Support d'exécution à base de tâches et programmation de haut niveau pour la simulation par éléments finis

Abdelbarie El Metni

Rapport de stage d'application

Support d'exécution à base de tâches et programmation de haut niveau pour la simulation par éléments finis

---

EL METNI Abdelbarie

Responsable de stage :

AUMAGE Olivier

Tuteur école :

BARRAL Nicolas

2023/2024

Sommaire

1. Introduction

1.1. Cadre de stage

Dans le cadre de mon stage, j'ai l'opportunité de travailler au sein de l'équipe STORM [1] à l'Inria Bordeaux, en collaboration avec le département Calcul Haute Performance (HPC) du laboratoire SIMULA en Norvège, dans le cadre du projet collaboratif Maelstrom.

L'équipe STORM, est un projet commun entre l'Inria, le CNRS, l'Université de Bordeaux et Bordeaux-INP, se concentre sur le développement de méthodologies et d'outils pour optimiser les calculs sur des architectures de calcul haute performance, en développant des supports d'exécution ainsi que des outils pour l'analyse de performance et l'optimisation énergétique.

Le projet Maelstrom, qui a débuté en 2022, a pour objectif d'intégrer des techniques avancées d'optimisation au sein de l'environnement de programmation de haut niveau FEniCS, utilisé pour résoudre numériquement les équations aux dérivées partielles avec la méthode des éléments finis.

1.2. Contexte

Les logiciels de simulation scientifique jouent un rôle crucial dans de nombreux domaines de la recherche et de l'industrie, permettant de modéliser et de résoudre des problèmes complexes qui ne pourraient pas être abordés expérimentalement. Ces logiciels sont particulièrement utilisés pour résoudre des équations aux dérivées partielles et simuler des phénomènes physiques. Parmi ces logiciels, on trouve FEniCS, une plateforme de calcul open-source permettant de résoudre des EDP [2] en utilisant la méthode des éléments finis. Elle transforme des modèles scientifiques complexes en un code de haut niveau via des interfaces Python ou C++.

L'évolution rapide des technologies de calcul et la demande croissante pour des simulations plus précises et plus rapides ont conduit au développement d'une bibliothèque de programmation par tâches appelée StarPU.

StarPU est un support d'exécution à base de tâches, développé principalement par l'équipe STORM d'Inria, et conçu pour optimiser les performances des applications sur des architectures hétérogènes combinant des CPU et des GPU. StarPU offre une gestion dynamique des tâches et un ordonnancement adaptable, permettant d'exploiter efficacement les ressources de calcul disponibles. Ce système permet aux développeurs de se concentrer sur les aspects algorithmiques de leurs applications sans se soucier des détails de bas niveau liés à la gestion des ressources.

1. Static Optimizations and Runtime Methods
2. Partial Differential Equation

1.3. Sujet de stage

L'amélioration des logiciels de simulations numériques est un enjeu majeur dans le domaine du calcul scientifique. Afin de répondre à cette problématique, mon stage porte sur l'adaptation de l'environnement FEniCS au-dessus de StarPU, en se concentrant sur la partie assemblage de la méthode des éléments finis (section 2), tout en introduisant certaines stratégies de parallélisme à base de tâches avec StarPU.

Lors d'une visite au département HPC de SIMULA en Norvège, nous avons discuté de la possibilité d'intégrer StarPU avec un logiciel de simulation de l'électrophysiologie à l'échelle cellulaire, un projet sur lequel l'équipe travaille. Cette collaboration a conduit à une nouvelle phase de mon stage, où les derniers mois ont été consacrés à l'exploration et au développement de cette intégration.

1.4. Objectifs

L'objectif de l'adaptation des logiciels de simulation numérique avec le support exécutif StarPU est d'améliorer leurs performances, leur efficacité et leur flexibilité. StarPU permet un ordonnancement dynamique des tâches et une gestion avancée des transferts de données en exploitant les ressources de calcul hétérogènes disponibles sur une configuration matérielle. Cela optimise l'utilisation des ressources, minimise les temps d'exécution et améliore la scalabilité des simulations. Grâce à cette approche, les logiciels peuvent tirer parti de la puissance de calcul des CPU et GPU, et adapter leur comportement en temps réel pour maximiser leurs performances.

1.5. Démarche suivie

Le stage a commencé par une phase de recherche sur l'environnement FEniCS 3 et la bibliothèque StarPU 4. L'objectif était de comprendre les mécanismes internes de l'assemblage de la méthode des éléments finis 5.1 et d'identifier les points clés pour intégrer StarPU afin d'améliorer le parallélisme et l'efficacité. Une adaptation antérieure avait déjà été réalisée, mais elle s'appuyait sur une ancienne version de FEniCS et se limitait à un code benchmark simplifié [1], ce qui réduisait la complexité de l'intégration avec le logiciel complet FEniCS.

En se basant sur ces développements antérieures, nous avons entrepris l'adaptation de FEniCS au-dessus de StarPU, en mettant en œuvre des stratégies de parallélisme à base de tâches. Cela a impliqué le développement de différentes méthodes de parallélisation, telles que l'utilisation de mutex 5.3, l'algorithme row-wise 5.4, et la stratégie de coloration de graphes 5.5, afin de minimiser les conflits d'accès concurrents et d'optimiser l'utilisation des ressources de calcul.

Enfin, l'intégration de StarPU dans un logiciel de simulation de l'électrophysiologie cellulaire a été explorée 6.

2. Méthode des éléments finis

La méthode des éléments finis est employée en analyse numérique pour résoudre numériquement les équations aux dérivées partielles. Elle est largement utilisée dans des domaines tels que la mécanique des structures, la mécanique des fluides, la thermodynamique et l'électromagnétisme, avec des applications concrètes comme la simulation de la dynamique des fluides et la modélisation de la diffusion de la chaleur.

La méthode des éléments finis (FEM) consiste à subdiviser un domaine complexe en de nombreux sous-domaines plus simples, appelés éléments finis, tels que des triangles ou des quadrilatères en 2D, et des tétraèdres ou des hexaèdres en 3D. Ces éléments sont connectés par des points nommés nœuds. Des fonctions de base, définies sur chaque élément, sont utilisées pour approximer la solution sur l'ensemble du domaine.

La mise en œuvre de la méthode des éléments finis (FEM) se déroule en plusieurs étapes clés, voici une description des principales étapes :

- **Discrétisation du domaine** : consiste à diviser le domaine d'étude en un maillage constitué de sous-domaines.
- **Formulation faible** : permet de reformuler les équations aux dérivées partielles (EDP) du problème en une forme intégrale.
- **Assemblage du système** : Une fois la formulation faible obtenue, le système d'équations linéaires ou non linéaires correspondant est assemblé. Cela implique de combiner les contributions de chaque élément fini pour former un système global qui représente le problème sur l'ensemble du domaine.
- **Application des conditions aux limites** : Les conditions aux limites, qui décrivent le comportement de la solution sur les bords du domaine, sont appliquées au système d'équations.
- **Résolution du système** : Le système d'équations résultant est ensuite résolu à l'aide de solveurs numériques

3. L'environnement FEniCS

FEniCS est un logiciel open source conçu pour résoudre les équations aux dérivées partielles avec la méthode des éléments finis [2]. Il traduit un modèle scientifique en un code de haut

niveau via des interfaces Python ou C++, permettant ainsi à des personnes non spécialistes en informatique de développer rapidement des codes de simulation efficaces. La dernière version de FEniCS, FEniCSx, comprend plusieurs composants :

- **DOLFINx** : un environnement de calcul haute performance permettant de gérer le maillage, d'assembler les éléments finis dans une matrice et comprenant également une interface vers les solveurs d'algèbre linéaire.
- **FFCx** : un compilateur qui traduit une formulation variationnelle implémentée avec UFL en un code C de bas niveau.
- **UFL** : permet de déclarer la formulation variationnelle avec une syntaxe de haut niveau proche de la notation mathématique.
- **Basix** : permet la génération des fonctions de base des éléments finis.

```
# Creer un maillage triangulaire du domaine [0, 1] x [0, 1]
mesh = RectangleMesh(MPI.COMM_WORLD, [np.array([0, 0, 0]), np.array([1, 1, 0])], [32, 32])

# Definir un espace de fonctions de Lagrange de degre 1
V = FunctionSpace(mesh, ("Lagrange", 1))

# Definir la condition aux limites de Dirichlet u = 0 sur les bords x = 0 et x = 1
u0 = Function(V)
u0.vector.set(0.0)

# Localiser les degres de liberte sur le bord pour appliquer la condition de Dirichlet
facets = locate_entities_boundary(mesh, 1, lambda x: np.isclose(x[0], [0, 1]))
bc = DirichletBC(u0, locate_dofs_topological(V, 1, facets))

# Definir les fonctions u et v
u = TrialFunction(V)
v = TestFunction(V)

# Definir la fonction source f et la condition de Neumann g
x = SpatialCoordinate(mesh)
f = 10 * exp(-((x[0] - 0.5)**2 + (x[1] - 0.5)**2) / 0.02)
g = sin(5 * x[0])

# Definir les formes bilineaire a(u, v) et lineaire L(v)
a = inner(grad(u), grad(v)) * dx
L = inner(f, v) * dx + inner(g, v) * ds
```

```
27
28 # Resoudre le probleme variationnel a(u, v) = L(v)
29 u = Function(V)  # Solution du probleme
30 solve(a == L, u, bc, petsc_options={"ksp_type": "preonly", "pc_type": "lu"})
```

Code 1 – Exemple de simulation en FEniCSx

FEniCSx simplifie le processus de résolution des équations aux dérivées partielles (EDP) en utilisant la méthode des éléments finis, permettant ainsi aux utilisateurs de se concentrer sur la modélisation physique plutôt que sur les détails de l'implémentation numérique. Le code 1 illustre l'implémentation de l'interface Python de FEniCSx pour résoudre l'équation de Poisson sur un domaine carré $[0, 1] \times [0, 1]$.

4. StarPU

StarPU est un support d'exécution conçu pour gérer et ordonnancer des tâches sur des architectures hétérogènes, telles que celles combinant des CPU et des GPU. StarPU offre un cadre permettant de créer et de gérer des tâches parallèles, optimisant ainsi leur exécution sur différents types d'unités de traitement.

StarPU utilise un modèle d'exécution à bases des tâches, où les applications sont décomposées en plusieurs tâches qui peuvent avoir plusieurs implémentations pour différents types de processeurs (par exemple, CPU, CUDA, OpenCL). Une tâche specifie une unité de travail à exécuter sur différents types de processeurs, permettant ainsi à StarPU de choisir dynamiquement le processeur le plus adapté à chacune des tâches.

Les tâches peuvent avoir des dépendances, c'est-à-dire que certaines tâches doivent être exécutées avant d'autres pour assurer la cohérence des données et le bon fonctionnement de l'application. StarPU gère automatiquement ces dépendances pour s'assurer que chaque tâche ne s'exécute que lorsque toutes les tâches dont elle dépend sont terminées [3]. La figure 1 illustre un exemple de graphe de tâches, avec les nœuds représentant des tâches individuelles, tandis que les flèches indiquent les dépendances entre ces tâches, définissant ainsi l'ordre d'exécution.

StarPU utilise des algorithmes sophistiqués pour planifier l'exécution des tâches sur différentes unités de traitement de manière optimale. Ces algorithmes prennent en compte les caractéristiques spécifiques des tâches et des ressources, comme la disponibilité des processeurs et les coûts de communication entre différentes unités de calcul. StarPU peut utiliser des modèles de performance pour prédire le temps d'exécution des tâches sur différents types de processeurs et sélectionner la meilleure option pour maximiser les performances globales.

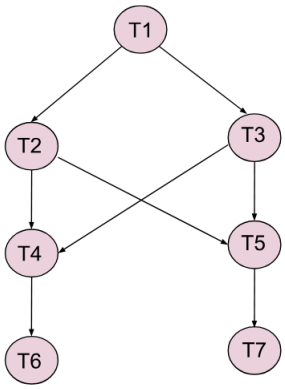

FIGURE 1 – Graphe de tâches

5. Parallélisation de la partie assemblage de DOLFINx avec StarPU

5.1. Algorithme d'assemblage

Dans la méthode des éléments finis, l'étape d'assemblage du système est particulièrement coûteuse en termes de calcul, en raison de la complexité des intégrations et du nombre élevé d'éléments impliqués. La structure du code implémenté sur DOLFINx pour l'assemblage de matrice par cellule (cellwise algorithm) est la suivante : on parcourt les éléments un par un. Pour chaque élément K, on construit les matrices élémentaires correspondantes. Ensuite, on assemble la matrice A en ajoutant les termes nécessaires à $A_{i,j}$ si K contient les sommets i et j. D'où l'algorithme d'assemblage :

Algorithm 1 Algorithme d'Assemblage

```
Lire le maillage (nœuds, éléments)
Initialiser A = 0
for chaque élément K dans elements do
    K_local ← CalculerMatriceLocal(K)
    for i = 0 to Taille(degre_de_liberte_local) − 1 do
        for j = 0 to Taille(degre_de_liberte_local) − 1 do
            I ← indice_global[i]
            J ← indice_global[j]
            A[I, J] += K_local[i, j]
        end for
    end for
end for
```

5.2. Accès concurrent des threads

Pour optimiser cette phase, nous avons utilisé StarPU pour décomposer le travail en tâches. Chaque tâche correspond à une portion de la boucle principale d'assemblage, typiquement un

ensemble d'indices de cellules élémentaires du maillage, par exemple, les dix premiers indices.

Un problème d'accès concurrent se pose lorsque deux cellules, traitées par des threads différents, partagent un même degré de liberté. Dans ce cas, les deux threads peuvent essayer d'accéder simultanément à la même entrée dans la matrice globale, ce qui peut entraîner des accès concurrents et des incohérences dans les résultats.

Pour éviter ces problèmes, des mécanismes de synchronisation, tels que les verrous (locks) et les opérations atomiques sont utilisés pour assurer que chaque thread accède de manière cohérente aux portions de la matrice qu'il doit modifier, ainsi que d'autres stratégies de parallélisation que nous aborderons en détail dans la suite. Ces techniques garantissent la cohérence des résultats tout en tirant parti des gains de performance offerts par la parallélisation.

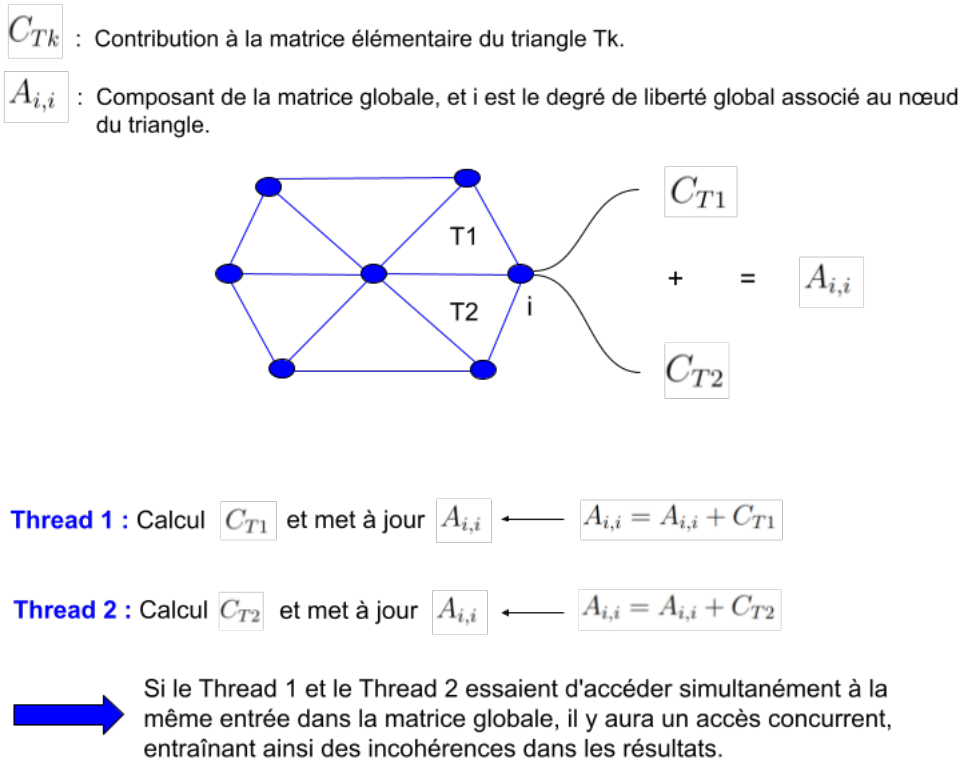

FIGURE 2 – Illustration de l'accès concurrent des threads pour une exécution parallèle

5.3. Synchronisation et gestion des accès concurrents avec Mutex

StarPU est utilisé pour paralléliser les tâches associées à des portions de la boucle sur les cellules élémentaires du maillage. Chaque tâche traite un sous-ensemble de cellules, permettant ainsi de distribuer la charge de travail sur plusieurs unités de traitement.

Lors de l'assemblage de la matrice, il est crucial de gérer les accès concurrents aux entrées partagées de la matrice globale. Une fois les données de chaque cellule (comme les coordonnées et la géométrie) récupérées et les matrices élémentaires calculées, des mutex sont utilisés pour

synchroniser et protéger l'accès concurrent aux entrées de la matrice, en bloquant l'accès aux autres threads pendant qu'un thread modifie la matrice.

Algorithm 2 Synchronisation de l'accès à la matrice globale

```
mtx.lock()
A[I, J] ← A[I, J] + K_local[i, j]
mtx.unlock()
```

L'utilisation de mutex est simple pour gérer les accès concurrents, sans nécessiter de modifications complexes dans le code existant, assurant ainsi que les données partagées sont modifiées de manière cohérente. Mais son utilisation peut introduire des surcoûts de synchronisation et ralentir le programme si les accès concurrents sont fréquents. Il est donc préférable d'adopter d'autres stratégies de la gestion d'accès concurrent pour éviter ce surcoût de synchronisation et mieux profiter de la parallélisation.

5.4. Algorithme d'assemblage par ligne "Row-wise"

Dans cette partie nous exploitons une stratégie differente, en changeant l'algorithme originale "cellwise" avec un nouvel algorithme appellé"row-wise". L'algorithme "row-wise" est une méthode efficace pour l'assemblage de matrices dans la méthode des éléments finis. Il permet d'éviter les accès concurrents et d'améliorer le parallélisme. Contrairement à l'algorithme "cellwise", qui itère sur les cellules du maillage, l'algorithme "row-wise" parcourt les lignes de la matrice globale (chaque ligne de la matrice correspondant à un degré de liberté). Voici une description de son fonctionnement :

Pour chaque degré de liberté, on identifie les éléments du maillage qui incluent ce nœud. Pour chaque élément K associé à ce degré de liberté, on extrait la ligne correspondante de la matrice élémentaire locale. Ensuite, on ajoute les contributions locales calculées pour ce degré de liberté à la ligne correspondante de la matrice globale A.

Algorithm 3 Algorithme d'assemblage par ligne

for chaque degré de liberté I **do**
  Récupérer les éléments K associés à I
  for chaque élément K **do**
    $i \leftarrow indice\_local[I]$
    Calculer la ligne i de la matrice élémentaire K_{local} associé à l'élément K
    Ajouter la contribution à la ligne I de la matrice globale A
  end for
end for

Pour ce nouvel algorithme, StarPU divise la boucle principale sur les degrés de liberté en tâches indépendantes. Ces tâches sont réparties sur différentes unités de traitement, chacune

étant responsable d'un ensemble spécifique de lignes, ce qui permet une exécution parallèle sans conflits d'accès. Bien que la boucle sur les degrés de liberté soit plus grande, le fait que chaque tâche traite des lignes distinctes de la matrice globale permet une parallélisation plus efficace et une meilleure utilisation des ressources sur les architectures multi-cœurs.

5.5. Coloration de graphe

Dans cette approche, nous utilisons une stratégie de coloration de graphe, qui attribue une couleur à chaque portion de la boucle, représentant un ensemble de cellules. Cela permet d'organiser le calcul de manière à optimiser le parallélisme et à éliminer les conflits d'accès. Voici une description détaillée des différentes étapes d'implémentation :

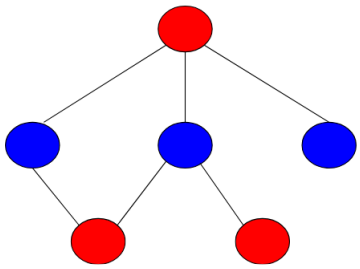

FIGURE 3 – Coloration d'un graphe : Exemple de deux couleurs

Création du graphe :

— Un graphe est construit où chaque nœud représente une portion de la boucle principale qui intègre un nombre défini de cellules (éléments de maillage).

— Les arêtes du graphe relient les portions adjacentes, indiquant qu'ils partagent une frontière et pourraient donc causer des accès concurrents s'ils sont traités en même temps.

Coloration de graphe :

— Un algorithme de coloration de graphe 4 est appliqué pour assigner des couleurs aux nœuds. L'algorithme glouton est utilisé pour sa simplicité et son efficacité.

— L'objectif est d'attribuer le minimum de couleurs possibles de sorte que deux portions adjacentes n'aient pas la même couleur, garantissant qu'ils ne seront pas exécutés en parallèle.

Algorithm 4 Algorithme de coloration

Require: Graphe $G = (V, E)$ où V est l'ensemble des sommets et E est l'ensemble des arêtes.
  Initialiser à zéro une liste couleurs pour chaque sommet
  for chaque sommet $u \in V$ **do**
    Marquer les couleurs des voisins de u comme indisponibles
    Attribuer à u la première couleur disponible
  end for
  return La liste couleurs

Exécution parallèle avec StarPU :

— Chaque portion colorée est incluse dans une tâche StarPU.

— Les tâches de même couleur sont lancées simultanément, tandis que les dépendances entre tâches de couleurs différentes sont respectées pour éviter les conflits d'accès.

Cette approche de coloration de graphe offre une solution efficace pour optimiser le calcul parallèle tout en minimisant les conflits d'accès. Cependant, une gestion efficace est nécessaire pour équilibrer la charge de travail et éviter les goulots d'étranglement qui peuvent être posés par la répartition inégale des portions de la boucle entre les couleurs. De plus, la construction et la gestion du graphe, ainsi que l'intégration de StarPU, peuvent ajouter de la complexité à l'implémentation, ce qui nécessite une évaluation attentive de son impact

5.6. Validation de l'adaptation de DOLFINx sous StarPU

Pour valider l'adaptation de DOLFINx sous StarPU, nous avons résolu l'équation de Poisson sur un domaine rectangulaire avec des conditions aux limites de Dirichlet et Neumann. Le problème consiste à résoudre l'équation : $-\nabla^2 u = f$ avec $u = 0$ sur les bords gauche et droit (Dirichlet) et une condition de flux normal $g = \sin(5x)$ sur les bords supérieur et inférieur (Neumann). Le terme source f est défini par $f = 10 \exp\left(-\frac{(x-0.5)^2+(y-0.5)^2}{0.02}\right)$.

Le maillage utilisé comprend 32x32 éléments, divisés en triangles, sur lequel est définie une fonction de base de type Lagrange de premier ordre.

Les résultats obtenus après l'adaptation ont été comparés à ceux de l'exécution standard de DOLFINx sans StarPU. Cette comparaison a permis de valider l'adaptation de DOLFINx sous StarPU en montrant une correspondance entre les solutions obtenues avant et après l'intégration.

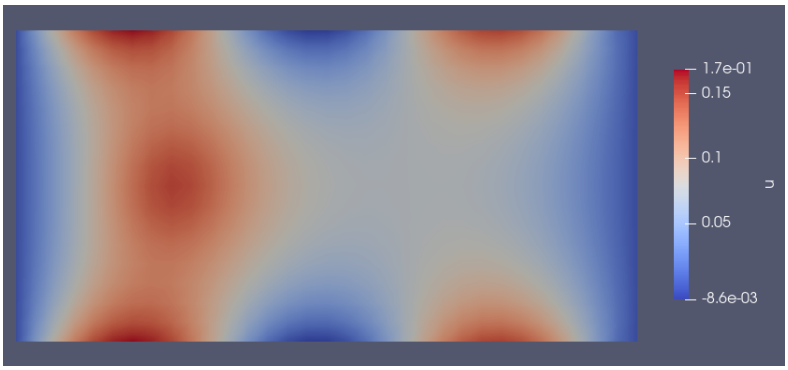

FIGURE 4 – Distribution de la solution de l'équation de Poisson résolue avec FEniCSx

6. Simulation de l'électrophysiologie cardiaque

6.1. Logiciel DEMI pour la simulation à l'échelle cellulaire

Le logiciel DEMI (Discrete EMI Model) est conçu pour simuler l'électrophysiologie des cellules cardiaques en modélisant de manière précise les domaines extracellulaire (E), membranaire (M), et intracellulaire (I). Ce modèle offre une meilleure représentation des cellules excitables comparé aux modèles homogénéisés traditionnels, en permettant de simuler l'activité électrique à l'échelle cellulaire avec une grande précision. Cependant, le modèle EMI nécessite des ressources de calcul importantes en raison de la complexité des systèmes linéaires qu'il génère et de la nécessité d'utiliser des maillages computationnels fins pour représenter avec précision la géométrie cellulaire [4].

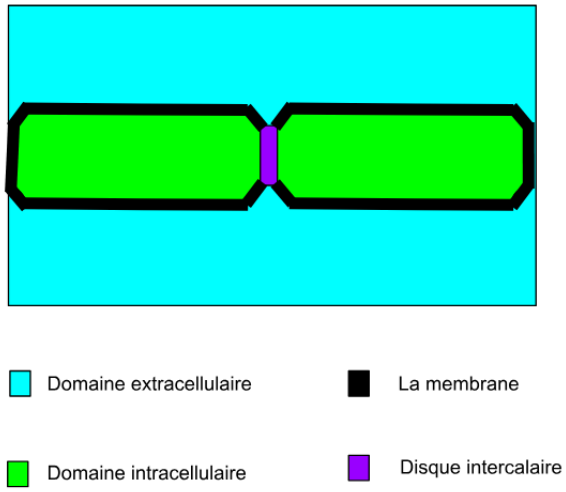

FIGURE 5 – Illustration des domaines intracellulaire, extracellulaire, et membranaire

6.2. Processus de simulation

À chaque instant t, la simulation commence par la résolution des équations membranaires. Ensuite, les disques intercalaires sont pris en compte, car ils jouent un rôle important dans la transmission des signaux électriques entre les cellules. Après avoir intégré les effets des disques intercalaires, les équations intracellulaires sont résolues pour déterminer la distribution du potentiel électrique à l'intérieur de chaque cellule. Enfin, le potentiel électrique dans l'environnement extracellulaire est calculé.

6.3. Adaptation du logiciel DEMI sous StarPU

Pour améliorer l'efficacité des calculs, DEMI a été adapté pour fonctionner avec StarPU. L'adaptation a commencé par la parallélisation de l'étape de résolution du potentiel intracellulaire, choisie pour sa simplicité et l'absence de dépendances entre les cellules. Chaque cellule est traitée comme une tâche indépendante. Cette étape a permis de valider l'intégration de StarPU, en assurant son bon fonctionnement avec le logiciel DEMI et en posant les bases pour une amélioration plus complexe des autres étapes du modèle EMI, permettant ainsi des simulations plus rapides et efficaces.

6.4. Validation des résultats de l'adaptation de DEMI sous StarPU

Pour valider l'adaptation de DEMI sous StarPU, un exemple de simulation des équations EMI en 3D a été utilisé 6. Le cas test consiste à résoudre les équations dans une grille de 5x1 cardiomyocytes, chaque cellule ayant des dimensions de 180x18x18 µm, plongée dans un domaine extracellulaire de dimensions 564x50x26 µm. L'adaptation de DEMI sous StarPU a

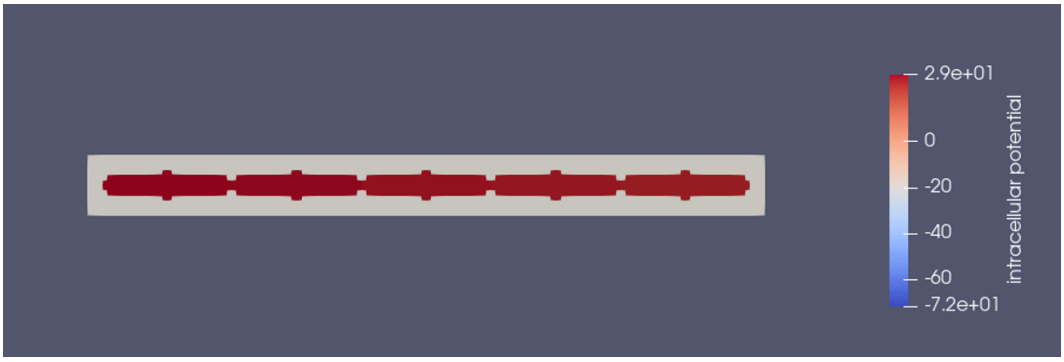

FIGURE 6 – Distribution du potentiel intracellulaire à l'instant final t=2.5 ms

été validée en comparant les résultats obtenus avant et après l'intégration de StarPU.

7. Conclusion et perspectives

Ce stage a été consacré à l'intégration de la bibliothèque StarPU avec la nouvelle version de FEniCSx pour améliorer le parallélisme et l'efficacité de l'étape d'assemblage du système dans la méthode des éléments finis. Contrairement aux adaptations précédentes, réalisées sur une ancienne version de FEniCS et un code de référence simplifié, notre travail a permis d'exploiter pleinement la nouvelle version de FEniCSx, qui intègre des fonctionnalités avancées.

Nous avons parallélisé à base de tâches l'assemblage de matrices. Dans un premier temps, l'intégration a nécessité l'utilisation de mutex pour gérer les accès concurrents aux données, constituant ainsi une première version fonctionnelle de la parallélisation. Cependant, pour ex-

ploiter plus efficacement les différentes ressources de calcul et améliorer le parallélisme, nous avons introduit des stratégies avancées telles que l'algorithme row-wise et la stratégie de coloration de graphes. Ces approches ont permis d'optimiser l'utilisation des ressources et de réduire les conflits d'accès concurrents. Les stratégies développées ont été validées et testées. Par ailleurs, l'adaptation du logiciel DEMI avec StarPU pour la simulation de l'électrophysiologie cardiaque à l'échelle cellulaire a prouvé que notre méthode fonctionne correctement, validant ainsi l'intégration et posant les bases pour des améliorations futures.

Pour aller plus loin, il est crucial d'étendre les capacités de StarPU en ajoutant des implémentations spécifiques pour les GPU afin d'accroître les performances des simulations en exploitant pleinement les capacités des architectures modernes, permettant ainsi une exécution efficace des calculs sur CPU et GPU.

En outre, il est essentiel de mener des mesures de performance pour évaluer l'impact réel des différentes stratégies d'adaptation sur l'efficacité des simulations. Comparer les performances des approches mutex, row-wise, et de coloration de graphes permettra d'identifier les méthodes les plus efficaces et d'orienter les améliorations futures. Ces perspectives offrent de nouvelles opportunités de recherche et d'innovation, renforçant ainsi les progrès réalisés durant ce stage.

Références

[1] Thomas Morin. Exploring the collaboration between FEniCSx and StarPU. Master's thesis, Université de Bordeaux (UB), FRA., December 2022.

[2] Anders Logg, Kent-Andre Mardal, Garth N. Wells, et al. *Automated Solution of Differential Equations by the Finite Element Method.* Springer, 2012.

[3] Olivier Aumage, Paul Carpenter, and Siegfried Benkner. Task-Based Performance Portability in HPC. October 2021.

[4] Aslak Tveito Kristian Gregorius Hustad Xing Cai, Karoline Horgmo Jæger. Efficient numerical solution of the emi model representing the extracellular space (e), cell membrane (m) and intracellular space (i) of a collection of cardiac cells. 2021.

A. Annexe :

A.1. Présentation du lieu de stage :

Mon stage s'est déroulé à l'Inria Bordeaux - Sud-Ouest, au sein de l'équipe STORM (Static Optimizations, Runtime Methods). Inria Bordeaux est un centre de recherche situé dans le campus universitaire de Bordeaux, à Talence. Le centre regroupe 19 équipes-projets, dont STORM, qui est un projet commune entre Inria, le CNRS, l'Université de Bordeaux et Bordeaux INP.

L'équipe STORM se concentre sur le développement de méthodes et d'outils pour exploiter au mieux les architectures de calcul parallèles modernes. Son travail porte principalement sur l'optimisation statique et les méthodes d'exécution dynamique, en vue de fournir aux programmeurs des solutions efficaces et portables pour les systèmes hétérogènes et multicœurs. Leurs recherches couvrent des domaines tels que les systèmes d'exécution pour les plateformes hétérogènes, et les outils pour l'analyse de performance et l'optimisation énergétique.

En plus de leur expertise technique, l'équipe STORM collabore étroitement avec d'autres institutions et laboratoires, notamment à travers des projets européens de grande envergure, tels que les initiatives Exascale, qui visent à repousser les limites des capacités de calcul modernes.