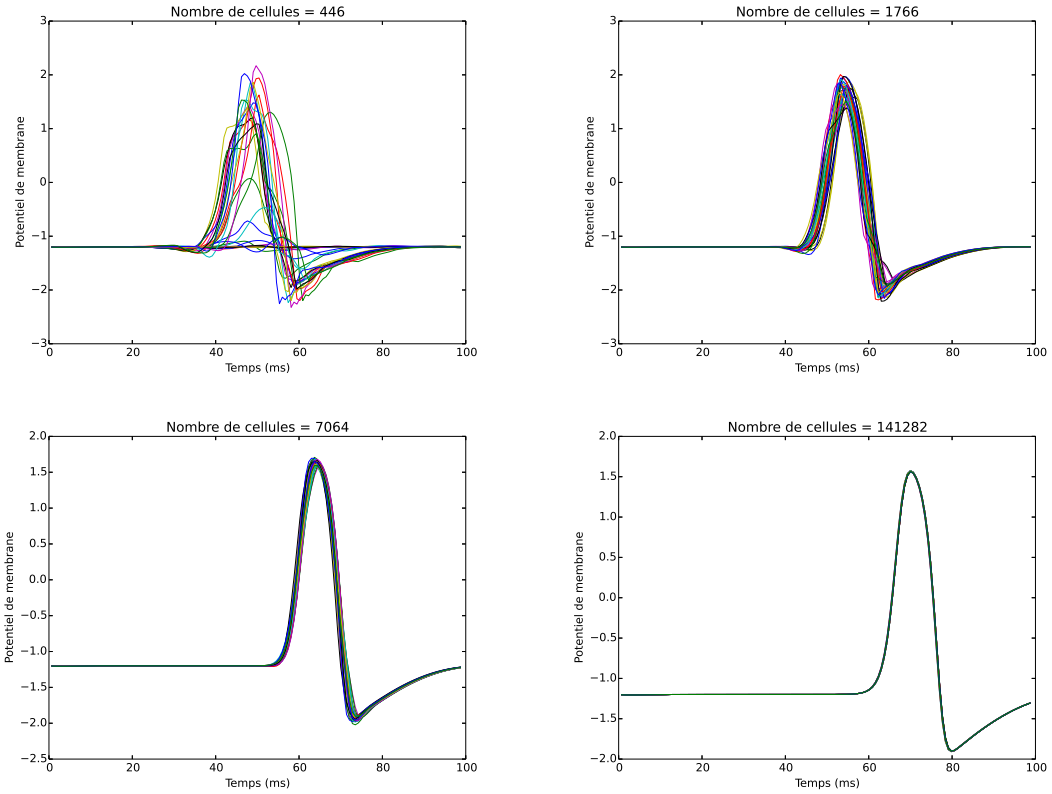

Figure 7.3 – Potentiel de membrane normalisé enregistré en différents points appartenant au même cercle de rayon 0,5 pour différentes tailles de maillage.

7.2.3 Application de la méthode pararéel en MPI

Nous avons observé dans le chapitre 6 que le temps de résolution devient très important lorsque une précision importante est demandée, aussi bien en espace qu'en temps. Le problème est identique ici, c'est pour cela que j'ai aussi adapté la méthode pararéel en FreeFem afin de répondre aux exigences de certains utilisateurs.

Dans un premier temps, il convient de définir deux problèmes en FreeFem dépendant d'un pas de temps différents. Ils serviront pour la résolution grossière et fine. En FreeFem, un problème doit-être défini comme une application bilinéaire, il n'est donc pas possible de mettre un argument supplémentaire lors de sa déclaration. Il est donc obligatoire de passer par deux problèmes ayant leur propre pas de temps. Le problème fin utilisera le pas de temps fin δt et le problème grossier, le pas de temps grossier Δt.

A chaque fin de résolution sur un sous intervalle, chaque processus envoie son résultat corrigé

Paramètres	Valeurs
T_{final}	100 ms
Δt	0,3125 ms
δt	0,01 ms
Nombre de triangles	15670
λ	0,5
A_m	2000 cm^{-1}
C_m	1 $\mu F.cm^{-1}$
c_x	1 $mS.cm^{-1}$
c_y	1 $mS.cm^{-1}$
$[T_i; T_f]_{stim}$	[10;100]
Rayon de la stimulation	0,1
I_{app}	0,4 μA

Tableau 7.4 – Valeurs des paramètres de simulation du modèle monodomaine en dimension deux.

au suivant, afin qu'il puisse poursuivre son travail à la prochaine itération. Or, comme dans le cas des différences finies (et tout problème d'EDP), un processus doit envoyer les informations sur chaque cellule du maillage, il est donc nécessaire de définir des tableaux contenant toutes les informations sur le maillage à cet instant donné. Ces tableaux doivent contenir les coordonnées de la solution dans la base de fonctions propres. Ce tableau peut très vite devenir important, et ainsi atteindre la limite théorique de la fonction `Send()` et `Recv()` de MPI. Si c'est le cas, il est nécessaire de découper les envois en plusieurs paquets. Notamment ici, il est essentiel d'envoyer séparément les informations liées au potentiel de membrane V_m et ceux concernant la variable de recouvrement w. De toute façon, il est impossible en FreeFem de déclarer des tableaux d'espace V_h à deux dimensions, pour une question de mémoire.

Le critère d'arrêt de l'algorithme pararéel est basé sur le calcul de la norme L^2 entre les solutions de deux itérations successives.

Pour les différents tests, on applique le protocole de simulation présenté dans le Tableau 7.4. Une simulation séquentielle est lancée afin de pouvoir comparer efficacement les différents résultats obtenus en simulation parallèle.

La Figure 7.4 synthétise les erreurs L^2 déterminées suite à l'obtention des différents résultats. On constate que l'erreur baisse lorsque l'itération augmente, confortant ainsi tous les résultats obtenus précédemment. On notera que l'erreur augmente suivant le nombre de processus, venant de cette propagation de l'erreur qui est plus longue à corriger lorsque les processus sont nombreux. On constate également que l'erreur stagne à 10^{-4}, contrairement à l'application aux différences finies, où l'erreur atteignait des valeurs bien plus basses.

On répertorie les accélérations, découlant des différentes simulations, dans la Figure 7.5. A nouveau, le gain de temps augmente en fonction du nombre de processus, pour atteindre un maximum de 4,8 lorsque ce nombre est à 32. Le gain est encore une fois non-négligeable,

entraînant ainsi un avantage certain à l'utilisation de cette méthode.

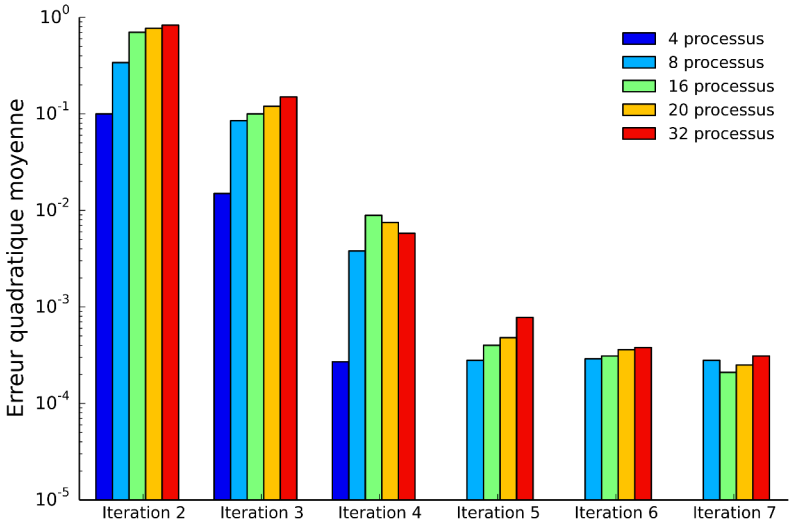

Figure 7.4 – Comparaison de l'erreur quadratique moyenne à chaque itération pour différents nombres de processus MPI.

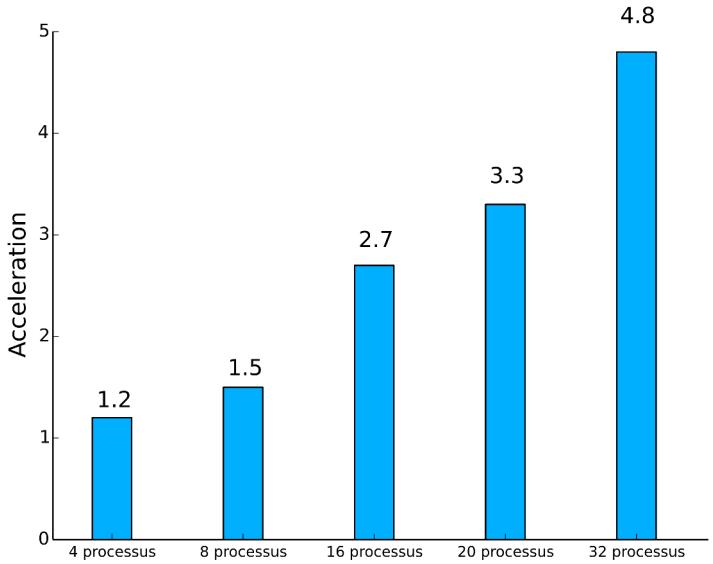

Figure 7.5 – Speed-up lors de l'utilisation de l'algorithme pararéel en FreeFem par rapport au temps de simulation séquentiel.

7.2.4 Application de l'algorithme pararéel couplé à la décomposition de domaine en MPI

En plus d'un gain indéniable en appliquant la méthode pararéel lors de la simulation du modèle, son utilité s'étend à l'utilisation de deux échelles de temps différents où deux dynamiques du modèle peuvent être représentées. Mais, il peut être également utile de posséder deux niveaux de discrétisation de l'espace défini par le domaine de définition.

L'intérêt est de pouvoir découper l'espace en plusieurs morceaux, afin de définir des sous-problèmes sur chacune de ces parties, puis de les résoudre indépendamment sur des processus différents afin de diminuer la complexité du problème, mais également le temps de calcul. Pour cela, j'utilise des méthodes de décomposition de domaine présentées dans la section 3.2. Pour mon cas d'application, je me restreins à la décomposition de domaine sans recouvrement, et dans un premier temps, à une division par deux de l'espace. Le problème peut donc s'écrire de la manière suivante, avec $i \in \{1;2\}$:

$$\left\{\begin{array}{rcll} A_m\left(C_m\dfrac{\partial V_{m,i}}{\partial t}+f(V_{m,i},w_i)\right) & = & \dfrac{1}{\lambda+1}\nabla\cdot(M_i\nabla V_{m,i}) & \text{dans } \Omega_i \\ \dfrac{\partial w_i}{\partial t} & = & g(V_{m,i},w_i) & \text{dans } \Omega_i \\ M_i\nabla V_m\cdot\mathbf{n}_{\partial\Omega} & = & 0 & \text{sur } \partial\Omega \\ V_{m,i} & = & \alpha & \text{sur } \Gamma_i \end{array}\right. , \tag{7.4}$$

où Γ_i est la frontière commune entre les deux domaines Ω_1 et Ω_2, V_m a des conditions initiales. Le problème consiste à trouver α tel que $V_{m,1} = V_{m,2}$ sur Γ_i.

Techniquement, je découpe le cercle en deux parts en créant, grâce à FreeFem, deux `border` délimitant chacun des sous-domaines, puis je construit le maillage en faisant appel à la fonction `buildmesh()` de FreeFem.

L'idée générale de l'utilisation de cette méthode est de pouvoir la coupler avec la méthode pararéel utilisée seule dans la section 7.2.3. Pour cela, je suppose toujours, pour des questions pratiques, que le domaine est scindé en deux sous-domaines. Lors du lancement de l'algorithme sur N processus, chacun va s'attacher à résoudre le problème sur son sous-intervalle de temps, comme l'algorithme pararéel standard. Mais lors de la résolution spatiale, chaque processus va faire appel à un processus supplémentaire, afin d'effectuer la méthode de décomposition de domaine. Cette décomposition va donc s'opérer sur deux processus distincts. Pour éviter tout conflit avec les processus s'occupant des autres sous-intervalles en temps, un processus supplémentaire sera alloué pour chaque processus s'occupant de la parallélisation en temps (Figure 7.6). Cela équivaut donc à utiliser $2N$ processus. C'est pour cette raison que le découpage du domaine en plus de deux sous-domaines n'a pas été envisagé pour les travaux présentés dans ce manuscrit.

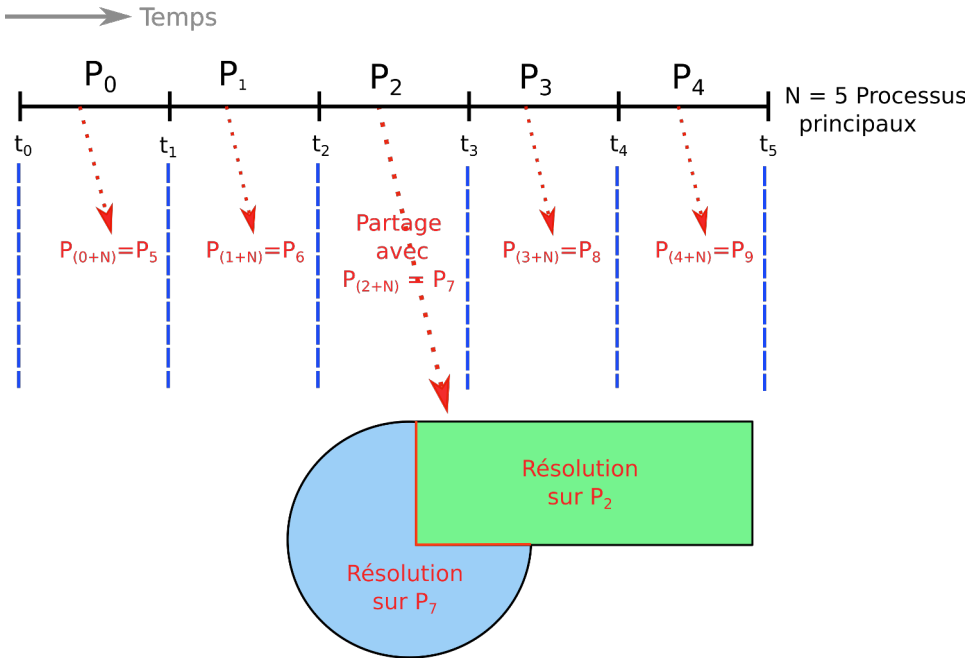

Figure 7.6 – Représentation schématique de la méthode couplant les deux formes de parallélisme.

Ainsi, chaque sous-intervalle en temps possède deux processus s'attachant à résoudre le sous-problème, permettant ainsi de coupler les deux parallélisations. En comparaison à la méthode GPU, il est compliqué de généraliser cette solution à M sous-domaines et N sous-intervalles en temps. Le travail de développement peut devenir très fastidieux, ce qui est déjà le cas, pour deux sous-domaines. Il faut faire en sorte que les bons processus communiquent entre-eux, et cela au bon moment.

J'effectue des tests de simulations, en appliquant la méthode présentée ci-dessus, au modèle monodomaine. J'applique le même protocole de simulation que ceux de la section 7.2.3, afin de pouvoir comparer au mieux les différents résultats. La Figure 7.7 présente les erreurs quadratiques entre les solutions séquentielle et parallèle, en fonction de l'itération de l'algorithme pararéel. On constate que les erreurs ne diminuent pas de façon aussi importante que celles de la section 7.2.3. La raison en est que le plus gros de l'erreur, entraînant cette différence, est commis lors de l'application de la méthode de décomposition de domaine. Ce sont des méthodes itératives, comme l'algorithme pararéel, dont l'arrêt est provoqué lorsque la tolérance définie par l'utilisateur est atteinte. Bien évidemment, plus cette tolérance est sévère, plus le temps de résolution va augmenter. Cette erreur conjuguée à celle commise par l'algorithme pararéel, provoque une erreur finale plus importante que celle constatée jusque là.

De plus, en observant les accélérations (Figure 7.8) obtenues lors de ces simulations, on note qu'aucun gain de temps n'est obtenu lors de l'utilisation de ces deux méthodes, au contraire

de ce qu'on a pu observer auparavant, notamment avec l'application des différences finies sur GPU. Néanmoins, cela n'ôte pas l'intérêt de cette manière de procéder, qui va permettre de définir des zones d'intérêts, où le maillage sera moins important, entraînant un temps de résolution global moins important, alors qu'un maillage uniforme est utilisé lorsque le domaine est considéré dans sa globalité.

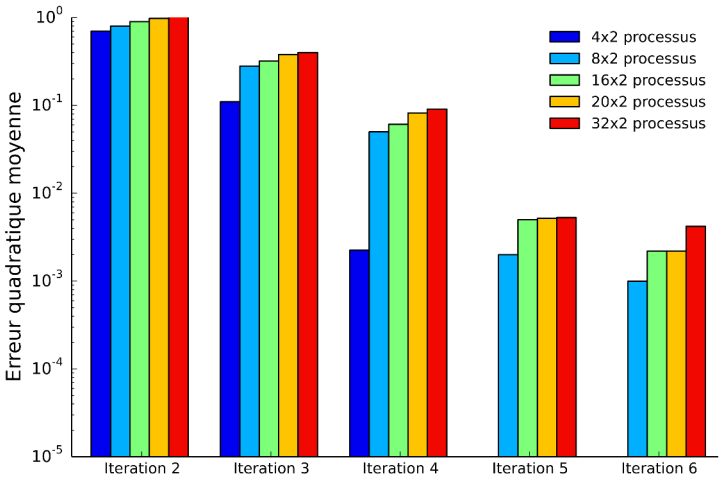

Figure 7.7 – Comparaison de l'erreur quadratique moyenne à chaque itération pour différents nombres de processus MPI, lors de l'application de la méthode pararéel couplé à la décomposition de domaine.

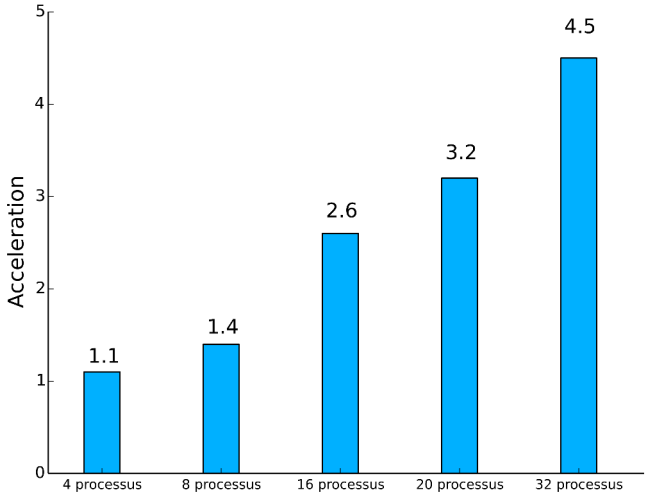

Figure 7.8 – Accélérations lors de l'utilisation de l'algorithme pararéel couplé à la décomposition de domaine, par rapport au temps de simulation séquentiel.

7.3 Simulations en dimension trois

Dans cette section, la méthode décrite précédemment est appliquée afin de pouvoir simuler le modèle monodomaine en dimensions trois, sur une géométrie complexe, demandant une grande ressource de calcul. Dans un premier temps, j'explique comment le maillage d'une géométrie complexe du cerveau avec une décomposition de domaine a été construit, puis une simulation avec des paramètres réalistes du modèle est effectuée. Les méthodes de résolution développées précédemment vont permettre de pouvoir effectuer cette simulation, afin d'obtenir une solution de bonne qualité, en un minimum de temps.

7.3.1 Création du maillage en trois dimensions

On part d'une surface numérique créée par des données obtenues en imagerie médicale, venant de Lancaster *et al.* (2007). A partir de ce fichier, contenant les coordonnées des points formant la surface du cerveau, ainsi qu'un premier maillage de cette surface par des triangles, on créé un nouveau fichier au format `mesh` supporté par le logiciel FreeFem. Ce fichier doit contenir différentes informations, notamment la dimension de l'espace, le nombre de sommets suivi de leurs coordonnées, et le nombre de triangles maillant la surface, suivi des liaisons entre eux (Code 7.1).

Code 7.1 – Aperçu d'un fichier au format mesh.

```
MeshVersionFormatted 1

Dimension 3

Vertices
81924
-2.832152 25.175578 14.043072 1
-13.824459 -55.618156 21.557579 1
-39.844431 9.443311 7.664743 1
-44.495649 -44.822843 18.692561 1

...


Triangles
163840
3 10243 10245 0
10243 2563 10244 0
10245 10243 10244 0
10245 10244 2565 0
```

A partir de ces informations qu'il a fallu transformer pour qu'elles soient exploitables, j'ai créé un maillage volumique. Pour réaliser cela, j'ai utilisé des fonctions de la bibliothèque TetGen. A la base, TetGen est un logiciel développé par le Dr. Hang Si de l'Institut Weierstrass à Berlin, en Allemagne (Si, 2015). Il permet de créer un maillage tétraédrique d'un domaine en trois

dimensions défini par ses limites.

La lancement de TetGen est effectué grâce au mot clé `tetg`. Ensuite, le maillage en trois dimensions peut être raffiné en utilisant la commande `tetgreconstruction` en lui définissant, par exemple, le taille du volume des tétraèdres. Pour plus d'informations, le lecteur intéressé est renvoyé à Si (2015) et Hecht (2012).

Au final, on obtient le maillage tétraédrique représenté par la Figure 7.9.

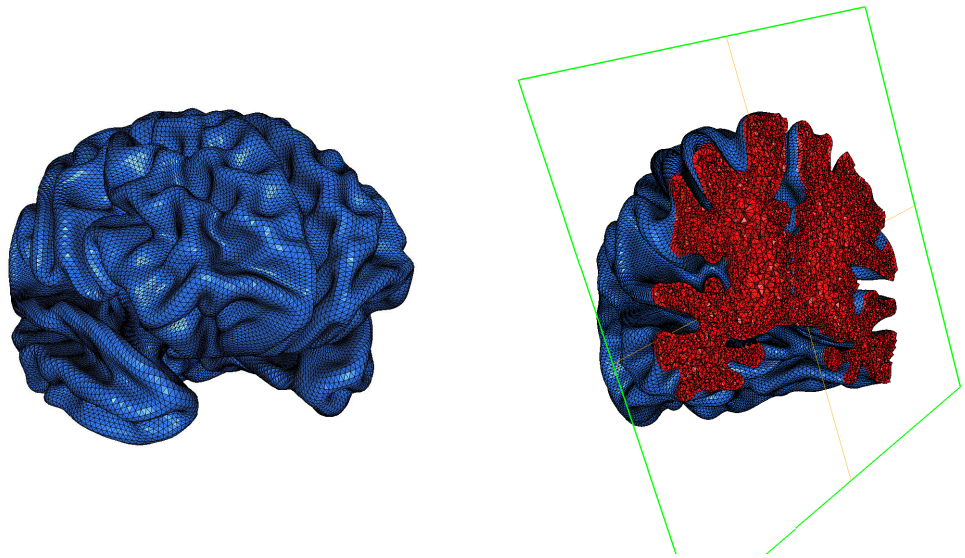

Figure 7.9 – Maillage tridimensionnel du cerveau.

7.3.2 Décomposition du maillage tridimensionnel

Une fois le maillage établi, je dois encore le découper en deux sous-domaines, afin de pouvoir appliquer la méthode de parallélisation. Dans ce cas, le découpage est bien moins trivial que le découpage d'un simple cercle en deux parties. Des logiciels tel que METIS (Karypis et Kumar, 1995, 1998), permettant de partitioner des maillages ou encore de réduire l'ordre des matrices creuses sont utilisés. Les algorithmes implémentés dans METIS sont basés sur des récursions-bisections multi-niveaux, des k-way multi-niveaux, et des partitions multi-contraintes. Ce logiciel va donc permettre de créer des partitions avec chevauchement ou sans chevauchement. Le découpage de notre maillage tridimensionnel est illustré dans la Figure 7.10.

7.3.3 Résultats numériques

Dans le maillage du cerveau illustré en Figure 7.9, on peut définir une zone représentant l'hippocampe, une structure du cerveau jouant un rôle central dans la mémoire et la navigation

spatiale. Elle se situe dans le lobe temporal médian, sous le cortex. Beaucoup d'aspects de cette structure ont été étudiés au sein des précédentes thèses de Rhenovia (Greget, 2011).

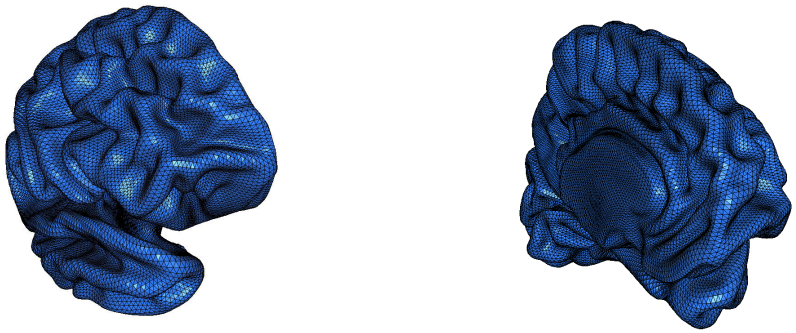

Figure 7.10 – Partition en deux parties distinctes, sans chevauchement, du maillage tridimensionnel.

Bien évidemment, de nombreuses autres structures composent le cerveau, mais dans un souci de simplification, nous ne considérons qu'une seule structure. On peut trouver dans certaines références une estimation des coordonnées, en centimètres, de la localisation de l'hippocampe : $(x\,;\,y\,;\,z) = (25,03\,;\,20,74\,;\,10,13)$. De plus, on considère le cerveau comme un milieu isotrope et homogène, même si en réalité les conductivités peuvent être différentes suivant les zones du cerveau. D'après Haueisen *et al.* (2002), les conductivités moyennes au sein de la matière grise et de la matière blanche sont données dans le tableau 7.5. Comme le milieu est isotrope et homogène, on retient qu'il n'y aucune distinction entre les différentes matières, on prend la moyenne de ces conductances : $0,37\ S.m^{-1}$.

A partir de ces différentes informations, nous allons exécuter une simulation à l'aide des méthodes décrites dans la section 6.3. On suppose que la propagation du potentiel d'action démarre au sein de l'hippocampe, entraînant une propagation du signal électrique dans le reste du cerveau, comme le montre la figure 7.10.

	Matière grise ($S.m^{-1}$)	Matière blanche ($S.m^{-1}$)
c_x	0,41	0,29
c_y	0,47	0,61
c_z	0,16	0,28

Tableau 7.5 – Valeurs des tenseurs de conductivités dans les deux milieux composant le cerveau selon Haueisen *et al.* (2002).

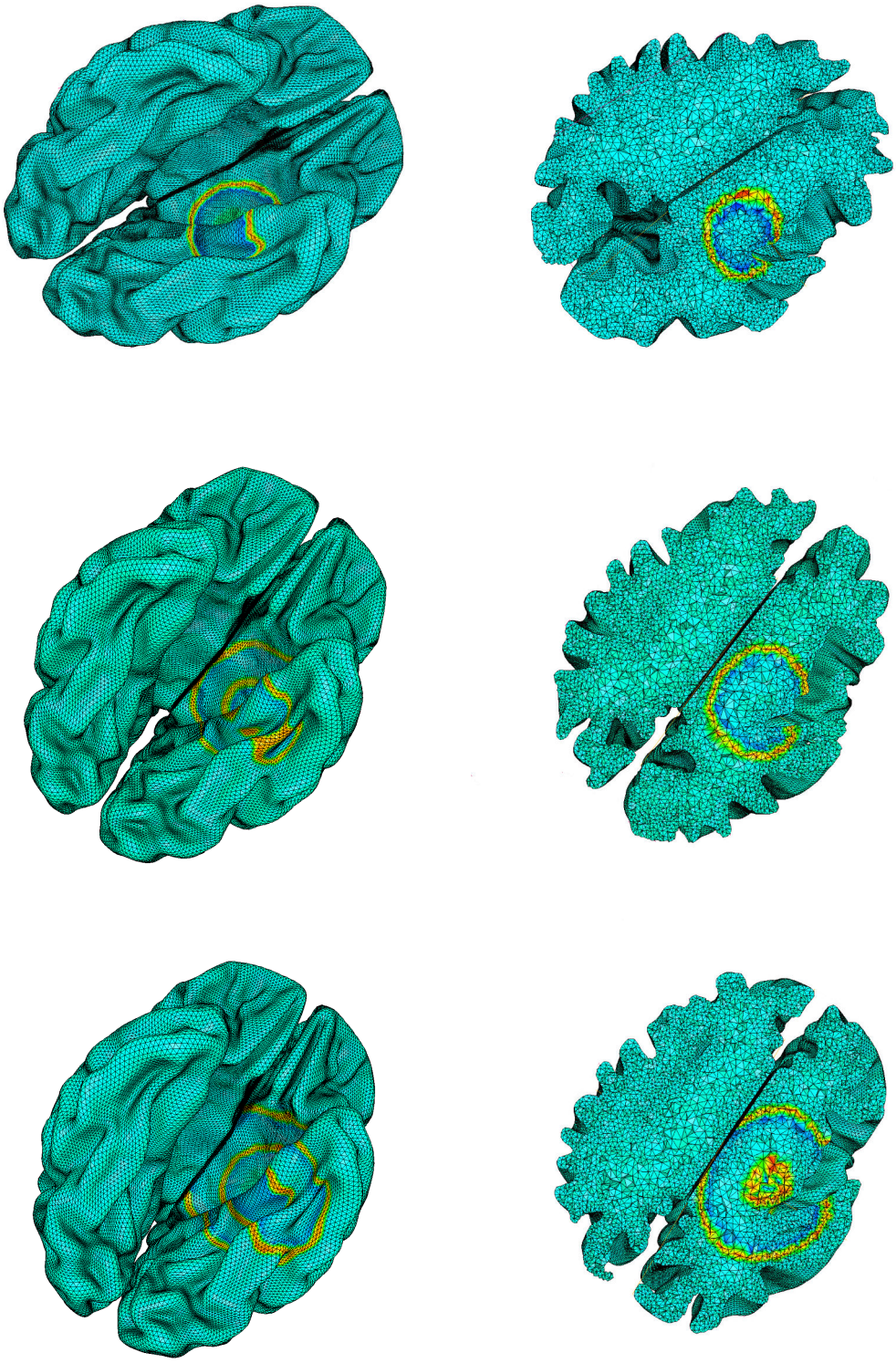

Figure 7.11 – Propagation du signal électrique, en surface (à gauche) et en profondeur (à droite) aux temps 45 ms, 65 ms et 85 ms (de haut en bas), après le déclenchement du premier potentiel d'action.

Il est à noter que cette simulation est un exemple parmi tant d'autres. La difficulté de devoir décrire l'activité électrique du cerveau dans sa globalité est la présence de connexions entre différentes régions d'intérêts 2.2.10 qui ne sont pas du tout prise en compte ici. L'intérêt est de pouvoir comparer les enregistrements obtenues avec des données réelles observées lors de diverses expériences "in-vivo", mais suite au condition particulière de fin de thèse, cette étape n'a pu être effectuée correctement.

7.4 Conclusion

A l'aide de la méthode des éléments finis, une géométrie plus complexe, décrivant au mieux celle du cerveau a été définie. Cela engendre d'énormes calculs, requérant un temps de simulation non-négligeable. Pour contourner ce problème, j'ai développé une méthode couplant une intégration pararéel et une décomposition de domaine. Cette dernière, à précision égale du pararéel seul, ne fait pas gagner de temps, mais elle devient utile lorsque l'on souhaite distinguer différentes zones, dont certaines demanderont un maillage moins important.

Partie IV

Simulations et applications des méthodes de parallélisation au modèle bidomaine

8 Simulations du modèle bidomaine à l'aide des différences finies

8.1 Introduction

Dans la partie précédente, j'ai développé et appliqué des méthodes de résolution d'EDP massivement parallèle. Pour cela, il a fallu découper aussi bien le domaine temporel que spatial, puis répartir les tâches afin de pouvoir les résoudre de façon totalement autonome. Ces méthodes ont été testées, puis validées sur le modèle monodomaine, simplification du modèle bidomaine.

La suite logique est d'appliquer ces différentes méthodes au modèle bidomaine, plus complexe. Ce modèle devient utile lorsque l'on souhaite simuler la propagation du signal électrique, au sein d'un ensemble couplé tel que le cerveau + crâne. Cependant, les simulations présentées ici se font sur un domaine simple représentant un tissu neuronal, ou encore le cerveau, comme lors de la résolution numérique du problème monodomaine.

Variables :

ϕ	Potentiel	[mV]
V_m	Potentiel de membrane	[mV]
C_m	Capacité membranaire	$[\mu F.cm^{-2}]$
A_m	Rapport d'aspect membranaire	$[cm^{-1}]$
M	Tenseur de conductivité	$[ms.cm-1]$
I_{ion}	Courant ionique membranaire	$[\mu A]$
t	Variable de temps	$[ms]$
x	Variable d'espace	$[cm]$

Indices :

i	Intra-cellulaire
e	Extra-cellulaire
B	Cerveau

On rappelle les notations et quantités intervenant dans le modèle bidomaine, qui ont été introduites dans la section 2.2.

On considère le domaine de définition global B représentant une fraction du cerveau ou le cerveau entier. Le modèle bidomaine se décompose en un problème en espace et deux problèmes en temps.

Problème en espace :

$$\nabla \cdot ((M_i + M_e)\nabla \phi_e) = -\nabla \cdot (M_i \nabla V_m) \text{ sur } B \quad , \tag{8.1}$$

dont les conditions aux limites sont de types Neumann :

$$\begin{cases} M_e \nabla \phi_e \cdot n_{\partial B} = 0 \quad \text{sur} \quad \partial B \\ M_i \nabla V_m \cdot n_{\partial B} = 0 \quad \text{sur} \quad \partial B \quad . \end{cases} \tag{8.2}$$

Problème d'évolution sur le potentiel de membrane :

$$-\nabla \cdot (M_i \nabla V_m) + A_m(C_m \partial_t V_m + I_{ion}) = \nabla \cdot (M_i \nabla \phi_e) \text{ dans } B \quad . \tag{8.3}$$

On rappelle également que le courant ionique I_{ion} est déterminé en chaque point de B grâce au modèle de Fitzhugh-Nagumo décrit par une équation d'évolution :

$$\begin{cases} I_{ion} = f(V_m, w) \\ \dfrac{\partial w}{\partial t} = g(V_m, w) \quad . \end{cases} \tag{8.4}$$

L'existence et l'unicité des solutions pour un tel problème ont été démontrées par Franzone et Savaré (2002), notamment lorsque le courant ionique est basé sur le modèle de Fitzhugh-Nagumo.

8.2 Simulations en dimension un

8.2.1 Résultats numériques

On considère le modèle bidomaine décrit par les équations (8.1) (8.3) (8.4) sur $\mathbb{R}$, où on suppose que les tenseurs de conductivités M_i, M_e sont des scalaires et les fonctions f, g sont

de type Fitzhugh-Nagumo :

$$\begin{cases} f(V_m, w) = V_m - V_m^3 - w - I(t) \\ g(V_m, w) = V_m - a - bw \quad . \end{cases} \tag{8.5}$$

Dans le cadre de nos simulations, on se place sur le segment $[0;1]$, en complétant ces équations par les conditions aux limites de Neumann suivantes :

$$\begin{cases} \frac{\partial V_m}{\partial x}(0) = 0 \quad , \quad \frac{\partial V_m}{\partial x}(1) = 0 \\ \frac{\partial \phi_e}{\partial x}(0) = 0 \quad , \quad \frac{\partial \phi_e}{\partial x}(1) = 0 \quad . \end{cases} \tag{8.6}$$

On créé un maillage uniforme du segment $[0;1]$ grâce à un découpage en N sous-segments de même taille $\Delta x = 1/N$. On note respectivement $V_{m,j}^n$, $\phi_{e,j}^n$ et w_j^n les approximations respectives de $V_m(x_j, t_n)$, $\phi_e(x_j, t_n)$ et $w(x_j, t_n)$ où $x_j = j\Delta x$, $t_n = n\Delta t$.

On définit les vecteurs V_m^n, Φ^n et W^n ayant pour coordonnées respectives $V_{m,j}^n$, $\phi_{e,j}^n$ et w_j^n. On discrétise les équations (8.1) (8.3) (8.4) à l'aide d'un schéma d'Euler explicite en temps, donnant ainsi :

$$\begin{cases} V_m^{n+1} = V_m^n + \dfrac{\Delta t}{A_m C_m}\left(A \cdot (V_m^n + \Phi^n) - A_m f(V_m^n, W^n)\right) \\ B \cdot \Phi^{n+1} = A \cdot V_m^{n+1} \\ W^{n+1} = g(V_m^n, w^n) \quad , \end{cases} \tag{8.7}$$

où A et B sont des matrices de tailles $N \times N$ déterminées grâce à la discrétisation des opérateurs $\nabla \cdot (M_i \nabla V_m)$, $\nabla \cdot (M_i \nabla \phi_e)$ et $\nabla \cdot ((M_i + M_e)\nabla \phi_e)$ respectivement. En dimension un, les tenseurs sont des scalaires $m_i, m_e \in \mathbb{R}$, ce qui permet une simplification de ces opérateurs donnés par $m_i \Delta V_m$, $m_i \Delta \phi_e$ et $(m_i + m_e)\Delta \phi_e$. Les matrices A et B s'écrivent :

$$A = m_i \begin{pmatrix} -1 & 1 & & & \\ 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -2 & 1 \\ & & & 1 & -1 \end{pmatrix} \quad ; \quad B = (m_i + m_e) \begin{pmatrix} -1 & 1 & & & \\ 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -2 & 1 \\ & & & 1 & -1 \end{pmatrix} \quad ,$$

Afin d'éviter de stocker inutilement les zéros de ces différentes matrices, et en anticipant les résolutions numériques sur des dimensions supérieures, on choisit de profiter de leur caractéristique creuse. Seul les éléments différents de zéro sont ainsi sauvegardés permettant ainsi de diminuer grandement l'espace mémoire requis, mais également les temps de résolution de certaines étapes de calcul, notamment l'inversion de la matrice B.

En pratique, à chaque étape, on détermine la valeur de V_m^{n+1} grâce aux précédentes valeurs

V_m^n, Φ^n et W^n, puis on effectue la mise à jour de W en utilisant le modèle de courant ionique retenu, et enfin grâce à l'actualisation de V_m au temps t_{n+1}, on détermine la valeur de Φ^{n+1}.

En ce qui concerne la résolution en elle-même, il est possible d'utiliser des méthodes de décomposition de matrices classiques, car il est utile de perdre le moins de temps CPU possible lors de la mise à jour de Φ. De plus, la matrice B étant non inversible, on se tourne donc vers la méthode du gradient biconjugué stabilisé (BICGSTAB). Plusieurs études ont comparé différentes méthodes dans le but de résoudre ce problème, tel que l'algorithme du gradient biconjugué (Press *et al.*, 2007) ou encore la méthode CGS (biconjugate gradient squared) (Sonneveld, 1989). D'autres applications ont montré que d'autres variantes comme QMRCGSTAB (Gallopoulos *et al.*, 1994) ne donnent aucune amélioration par rapport à BICGSTAB.

L'algorithme BICGSTAB est une méthode itérative utilisant les sous-espaces de Krylov. Son itération s'arrête lorsque la norme du résidu arrive à une tolérance spécifiée par l'utilisateur (ou $1,0 \times 10^{-5}$ par défaut).

La résolution du problème en différences finies a été développée en Python, sur le même principe que pour le monodomaine, c'est-à-dire le développement de fonctions et de classes, permettant à l'utilisateur de pouvoir effectuer des simulations rapides, en donnant les valeurs des différents paramètres de simulation et de stimulation du modèle.

On fixe des paramètres de simulation, comme pour la simulation du modèle monodomaine (chapitre 6), dont on spécifie les valeurs dans le tableau 8.1.

Paramètres	Valeurs
T_{final}	$100\ ms$
Δt	$1,0 \times 10^{-4}\ ms$
Δx	$1,0 \times 10^{-3}$
A_m	$2000\ cm^{-1}$
C_m	$1\ \mu F.cm^{-1}$
m_i	$1\ mS.cm^{-1}$
m_e	$1\ mS.cm^{-1}$
$[T_i; T_f]_{stim}$	[10; 100]
$[x_i; x_f]_{stim}$	[0,1; 0,2]
I_{app}	$0,4\ \mu A$

Tableau 8.1 – Valeurs des paramètres de simulation du modèle bidomaine en une dimension.

Un objet `SimParam` est ainsi créé avec ces différentes valeurs. L'utilisateur n'a plus qu'à lancer sa simulation en faisant appel à la fonction `BidomainSimulation1D(...)`. La Figure 8.1 présente les résultats à différentes localisations au cours du temps. On constate effectivement un déplacement du potentiel d'action le long du segment.

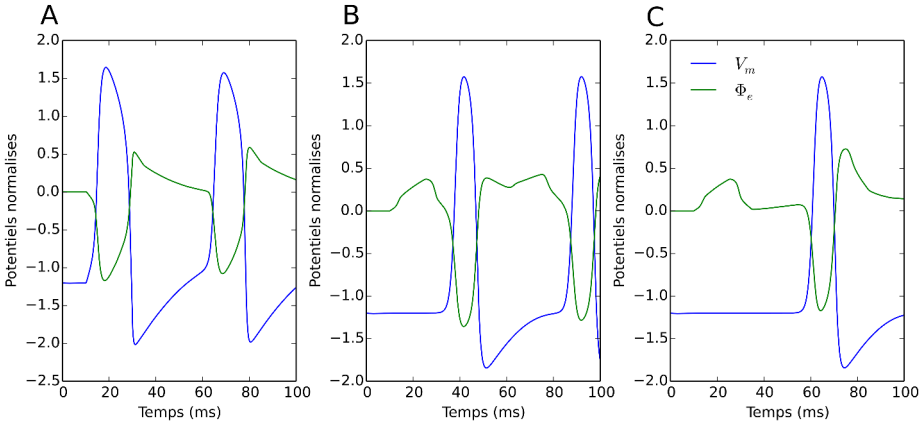

Figure 8.1 – Représentation graphique du potentiel de membrane, suite à la simulation du modèle bidomaine en une dimension en différentes positions : **A** : x = 0,2 ; **B** : x = 0,4 et **C** : x = 0,6.

8.3 Simulations en dimension deux

On s'attache dans cette section à effectuer la simulation du modèle bidomaine en deux dimensions, d'abord séquentiellement, puis en appliquant les méthodes de parallélisation vues dans les chapitres précédents. Nous partons des équations (8.1) (8.3) (8.4) où M_i et M_e sont des matrices de tailles 2×2 telles que :

$$M_i = \begin{pmatrix} c_{i,x} & 0 \\ 0 & c_{i,y} \end{pmatrix} \quad M_e = \begin{pmatrix} c_{e,x} & 0 \\ 0 & c_{e,y} \end{pmatrix} \quad ,$$

où $c_{i,x}$, $c_{i,y}$ sont les conductivités intra-cellulaires selon les axes x, y, et $c_{e,x}$, $c_{e,y}$ les conductivités extra-cellulaires selon ces deux mêmes axes. On suppose que le tissu neuronal est représenté par le carré $[0;1]^2$ sur lequel on applique le modèle bidomaine, avec des conditions aux limites de Neumann sur les variables V_m et Φ_e (Equation 8.2).

Soient N_x et N_y deux entiers représentant le nombre de cellules dans les directions x et y. On suppose que $N_x = N_y$ afin d'obtenir un maillage uniforme dont la taille des cellules est donnée par $\Delta x = \Delta y = 1/N_x$.

On cherche une approximation $V^n_{m,i,j}$, $\phi^n_{e,i,j}$ et $w^n_{i,j}$ des valeurs de $V_m(x_i; y_j; t_n)$, $\phi_e(x_j; y_j; t_n)$ et $w(x_j; y_j; t_n)$.

Cette fois, on doit construire différentes lors de la discrétisation des opérateurs $\nabla \cdot (M_i \nabla V_m)$, $\nabla \cdot (M_i \nabla \phi_e)$ et $\nabla \cdot ((M_i + M_e) \nabla \phi_e)$, car les valeurs des tenseurs sont différentes. Lors de la résolution numérique en dimension un, la même matrice pouvait être utilisée, puisque une simple multiplication scalaire suffisait pour obtenir la bonne matrice de discrétisation. On a

les formules suivantes :

$$\begin{cases} \nabla \cdot (M_i \nabla V_m) & = & c_{i,x} \dfrac{\partial^2 V_m}{\partial x^2} + c_{i,y} \dfrac{\partial^2 V_m}{\partial y^2} \\ \nabla \cdot (M_i \nabla \phi_e) & = & c_{i,x} \dfrac{\partial^2 \phi_e}{\partial x^2} + c_{i,y} \dfrac{\partial^2 \phi_e}{\partial y^2} \\ \nabla \cdot ((M_i + M_e) \nabla V_m) & = & (c_{i,x} + c_{e,x}) \dfrac{\partial^2 V_m}{\partial x^2} + (c_{i,y} + c_{e,y}) \dfrac{\partial^2 V_m}{\partial y^2} \quad . \end{cases} \tag{8.8}$$

En utilisant les différences finies, on obtient la relation suivante :

$$\alpha \frac{\partial^2 u}{\partial x^2} + \beta \frac{\partial^2 u}{\partial y^2} = \alpha \frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{h^2} + \beta \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{h^2} \quad , \tag{8.9}$$

où $u = \{V_m \; ; \; \phi_e\}$, $h = \Delta x = \Delta y$, $\alpha = \{c_{i,x} \; ; \; c_{i,x} + c_{e,x}\}$ et $\beta = \{c_{i,y} \; ; \; c_{i,y} + c_{e,y}\}$.

Le système discret 8.7 utilisent donc des matrices A et B de tailles $N_x^2 \times N_y^2$ définies de façon quasi-similaire à la matrice A du chapitre 6. Les vecteurs V_m, Φ et W sont également de taille N_x^2.

8.3.1 Résolution séquentielle

On profite à nouveau des caractéristiques creuses des différentes matrices afin de prendre le moins d'espace mémoire possible, ainsi que de réduire le temps de résolution du système globale. Il est encore plus important d'utiliser des matrices creuses par rapport à l'application en une dimension, car les matrices peuvent devenir très importantes et sont au nombre de deux.

On stimule le modèle bidomaine à l'aide d'un courant externe appliqué sur une fraction carré du tissu, modélisé par l'ensemble $[0;1]^2$. Tous les paramètres de la simulation, donnés dans le tableau 8.2, sont définis dans des objets des classes `SimParam` et `CurrentApplied` déjà utilisés dans la section 6.3.

Paramètres	Valeurs
T_{final}	100 ms
Δt	$1,0 \times 10^{-3}$ ms
Δx ; Δy	$5,0 \times 10^{-2}$; $5,0 \times 10^{-2}$
A_m	2000 cm^{-1}
C_m	1 $\mu F.cm^{-1}$
$[c_{i,x}$; $c_{i,y}]$	[1 $mS.cm^{-1}$; 1 $mS.cm^{-1}$]
$[c_{e,x}$; $c_{e,y}]$	[1 $mS.cm^{-1}$; 1 $mS.cm^{-1}$]
$[T_i; T_f]_{stim}$	[10;100]
$[x_i; x_f]_{stim} \times [y_i; y_f]_{stim}$	[0,4;0,6] × [0,4;0,6]
I_{app}	0,4 μA

Tableau 8.2 – Valeurs des paramètres de simulation du modèle bidomaine en dimension deux

L'intérêt de cette simulation, dont les résultats sont présentés Figure 8.2, est de pouvoir tester et valider la résolution séquentielle, afin de pouvoir la comparer avec les implémentations parallèles de ce problème. Plus l'exigence demandée devient importante en termes de finesse des résultats, plus le temps de simulation va devenir important. Ce temps est bien plus important que pour la simulation du modèle monodomaine, car il faut garder à l'esprit qu'il est nécessaire de résoudre un problème elliptique à chaque pas de temps, en plus des deux équations d'évolution, déjà présentes dans le modèle monodomaine.

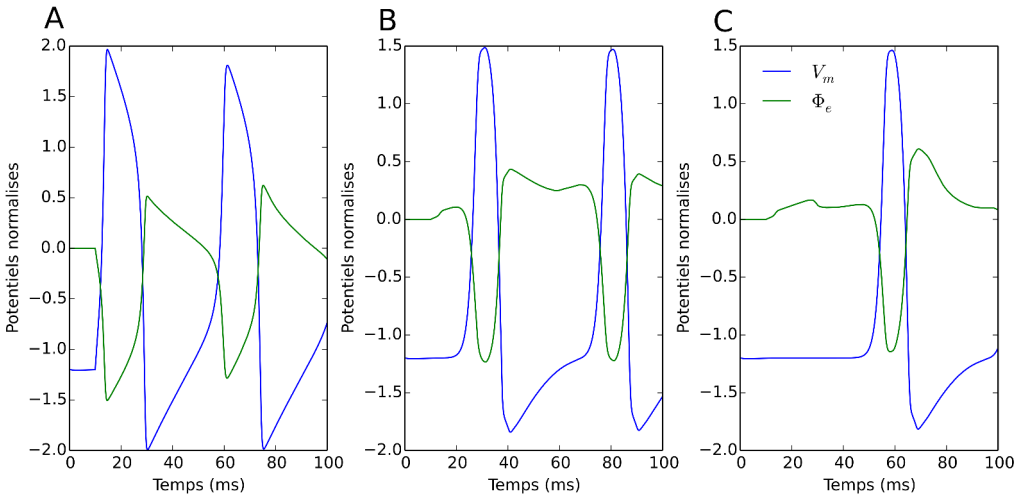

Figure 8.2 – Représentation graphique du potentiel de membrane, suite à la simulation du modèle bidomaine en deux dimensions en différentes positions : **A** : (x ; y) = (0,5 ; 0,5) ; **B** : (x ; y) = (0,5 ; 0,7) et **C** : (x ; y) = (0,5 ; 0,9).

Le temps d'intégration requis pour cette configuration est de 1380 s là où il fallait 2500 s, sur un nœud du cluster de Rhenovia, pour le modèle monodomaine avec un pas de temps dix fois plus petit. De plus, on utilise cette fois des matrices creuses censées diminuer le temps de calcul. Cela donne un aperçu du temps nécessaire requis. Il est donc impératif d'utiliser des méthodes de calcul parallèle pour permettre de déterminer rapidement des solutions précises.

8.3.2 Simulations avec la méthode pararéel en MPI

On souhaite maintenant appliquer la méthode développée dans la section 6.3 afin de diminuer le temps de résolution. Par rapport au modèle monodomaine, il convient, à chaque processus, de communiquer la troisième variable ϕ_e au processus suivant, à la fin de son travail de l'itération courante. Comme le processus doit communiquer les valeurs à chaque noeud, il doit faire transiter $3N_x^2$ valeurs à chaque communication. Les différentes matrices sont construites par un seul processus, avant le début de la résolution, puis la mémoire est partagée afin d'éviter de stocker inutilement des matrices identiques, pouvant saturer la mémoire.

On utilise à nouveau le schéma d'Euler explicite lors de la résolution sur la grille de temps fine, et le schéma RK4 sur la grille grossière. On applique le même protocole de simulation vu dans la table 8.2, permettant une comparaison entre les solutions parallèle et séquentielle. Les différents tests sont lancés sur un nombre croissant de processus, permettant de constater (Figure 8.3) l'évolution de l'erreur au cours des itérations, en fonction du nombre de processus.

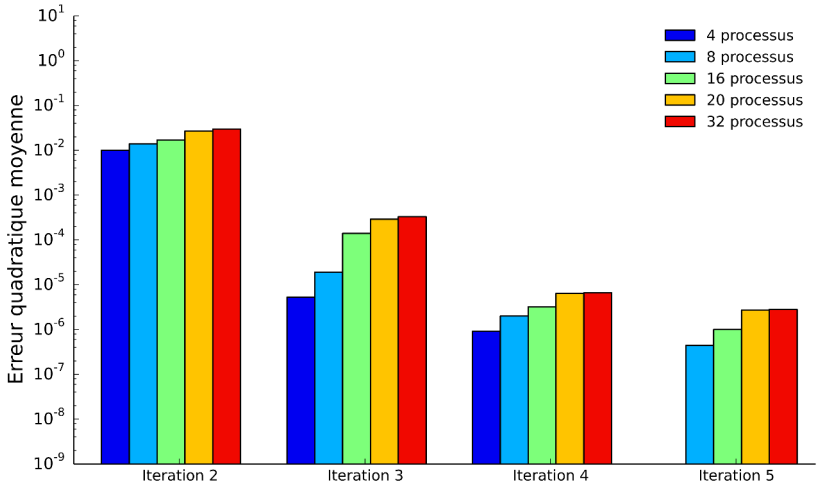

Figure 8.3 – Comparaison pour différents nombres de processus, de l'erreur quadratique moyenne à chaque itération, lors de l'utilisation de la méthode pararéel en MPI.

Cette erreur quadratique moyenne décroit lorsque le nombre d'itérations augmente, jusqu'à arriver à une tolérance acceptable sur cette dernière. En couplant ces données avec les accélérations obtenues (Figure 8.4), on constate que le gain de temps est d'un facteur 8 environ. Ces valeurs sont bien plus importantes que celles observées jusqu'ici. L'erreur commise diminue rapidement dès les premières itérations, arrivant rapidement à atteindre la tolérance requise vers la troisième itération.

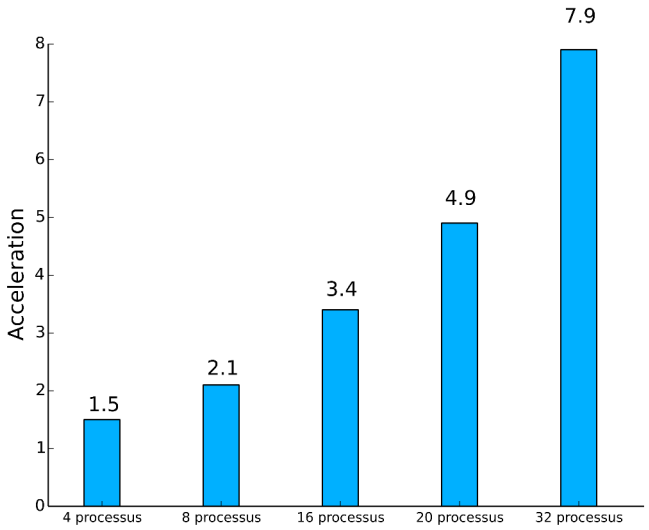

Figure 8.4 – Accélérations obtenues à l'aide de MPI, pour chaque nombre de processus.

8.3.3 Simulations avec la méthode pararéel en CUDA

Afin de pouvoir augmenter la précision sur la solution en temps, mais également en espace, la méthode présentée dans la section 6.3.3 a été adaptée au modèle bidomaine. Le principe est toujours identique, on lance l'algorithme pararéel au sein d'une grille nommée grille parent, puis cette dernière afin d'accélérer encore le temps de résolution du problème, lance les calculs des laplaciens et du problème elliptique au sein d'une grille fille. On utilise à nouveau des streams CUDA afin de pouvoir obtenir le maximum de concurrence entre les différents exécutions des grilles.

Il est par contre nécessaire d'utiliser une approche autre que la méthode BICGSTAB afin de résoudre le problème elliptique. Comme pour la résolution de l'équation d'évolution selon V_m, on souhaite qu'un thread de la grille fille représente un noeud du maillage. Cela évitera de devoir stocker les matrices A et B en mémoire, permettant ainsi de dégager de la place pour tout autre besoin. A nouveau, on profite de l'écriture explicite en différences finies de

l'équation (8.1) à l'instant t_n (non mentionné ici, afin de simplifier l'écriture) :

$$(c_{i,x} + c_{e,x})\frac{\partial \phi_e}{\partial x^2} + (c_{i,y} + c_{e,y})\frac{\partial \phi_e}{\partial y^2} = -\left(c_{i,x}\frac{\partial V_m}{\partial x^2} + c_{i,y}\frac{\partial V_m}{\partial y^2}\right) \quad , \tag{8.10}$$

qui devient, dans sa forme discrétisée :

$$\begin{aligned}(c_{i,x} + c_{e,x})\frac{\phi_{e,i-1,j} - 2\phi_{e,i,j} + \phi_{e,i+1,j}}{h^2} + (c_{i,y} + c_{e,y})\frac{\phi_{e,i,j-1} - 2\phi_{e,i,j} + \phi_{e,i,j-1}}{h^2} = \\ -\left(c_{i,x}\frac{V_{m,i-1,j} - 2V_{m,i,j} + V_{m,i+1,j}}{h^2} + c_{i,y}\frac{V_{m,i,j-1} - 2V_{m,i,j} + V_{m,i,j+1}}{h^2}\right) \quad .\end{aligned} \tag{8.11}$$

Au sein du thread $(i; j)$, la valeur à déterminer est $\phi_{e,i,j}$, tout le reste est censé être connu. Ce thread doit donc faire appel à tous ces threads voisins, afin d'obtenir les autres valeurs. Bien évidemment, un grand nombre d'accès mémoire est requis, ce qui peut ralentir grandement la résolution, étant donné que l'on utilise en grande majorité de la mémoire globale. Pour diminuer les accès, on décide de calculer séparément, sur une grille fille, la valeur de $\nabla \cdot (M_i \nabla V_m)$ à chaque coordonnée $(i; j)$. Etant donné qu'on réutilise ces valeurs lors de la mise à jour de V_m, il suffira de faire un appel mémoire contre trois habituellement.

Il faut aussi prendre en considération les conditions de Neumann, que l'on applique au modèle. On doit donc distinguer les cas où les coordonnées des threads se trouvent au bord du domaine de définition.

Au final, on détermine la valeur de $\phi_{e,i,j}$, en l'isolant, puis en appliquant des itérations de Jacobi, grâce à la relation (8.11).

On applique le protocole de simulations 8.2 au modèle bidomaine, puis on compare les résultats séquentiel et parallèle, en déterminant l'erreur quadratique moyenne. La Figure 8.5 permet de confirmer les résultats obtenus avec MPI, lorsque l'itération augmente, l'erreur diminue jusqu'à atteindre un niveau de tolérance acceptable. Les accélérations (Figure 8.6) permettent de conforter les accélérations obtenues dans les chapitres précédents. Néanmoins, l'erreur stagne vers $1,0 \times 10^{-4}$, en partie due à la résolution du problème elliptique, qui demande un temps plus long entraînant des accélérations moins impressionnantes que lors de la simulation du modèle monodomaine.

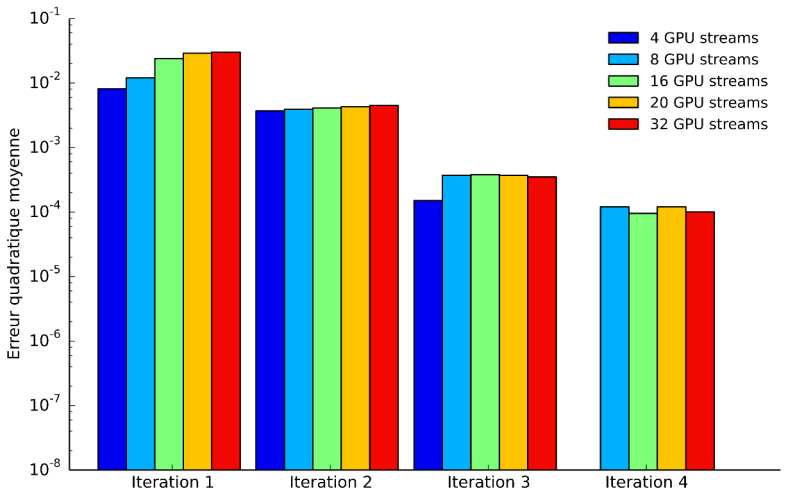

Figure 8.5 – Comparaison pour différents nombres de streams CUDA, de l'erreur quadratique moyenne à chaque itération, lors de l'utilisation de la méthode pararéel sur GPU.

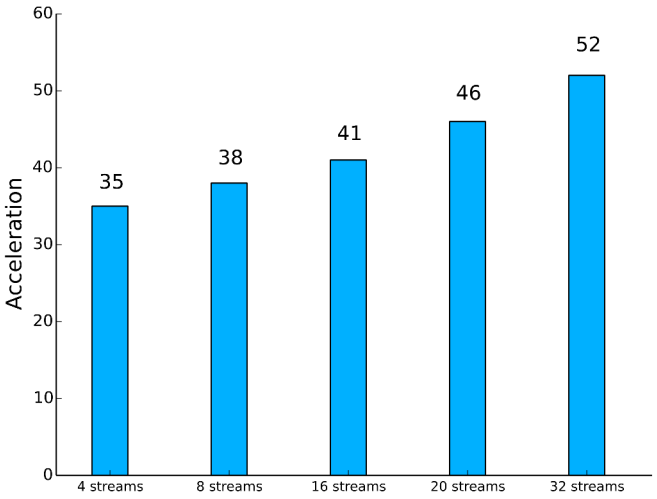

Figure 8.6 – Accélérations obtenues à l'aide de CUDA, pour chaque nombre de streams CUDA utilisé.

8.4 Simulations en dimension trois

On suppose que les matrices de conductivités M_i et M_e de taille 3×3 sont de la forme :

$$M_i = \begin{pmatrix} c_{i,x} & 0 & 0 \\ 0 & c_{i,y} & 0 \\ 0 & 0 & c_{i,z} \end{pmatrix} \quad M_e = \begin{pmatrix} c_{e,x} & 0 & 0 \\ 0 & c_{e,y} & 0 \\ 0 & 0 & c_{e,z} \end{pmatrix} \quad ,$$

avec respectivement $c_{i,x}$, $c_{i,y}$, $c_{i,z}$, $c_{e,x}$, $c_{e,y}$, $c_{e,z}$ les conductivités intra-cellulaire, extra-cellulaire selon les axes x, y et z.

On représente le tissu neuronal sur lequel on applique le modèle bidomaine par le cube unitaire $[0;1]^3$. On rappelle également que le modèle est complété par des conditions de Neumann.

On suppose que le maillage est formé de $N_x \times N_y \times N_z$ cellules de taille $\Delta_x \times \Delta_y \times \Delta_z$ identique. Comme le maillage est supposé uniforme, on a : $N_x = N_y = N_z$ et donc : $\Delta_x = \Delta_y = \Delta_z$.

Le système d'équations est discrétisé par le système (8.7) dont les matrices A et B sont de tailles : $N_x^3 \times N_x^3$.

8.4.1 Résolution séquentielle

On applique le protocole de stimulation définit par les objets Python, qui sont des instances des classes `SimParam` et `CurrentApplied`, dont les différents paramètres sont donnés dans le tableau 8.3.

Paramètres	Valeurs
T_{final}	$100\ ms$
Δt	$1,0 \times 10^{-3}\ ms$
$\Delta x\ ;\ \Delta y\ ;\ \Delta z$	$1,0 \times 10^{-1}\ ;\ 1,0 \times 10^{-1}\ ;\ 1,0 \times 10^{-1}$
A_m	$2000\ cm^{-1}$
C_m	$1\ \mu F.cm^{-1}$
$[c_{i,x}\ ;\ c_{i,y}\ ;\ c_{i,z}]$	$[1\ mS.cm^{-1}\ ;\ 1\ mS.cm^{-1}\ ;\ 1\ mS.cm^{-1}]$
$[c_{e,x}\ ;\ c_{e,y}\ ;\ c_{e,z}]$	$[1\ mS.cm^{-1}\ ;\ 1\ mS.cm^{-1}\ ;\ 1\ mS.cm^{-1}]$
$[T_i;T_f]_{stim}$	$[10;100]$
$[x_i;x_f]_{stim} \times [y_i;y_f]_{stim} \times [z_i;z_f]_{stim}$	$[0,4;0,6] \times [0,4;0,6] \times [0,4;0,6]$
I_{app}	$0,4\ \mu A$

Tableau 8.3 – Valeurs des paramètres de simulation du modèle bidomaine en dimension trois.

Sur le même principe que la simulation en dimension deux, on profite de l'aspect creux des matrices A et B, car les ressources en dimension trois deviennent très vite importantes. La taille

des matrices étant de $N_x^3 \times N_x^3$, la moindre augmentation au niveau de la taille du maillage devient très vite importante, pour au final sept données intéressantes sur une ligne de la matrice. Il est donc indispensable de se débarrasser des données superflues. Les constructions des matrices sont plus longues, mais le gain de temps se fait ressentir ensuite lors de la résolution. Après la construction, il est fortement recommandé de tout de même vider la mémoire des tableaux qui ne seront plus utilisés par la suite, en utilisant la fonction `del` de Python. On présente une série de résultats dans la Figure 8.7 représentant l'évolution du potentiel d'action ainsi que du potentiel extra-cellulaire, en différentes localisations.

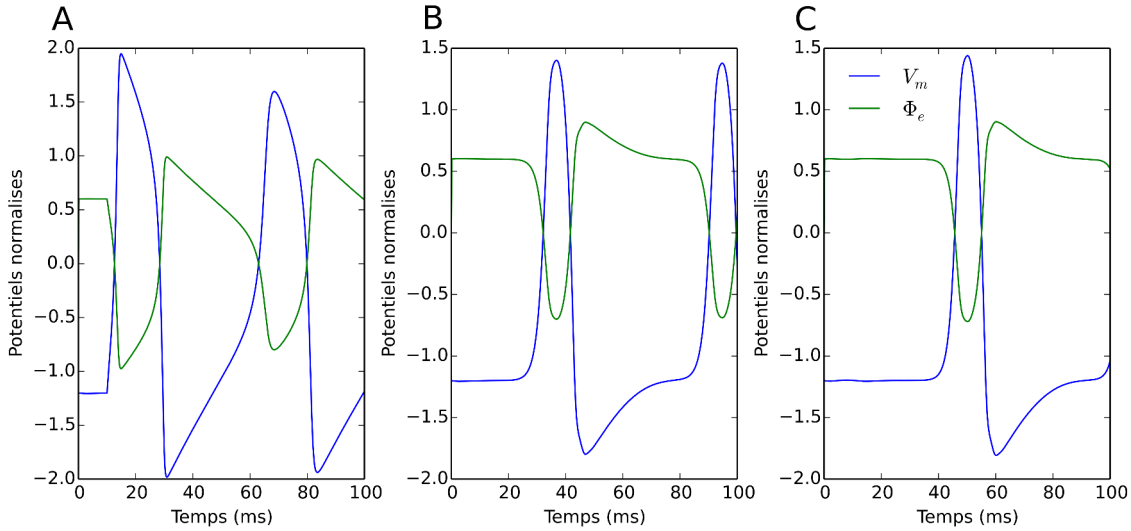

Figure 8.7 – Représentation graphique du potentiel de membrane, suite à la simulation du modèle bidomaine en trois dimensions en différentes positions : **A** : (x ; y ; z) = (0,5 ; 0,5 ; 0,5) ; **B** : (x ; y ; z) = (0,5 ; 0,5 ; 0,75) et **C** : (x ; y ; z) = (0,5 ; 0,5 ; 0,875).

8.4.2 Simulations avec la méthode pararéel en MPI

Lorsque l'on souhaite une excellente précision de résultats, le problème bidomaine devient très long à résoudre en dimension deux, ce qui est encore plus vrai lorsque l'on se place en dimension trois. Ainsi, au vu des résultats précédents, la suite logique était d'appliquer les différentes techniques et méthodes vu précédemment à ce problème.

La figure 8.8 montre que l'erreur décroit en fonction du nombre d'itération et de façon identique au problème en dimension deux, permettant ainsi de pouvoir obtenir des performances acceptables (Figure 8.9). Nous sommes loin des valeurs théoriques, encore une fois, mais ce gain est loin d'être négligeable surtout quand le temps de résolution en séquentiel est aussi très long. Il devient envisageable de pouvoir obtenir une solution de bonne qualité du problème bidomaine.

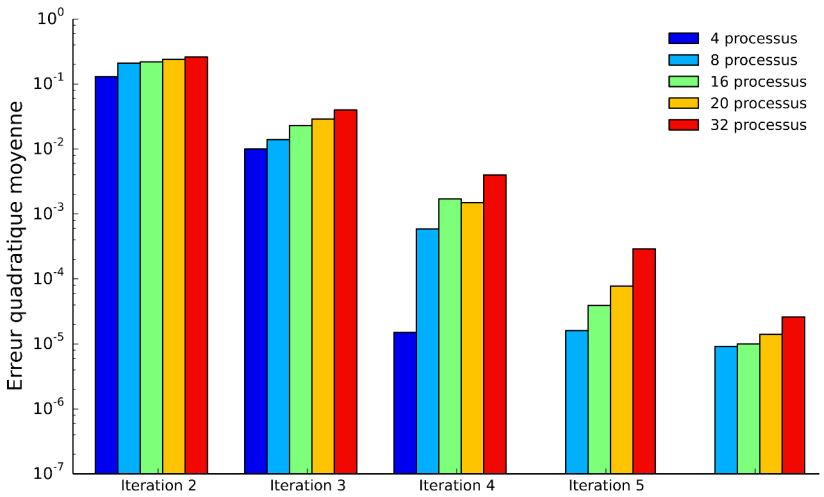

Figure 8.8 – Comparaison pour différents nombres de processus, de l'erreur quadratique moyenne à chaque itération, lors de l'utilisation de la méthode pararéel en MPI.

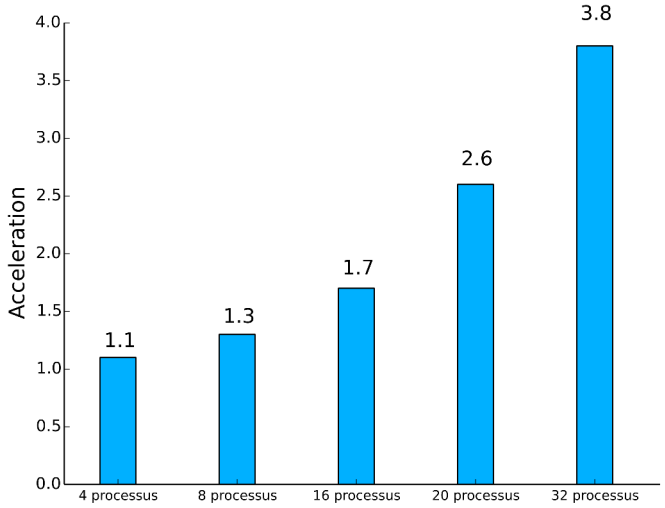

Figure 8.9 – Accélérations obtenues à l'aide de MPI, pour chaque nombre de processus.

8.4.3 Simulations avec la méthode pararéel en CUDA

On souhaite maintenant appliquer la méthode permettant de paralléliser massivement le modèle bidomaine à l'aide de CUDA. La technique est la même que dans la section 8.3.3 à

ceci près que cette fois ci, nous devons définir des grilles filles en trois dimensions, afin de représenter au mieux la discrétisation spatiale du problème.

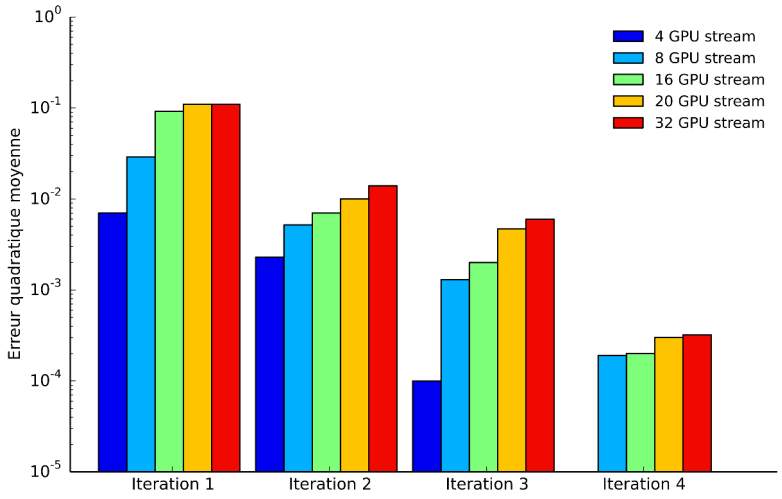

Figure 8.10 – Comparaison pour différents nombres de streams CUDA, de l'erreur quadratique moyenne à chaque itération, lors de l'utilisation de la méthode pararéel sur GPU.

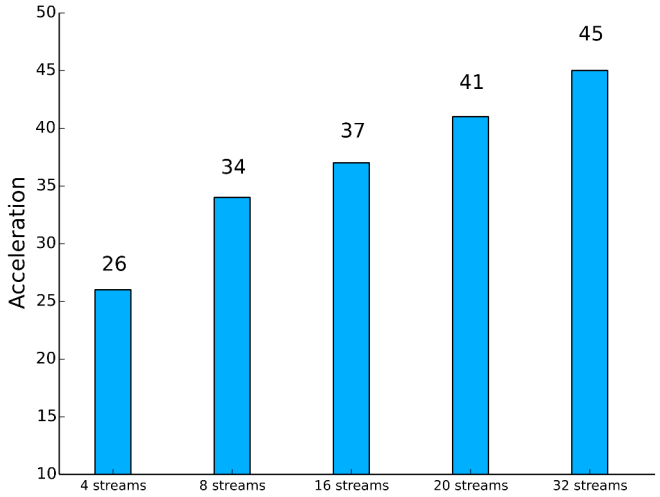

Figure 8.11 – Accélérations obtenues à l'aide de CUDA, pour chaque nombre de streams CUDA utilisé.

On constate que le gain de temps obtenu est important (Figure 8.10 - Figure 8.11), permettant de pouvoir résoudre le problème bidomaine sur de très grands maillages, dans la limite

technique du GPU (voir section 6.4.3). En général, on constate que le temps de résolution est quasiment identique à celui atteint en dimension deux. Ceci est logique, car le nombre d'opérations par thread est toujours le même : on définit, au mieux, les grilles filles CUDA de manière à ce qu'un noeud du maillage puisse être traité par un thread.

8.5 Conclusion

Le modèle bidomaine a été simulé à l'aide de la méthode des différences finies. Cela a permis de valider le modèle, puis pour diminuer le temps de calcul des différentes résolutions, les méthodes utilisées dans le chapitre 6 ont été appliquées. Une adaptation est nécessaire afin qu'elles puissent au mieux intégrer le problème bidomaine, plus complexe et plus coûteux à résoudre. Les méthodes MPI et CUDA ont notamment aidé à atteindre cet objectif.

Dans le prochain chapitre, la méthode des éléments finis va être appliquée à ce modèle, permettant d'utiliser des géométries plus complexes telle que le cerveau. Les différentes techniques développées dans le chapitre 7 seront aussi utilisées, afin de paralléliser au mieux la simulation du modèle.

9 Simulations du modèle bidomaine à l'aide des éléments finis

9.1 Introduction

Afin de prendre en compte des géométries plus complexes, comme pour le modèle monodomaine, je vais utiliser la méthode des éléments finis afin de réaliser la simulation du modèle bidomaine. Dans un premier temps, j'aborderai la convergence de cette méthode appliquée au modèle bidomaine, en utilisant les techniques déjà utilisées dans le chapitre 7, puis nous appliquerons les méthodes parallèles déjà utilisées dans le but d'effectuer des tests sur la géométrie complexe du cerveau.

9.2 Formulation faible du problème bidomaine

Dans un premier temps, nous allons développer la formulation faible du modèle bidomaine décrit par les équations (8.1) (8.3) (8.4), à l'aide des éléments finis. Les inconnues du problème bidomaine sont les suivantes :

$$\phi_e \in V = \quad \left\{ v \in H^1(B) \quad : \quad \int_B \phi_e dx = 0 \right\} \tag{9.1}$$

$$V_m \in C(0,T;V) \quad , \tag{9.2}$$

où B représente le domaine du cerveau. Ainsi, on obtient les formulations faibles du problème :

$$-\int_B \nabla \cdot (M_i \nabla V_m)\eta + \int_B A_m(C_m(\partial_t V_m) + f(V_m, w))\eta = \int_B \nabla \cdot (M_i \nabla \phi_e)\eta \quad , \tag{9.3}$$

avec $\eta \in H^1(B)$. Grâce à une intégration par partie, on obtient :

$$\int_B M_i \nabla V_m \nabla \eta - \int_{\partial B} M_i \nabla V_m \cdot n_{\partial B} \eta + \int_B A_m (C_m(\partial_t V_m) + f(V_m, w))\eta = \int_B \nabla \cdot (M_i \nabla \phi_e)\eta \qquad (9.4)$$

$$\Leftrightarrow \int_B M_i \nabla V_m \nabla \eta + \int_{\partial B} M_i \nabla \phi_e \cdot n_{\partial B} \eta + \int_B A_m (C_m(\partial_t V_m) + f(V_m, w))\eta = \int_B \nabla \cdot (M_i \nabla \phi_e)\eta \quad . \qquad (9.5)$$

En insérant les différentes conditions de Neumann, on a la formulation faible de l'équation d'évolution :

$$\int_B M_i \nabla V_m \nabla \eta + \int_B A_m (C_m(\partial_t V_m) + f(V_m, w))\eta = -\int_B M_i \nabla \phi_e \nabla \eta \quad . \qquad (9.6)$$

On fait de même pour l'équation elliptique du modèle :

$$-\int_B (M_i + M_e) \nabla \phi_e \nabla \eta = \int_B M_i \nabla V_m \nabla \eta \quad . \qquad (9.7)$$

9.3 Simulations en dimension deux

On considère que le domaine B est représenté par le cercle unité. Un maillage est créé grâce à une triangularisation du domaine (voir section 7.2.1). Puis, on utilise un schéma implicite en temps pour effectuer la résolution du modèle. Afin de vérifier la convergence, les paramètres de simulation sont fixées aux valeurs fournies dans la table 9.1. Une stimulation est ainsi générée sur un disque de rayon 0,1 pendant 10 ms, dans le but d'activer un potentiel d'action.

Paramètres	Valeurs
T_{final}	100 ms
Δt	0,7 ms
A_m	2000 cm^{-1}
C_m	1 $\mu F.cm^{-1}$
$[c_{i,x}\ ;\ c_{i,y}]$	1 $mS.cm{-}1$
$[c_{e,x}\ ;\ c_{e,y}]$	1 $mS.cm{-}1$
$[T_i; T_f]_{stim}$	[10;20]
Rayon de la stimulation	0,1
I_{app}	0,4 μA

Tableau 9.1 – Valeurs des paramètres de simulation du modèle bidomaine en dimension deux.

Suite aux observations faites dans la section 7.2.1, l'utilisation du gradient conjugué comme solveur est pertinente. Les résultats obtenus lors de l'utilisation d'un maillage de plus en plus fins sont présentés dans le tableau 9.1. On constate que la géométrie du front d'onde est circulaire ainsi qu'une propagation de l'onde moins rapide, lorsque le maillage devient fin. Cette constatation est confortée est par une analyse de la convergence.

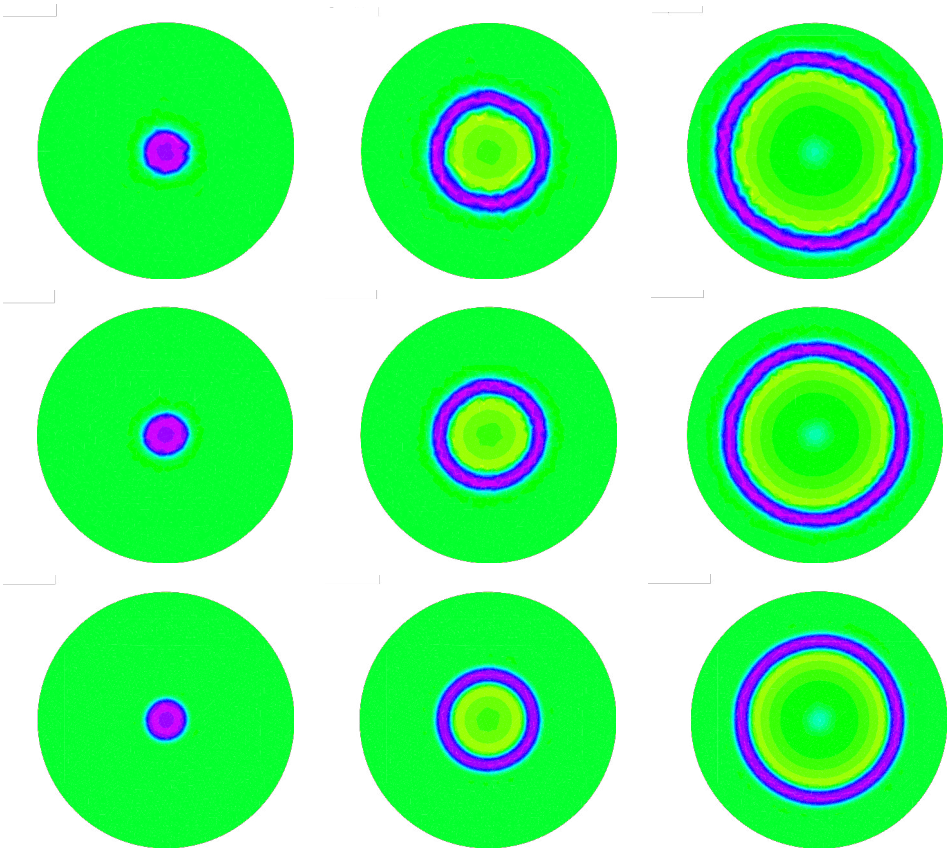

Figure 9.1 – Propagation du potentiel d'action au cours du temps (de gauche à droite) obtenus sur des maillages de 1766 triangles (en haut), de 2978 triangles (au milieu) et de 7064 triangles (en bas).

9.3.1 Convergence

On applique la technique introduite dans la section 7.2.2 afin de démontrer la convergence. On compare ainsi plusieurs caractéristiques telles que la moyenne $\bar{\tau}$ des temps précis τ où le maximum de chaque enregistrement est atteint, la différence $\Delta\tau$ entre les temps du premier et le dernier maximum enregistré, et l'écart-type σ_τ de ces différents temps. Ces différents résultats sont synthétisés dans le tableau 9.2.

La Figure 9.2 montre les enregistrements des différentes électrodes simulées et placées sur le cercle de rayon $0,5$. On constate ainsi que les différents signaux finissent par se juxtaposer, puisque l'écart-type tend vers 0, lorsque la taille du maillage augmente. L'impression de vitesse

de propagation qui se réduit petit à petit est confirmée par les temps moyens obtenus.

Nombre d'élèments	$\bar{\tau}$	$\Delta\tau$	σ_τ	Δr	Δx
446	50,75	9,1	2,28	0,089	0,042
1 766	52,27	3,5	0,909	0,033	0,021
3 924	58,94	3,5	0,929	0,029	0,014
7 064	61,74	1,4	0,491	0,011	0,010
15 508	65,29	0,7	0,309	0,005	0,007
43 312	67,51	0,7	0,347	0,005	0,004
85 306	68,60	0	0,152	0	0,003
141 282	69,21	0	0,006	0	0,0023
207 856	70	0	0	0	0,0019

Tableau 9.2 – Analyse des résultats du modèle bidomaine 2D couplé avec le modèle Fitzhugh-Nagumo

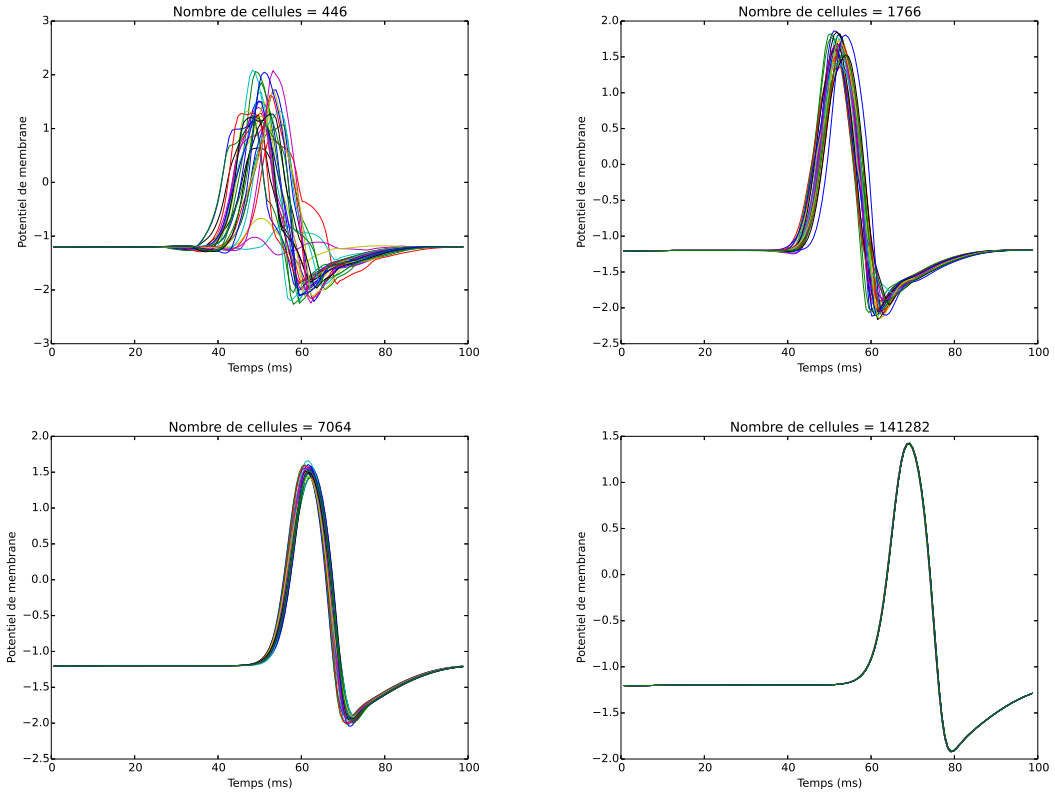

Figure 9.2 – Potentiel de membrane normalisé enregistré en différents points appartenant au même cercle de rayon 0,5 pour différentes tailles de maillage, lors de la simulation du modèle bidomaine en dimension deux.

9.3.2 Application de la méthode pararéel en MPI

Lorsque nous voulons utiliser un maillage relativement petit, comportant 15670 cellules, mais avec un pas de temps de l'ordre de 10^{-4}, la solution est obtenue en environ 3000 s, soit 50 min. On utilise la méthode pararéel développée en FreeFem, afin de l'appliquer au problème bidomaine. Cela permettra d'obtenir une solution de meilleure qualité, en diminuant le temps de résolution. Lors de l'utilisation de l'algorithme pararéel pour la simulation du modèle bidomaine, il est nécessaire de communiquer les valeurs du triptyque (V_m; ϕ_e; w) suite à l'achèvement des tâches de chaque processus. A nouveau, ce sont les tableaux contenant les coordonnées de la solution dans la base de fonctions propres. Pour éviter de surcharger les communications, on envoie séparément les trois variables. Le critère d'arrêt est basé sur la norme L^2 entre les solutions obtenues lors de deux itérations successives.

Lors des différents tests, on applique le protocole de stimulation suivant (tableau 9.3) lors des simulations séquentielle et parallèles. Tous les paramètres sont identiques à ceux du tableau 9.1, hormis le pas de temps grossier Δt plus petit.

Paramètres	Valeurs
T_{final}	100 ms
Δt	0,3125 ms
δt	$1,0 \times 10^{-2}$ ms
A_m	2000 cm^{-1}
C_m	1 $\mu F.cm^{-1}$
$[c_{i,x}\ ;\ c_{i,y}]$	1 $mS.cm-1$
$[c_{e,x}\ ;\ c_{e,y}]$	1 $mS.cm-1$
$[T_i; T_f]_{stim}$	[10;20]
Rayon de la stimulation	0,1
I_{app}	0,4 μA

Tableau 9.3 – Valeurs des paramètres de simulation du modèle bidomaine en dimension deux.

Les différents résultats obtenus sont présentés Figures 9.4 et 9.5. On constate que l'erreur diminue effectivement au fil des itérations jusqu'à atteindre au bout de sept itérations un ordre de grandeur de 10^{-4}. L'algorithme fonctionne donc très bien sur ce modèle en FreeFem, même si les résultats sont en deçà de ceux obtenus avec le modèle monodomaine.

En observant les accélérations (Figure 9.4), on observe un gain non négligeable lorsque le nombre de processus augmente, en restant notamment assez loin des valeurs de speed-up maximales théoriques. On remarquera que ces valeurs sont moins importantes que lors de l'étude du modèle monodomaine, où une utilisation de 32 processus permettait une diminution du temps de calcul d'un facteur de 4,8.

Néanmoins ce gain est non négligeable et permet d'obtenir des résultats plus rapidement, en faisant appel à des critères de simulations exigeants.

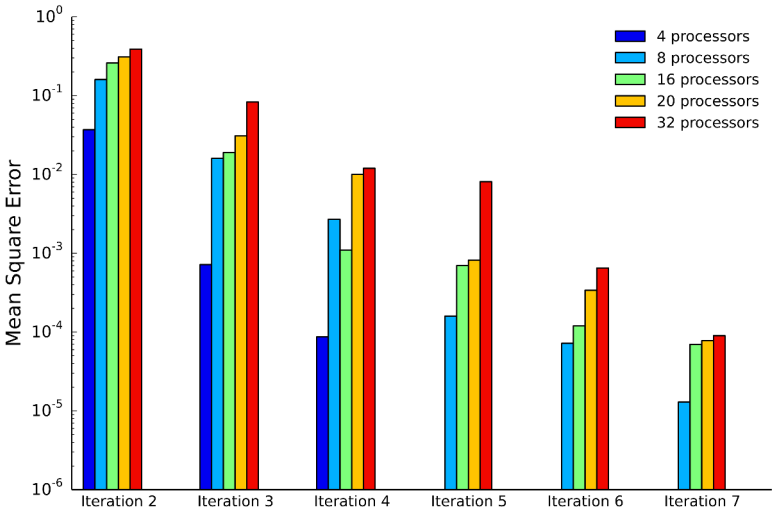

Figure 9.3 – Comparaison de l'erreur quadratique moyenne à chaque itération pour différents nombres de processus MPI.

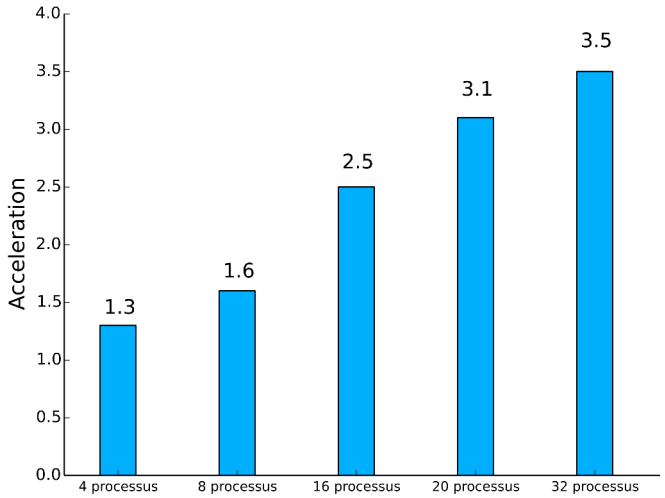

Figure 9.4 – Accélérations lors de l'utilisation de l'algorithme pararéel en FreeFem par rapport au temps de simulation séquentiel.

9.3.3 Application de l'algorithme pararéel couplé à la décomposition de domaine en MPI

On souhaite maintenant appliquer la méthode décrite dans la section 7.2.4 au modèle bidomaine. On découpe l'espace de manière analogue, afin de résoudre le problème sur chaque sous-espace en complément de l'utilisation de la méthode pararéel. Comme précédemment on suppose que le domaine de définition Ω est découpé en deux sous-espace Ω_1 et Ω_2. Puisqu'on suppose que N processus sont utilisés lors de l'utilisation de la méthode pararéel, on définie N autres processus permettant de s'occuper de la décomposition de domaine.

J'ai effectué une série de simulations, en appliquant le protocole donné dans la table 9.3. Les erreurs obtenues au cours des différentes itérations sont récapitulées dans la Figure 9.5. On constate que l'erreur est minorée par 10^{-3}, entraînant des résultats légèrement moins bons que ceux obtenus lors de l'utilisation de la méthode pararéel seule. La même remarque a déjà été émise lors de l'application de cette méthode au modèle monodomaine. Le plus gros de l'erreur est commis lors de lors de l'application de la méthode de décomposition de domaine. Néanmoins, on observe une diminution de l'erreur lors des différentes itérations, ce qui suit la logique des résultats obtenus jusque là.

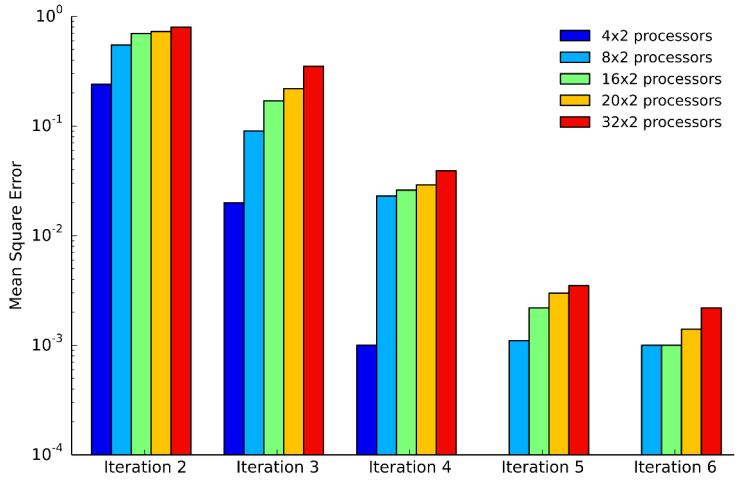

Figure 9.5 – Comparaison de l'erreur quadratique moyenne à chaque itération pour différents nombres de processus MPI, lors de l'application de la méthode pararéel couplé à la décomposition de domaine.

La Figure 9.6 permet de s'apercevoir que le gain de temps est anecdotique par rapport à l'utilisation de la méthode pararéel seule. Il faut noter que l'avantage de ce couplage ne réside pas uniquement dans le gain de temps, mais va permettre de définir des zones d'intérêts, où la taille du maillage sera plus importante.

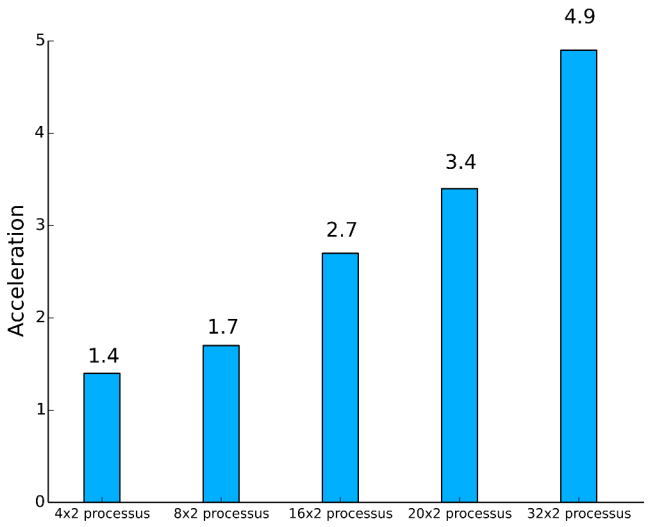

Figure 9.6 – Accélérations lors de l'utilisation de l'algorithme pararéel couplé à la décomposition de domaine, par rapport au temps de simulation séquentiel.

9.4 Simulations en dimension trois

Comme pour l'obtention des résultats numériques du modèle monodomaine en trois dimensions, nous utilisons les techniques de parallélisations vues dans les sections précédentes, afin de simuler convenablement le problème bidomaine sur une géométrie complexe en trois dimensions. Le maillage de la partie 7.3 est à nouveau utilisé, ainsi que les techniques pour fractionner le domaine.

Dans le modèle bidomaine, il convient de définir les différentes conductances intra et extra-cellulaire. Pour cela, on se base sur des valeurs suivantes (tableau 9.4) données dans Sadleir (2010) :

c_e	1,538
c_i	0,638

Tableau 9.4 – Valeurs des conductivités intra (c_i) et extra-cellulaire (c_e).

On suppose que les conductivités sont les mêmes dans toutes les directions. Il est impossible, à l'heure actuelle, de trouver des données précises sur les conductivités intra et extra-cellulaire en fonction des directions x, y et z. A l'aide de ces données, nous allons exécuter une simulation du modèle bidomaine sur notre géométrie complexe du cerveau. Une stimulation électrique est engendrée au niveau de l'hippocampe, puis on constate grâce à la Figure 9.7 une propagation du signal électrique dans toutes les directions.

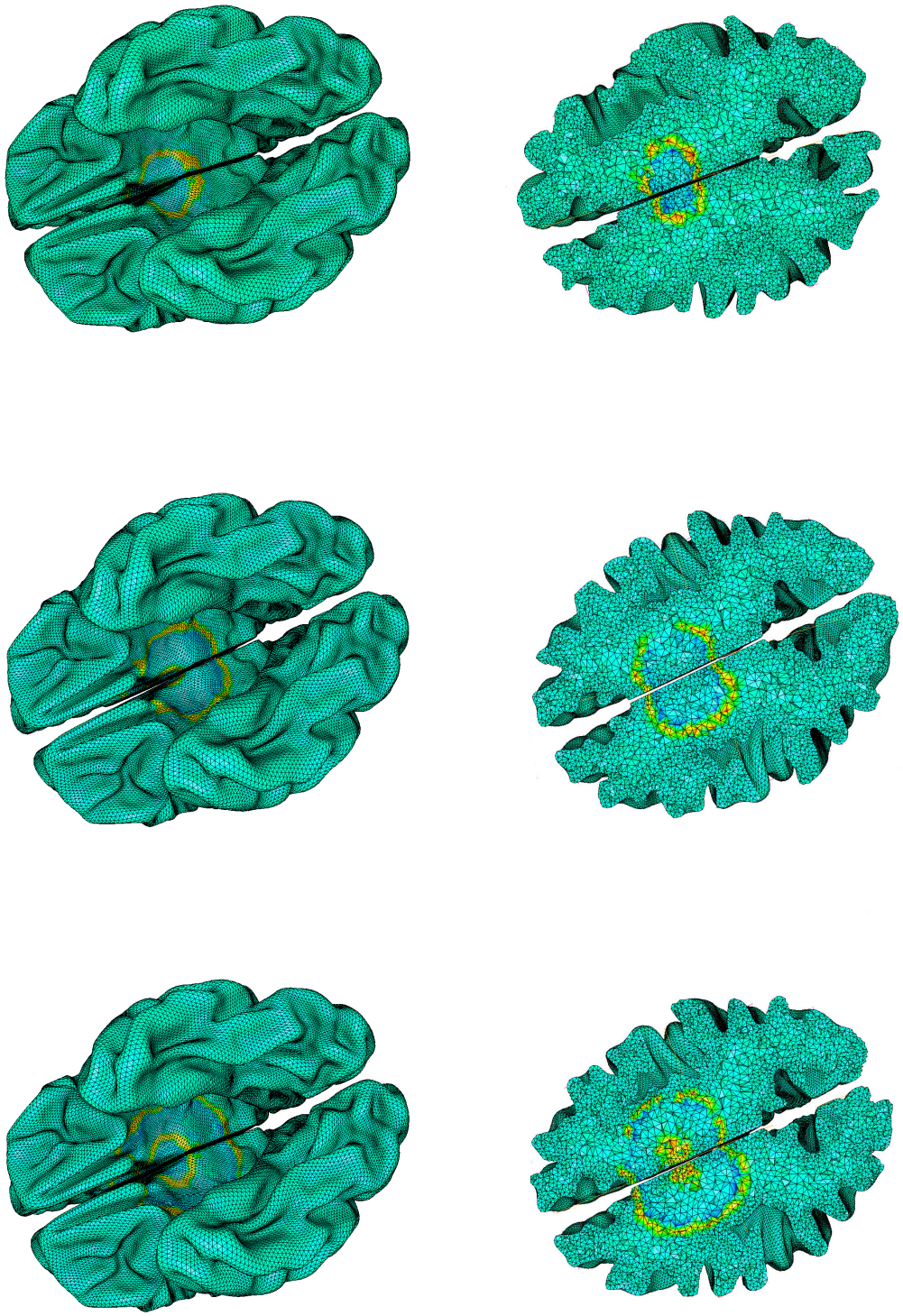

Figure 9.7 – Propagation du signal électrique, en surface (à gauche) et en profondeur (à droite) aux temps 45 ms, 65 ms et 85 ms (de haut en bas), après le déclenchement du premier potentiel d'action.

Le signal part effectivement au niveau de l'hippocampe, puis l'onde de dépolarisation se propage notamment dans les deux hémisphères du cerveau, suivi d'une vague de polarisation. Il serait bien entendu intéressant de distinguer plusieurs zones dans ce volume, avec des conductivités différentes, mais dans un souci de délai, il m'a été impossible de pouvoir effectuer ces simulations. Néanmoins, cela est une perspective pour la suite de ces travaux, c'est-à-dire développer une géométrie réaliste de cerveau avec différentes zones d'intérêts.

9.5 Conclusion

Dans ce chapitre, j'ai utilisé la méthode des éléments finis afin de simuler le modèle bidomaine sur des géométries plus complexes que celles permises par la méthode des différences finies. Pour réussir au mieux cet objectif dans un temps de résolution acceptable, tout en ne négligeant pas la qualité des résultats, les méthodes de parallélisation déjà introduite dans le chapitre 7 ont été utilisées. Dans cette étude, la décomposition de domaine couplé à l'algorithme pararéel n'apporte pas de gain de temps, il reste néanmoins intéressant lorsque des zones d'intérêt seront considérées sur la géométrie. Par exemple, il est nécessaire de distinguer les fonctionnements de différentes zones du cerveau, en utilisant des modèles de neurones différents. De plus les conductivités ne sont pas les même suivant les régions du cerveau (Sadleir, 2010). Tout ceci est à prendre en compte pour simuler de la manière la plus fidèle, l'activité électrique du cerveau. Une suite logique, bien que longue, est de pouvoir généraliser cette méthode en découpant le domaine en plus de deux sous-domaines.

Conclusion générale

Le travail de cette thèse, effectué dans une entreprise évoluant dans le domaine de la biosimulation du système nerveux central, avait pour objectif de réaliser une modélisation et une simulation multi-échelle de l'activité neuronale, à l'aide de techniques et technologies permettant d'effectuer des calculs en parallèle. J'ai ainsi présenté dans les premiers chapitres les différentes généralités permettant une meilleure compréhension du cadre d'étude. Une introduction au système nerveux central a été effectuée, puis j'ai détaillé les différentes caractéristiques et constituants d'un neurone. Dans ce même chapitre, j'ai également décrit son fonctionnement, en étudiant les différentes phases constituant un potentiel d'action. Après ces différentes explication d'un point de vue biologique, une description des différents modèles mathématiques décrivant l'activité d'un neurone a été faite. C'est ici notamment que le modèle de Hodgkin-Huxley, un modèle de neurone biologique à l'échelle cellulaire a été introduit. Ce modèle décrivant avec précision l'activité d'un neurone, il reste néanmoins coûteux, en terme de temps de calcul, dans sa simulation. C'est pour cela que j'ai introduit également divers modèle plus simple tel que le modèle de Fitzhugh-Nagumo. Suite à la présentation de ces différents modèles, j'ai présenté le modèle bidomaine, souvent utilisé dans la modélisation d'un tissu excitable comme le coeur. Ce modèle permet un regard à deux échelles sur le milieu étudié. Une simplification de ce modèle a également été présentée : le modèle monodomaine. Ces différents modèles étant très coûteux en temps de calcul lors de leur simulation, j'ai décidé d'implémenter des techniques de parallélisation permettant de découper les problèmes aussi bien sur l'échelle de temps que sur celle de l'espace. Dans un premier temps, j'ai introduit la méthode pararéel permettant d'intégrer un problème suivant l'échelle de temps. Une analyse de cette méthode, ainsi qu'une présentation des différentes variantes de cet algorithme ont été faites. Cela a permis de choisir la meilleure de ces méthodes pour la suite de mon étude. Dans un deuxième temps, j'ai présenté les méthodes de décomposition de domaine, permettant de résoudre des problèmes d'EDP suivant l'échelle spatiale. Pour implémenter ces différentes méthodes de manière optimale, j'ai dû faire appel à des technologies informatiques permettant de pouvoir réaliser des tâches en totales indépendances. Tout d'abord, j'ai décidé de profiter au mieux de l'aspect multi-cœur des CPU actuel, en utilisant la technologie MPI. Cela a permis l'utilisation de la méthode pararéel lors de la simulation des modèles de neurones, puis dans le développement d'une bibliothèque de calculs permettant la résolution d'EDO grâce à cette méthode. Cette implémentation a débouché sur une diminution du temps de calcul, en plus d'une précision accrue de la solution. Puis grâce à cette preuve de concept,

j'ai pu adapter cette technique pour la simulation du modèle monodomaine, puis bidomaine avec la méthode des différences finies. L'autre objectif de cette thèse a été de développer une méthode de résolution couplant deux techniques de parallélisations : la méthode pararéel et la décomposition de domaine en espace. Afin d'atteindre cette objectif j'ai décidé, dans un premier temps, d'utiliser la technologie des GPU avec l'environnement CUDA afin de tirer partie de la parallélisation massive permise grâce à ces outils. Pour cela, j'ai discrétisé les problèmes monodomaine puis bidomaine à l'aide de la méthode des différences finies. J'ai ainsi développer des outils permettant à divers utilisateurs de simuler ces différents modèles sur des géométries simples, en modifiant uniquement les paramètres des modèles. Ces outils permettent également d'envisager des simulations sur des intervalles de temps et d'espaces beaucoup plus grands que ceux utilisés lors d'une simulation séquentielle, car les gains apportés par cette technique sont importants. Suite aux développements de ces outils, j'ai décidé d'appliquer la méthode des éléments finis afin de discrétiser les problèmes monodomaine puis bidomaine. Cette méthode permet notamment de pouvoir envisager des simulations sur des géométries plus complexes que celles permises par la méthode des différences finies. A l'aide du logiciel FreeFem couplé à MPI, j'ai ainsi pu développer une méthode de résolution, couplant le pararéel avec une décomposition du domaine en deux parties. Le développement de tous ces outils a conduit à une simulation des différents modèles sur une géométrie réaliste du cerveau, avec des données physiologiques trouvées dans la littérature.

Dans un souci de temps, il n'a pas était possible de généraliser le décomposition de domaine à plus de deux sous-domaines, cela fait néanmoins partie des perspectives. Pour une simulation plus pertinente, il serait nécessaire de définir des zones d'intérêts au sein du cerveau, permettant ainsi de pouvoir appliquer un maillage plus ou moins fin. Une autre perspective concerne également l'extension des méthodes CUDA à la méthode des éléments finis. Pour cela, certaines bibliothèques sont d'ores et déjà disponible, tel que AmgX. Il est également envisageable de coupler les technologies MPI et CUDA, mais il convient de posséder plusieurs cartes graphique. Même si Rhenovia n'a pas la puissance informatique pour accomplir des tâches aussi compliquées, en terme de coût matériel, il serait utile de construire un partenariat avec des centres de calcul, comme le méso-centre de Strasbourg. L'avantage de ce méso-centre est sa labélisation NVIDIA permettant de bénéficier d'un accès privilégié aux nouveaux matériels et bibliothèques. Cela explique notamment le choix de l'environnement CUDA par rapport à OpenCL, dont les mises à jour ainsi que les bibliothèques externes sont abondantes. La preuve de concept étant faite, ceci aurait permis à Rhenovia dans le futur, de se doter d'une baie de GPU. Car le calcul GPU ne s'arrête pas à cette seule étude, un logiciel comme Neuron, utilisé au sein de Rhenovia, continue son développement afin que le calcul GPU soit possible. Cela permettrait également à l'entreprise d'ajouter cette compétence à son savoir faire. Il faut néanmoins garder à l'esprit que le calcul GPU n'est pas une solution à tout, car tous les calculs ne peuvent pas se paralléliser de façon optimale, afin d'obtenir un gain de temps important. Il convient donc de bien définir le problème et d'établir les différentes limites avant toute programmation fastidieuse en CUDA ou même en MPI.

Une autre perspective concerne la simulation de l'activité électrique du cerveau à l'aide du mo-

dèle monodomaine et/ou bidomaine. Le cerveau est un organe complexe, possédant plusieurs régions ayant leurs propres caractéristiques physiologiques qu'il faut bien entendu distinguer, afin d'obtenir la simulation la plus réaliste possible. J'ai pu constater au cours de ce travail de thèse que l'application de ces deux modèles au cas de l'activité neuronale n'est pas aussi aisée que dans le cas de l'activité cardiaque. Notamment, il est difficile de vouloir modéliser l'activité électrique du cerveau dans sa globalité avec ce seul modèle. La raison principale est que la propagation moyenne de l'information, à la différence du cœur, se propage tel une onde dans une petite région mais pas au niveau du cerveau total. De nombreuses études (Sporns *et al.*, 2005; Achard et Bullmore, 2007; Bassett *et al.*, 2006) de cartographie du cerveau ont notamment pu distinguer différentes régions d'intérêts connectées entre elles. Ainsi on se rapproche de la théorie des graphes puisque des poids de connexion sont associés à chaque liaison, afin de déterminer l'affinité entre les différentes régions. Toute la complexité de l'application du modèle monodomaine/bidomaine au cerveau entier réside dans la traduction de cette matrice de connectivité. De plus, il convient de trouver les différentes conductivités au sein des différentes couches du cerveau, mais aussi des différentes régions. Ce sont beaucoup de données qui ne sont pas évidentes à trouver dans la littérature, conjugué à un manque de temps lié à la fin d'activité de Rhenovia. Notamment un des manques de cette fin d'étude est de ne pas avoir eu le temps de confronter les données simulées avec des données réelles, car l'intérêt final était de pouvoir obtenir un modèle simulant l'activité électrique du cerveau lors de crise épileptique. Evidemment pour être plus complet, il aurait fallu pouvoir déterminer morphologiquement les différentes régions d'intérêts avec leurs caractéristiques ainsi que les différentes connexions entre elles. Tout ceci aurait permis de pouvoir obtenir des résultats concrets.

D'autres perspectives sont évidemment envisageable à l'aide de ce modèle, comme ceux introduits dans la partie 2.2.10. Il est possible de l'utiliser comme méthode alternative lors du calcul du Local Field Potential. Pour le moment, Rhenovia utilise une bibliothèque Python nommé LFPy (Klas H Pettersen, 2010) permettant de déterminer le potentiel extra-cellulaire provenant de plusieurs compartiments neuronaux. Il était donc également envisagé de faire appel à ce modèle afin d'utiliser une autre solution lors du calcul ce potentiel, et d'utiliser les techniques de parallélisation décrites pour diminuer grandement le calcul du Local Field Potential, surtout lorsqu'il devait être calculé sur de grands réseaux de neurones. Une autre perspective, déjà aperçue dans la littérature (R. Szmurlo, 2007), est d'appliquer le modèle monodomaine/bidomaine pour la propagation du signal électrique au sein d'un nerf.

De plus, il ne faut pas oublier que le modèle monodomaine/bidomaine est utilisé lors de la simulation de l'activité électrique du coeur, donc toutes les méthodes de parallélisation présentées dans cet ouvrage s'appliquent très bien à cette problématique. Je noterai encore qu'une perspective liée à ces méthodes est d'utiliser d'autres schémas de résolutions en partenariat avec des bibliothèques de calculs CUDA afin d'améliorer encore l'efficacité des outils présentés ici. Par exemple, il serait intéressant de travailler avec des méthodes implicites, plus complexes en terme de développement CUDA, que cela soit au niveau de l'algorithme mais également de la gestion de la mémoire, car il faut passer cette fois-ci par une formulation

matricielle du problème.

Pour finir, ces travaux ont permis d'apporter un nouveau regard sur les outils mathématiques et informatiques dans le domaine des neurosciences computationnelles, ainsi qu'une nouvelle expertise au sein des équipes de Rhenovia. Cette thèse a également permis de m'acclimater à de nouvelles technologies, ainsi qu'approfondir mes connaissances sur le thème très intéressant des neurosciences grâce à l'aide de mes collègues. Ces connaissances permettent notamment une plus grande flexibilité dans la poursuite de ces travaux, mais également dans la création de futurs projets.

Annexes

Partie V

Annexe 1 : Formulation Matricielle en dimension trois

Dans cette annexe, je vais détailler la construction de la matrice globale A, lors de la discrétisation avec la méthode des différences finies, du modèle monodomaine en dimension trois.

On considère les entiers N_x, N_y et N_z définissant le nombre de nœuds dans les directions x, y et z respectivement. Pour des questions de commodités, on suppose que ces nombres sont égaux, autrement dit : $N_x = N_y = N_z$.

On suppose que chaque élément constituant le maillage du domaine, sur lequel le problème est considéré, est un cube dont le volume est $\Delta x \times \Delta y \times \Delta z$, avec $\Delta x = \Delta y = \Delta z = h = 1 / N_x$.

Chaque maille est caractérisé par des coordonnées uniques dans l'espace : $(x_i; y_j; z_k) = (i\Delta x; j\Delta y; k\Delta z)$.

On rappel que le système discret du modèle monodomaine est donné par :

$$\begin{cases} U^{n+1} = U^n + \dfrac{1}{A_m C_m (1+\lambda)} A U^n - \dfrac{1}{C_m} f(U^n, W^n) \\ W^{n+1} = g(U^n, W^n) \quad , \end{cases} \tag{9.8}$$

où U, W sont les vecteurs contenant les valeurs $u^n_{i,j,k}$, $w^n_{i,j,k}$ respectivement.

On cherche à construire la matrice A correspondant à la discrétisation de $\nabla \cdot (M\nabla)$, où M est le tenseur de conductivités défini par la matrice :

$$M = \begin{pmatrix} c_x & 0 & 0 \\ 0 & c_y & 0 \\ 0 & 0 & c_z \end{pmatrix} \quad ,$$

Dans un premier temps, on rappel que :

$$\nabla \cdot (M_i \nabla u) = c_x \frac{\partial^2 u}{\partial x^2} + c_y \frac{\partial^2 u}{\partial y^2} + c_z \frac{\partial^2 u}{\partial z^2} \quad . \tag{9.9}$$

Grâce à un développement limité, on obtient une approximation de (9.9) :

$$\frac{-2(c_x + c_y + c_z)u_{i,j,k} + c_x(u_{i-1,j,k} + u_{i+1,j,k}) + c_y(u_{i,j-1,k} + u_{i,j+1,k}) + c_z(u_{i,j,k-1} + u_{i,j,k+1})}{h^2} \quad .$$

On peut ainsi écrire le système global $\nabla \cdot (M \nabla U)$ comme un problème linéaire de la forme AU. Il reste néanmoins à prendre en considération les conditions aux limites de Neumann conduisant aux relations suivantes :

$$u_{-1,j,k} = u_{0,j,k} \quad ; \quad u_{i,-1,k} = u_{i,0,k} \quad ; \quad u_{i,j,-1} = u_{i,j,0} \quad . \tag{9.10}$$

Afin que ces conditions soient intégrées correctement, il faut distinguer l'ensemble d'indices définissant le bord du domaine. Dans le cas où le domaine est défini par un cube, on dénombre 8 cas particuliers correspondants aux sommets du cube, ainsi que 6 autres ensembles correspondants aux surfaces du cubes.

Au final, on définit la matrice globale $A \in \mathbb{R}^{N_x^3 \times N_x^3}$ de la manière suivante :

$$A = \begin{pmatrix} B & J & 0 & 0 & \dots & 0 \\ J & C & J & 0 & \dots & 0 \\ & \ddots & \ddots & \ddots & & \\ 0 & \dots & 0 & J & C & J \\ 0 & \dots & 0 & 0 & J & B \end{pmatrix} \quad ,$$

où les matrices par blocs B, C et $J \in \mathbb{R}^{N_x^2 \times N_x^2}$ sont définies par :

$$B = \begin{pmatrix} E_1 & D & 0 & 0 & \dots & 0 \\ D & F_1 & D & 0 & \dots & 0 \\ & \ddots & \ddots & \ddots & & \\ 0 & \dots & 0 & D & F_1 & D \\ 0 & \dots & 0 & 0 & D & E_1 \end{pmatrix} \quad ; \quad C = \begin{pmatrix} E_2 & D & 0 & 0 & \dots & 0 \\ D & F_2 & D & 0 & \dots & 0 \\ & \ddots & \ddots & \ddots & & \\ 0 & \dots & 0 & D & F_2 & D \\ 0 & \dots & 0 & 0 & D & E_2 \end{pmatrix} ;$$

$$J = \begin{pmatrix} c_z & 0 & 0 & 0 & \dots & 0 \\ 0 & c_z & 0 & 0 & \dots & 0 \\ & \ddots & \ddots & \ddots & & \\ 0 & \dots & 0 & 0 & 0 & c_z \end{pmatrix} ,$$

avec les matrices E_1, E_2, F_1, F_2 et $D \in \mathbb{R}^{N_x \times N_x}$ suivantes :

$$E_1 = \begin{pmatrix} -(c_x+c_y+c_z) & c_x & 0 & 0 & \dots & 0 \\ c_x & -(2c_x+c_y+c_z) & c_x & 0 & \dots & 0 \\ & \ddots & \ddots & \ddots & & \\ 0 & \dots & 0 & c_x & -(2c_x+c_y+c_z) & c_x \\ 0 & \dots & 0 & 0 & c_x & -(c_x+c_y+c_z) \end{pmatrix} ;$$

$$E_2 = \begin{pmatrix} -(c_x+c_y+2c_z) & c_x & 0 & 0 & \dots & 0 \\ c_x & -(2c_x+c_y+2c_z) & c_x & 0 & \dots & 0 \\ & \ddots & \ddots & \ddots & & \\ 0 & \dots & 0 & c_x & -(2c_x+c_y+2c_z) & c_x \\ 0 & \dots & 0 & 0 & c_x & -(c_x+c_y+2c_z) \end{pmatrix} ;$$

$$F_1 = \begin{pmatrix} -(c_x+2c_y+c_z) & c_x & 0 & 0 & \dots & 0 \\ c_x & -2(c_x+c_y+c_z) & c_x & 0 & \dots & 0 \\ & \ddots & \ddots & \ddots & & \\ 0 & \dots & 0 & c_x & -2(c_x+c_y+c_z) & c_x \\ 0 & \dots & 0 & 0 & c_x & -(c_x+2c_y+c_z) \end{pmatrix} ;$$

$$F_2 = \begin{pmatrix} -(c_x+2c_y+2c_z) & c_x & 0 & 0 & \dots & 0 \\ c_x & -2(c_x+c_y+2c_z) & c_x & 0 & \dots & 0 \\ & \ddots & \ddots & \ddots & & \\ 0 & \dots & 0 & c_x & -2(c_x+c_y+2c_z) & c_x \\ 0 & \dots & 0 & 0 & c_x & -(c_x+2c_y+2c_z) \end{pmatrix} ;$$

$$D = \begin{pmatrix} c_y & 0 & 0 & 0 & \dots & 0 \\ 0 & c_y & 0 & 0 & \dots & 0 \\ & \ddots & \ddots & \ddots & & \\ 0 & \dots & 0 & 0 & 0 & c_y \end{pmatrix} \quad ,$$

Table des figures

Liste des Algorithmes

Codes

Bibliographie

(2006). *Computing the Electrical Activity in the Heart*, volume 1 de *Monographs in Computational Science and Engineering*. Springer Berlin Heidelberg, Berlin, Heidelberg.

ABBOTT, L. F. (1999). Lapicque's introduction of the integrate-and-fire model neuron (1907). *Brain Research Bulletin*, 50(5-6):303–304.

ACHARD, S. et BULLMORE, E. (2007). Efficiency and Cost of Economical Brain Functional Networks. *PLoS Computational Biology*, 3(2).

ALLERHEILIGEN, S. R. B. (2010). Next-generation model-based drug discovery and development : quantitative and systems pharmacology. *Clinical Pharmacology and Therapeutics*, 88(1):135–137.

AUBANEL, E. (2011). Scheduling of tasks in the parareal algorithm. *Parallel Computing*, 37(3):172–182.

BASSETT, D. S., MEYER-LINDENBERG, A., ACHARD, S., DUKE, T. et BULLMORE, E. (2006). Adaptive reconfiguration of fractal small-world human brain functional networks. *Proceedings of the National Academy of Sciences of the United States of America*, 103(51):19518–19523.

BASTIAN, P., HORTON, G. et BURMEISTER, J. (1991). Implementation of a parallel multigrid method for parabolic partial differential equations.

BEN-YISHAI, R., BAR-OR, R. L. et SOMPOLINSKY, H. (1995). Theory of orientation tuning in visual cortex. *Proceedings of the National Academy of Sciences of the United States of America*, 92(9):3844–3848.

BENNANI, Y. L. (2011). Drug discovery in the next decade : innovation needed ASAP. *Drug Discovery Today*, 16(17-18):779–792.

BEURLE, R. L. (1956). Properties of a Mass of Cells Capable of Regenerating Pulses. *Philosophical Transactions of the Royal Society of London B : Biological Sciences*, 240(669):55–94.

BORSUK, M. (2010). *Transmission Problems for Elliptic Second-Order Equations in Non-Smooth Domains*. Frontiers in Mathematics. Springer Basel, Basel.

BRADY, L. S., WINSKY, L., GOODMAN, W., OLIVERI, M. E. et STOVER, E. (2008). NIMH Initiatives to Facilitate Collaborations Among Industry, Academia, and Government for the Discovery and Clinical Testing of Novel Models and Drugs for Psychiatric Disorders. *Neuropsychopharmacology*, 34(1):229–243.

BUHRY, L. (2010). *Estimation de paramètres de modèles de neurones biologiques sur une plate-forme de SNN (Spiking Neural Network) implantés "in silico"*. phdthesis, Université Sciences et Technologies - Bordeaux I.

BUTCHER, J. (1996). A history of runge-kutta methods. *Applied Numerical Mathematics - APPL NUMER MATH*.

BÉDARD, C., KRÖGER, H. et DESTEXHE, A. (2004). Modeling Extracellular Field Potentials and the Frequency-Filtering Properties of Extracellular Space. *Biophysical Journal*, 86(3):1829–1842.

CHAN, T. F. et MATHEW, T. P. (1994). Domain decomposition algorithms. *Acta Numerica*, 3:61–143.

CHARBEL FARHAT, M. C. (2003). Time-decomposed parallel time-integrators : Theory and feasibility studies for fluid, structure, and fluid-structure applications. *International Journal for Numerical Methods in Engineering*, 58(9):1397 – 1434.

CIARLET, P. (2007). *Introduction à l'analyse numérique matricielle et à l'optimisation - Dunod*. Dunod, 5 édition.

COURTEMANCHE, R., FUJII, N. et GRAYBIEL, A. M. (2003). Synchronous, focally modulated beta-band oscillations characterize local field potential activity in the striatum of awake behaving monkeys. *The Journal of Neuroscience : The Official Journal of the Society for Neuroscience*, 23(37):11741–11752.

COUTURIER, R. (2013). *Designing Scientific Applications on GPUs*.

DESTEXHE, A. et HUGUENARD, J. R. (2000). Nonlinear Thermodynamic Models of Voltage-Dependent Currents. *Journal of Computational Neuroscience*, 9(3):259–270.

DOSTROVSKY, J. et BERGMAN, H. (2004). Oscillatory activity in the basal ganglia—relationship to normal physiology and pathophysiology. *Brain*, 127(4):721–722.

DOWLING, J. (2005). Artificial human vision. *Expert Review of Medical Devices*, 2(1):73–85.

FAUGERAS, O., TOUBOUL, J. et CESSAC, B. (2009). A Constructive Mean-Field Analysis of Multi-Population Neural Networks with Random Synaptic Weights and Stochastic Inputs. *Frontiers in Computational Neuroscience*, 3.

FIFE, P. C. et MCLEOD, J. B. (1977). The approach of solutions of nonlinear diffusion equations to travelling front solutions. *Archive for Rational Mechanics and Analysis*, 65(4):335–361.

FITZHUGH, R. (1955). Mathematical models of threshold phenomena in the nerve membrane. *The bulletin of mathematical biophysics*, 17(4):257–278.

FITZHUGH, R. (1961). Impulses and Physiological States in Theoretical Models of Nerve Membrane. *Biophysical Journal*, 1(6):445–466.

FRANZONE, P. C. et SAVARÉ, G. (2002). Degenerate Evolution Systems Modeling the Cardiac Electric Field at Micro- and Macroscopic Level. *In* LORENZI, A. et RUF, B., éditeurs : *Evolution Equations, Semigroups and Functional Analysis*, numéro 50 de Progress in Nonlinear Differential Equations and Their Applications, pages 49–78. Birkhäuser Basel.

FUKAI, H., NOMURA, T., DOI, S. et SATO, S. (2000). Hopf bifurcations in multiple-parameter space of the hodgkin-huxley equations II. Singularity theoretic approach and highly degenerate bifurcations. *Biological Cybernetics*, 82(3):223–229.

GALLOPOULOS, E., SIMONCINI, V., SZETO, T., TONG, C. H., CHAN, T. F. et CHAN, T. F. (1994). A Quasi-Minimal Residual Variant Of The Bi-Cgstab Algorithm For Nonsymmetric Systems. *SIAM J. Sci. Comput*, 15:338–347.

GANDER, M. J. et VANDEWALLE, S. (2007). Analysis of the parareal time-parallel time-integration method.

GESELOWITZ, D. B. et III, W. T. M. (1983). A bidomain model for anisotropic cardiac muscle. *Annals of Biomedical Engineering*, 11(3-4):191–206.

GRABOWSKI, H. (2004). Are the economics of pharmaceutical research and development changing ? : productivity, patents and political pressures. *PharmacoEconomics*, 22(2 Suppl 2):15–24.

GRASELA, T. H. et SLUSSER, R. (2010). Improving productivity with model-based drug development : an enterprise perspective. *Clinical Pharmacology and Therapeutics*, 88(2):263–268.

GREGET, R. (2011). *Modélisation et simulation informatique de la transmission nerveuse*. Thèse de doctorat, Mulhouse.

GUCKENHEIMER, J. et LABOURIAU, J. S. (1993). Bifurcation of the Hodgkin and Huxley equations : A new twist. *Bulletin of Mathematical Biology*, 55(5):937–952.

HACKBUSCH, W. (2013). *Multi-Grid Methods and Applications*. Springer Science & Business Media.

HAGMANN, P., CAMMOUN, L., GIGANDET, X., MEULI, R., HONEY, C. J., WEDEEN, V. J. et SPORNS, O. (2008). Mapping the Structural Core of Human Cerebral Cortex. *PLoS Biology*, 6(7).

HAIRER, E. (1993). *Solving Ordinary Differential Equations I*, volume 8 de *Springer Series in Computational Mathematics*. Springer Berlin Heidelberg, Berlin, Heidelberg.

HAIRER, E. et WANNER, G. (1996). *Solving Ordinary Differential Equations II*, volume 14 de *Springer Series in Computational Mathematics*. Springer Berlin Heidelberg, Berlin, Heidelberg.

HASBOUN (2004). Polycopié de Neuroanatomie - Morphologie.

HASSARD, B. (1978). Bifurcation of periodic solutions of Hodgkin-Huxley model for the squid giant axon. *Journal of Theoretical Biology*, 71(3):401–420.

HASSARD, B. et SHIAU, L. J. (1996). A special point of Z2-codimension three Hopf bifurcation in the Hodgkin-Huxley model. *Applied Mathematics Letters*, 9(3):31–34.

HASTINGS, S. P. (1976). On the existence of homoclinic and periodic orbits for the Fitzhugh-Nagumo equations. *Quarterly Journal of Mathematics*, 27(1).

HAUEISEN, J., TUCH, D. S., RAMON, C., SCHIMPF, P. H., WEDEEN, V. J., GEORGE, J. S. et BELLIVEAU, J. W. (2002). The influence of brain tissue anisotropy on human EEG and MEG. *NeuroImage*, 15(1):159–166.

HE, B., éditeur (2013). *Neural Engineering*. Springer US, Boston, MA.

HECHT, F. (2012). New development in freefem++. *J. Numer. Math.*, 20(3-4):251–265.

HENRIQUEZ, C. S. (1993). Simulating the electrical behavior of cardiac tissue using the bidomain model. *Critical Reviews in Biomedical Engineering*, 21(1):1–77.

HODGKIN, A. L. et HUXLEY, A. F. (1952a). The components of membrane conductance in the giant axon of Loligo. *The Journal of Physiology*, 116(4):473–496.

HODGKIN, A. L. et HUXLEY, A. F. (1952b). Currents carried by sodium and potassium ions through the membrane of the giant axon of Loligo. *The Journal of Physiology*, 116(4):449–472.

HODGKIN, A. L. et HUXLEY, A. F. (1952c). The dual effect of membrane potential on sodium conductance in the giant axon of Loligo. *The Journal of Physiology*, 116(4):497–506.

HODGKIN, A. L. et HUXLEY, A. F. (1952d). Propagation of electrical signals along giant nerve fibers. *Proceedings of the Royal Society of London. Series B, Biological Sciences*, 140(899):177–183.

HODGKIN, A. L. et HUXLEY, A. F. (1952e). A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of Physiology*, 117(4):500–544.

IZHIKEVICH, E. M. (2004). Which model to use for cortical spiking neurons ? *IEEE transactions on neural networks / a publication of the IEEE Neural Networks Council*, 15(5):1063–1070.

IZHIKEVICH, E. M., GALLY, J. A. et EDELMAN, G. M. (2004). Spike-timing Dynamics of Neuronal Groups. *Cerebral Cortex*, 14(8):933–944.

JOAKIM SUNDNES, G. T. L. (2001). Efficient solution of ordinary differential equations modeling electrical activity in cardiac cells. *Mathematical Biosciences*, 172(2):55–72.

KAITIN, K. I. (2010). Deconstructing the drug development process : the new face of innovation. *Clinical Pharmacology and Therapeutics*, 87(3):356–361.

KANDEL (2012). *Principles of Neural Science, Fifth Edition*. McGraw-Hill Education / Medical, New York, 5th edition édition.

KARIMI, K., DICKSON, N. G. et HAMZE, F. (2010). A Performance Comparison of CUDA and OpenCL. *arXiv :1005.2581 [physics]*. arXiv : 1005.2581.

KARYPIS, G. et KUMAR, V. (1995). METIS – Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0. Rapport technique.

KARYPIS, G. et KUMAR, V. (1998). A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392.

KIRK, D. B. et HWU, W.-m. W. (2010). *Programming Massively Parallel Processors : A Hands-on Approach*. Morgan Kaufmann, Burlington, MA.

KLAS H PETTERSEN, H. L. (2010). Extracellular spikes and current-source density.

KRISHNAMOORTHI, S., SARKAR, M. et KLUG, W. S. (2013). Numerical quadrature and operator splitting in finite element methods for cardiac electrophysiology. *International Journal for Numerical Methods in Biomedical Engineering*, 29(11):1243–1266.

LAING, C. R. et TROY, W. C. (2003). *PDE Methods for Nonlocal Models*.

LANCASTER, J. L., TORDESILLAS-GUTIÉRREZ, D., MARTINEZ, M., SALINAS, F., EVANS, A., ZILLES, K., MAZZIOTTA, J. C. et FOX, P. T. (2007). Bias between MNI and Talairach coordinates analyzed using the ICBM-152 brain template. *Human Brain Mapping*, 28(11):1194–1205.

LIONS (2001). Introduction à l'analyse numérique matricielle et à l'optimisation - Dunod.

LIU, C., WU, H., FENG, L. et YANG, A. (2011). Parallel Fourth-Order Runge-Kutta Method to Solve Differential Equations. *In* LIU, B. et CHAI, C., éditeurs : *Information Computing and Applications*, numéro 7030 de Lecture Notes in Computer Science, pages 192–199. Springer Berlin Heidelberg.

LUCQUIN, B. (1998). *Introduction to Scientific Computing*. Wiley-Blackwell, Chichester ; New York.

MACKAY, D. J. C. (2003). *Information Theory, Inference and Learning Algorithms*. Cambridge University Press, Cambridge, UK ; New York.

MALSBURG, C. V. D. (1986). Frank Rosenblatt : Principles of Neurodynamics : Perceptrons and the Theory of Brain Mechanisms. *In* PALM, D. G. et AERTSEN, D. A., éditeurs : *Brain Theory*, pages 245–248. Springer Berlin Heidelberg. DOI : 10.1007/978-3-642-70911-1_20.

MANET, V. (2015). La Méthode des Éléments Finis : vulgarisation des aspects mathématiques, illustration des capacités de la méthode. page 422.

MUNOS, B. (2010). Can open-source drug R&D repower pharmaceutical innovation ? *Clinical Pharmacology and Therapeutics*, 87(5):534–536.

NAGUMO, J., ARIMOTO, S. et YOSHIZAWA, S. (1962). An Active Pulse Transmission Line Simulating Nerve Axon. *Proceedings of the IRE*, 50(10):2061–2070.

NEU, J. C. et KRASSOWSKA, W. (1993). Homogenization of syncytial tissues. *Critical Reviews in Biomedical Engineering*, 21(2):137–199.

NUNEZ, P. L. et SRINIVASAN, R. (2006). *Electric Fields of the Brain : The neurophysics of EEG*. OUP USA, second edition édition.

NVIDIA (2015). CUDA NVIDIA toolkit documentation. http://docs.nvidia.com/cuda/index.html#. Accessed : 2015-07-27.

OLVER, P. J. (2014). *Introduction to Partial Differential Equations*. Undergraduate Texts in Mathematics. Springer International Publishing, Cham.

PACHECO, P. S. (1996). *Parallel Programming with MPI*. Morgan Kaufmann, San Francisco, Calif.

PIERO COLLI FRANZONE, G. S. (2002). Degenerate Evolution Systems Modeling the Cardiac Electric Field at Micro- and Macroscopic Level.

PIERRE, C. (2005). *Modélisation et simulation de l'activité électrique du coeur dans le thorax, analyse numérique et méthodes de volumes finis*. phdthesis, Université de Nantes.

PRESS, W., TEUKOLSKY, S. et VETTERLING, W. (2007). *Numerical Recipes 3rd Edition : The Art of Scientific Computing*. Cambridge University Press, Cambridge, UK ; New York, 3 edition édition.

PURVES et AUGUSTINE (2001). *Neuroscience*. Sinauer Associates, 2nd édition.

QUINN, M. (2003). *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Science/Engineering/Math, Dubuque, Iowa, 1 edition édition.

R. SZMURLO, J. S. (2007). Multiscale Finite Element Model of the Electrically Active Neural Tissue.

ROBINSON, D. K. et SETHURAMAN, N. (2010). How innovative technology is moving biologics into the 21st century. *Clinical Pharmacology and Therapeutics*, 87(3):261–263.

ROSENBLATT, F. (1958). The perceptron : A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408.

SADLEIR, R. (2010). A Bidomain Model for Neural Tissue. *International Journal of Bioelectromagnetism*, 12(1):2–6.

SAHA, S., BHATTACHARYYA, S. S. et WOLF, W. (2006). A communication interface for multiprocessor signal processing systems. *In In Proc. of the IEEE Wkshp. on Embedded Systems for Real-Time Multimedia*, pages 127–132.

SANDERS, J. et KANDROT, E. (2010). *CUDA by Example : An Introduction to General-Purpose GPU Programming*. Addison Wesley, Upper Saddle River, NJ, 1 édition.

SANDERS, J. et KANDROT, E. (2011). *CUDA par l'exemple : Une introduction à la programmation parallèle de GPU*. Pearson, Paris.

SARMIS, M. (2013). *Étude de l'activité neuronale : optimisation du temps de simulation et stabilité des modèles*. Thèse de doctorat.

SCHMITT, O. H. (1969). Biological Information Processing Using the Concept of Interpenetrating Domains. *In* LEIBOVIC, K. N., éditeur : *Information Processing in The Nervous System*, pages 325–331. Springer Berlin Heidelberg.

SI, H. (2015). TetGen, a Delaunay-Based Quality Tetrahedral Mesh Generator. *ACM Transactions on Mathematical Software*, 41(2):1–36.

SKEEL, R. (1982). A Theoretical Framework for Proving Accuracy Results for Deferred Corrections. *SIAM Journal on Numerical Analysis*, 19(1):171–196.

SMITH, B., BJORSTAD, P. et GROPP, W. (2004). *Domain Decomposition : Parallel Multilevel Methods for Elliptic Partial Differential Equations*. Cambridge University Press.

SMOLLER, J. (1994). *Shock Waves and Reaction—Diffusion Equations*, volume 258 de *Grundlehren der mathematischen Wissenschaften*. Springer New York, New York, NY.

SNIR, M. (1998). *MPI : The Complete Reference*. Second edition édition.

SONNEVELD, P. (1989). CGS, A Fast Lanczos-Type Solver for Nonsymmetric Linear systems. *SIAM Journal on Scientific and Statistical Computing*, 10(1):36–52.

SPORNS, O., TONONI, G. et KÖTTER, R. (2005). The Human Connectome : A Structural Description of the Human Brain. *PLoS Computational Biology*, 1(4).

STAFF, G. A. et RØNQUIST, E. M. (2005). Stability of the Parareal Algorithm. *In* BARTH, T. J., GRIEBEL, M., KEYES, D. E., NIEMINEN, R. M., ROOSE, D., SCHLICK, T., KORNHUBER, R., HOPPE, R., PÉRIAUX, J., PIRONNEAU, O., WIDLUND, O. et XU, J., éditeurs : *Domain Decomposition Methods in Science and Engineering*, numéro 40 de Lecture Notes in Computational Science and Engineering, pages 449–456. Springer Berlin Heidelberg. DOI : 10.1007/3-540-26825-1_46.

SUNDNES, J., LINES, G. T. et TVEITO, A. (2005). An operator splitting method for solving the bidomain equations coupled to a volume conductor model for the torso. *Mathematical Biosciences*, 194(2):233–248.

SZMURLO, R., STARZYNSKI, J., SAWICKI, B. et WINCENCIAK, S. (2007). Multiscale Finite Element Model of the Electrically Active Neural Tissue. *In EUROCON, 2007. The International Conference on #34 ;Computer as a Tool #34 ;*, pages 2343–2348.

TALLEC, P. L. (1994). Domain decomposition methods in computational mechanics. *Computational Mechanics Advances*, 1(2).

TAYLOR, J. G. (1999). Neural 'bubble' dynamics in two dimensions : foundations. *Biological Cybernetics*, 80(6):393–409.

TORABI ZIARATGAHI, S., MARSH, M., SUNDNES, J. et SPITERI, R. (2014). Stable time integration suppresses unphysical oscillations in the bidomain model. *Frontiers in Physics*, 2:40.

TSUCHIYAMA, R., NAKAMURA, T., IIZUKA, T., ASAHARA, A., MIKI, S. et TAGAWA, S. (2010). *The OpenCL Programming Book*. Fixstars Corporation, 1 edition édition.

TSUMOTO, K., KITAJIMA, H., YOSHINAGA, T., AIHARA, K. et KAWAKAMI, H. (2006). Bifurcations in Morris–Lecar neuron model. *Neurocomputing*, 69(4–6):293–316.

TUNG, L. (1978). *A bi-domain model for describing ischemic myocardial d-c potentials*. Thesis, Massachusetts Institute of Technology.

WHITELEY, J. (2006). An Efficient Numerical Technique for the Solution of the Monodomain and Bidomain Equations. *IEEE Transactions on Biomedical Engineering*, 53(11):2139–2147.

WILSON, H. R. et COWAN, J. D. (1973). A mathematical theory of the functional dynamics of cortical and thalamic nervous tissue. *Kybernetik*, 13(2):55–80.