Méthodes numériques pour les plasmas sur architectures multicoeurs

Michel Massaro

UNIVERSITÉ DE STRASBOURG

ÉCOLE DOCTORALE MATHEMATIQUES, SCIENCES DE L'INFORMATION ET DE L'INGENIEUR

UMR 7501

THÈSE présentée par :

Michel MASSARO

soutenue le : 16 **décembre 2016**

pour obtenir le grade de : **Docteur de l'université de Strasbourg**

Discipline/ Spécialité : Mathématiques Appliquées

Méthodes numériques pour les plasmas sur architectures multicœurs

THÈSE dirigée par :

Philippe HELLUY	Professeur, université de Strasbourg

RAPPORTEURS :

Jacques BLUM	Professeur, université de Nice
Stéphane LANTERI	Directeur de Recherche, INRIA Sophia Antipolis

---

AUTRES MEMBRES DU JURY :

Vincent LOECHNER	Maître de Conférences, université de Strasbourg
Christophe PRUD'HOMME	Professeur, université de Strasbourg

Institut de Recherche Mathématique Avancée
Université de Strasbourg et C.N.R.S (UMR 7501)
7 rue René Descartes
67084 Strasbourg Cedex

THÈSE DE DOCTORAT DE L'UNIVERSITÉ DE STRASBOURG

École Doctorale de Mathématiques, Sciences de l'Information et de l'Ingénieur

Spécialité

Mathématiques Appliquées

Présentée par

Michel Massaro

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE STRASBOURG

Méthodes numériques pour les plasmas sur architectures multicoeurs

Soutenue le 16 décembre 2016 *devant le jury composé de* :

Rapporteur :	Jacques Blum	*Université de Nice*
Rapporteur :	Stéphane Lanteri	*INRIA Sophia Antipolis*
Directeur de thèse :	Philippe Helluy	*Université de Strasbourg*
Examinateur :	Michel Kern	*INRIA Rocquencourt*
Examinateur :	Vincent Loechner	*Université de Strasbourg*
Examinateur :	Christophe Prud'homme	*Université de Strasbourg*

Remerciements

Mes premiers remerciements s'adressent naturellement à mon directeur de thèse Philippe Helluy sans qui je n'écrirais pas ces mots en ce moment. Il m'a fait découvrir le monde du calcul numérique lors de mon master puis il m'a permis d'évoluer dans ce domaine au cours de ces quatre dernières années. En plus de faire preuve d'une grande patience à mon égard et de son expertise du point de vue mathématique et informatique, il a également su m'apporter de précieux conseils d'ordre non professionnel. Je remercie également Vincent Loechner de m'avoir mis le pied à l'étrier en matière d'optimisation de code. Il a su me faire comprendre l'utilité et le fonctionnement des options de compilations et de la vectorisation des opérations que je négligeais bien trop jusque là. Je le remercie également pour sa participation non négligeable dans l'implémentation OpenMP et le parcours par tuiles de l'algorithme. Enfin merci pous sa présence au sein du jury.

Je remercie chaleureusement Jacques Blum et Stéphane Lanteri d'avoir accepté d'être les rapporteurs de ma thèse. En plus de la relecture et de leurs remarques sur le manuscrit, leur participation les conduit à quitter, brièvement, la douceur de Nice pour la fraicheur alsacienne à la veille des vacances de fin d'année.

Je souhaite également remercier Hubert Baty et Jérôme Pietri du laboratoire d'astrophysique pour la présentation des cas tests ainsi que leurs conseils en matière de MHD.

Je tiens également à remercier Jonathan Jung, Thomas Strub et Malcolm Roberts qui ont toujours été présents pour répondre à mes multiples questions et qui m'ont ainsi permis d'avancer tout au long de la thèse.

Un grand merci également à Vincent Huber, avec qui j'ai partagé le bureau pendant plus d'un an et Alexandre Ancel qui m'a viré de ce même bureau pour prendre ma place... Ils ont toujours été présents pour répondre à mes questions de tout ordre et les moments partagés autour d'un café (ou pas) ont toujours été agréables.

Sur le plan professionnel, je souhaite enfin remercier les différentes équipes des services informatiques m'ayant permis de continuellement bénéficier d'un environnement de travail fonctionnel ainsi que les équipes des services administratifs de l'université.

Je remercie également le personnel de l'ENSIIE, Stéphane, Christiane, Carole, et par ordre de préférence, Pierre, Pierre et Pierre. Ils m'ont accueilli dans leur équipe lors de mon année d'ATER où là encore j'ai énormément appris.

J'en viens à présent à remercier toutes les personnes avec qui j'ai pu partager des repas, café, discutions, qui n'étaient pas nécessairement tourné autour du travail. Je pense notamment à Raphaël, Philippe, Sever, Guillaume, Nhung, Christophe.

Merci à toute la famille de Diane pour son soutien et ses encouragements et merci à ma famille pour les mêmes raisons, mais également pour avoir accepté et subi ma faible présence tout au long de ces longues années d'études.

Enfin, je n'oublie pas Diane qui fait partie de ma vie depuis onze ans maintenant et qui n'a eu de cesse de me pousser à donner le maximum de moi-même. Elle a été là dans tout les moments de doute et de déprime, elle est, pour moi, un réel modèle de courage et de persévérance. Je souhaite qu'elle soit ma source d'inspiration encore pendant de nombreuses années.

Comme beaucoup de listes, la liste des personnes citées ci-dessus ne peut pas être complète je souhaiterais donc pour finir remercier toutes les personnes qui ont été oubliées.

A Diane.

Table des matières

Introduction **1**

1 Les équations de la MHD idéales **5**

1.1 Modèle mathématique 6

1.1.1 Équations de la mécanique des fluides 6

1.1.2 Équations de Maxwell 7

1.1.3 Équations de la MHD idéale 7

1.2 La condition de divergence nulle 10

1.2.1 Méthode de Powell 11

1.2.2 Méthode de Dedner 11

1.3 Hyperbolicité 12

1.3.1 Dimension 1 15

1.3.2 Dimension $\geqslant 2$ 17

1.4 Conclusion 23

2 Étude des équations en dimension 1 **25**

2.1 Schéma de volumes finis 25

2.1.1 Méthode Galerkin Discontinu (GD) 25

2.1.2 Formulation générale de la méthode des volumes finis 27

2.1.3 Convergence 29

2.1.4 Limiteur de pentes 30

2.2 Flux numérique 32
2.2.1 Flux de Rusanov 32
2.2.2 Flux VFRoe 33
2.2.3 Schéma décentré *upwind* 35
2.3 Ordre et coût 40
2.4 Résultats 40
2.5 Conclusion 43

3 Implémentation et passage en dimension 2 **49**
3.1 Implémentation 50
3.2 Optimisation et parallélisation automatique 51
3.2.1 Fonctionnement d'un processeur (CPU) 52
3.2.2 Optimisations du compilateur 56
3.2.3 Un premier code parallèle avec OpenMP 56
3.3 Optimisations manuelles 57
3.3.1 "Loop tiling" ou parcours par tuiles 57
3.4 Résultats 60
3.5 Conclusion 60

4 Implémentation GPU/OpenCL **63**
4.1 Introduction à la programmation GPU 65
4.1.1 Architecture 65
4.1.2 Introduction à la programmation OpenCL 65
4.2 Programmation GPU du problème de la MHD 69
4.2.1 Splitting de Strang 69
4.2.2 Distribution des données en mémoire 70
4.2.3 Méthode de transposition 72
4.2.4 Distribution des tâches entre les processeurs 73

4.2.5 Déroulement d'un pas de temps . . . 73
4.3 Performances . . . 74
4.3.1 Modèle *roofline* . . . 74
4.3.2 Temps mesurés . . . 75
4.4 Multi-GPU . . . 77
4.4.1 Décomposition de domaine . . . 78
4.4.2 Temps mesurés . . . 79
4.5 Résultats numériques . . . 79
4.6 Conclusion . . . 84

5 SCHNAPS : Solveur Conservatif Hyperbolique Non-linéaire Appliqué aux PlaSmas 85

5.1 Introduction . . . 86
5.2 Méthode Galerkin Discontinu (GD) . . . 86
5.2.1 Systèmes de lois de conservation . . . 86
5.2.2 Flux numérique . . . 87
5.2.3 Formalisme GD général . . . 89
5.3 SCHNAPS : philosophie . . . 91
5.3.1 StarPU . . . 91
5.3.2 MPI . . . 92
5.3.3 OpenCL . . . 92
5.3.4 Modèles physiques . . . 93
5.3.5 Sous-domaines . . . 94
5.3.6 Macrocellules et champs . . . 94
5.3.7 Simulation . . . 95
5.3.8 Éléments géométriques . . . 97
5.3.9 Interpolation . . . 100
5.3.10 Interpolation de Gauss-Lobatto avec des sous-cellules . . . 101

5.3.11 Schéma GD pour l'interpolation de Gauss-Lobatto sur des sous-cellules . . . 104
5.4 Programmation de la MHD dans SCHNAPS . . . 106
5.4.1 Commentaires sur la programmation . . . 106
5.4.2 Premières validations . . . 107
5.4.3 Tests de performances . . . 112
5.5 Conclusion . . . 118

6 Applications de la MHD **121**
6.1 Instabilité de Kelvin-Helmoltz . . . 121
6.2 Nappes de courant oscillantes . . . 124

Conclusion **131**

A Exemple complet d'un code OpenCL **133**

Bibliographie **141**

Table des figures

2.1 Règle de nommage. 26

2.2 Propagtion de l'information à une distance inférieure à Δt. 29

2.3 Propagtion de l'information à une distance supérieure à Δt. 30

2.4 Fonctions d'approximations de la valeur absolue. 37

2.5 Condition initiale et solution exacte pour la variable ρ 41

2.6 Conditions initiales pour le cas test du choc fort 41

2.7 Solution obtenue pour la densité à l'ordre 1. 42

2.8 Solution obtenue pour la densité à l'ordre 2 avec le limiteur de pentes minmod 42

2.9 Solution obtenue pour la densité à l'ordre 2 avec le limiteur de pentes minmod2. 43

2.10 Solution obtenue pour la vitesse dans la direction y à l'ordre 1. 43

2.11 Solution obtenue pour la vitesse dans la direction y à l'ordre 2 avec le limiteur de pentes minmod. 44

2.12 Solution obtenue pour la vitesse dans la direction y à l'ordre 2 avec le limiteur de pentes minmod2. 44

2.13 Comparaison des différents schémas numériques à l'ordre 1 45

2.14 Comparaison des différents schémas numériques à l'ordre 2 avec le limiteur de pentes minmod 45

2.15 Comparaison des différents schémas numériques à l'ordre 2 avec le limiteur de pentes minmod2 46

2.16 Comparaison des temps d'exécution des schémas 46

2.17 Comparaison des ordres de convergence pour les méthodes d'ordre 1 et d'ordre 2. 47

3.1 Représentation schématique des données du problème. 50

3.2 Représentation de l'étape de mise à jour 51

3.3 Représentation du passage des conditions de bord périodique 52

3.4 Architecture simplifiée d'un CPU. 52

3.5 Pipeline d'instructions. 53

3.6 Pipeline d'instructions pour l'addition de deux vecteurs. 54

3.7 Déroulement schématique d'un code utilisant la parallélisation à l'aide de OpenMP. 57

3.8 Parcours traditionnel d'une grille 58

3.9 Représentation des cellules dans la mémoire cache 58

3.10 Parcours par tuile d'une grille 59

3.11 Représentation des cellules dans la mémoire cache 60

4.1 Superordinateurs *Cray-I* et *Tianhe-2* 64

4.2 Représentation schématique de l'architecture d'un GPU 66

4.3 Lecture de cinq valeurs en mémoire réparties de manières différentes dans la mémoire. 71

4.4 Lecture des données dans une grille bi dimensionnelle et une grille linéarisée. 71

4.5 Répartition des champs de la figure 4.4 après transposition de la grille. . . 72

4.6 Représentation des étapes effectuées lors de la transposition optimisée. . . 73

4.7 Distribution des tâches dans le GPU 74

4.8 Modèle roofline pour la méthode volumes finis à l'ordre 1 et 2 76

4.9 Représentation schématique d'un ensemble de GPU 78

4.10 Exemple du partage d'un maillage sur quatre cartes graphiques 78

4.11 Cellules de recouvrement entre les sous-parties à l'ordre 1 79

4.12 Conditions initiales pour le cas test du vortex de Orzsag-Tang 80

4.13 Solution pour le cas test de Orzsag-Tang au temps $0,5$. Simulation effectuée à l'ordre 2 avec le flux $P2$ sur un maillage de taille 4000×4000 . . . 81

4.14 Solution pour le cas test de Orzsag-Tang au temps $1,5$. Simulation effectuée à l'ordre 2 avec le flux $P2$ sur un maillage de taille 4000×4000 . . . 81

4.15 Solution pour le cas test de Orzsag-Tang au temps $2,5$. Simulation effectuée à l'ordre 2 avec le flux $P2$ sur un maillage de taille 4000×4000 . . . 82

4.16 Solution pour le cas test de Orzsag-Tang au temps 3. Simulation effectuée à l'ordre 2 avec le flux $P2$ sur un maillage de taille 4000×4000 82

4.17 Zoom sur une partie de la solution pour le cas test de Orzsag-Tang au temps 1. Simulation effectuée à l'ordre 2 avec le flux $P2$ sur un maillage de taille 15000×15000 83

4.18 Zoom sur une partie de la solution pour le cas test de Orzsag-Tang au temps 2. Simulation effectuée à l'ordre 2 avec le flux $P2$ sur un maillage de taille 15000×15000 84

5.1 Transformation géométrique 94

5.2 Graphe des tâches pour un calcul dans un sous-domaine contenant 3 zones. Les tâches sont représentées par les noeuds du graphe et les flèches représentent les dépendances entre les tâches (une tâche doit attendre la fin des tâches dont elle dépend avant de démarrer). 96

5.3 Numérotation des faces 99

5.4 Numérotation dans les sous-cellules 102

5.5 Solutions obtenues par SCHNAPS pour le cas test du choc fort au degré 1. 108

5.6 Solutions obtenues par SCHNAPS pour le cas test du choc fort au degré 2. 108

5.7 Solutions obtenues par SCHNAPS pour le cas test du choc fort au degré 3. 109

5.8 Comparaison des ordres de convergence pour les méthodes d'ordre 1 et d'ordre 2. 109

5.9 Solution au temps $t = 1$ pour le cas test de Orzsag-Tang 110

5.10 Solution au temps $t = 1.5$ pour le cas test de Orzsag-Tang 111

5.11 Zoom sur une partie de la solution au temps $t = 1$ pour le cas test de Orzsag-Tang 111

5.12 Comparaison des temps d'exécution entre la version C et la version OpenCL de SCHNAPS sur un seul coeur de calcul 112

5.13 Comparaison des temps d'exécution de la méthode RK2_CL pour les degrés 1, 2 et 3 de la méthode GD. 113

5.14 Comparaison des temps d'exécution entre la version C et la version OpenCL de SCHNAPS sur un seul coeur de calcul 115

5.15 Comparaison des temps d'exécution de la version StarPU de SCHNAPS pour les codelets C pour différents nombres de coeurs 115

5.16 Trace de l'exécution du programme pour un macromaillage 3×3 sur 4, 9 et 12 CPU. 116

5.17 Trace de l'exécution du programme pour un macromaillage 3×3 sur 4, 9 et 12 CPU. 117

6.1 Conditions initiales pour le cas test des instabilités de Kelvin-Helmoltz . . 122

6.2 Densité pour le cas test des instabilités de Kelvin-Helmoltz au temps 1.5. Simulation effectuée à l'ordre 2 avec le flux $P2$ sur un maillage de taille 4000×4000 122

6.3 Champ magnétique dans la direction x pour le cas test des instabilités de Kelvin-Helmoltz au temps 1.5. Simulation effectuée à l'ordre 2 avec le flux $P2$ sur un maillage de taille 4000×4000 123

6.4 Densité pour le cas test des instabilités de Kelvin-Helmoltz au temps 2. Simulation effectuée à l'ordre 2 avec le flux $P2$ sur un maillage de taille 4000×4000 123

6.5 Densité pour le cas test des instabilités de Kelvin-Helmoltz au temps 2. Simulation effectuée à l'ordre 2 avec le flux $P2$ sur un maillage de taille 4000×4000 124

6.6 Conditions initiales pour le cas test des nappes de courants oscillantes . . 125

6.7 Champ magnétique dans la direction y pour le cas test des nappes de courant oscillantes au temps 0.5. Simulation effectuée à l'ordre 2 avec le flux $P2$ sur un maillage de taille 2500×2500 125

6.8 Champ magnétique dans la direction y pour le cas test des nappes de courant oscillantes au temps 1.5. Simulation effectuée à l'ordre 2 avec le flux $P2$ sur un maillage de taille 2500×2500 126

6.9 Champ magnétique dans la direction y pour le cas test des nappes de courant oscillantes au temps 2. Simulation effectuée à l'ordre 2 avec le flux $P2$ sur un maillage de taille 2500×2500 126

6.10 Champ magnétique dans la direction y pour le cas test des nappes de courant oscillantes au temps 2.5. Simulation effectuée à l'ordre 2 avec le flux $P2$ sur un maillage de taille 2500×2500 127

6.11 Pression pour le cas test des nappes de courant oscillantes au temps 1.5. Simulation effectuée à l'ordre 2 avec le flux $P2$ sur un maillage de taille 2500×2500 128

6.12 Pression pour le cas test des nappes de courant oscillantes au temps 2. Simulation effectuée à l'ordre 2 avec le flux $P2$ sur un maillage de taille 2500×2500 128

6.13 Pression pour le cas test des nappes de courant oscillantes au temps 2.5. Simulation effectuée à l'ordre 2 avec le flux $P2$ sur un maillage de taille 2500×2500 129

Liste des symboles

Chapitre 1

Ω	Domaine de définition du problème, $\Omega \in \mathbb{R}^3$.
ρ	Masse volumique d'un gaz incrompressible.
$\mathbf{u}$	Champs de vecteur dans $\mathbb{R}^3$.
p	Pression d'un gaz.
γ	Ratio entre la capacité thermique à pression constante et la capacité thermique à volume constant d'un gaz.
Q	Energie totale du système.
$\mathbf{B}$	Champ magnétique dans $\mathbb{R}^3$.
$\mathbf{E}$	Champ électrique dans $\mathbb{R}^3$.
$\mathbf{J}$	Densité de courant.
$q_i,\ q_e$	Densité de charge des ions et des électrons.
σ	Conductivité.
$\varepsilon_0,\ \mu_0$	Permittivité et perméabilité d'un gaz.
$\mathcal{D}$	Opérateur différentiel linéaire.
c_h	Vitesse d'évacuation de la divergence.
$\mathbf{W}$	Vecteur de variables conservatives.
$\mathbf{F}(\mathbf{W}, \mathbf{n})$	Flux du système.
$\mathbf{A}(\mathbf{W}, \mathbf{n})$	Matrice jacobienne du système.
$\mathbf{Y}$	Vecteur de variables primitives.
c_A	Vitesse d'Alfvèn.
$\lambda_i\ r_i$	Valeurs et vecteurs propres du système.
$c_s\ c_f$	Vitesses magnétoaccoustique lentes et rapide.

Chapitre 2

$\mathcal{T}$	Maillage physique de Ω.
K	Elément quelconque de $\mathcal{T}$.
φ_i^K	Fonction test sur un élément K. La fonction vaut 1 sur le point de Gauss numéro i et 0 ailleurs.
$\mathbf{W}_{L/R}$	États du système de part et d'autre de la séparation de deux cellules voisines L et R.
$\mathbf{n}_{LR}$	Vecteur unitaire sortant dirigé de la cellule L vers la cellule R.
$\mathbf{F}(\mathbf{W}_L, \mathbf{W}_R, \mathbf{n}_{LR})$	Flux numérique du système.
$\boldsymbol{\zeta}$, $\mathbf{r}$	Dérivés en espace et en temps du champ $\mathbf{W}$.

Chapitre 5

$\widehat{\star}$	Certains symboles peuvent être accentués par l'accent circonflexe signifiant que l'élément est pris dans l'espace de référence.
L	Elément géométrique quelconque.
$\mathbf{X}$	Coordonées des noeuds géométriques.
$\boldsymbol{\psi}$	Fonctions de bases géométriques.
τ_L	Transformation géométrique envoyant $\hat{L}$ sur L.
I_L	Le k^{e}polynôme de Lagrange.
I_L	Le k^{e}polynôme de Lagrange.
$\boldsymbol{\xi}$	Vecteur de coordonnées dans l'espace physique des points de Gauss-Lobatto.
$\mathbf{P}i$, π	Transformation faisant passé de la numérotation sur les bord de L à la numérotation dans l'élément L.
$\mathbf{Y}$, λ	Points et poids de Gauss-Lobatto.

Introduction

Apparue en même temps que l'informatique, la simulation numérique consiste à modéliser des modèles physiques via un programme informatique. Elle est omniprésente dans de nombreuses disciplines, par exemple pour les prévisions climatiques. Toutes les simulations existantes reposent sur un modèle mathématique théorique du phénomène physique réel à étudier. Dans cette thèse, nous sommes intéressés par l'étude des plasmas et plus particulièrement par l'étude des équations de la Magnéto-Hydro-Dynamique (MHD) qui modélisent l'évolution d'un fluide conducteur dans un champ magnétique. Ces équations écrites pour la première fois par Alfvèn [1] sont composées des équations de la mécanique des fluides couplées aux équations de Maxwell, . Ces équations apparaissent par exemple dans le cadre du projet de fusion thermonucléaire ITER puisque le fonctionnement d'un réacteur à fusion consiste à chauffer du plasma à très haute température. Ce plasma est soumis à un champ magnétique intense. L'étude des équations de la MHD nous permet dans ce cas de prédire les instabilités de bord du plasma. Dans un autre contexte, les instabilités MHD sont également la source du champ magnétique terrestre.

Les méthodes de résolution pour des simulations numériques sont multiples. Les problèmes d'équations aux dérivées partielles sont très souvent résolus à l'aide d'une méthode d'éléments finis ou de volumes finis qui se base sur une formulation faible du problème. Dans notre cas, le système MHD est un système hyperbolique de lois de conservation. Pour ce type de problème, la méthode des volumes finis présente l'avantage d'être plus rapide et moins coûteuse en matière d'espace mémoire. Elle est également assez robuste lorsque les solutions obtenues présentent des ondes de choc. Cependant, lorsque la solution a de nombreuses fluctuations, la méthode des volumes finis requiert un maillage très fin. Pour capturer ce type de solutions à moindre coût, nous utilisons une méthode d'éléments finis discontinus : la méthode Galerkin Discontinu (GD). La méthode GD est une généralisation à des ordres quelconques de la méthode des volumes finis (VF).

Cette nécessité d'obtenir un résultat précis en un minimum de temps est un problème universel dans le domaine de la simulation numérique. Bien que la croissance exponentielle de la puissance de calcul des machines ait permis des réductions du temps d'exécution non négligeable, c'est l'utilisation du calcul parallèle qui permet de véritables augmentations de la performance des algorithmes aujourd'hui. Pour ces raisons, un lien important per-

siste entre les mathématiques et l'informatique et de ce fait, le domaine de la simulation numérique bénéficie de toutes les évolutions des méthodes de calcul sur ordinateurs. Dans ce manuscrit, nous étudions plusieurs méthodes d'accélérations pour les algorithmes. En particulier, nous implémentons des méthodes de calcul sur processeurs graphiques.

Dans le premier chapitre, nous posons les bases du problème de la MHD idéale. Pour cela, nous commençons par étudier les équations de la dynamique des fluides avant de voir les équations de l'électrodynamique de Maxwell. Le couplage de ces deux systèmes nous conduit finalement au problème de la MHD. En dimension quelconque, une des conditions essentielles du problème concerne la divergence du champ magnétique. En effet, cette valeur doit rester nulle au cours de la simulation. Cependant, il est difficile de respecter numériquement cette condition. Dans le but de corriger l'apparition d'une divergence non nulle, nous étudions les méthodes proposées par Dedner et Powell [18, 54]. Nous terminons ce chapitre en étudiant l'hyperbolicité du problème MHD.
Le chapitre 2 est consacré à la présentation d'une méthode de volumes finis pour la résolution du problème de la MHD. Pour commencer, nous nous plaçons dans le cas particulier de la dimension 1. Après avoir exposé la formulation générale de la méthode des volumes finis à l'ordre 1 et avoir discuté de la convergence d'un tel schéma, nous utilisons une méthode MUSCL de Van Leer [64] afin d'obtenir un schéma d'ordre 2. Le choix du flux numérique de la méthode des éléments finis étant un point important en termes de précision, mais également en termes de temps de calcul, nous comparons plusieurs flux numériques possibles selon ces deux critères. En particulier, nous évaluons plusieurs flux numériques possibles, dérivés d'un schéma décentré *upwind* proposé dans [19].
La partie de la thèse plus axée implémentations numériques et optimisations démarre au chapitre 3 où nous donnons les détails de l'implémentation de la méthode des volumes finis pour le problème de la MHD en dimension 2. Après une introduction sur le fonctionnement d'un processeur, nous étudions sur un exemple simple à l'aide de codes en assembleur, l'intérêt des optimisations par déroulage de boucles effectuées de manière automatique par les compilateurs. Nous restons ensuite sur la parallélisation automatique puisque nous décrivons rapidement le principe de la parallélisation à l'aide de la librairie OpenMP. Suite à cela, nous terminons ce chapitre en présentant une méthode permettant de parcourir une grille par tuile afin de profiter au maximum des effets de cache.
Dans le chapitre 4, nous commençons par une introduction à la programmation sur processeurs graphiques (GPU) où nous donnons des informations concernant l'architecture d'un processeur graphique. Nous donnons dans cette introduction un exemple détaillé permettant de comprendre l'utilisation de la librairie OpenCL. Nous donnons ensuite les détails de l'implémentation de notre modèle volumes finis pour le problème de la MHD. Pour cela, nous décrivons les méthodes de splitting directionnelle de Strang et de transposition optimisée que nous avons utilisées afin de conserver un alignement maximal des données en mémoire. En plus de la mesure du temps d'exécution de l'algorithme, nous étudions la performance de notre implémentation à l'aide du modèle *roofline*. Nous utilisons ensuite la librairie MPI (Message Passing Interface) [59] afin de pouvoir gérer un

transfert des données entre plusieurs GPU et ainsi d'accroitre la puissance de calcul.
L'avant-dernier chapitre de la thèse est consacré à la bibliothèque de fonctions SCHNAPS en développement à l'IRMA de Strasbourg depuis deux ans. Cette bibliothèque permet la résolution de systèmes de lois de conservation sur des architectures massivement parallèles. Nous commençons par un rappel du formalisme de la méthode Galerkin-Discontinue puis nous discutons de la philosophie de SCHNAPS qui utilise des méthodes d'optimisations récentes telles que la programmation sur processeurs graphiques et l'utilisation de la bibliothèque StarPU, qui permet une programmation par graphe des tâches. Une partie importante de ce chapitre concerne la représentation physique des modèles dans SCHNAPS, ainsi que l'utilisation d'un maillage à différents niveaux de finesse. Finalement, nous revenons sur le problème de la MHD en faisant une description de l'implémentation du modèle dans SCHNAPS. Le chapitre se termine par une étude de la précision et des temps obtenus avec SCHNAPS.
Enfin, pour conclure le manuscrit, nous donnons dans le chapitre 6 des exemples de problème MHD appliqués dans le domaine de l'astrophysique.

Chapitre 1

Les équations de la MHD idéales

> The most exciting phrase to hear in science, the one that heralds new discoveries, is not 'Eureka!' (I found it!) but rather, 'hmm.... that's funny...'
>
> Isaac Asimov

Sommaire

1.1 Modèle mathématique 6
1.1.1 Équations de la mécanique des fluides 6
1.1.2 Équations de Maxwell 7
1.1.3 Équations de la MHD idéale 7
1.2 La condition de divergence nulle 10
1.2.1 Méthode de Powell 11
1.2.2 Méthode de Dedner 11
1.3 Hyperbolicité 12
1.3.1 Dimension 1 15
1.3.2 Dimension $\geqslant 2$ 17
1.4 Conclusion 23

Les équations de la Magnéto-Hydro-Dynamique (MHD) modélisent le comportement de fluides chargés. Elles sont déduites des équations de la mécanique des fluides et des équations de Maxwell. Dans ce chapitre, nous étudions comment se fait le couplage entre ces deux modèles puis nous donnons le problème sous sa forme conservative. Les équations de la MHD sont couplées à une condition de divergence nulle du champ magnétique. Dans un deuxième temps, nous étudions certaines méthodes utilisées afin d'imposer cette condition de divergence nulle sur le plan numérique.

1.1 Modèle mathématique

1.1.1 Équations de la mécanique des fluides

Du point de vue des physiciens, les équations de la mécanique des fluides sont basées sur les bilans des masses, de la quantité de mouvement et de l'énergie. Sur un domaine $\Omega \in \mathbb{R}^3$, pour un vecteur normal sortant $\mathbf{n}$, la conservation de la masse volumique $\rho(\mathbf{x}, t)$ d'un gaz imcompressible est donnée par

$$\frac{d}{dt}\int_\Omega \rho d\mathbf{x} = -\int_{\partial\Omega} \rho \mathbf{u}\cdot\mathbf{n}$$

où $\mathbf{u} = (u^1, u^2, u^3)^T$ représente le champ de vitesse. Nous pouvons réécrire cette équation sous la forme

$$\rho_t + \nabla\cdot(\rho\mathbf{u}) = 0. \tag{1.1}$$

De manière équivalente, la conservation de la quantité de mouvement s'écrit

$$\frac{d}{dt}\int_\Omega \rho\mathbf{u} d\mathbf{x} = -\int_{\partial\Omega} \mathbf{u}\cdot\mathbf{n}\rho\mathbf{u} - \int_{\partial\Omega} p\mathbf{n}$$

pour une pression p vérifiant la loi des gaz parfaits

$$p = (\gamma - 1)\rho e.$$

avec γ le ratio entre la capacité thermique à pression constante et la capacité thermique à volume constant du gaz. Dans notre cas, nous prenons principalement la valeur $5/3$ c'est-à-dire la valeur correspondante au ratio pour un gaz parfait monoatomique. Sous la forme d'une équation vectorielle, on obtient

$$(\rho\mathbf{u})_t + \nabla\cdot(\rho\mathbf{u}\otimes\mathbf{u}) + \nabla p = 0. \tag{1.2}$$

Enfin, l'énergie totale du système est la somme de l'énergie interne et de l'énergie cinétique

$$Q = \rho e + \rho\frac{\mathbf{u}\cdot\mathbf{u}}{2}. \tag{1.3}$$

La conservation de l'énergie est donnée par

$$\frac{d}{dt}\int_\Omega Q d\mathbf{x} = -\int_{\partial\Omega} Q\mathbf{u}\cdot\mathbf{n} - \int_{\partial\Omega} p\mathbf{u}\cdot\mathbf{n},$$

ou encore

$$Q_t + \nabla\cdot((Q+p)\mathbf{u}) = 0. \tag{1.4}$$

1.1.2 Équations de Maxwell

Les équations de Maxwell sont données par les équations de Maxwell Ampère, Maxwell Faraday, Maxwell Gauss et l'équation locale de Maxwell qui sont respectivement

$$\begin{cases} \mu_0\varepsilon_0\mathbf{E}_t + \nabla\times\mathbf{B} = \mu_0\mathbf{J} & (1.5)\\ \mathbf{B}_t + \nabla\times\mathbf{E} = 0 & (1.6)\\ \nabla\cdot\mathbf{E} = \dfrac{\rho}{\varepsilon_0} & (1.7)\\ \nabla\cdot\mathbf{B} = 0. & (1.8) \end{cases}$$

où μ_0 et ε_0, les perméabilité magnétique et permittivité diélectrique du vide, sont des constantes, $\mathbf{B} = (B^1, B^2, B^3)^T$ et $\mathbf{E} = (E^1, E^2, E^3)^T$ représentent les champs électrique et magnétique et $\mathbf{J}$ est le vecteur densité de courant. L'équation de conservation de l'énergie de ce système est donnée par

$$\partial_t\left(\frac{\mathbf{E}^2}{2\mu_0\varepsilon_0} + \frac{\mathbf{B}^2}{2}\right) + \nabla\cdot(\mathbf{E}\times\mathbf{B}) = 0 \tag{1.9}$$

où le vecteur $\mathbf{E}\times\mathbf{B}$ est appelé le vecteur de poynting, il indique la direction de propagation de l'énergie d'une onde électromagnétique.

1.1.3 Équations de la MHD idéale

Le domaine de la Magnéto-Hydro-Dynamique est divisé entre la MHD idéale où le fluide est considéré comme un conducteur parfait et la MHD résistive où l'évolution du champ magnétique dans le fluide est donnée par l'équation d'induction résistive. Dans cette thèse, le travail est axé sur la MHD idéale. Nous considérons donc les hypothèses suivantes.

— Le fluide est non relativiste. Ce qui implique que le champ magnétique domine.
— Le fluide est quasi-neutre.
— Le fluide est infiniment conducteur.

Comme nous sommes intéressés par l'étude de fluides chargés nous ajoutons à l'équation d'énergie des équations d'Euler(1.3 page 6), la conservation d'énergie pour les équations de Mawell (1.9 page précédente). Comme nous nous plaçons dans le cas de la MHD idéale, les hypothèses ci-dessus impliquent que nous pouvons réécrire cette dernière équation

$$\partial_t \left(\frac{\mathbf{B}^2}{2} \right) = 0 \tag{1.10}$$

et donc que pour un fluide chargé, l'énergie totale du système s'écrit

$$Q = \rho e + \rho \frac{\mathbf{u} \cdot \mathbf{u}}{2} + \frac{\mathbf{B} \cdot \mathbf{B}}{2}. \tag{1.11}$$

L'hypothèse de quasi-neutralité du système implique que la charge magnétique est nulle. En effet, les ions chargés positivement sont immobiles face aux électrons beaucoup plus légers. Les électrons viennent donc très rapidement compenser les charges dans le système. En notant q_i et q_e les densités de charges respectives des ions et des électrons, nous avons donc

$$n(q_i - q_e)\mathbf{E} = 0.$$

Cependant, si la charge globale est nulle, ce n'est pas le cas du courant. En effet, les électrons se déplacent rapidement et génèrent donc un courant. L'évolution des champs magnétique et électrique s'obtient en écrivant la loi d'Ohm dans un repère lié au fluide

$$\mathbf{J} = \sigma(\mathbf{E} + \mathbf{u} \times \mathbf{B}).$$

Or nous avons supposé dans le cas de la MHD idéale, que la conductivité σ tend vers l'infini. Nous avons l'égalité

$$\mathbf{E} = -\mathbf{u} \times \mathbf{B}$$

qui nous conduit alors, en utilisant (1.6 page précédente), à l'équation d'induction du champ magnétique

$$\mathbf{B}_t - \nabla \times (\mathbf{u} \times \mathbf{B}) = 0. \tag{1.12}$$

En utilisant (1.8 page 7), nous réécrivons à l'aide du produit tensoriel

$$\mathbf{B}_t + \nabla \cdot (\mathbf{u} \otimes \mathbf{B} - \mathbf{B} \otimes \mathbf{u}) = 0. \tag{1.13}$$

De plus, dans le cas d'un fluide chargé, la force volumique exercée sur le fluide est une force de Lorentz donnée par

$$q\mathbf{E} + \mathbf{J} \times \mathbf{B}$$

Nous avons vu que d'après l'hypothèse de quasi-neutralité nous pouvons négliger le terme $q\mathbf{E}$ et l'équation (1.2 page 6) devient

$$(\rho\mathbf{u})_t + \nabla \cdot (\rho\mathbf{u} \otimes \mathbf{u}) + \nabla p = \mathbf{J} \times \mathbf{B}$$

Enfin, en réécrivant la loi de Maxwell-Ampère (1.5 page 7) sous la forme

$$\nabla \times \mathbf{B} = \mu_0 \mathbf{J} - \mu_0 \varepsilon_0 \mathbf{E}_t$$

nous constatons que le terme $\mu_0 \varepsilon_0 \mathbf{E}_t$ est négligeable puisque la permittivité du vide est de l'ordre de 10^{-12}. De plus, dans les matériaux ne pouvant pas canaliser le champ magnétique comme le vide où les gaz, la perméabilité μ_0 vaut 1. Nous avons ainsi la nouvelle expression pour le rotationnel du champ magnétique

$$\nabla \times \mathbf{B} = \mathbf{J}$$

donc en utilisant (1.8 page 7) nous pouvons réécrire

$$(\rho\mathbf{u})_t + \nabla \cdot \left(\rho\mathbf{u} \otimes \mathbf{u} + \left(p + \frac{\mathbf{B} \cdot \mathbf{B}}{2} \right) \mathbf{I} - \mathbf{B} \otimes \mathbf{B} \right) = 0 \tag{1.14}$$

où $\mathbf{I}$ représente la matrice identité.

Finalement, en regroupant les équations (1.1 page 6), (1.14), (1.4 page 7) et (1.13) nous pouvons écrire le système d'équations pour la MHD idéale

$$\partial_t \begin{pmatrix} \rho \\ \rho\mathbf{u} \\ Q \\ \mathbf{B} \end{pmatrix} + \nabla \cdot \begin{pmatrix} \rho\mathbf{u} \\ \rho\mathbf{u}\otimes\mathbf{u} + (p + \frac{\mathbf{B}\cdot\mathbf{B}}{2})\mathbf{I} - \mathbf{B}\otimes\mathbf{B} \\ (Q + p + \frac{\mathbf{B}\cdot\mathbf{B}}{2})\mathbf{u} - (\mathbf{B}\cdot\mathbf{u})\mathbf{B} \\ \mathbf{u}\otimes\mathbf{B} - \mathbf{B}\otimes\mathbf{u} \end{pmatrix} = 0, \tag{1.15}$$

$$Q = \rho e + \rho\frac{\mathbf{u}\cdot\mathbf{u}}{2} + \frac{\mathbf{B}\cdot\mathbf{B}}{2}.$$

$$p = P(\rho, e) = (\gamma - 1)\rho e, \quad \gamma > 1.$$

À l'état initial, le champ magnétique est considéré à divergence nulle. En prenant la divergence de l'équation d'induction du champ magnétique (1.12 page précédente), nous obtenons

$$\partial_t(\nabla\cdot\mathbf{B}) - \nabla\cdot(\nabla\times(\mathbf{u}\times\mathbf{B})) = 0$$

et comme la divergence d'un rotationnel est nulle nous en déduisons que

$$\partial_t(\nabla\cdot\mathbf{B})) = 0. \tag{1.16}$$

et donc $\nabla\cdot\mathbf{B}$ est constante. Nous avons donc une dernière équation à respecter pour le modèle de la MHD idéale,

$$\nabla\cdot\mathbf{B} = 0. \tag{1.17}$$

1.2 La condition de divergence nulle

Comme nous venons de le voir, la condition de divergence est importante pour assurer qu'il n'y ait pas de charge magnétique. Cependant, certains schémas numériques peuvent créer des charges magnétiques non physiques. L'équation d'induction (1.13 page précédente) va alors les préserver et la divergence non nulle va provoquer l'apparition dans le modèle de forces magnétiques parallèles aux lignes de champs. Ces effets vont s'accumuler au cours de la simulation et ainsi provoquer des instabilités numériques.

1.2.1 Méthode de Powell

Une première approche pour résoudre ce problème en dimension supérieure est donnée par Powell [54]. Il propose d'introduire un terme magnétique dans les équations de la MHD. Les équations (1.14 page 9) et (1.13 page 9) deviennent alors

$$(\rho\mathbf{u})_t + \nabla\cdot\left(\rho\mathbf{u}\otimes\mathbf{u} + \left(p + \frac{\mathbf{B}\cdot\mathbf{B}}{2}\right)\mathbf{I} - \mathbf{B}\otimes\mathbf{B}\right) = -(\nabla\cdot\mathbf{B})\mathbf{B} \tag{1.18}$$

$$\mathbf{B}_t + \nabla\cdot(\mathbf{u}\otimes\mathbf{B} - \mathbf{B}\otimes\mathbf{u}) = -(\nabla\cdot\mathbf{B})\mathbf{u} \tag{1.19}$$

et le problème (1.15 page précédente) s'écrit maintenant à l'aide d'un terme source

$$\partial_t\begin{pmatrix}\rho\\ \rho\mathbf{u}\\ Q\\ \mathbf{B}\end{pmatrix} + \nabla\cdot\begin{pmatrix}\rho\mathbf{u}\\ \rho\mathbf{u}\otimes\mathbf{u} + (p+\frac{\mathbf{B}\cdot\mathbf{B}}{2})\mathbf{I} - \mathbf{B}\otimes\mathbf{B}\\ (Q+p+\frac{\mathbf{B}\cdot\mathbf{B}}{2})\mathbf{u} - (\mathbf{B}\cdot\mathbf{u})\mathbf{B}\\ \mathbf{u}\otimes\mathbf{B} - \mathbf{B}\otimes\mathbf{u}\end{pmatrix} = -\nabla\cdot\mathbf{B}\begin{pmatrix}0\\ \mathbf{B}\\ \mathbf{u}\cdot\mathbf{B}\\ \mathbf{u}\end{pmatrix}. \tag{1.20}$$

De cette manière, sept des huit valeurs propres et vecteurs propres de ce système par rapport au système initial (1.15 page ci-contre) sont identiques et seule une valeur propre est modifiée. De plus, à l'état initial où nous avons $\nabla\cdot\mathbf{B} = 0$ les deux systèmes sont équivalents. En revanche, si une perturbation est introduite sur la divergence, en utilisant des conditions aux bords adéquates, la perturbation sera propagée à la vitesse $\mathbf{u}$. Bien que la méthode permette de diminuer les erreurs de divergence, le terme source rend le système non conservatif. Cela aura donc pour effet d'ajouter des solutions non physiques au problème (en particulier au niveau des chocs). Toutefois, l'erreur ajoutée est locale donc le résultat global est amélioré.

1.2.2 Méthode de Dedner

Afin d'améliorer numériquement la condition de divergence nulle tout en gardant un système conservatif, une autre approche a été proposée par Dedner *et al.* [18]. Cette méthode consiste à coupler le système (1.15 page précédente) et l'équation (1.16 page ci-contre) à l'aide d'une nouvelle fonction inconnue ψ. Les équations (1.8 page 7) et (1.13 page 9) deviennent alors

$$\mathbf{B}_t + \nabla\cdot(\mathbf{u}\otimes\mathbf{B} - \mathbf{B}\otimes\mathbf{u}) + \nabla\psi = 0. \tag{1.21}$$

$$\nabla \cdot \mathbf{B} + \mathcal{D}(\psi) = 0 \tag{1.22}$$

où $\mathcal{D}$ représente un opérateur linéaire différentiel. Plusieurs choix concernant l'opérateur $\mathcal{D}$ sont étudiés dans [18] conduisant les équations à devenir, selon le cas elliptique, parabolique ou hyperbolique. Dans cette thèse, nous souhaitons travailler sur des systèmes hyperboliques. Pour cela, l'opérateur $\mathcal{D}$ le plus approprié est alors

$$\mathcal{D}(\psi) = \frac{1}{c_h^2}\partial_t \psi.$$

où c_h est une constante représentant la vitesse à laquelle est évacuée la divergence. L'équation sur la divergence (1.22) devient alors

$$\partial_t \psi + c_h^2 \nabla \cdot \mathbf{B} = 0.$$

et le système (1.15 page 10) s'écrit maintenant avec une inconnue et une équation supplémentaire

$$\partial_t \begin{pmatrix} \rho \\ \rho\mathbf{u} \\ Q \\ \mathbf{B} \\ \psi \end{pmatrix} + \nabla \cdot \begin{pmatrix} \rho\mathbf{u} \\ \rho\mathbf{u} \otimes \mathbf{u} + (p + \frac{\mathbf{B}\cdot\mathbf{B}}{2})\mathbf{I} - \mathbf{B} \otimes \mathbf{B} \\ (Q + p + \frac{\mathbf{B}\cdot\mathbf{B}}{2})\mathbf{u} - (\mathbf{B}\cdot\mathbf{u})\mathbf{B} \\ \mathbf{u} \otimes \mathbf{B} - \mathbf{B} \otimes \mathbf{u} + \psi\mathbf{I} \\ c_h^2\mathbf{B} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}. \tag{1.23}$$

Ce problème contient le problème précédent puisque nous remarquons aisément que si nous avons

$$\nabla \cdot \mathbf{B} = 0 \text{ et } \psi = C$$

avec C une valeur constante, le système modifié (1.23) est équivalent au système initial (1.15 page 10).

1.3 Hyperbolicité

Afin d'étudier l'hyperbolicité du système de la MHD idéale (1.15 page 10), nous commençons par l'écrire sous la forme d'un système de lois de conservation.

Soit $\mathbf{W}$ le vecteur de variables conservatives

$$\mathbf{W} = (\rho, \rho\mathbf{u}, Q, \mathbf{B})^T$$

nous pouvons alors réécrire l'équation (1.15 page 10) sous la forme d'une équation conservative

$$\frac{\partial}{\partial t}\mathbf{W} + \sum_{i=1}^{d} \frac{\partial}{\partial x_i}\mathbf{F}^i(\mathbf{W}) = 0.$$

De plus, en utilisant les conventions d'écriture

$$\partial\star = \frac{\partial}{\partial x_\star} \quad \text{et} \quad \partial_i \mathbf{F}^i(\mathbf{W}) = \sum_{i=1}^{d} \frac{\partial}{\partial x_i}\mathbf{F}^i(\mathbf{W})$$

nous avons alors l'équation

$$\partial_t \mathbf{W} + \partial_i \mathbf{F}^i(\mathbf{W}) = 0. \tag{1.24}$$

Pour tout vecteur $\mathbf{n} = (n_1, n_2, n_3) \in \mathbb{R}^3$, nous définissons le flux par le vecteur

$$\mathbf{F}(\mathbf{W}, \mathbf{n}) = \mathbf{F}(\mathbf{W}) \cdot \mathbf{n}.$$

Dans le cas de la MHD idéale ce flux numérique est donné par

$$\mathbf{F}(\mathbf{W}, \mathbf{n}) = \begin{pmatrix} \rho\mathbf{u}\cdot\mathbf{n} \\ \rho(\mathbf{u}\cdot\mathbf{n})\mathbf{u} + (p + \frac{\mathbf{B}\cdot\mathbf{B}}{2})\mathbf{n} - (\mathbf{B}\cdot\mathbf{n})\mathbf{B} \\ (Q + p + \frac{\mathbf{B}\cdot\mathbf{B}}{2})\mathbf{u}\cdot\mathbf{n} - (\mathbf{B}\cdot\mathbf{u})(\mathbf{B}\cdot\mathbf{n}) \\ (\mathbf{u}\cdot\mathbf{n})\mathbf{B} - (\mathbf{B}\cdot\mathbf{n})\mathbf{u} \end{pmatrix}. \tag{1.25}$$

La matrice jacobienne du système est définie par la dérivée par rapport au vecteur de variables conservatives, nous le notons

$$\mathbf{A}(\mathbf{W}, \mathbf{n}) = D_{\mathbf{W}}\mathbf{F}(\mathbf{W}, \mathbf{n}) \tag{1.26}$$

Afin de simplifier l'étude de cette matrice et de l'équation, nous réécrivons le problème à l'aide de ses variables primitives. Nous notons $\mathbf{Y}$ le vecteur de variables primitives du système défini par

$$\mathbf{Y} = (\rho, \mathbf{u}, p, \mathbf{B})^T.$$

Soient Y_i et W_i, les huit variables primitives et conservatives prisent dans l'ordre de lecture du vecteur. Le passage des variables conservatives vers les variables primitives se fait à travers les équations suivantes

$$\begin{aligned} Y_1 &= W_1 \\ Y_2 &= \frac{W_2}{W_1} \\ Y_3 &= \frac{W_3}{W_1} \\ Y_4 &= \frac{W_4}{W_1} \end{aligned}$$

$$Y_5 = (\gamma - 1)\left(W_5 - \frac{W_1}{2}\left(\left(\frac{W_2}{W_1}\right)^2 + \left(\frac{W_3}{W_1}\right)^2 + \left(\frac{W_4}{W_1}\right)^2 + \right) - \frac{1}{2}\left(W_6^2 + W_7^2 + W_8^2\right)\right)$$

$$\begin{aligned} Y_6 &= W_6 \\ Y_7 &= W_7 \\ Y_8 &= W_8 \end{aligned}$$

Les variables primitives sont ainsi définies par

$$\begin{aligned} \rho &= \rho \\ \mathbf{u} &= \frac{1}{\rho}(\rho\mathbf{u}) \\ p &= (\gamma - 1)\left(Q - \frac{1}{2\rho}(\rho\mathbf{u})^2 - \frac{1}{2}\mathbf{B}^2\right) \\ \mathbf{B} &= \mathbf{B} \end{aligned}$$

À l'aide de ce changement de variables, nous pouvons écrire la matrice jacobienne dans les variables primitives, nous obtenons

$$\mathbf{A}(\mathbf{Y},\mathbf{n}) = \begin{pmatrix} \mathbf{u}\cdot\mathbf{n} & \rho n_1 & \rho n_2 & \rho n_3 & 0 & 0 & 0 & 0 \\ 0 & \mathbf{u}\cdot\mathbf{n} & 0 & 0 & \frac{n_1}{\rho} & -\frac{\mathbf{B}\cdot\mathbf{n}}{\rho} & \tau_{2,1} & \tau_{3,1} \\ 0 & 0 & \mathbf{u}\cdot\mathbf{n} & 0 & \frac{n_2}{\rho} & \tau_{1,2} & -\frac{\mathbf{B}\cdot\mathbf{n}}{\rho} & \tau_{3,2} \\ 0 & 0 & 0 & \mathbf{u}\cdot\mathbf{n} & \frac{n_3}{\rho} & \tau_{1,3} & \tau_{2,3} & -\frac{\mathbf{B}\cdot\mathbf{n}}{\rho} \\ 0 & \gamma p n_1 & \gamma p n_2 & \gamma p n_3 & \mathbf{u}\cdot\mathbf{n} & (\gamma-1)\mathbf{B}\cdot\mathbf{u}n_1 & (\gamma-1)\mathbf{B}\cdot\mathbf{u}n_2 & (\gamma-1)\mathbf{B}\cdot\mathbf{u}n_3 \\ 0 & \kappa_{2,3} & B^1 n_2 & B^1 n_3 & 0 & u^2 n_2 + u^3 n_3 & -u^1 n_2 & -u^1 n_3 \\ 0 & B^2 n_1 & \kappa_{1,3} & B^2 n_3 & 0 & -u^2 n_1 & u^1 n_1 + u^3 n_3 & -u^2 n_3 \\ 0 & B^3 n_1 & B^3 n_2 & \kappa_{1,2} & 0 & -u^3 n_1 & -u^3 n_2 & u^1 n_1 + u^2 n_2 \end{pmatrix} \tag{1.27}$$

1.3.1 Dimension 1

Le système de la MHD idéale (1.15 page 10) étant invariant par rotation, nous pouvons étudier l'hyperbolicité de ce problème en dimension 1 et donc prendre le vecteur normal $\mathbf{n}_x = (1,0,0)$. Le champ magnétique $\mathbf{B} = (B^1, B^2, B^3)$ est décomposé en une composante normale $b = B^1$ et une composante tangentielle $\mathbf{b} = (B^2, B^3)$. Comme le système vérifie la condition $\nabla \cdot \mathbf{B} = 0$, en dimension 1, cela se traduit par la condition $B^1 =$ constant. Dans notre cas, nous avons donc, dans une direction fixée, la composante normale $b =$ constante. De la même façon, nous décomposons également la vitesse $\mathbf{u}$, $u = u^1$, $\mathbf{v} = (u^2, u^3)$. La constance de la composante normale du champ magnétique permet de libérer le système d'une équation. Celui-ci devient alors en dimension 1

$$\partial_t \begin{pmatrix} \rho \\ \rho u \\ \rho \mathbf{v} \\ Q \\ \mathbf{b} \end{pmatrix} + \partial_x \begin{pmatrix} \rho u \\ \rho u^2 + p + \frac{1}{2}\mathbf{b}^2 \\ \rho u \mathbf{v} - b\mathbf{b} \\ (\rho u^2 + p + \frac{1}{2}\mathbf{b}^2)u - b\mathbf{b}\cdot v \\ u\mathbf{b} - b\mathbf{v} \end{pmatrix} = 0 \tag{1.28}$$

Les vecteurs de variables conservatives et primitives s'écrivent à présent

$$\mathbf{W} = (\rho, \rho u, \rho \mathbf{v}, Q, \mathbf{b})^T, \text{ et } \mathbf{Y} = (\rho, u, \mathbf{v}, Q, \mathbf{b})^T.$$

Enfin, le système conservatif s'écrit en dimension 1 et dans les variables primitives sous la forme

$$\partial t \mathbf{Y} + \mathbf{A}(\mathbf{Y}, \mathbf{n}_x)\partial x \mathbf{Y} = 0,$$

avec

$$\mathbf{A}(\mathbf{Y},\mathbf{n}_x)=\begin{pmatrix} u & \rho & 0 & 0 & 0 & 0 & 0\\ 0 & u & 0 & 0 & \frac{1}{\rho} & \frac{B^2}{\rho} & \frac{B^3}{\rho}\\ 0 & 0 & u & 0 & 0 & -\frac{b}{\rho} & 0\\ 0 & 0 & 0 & u & 0 & 0 & -\frac{b}{\rho}\\ 0 & \gamma p & 0 & 0 & u & 0 & 0\\ 0 & B^2 & -b & 0 & 0 & u & 0\\ 0 & B^3 & 0 & -b & 0 & 0 & u \end{pmatrix}$$

Les valeurs propres de ce système se calculent analytiquement [49]. Nous définissons la vitesse d'Alfvèn

$$c_A=\frac{b}{\sqrt{\rho}},$$

la vitesse magnétoacoustique lente

$$c_s=\sqrt{\frac{1}{2}\left(\frac{\mathbf{B}\cdot\mathbf{B}}{\rho}+a^2\right)-\sqrt{\frac{1}{4}\left(\frac{\mathbf{B}\cdot\mathbf{B}}{\rho}+a^2\right)^2-\frac{a^2b^2}{\rho}}},$$

et la vitesse magnétoacoustique rapide

$$c_f=\sqrt{\frac{1}{2}\left(\frac{\mathbf{B}\cdot\mathbf{B}}{\rho}+a^2\right)+\sqrt{\frac{1}{4}\left(\frac{\mathbf{B}\cdot\mathbf{B}}{\rho}+a^2\right)^2-\frac{a^2b^2}{\rho}}},$$

où $b=\mathbf{B}\cdot\mathbf{n}$ et a représente la vitesse du son donnée par

$$a=\sqrt{\frac{\gamma p}{\rho}}.$$

Les valeurs propres sont alors données, de la plus petite à la plus grande, par

$$\begin{aligned}&\lambda_1=\mathbf{u}\cdot\mathbf{n}-c_f \quad \lambda_2=\mathbf{u}\cdot\mathbf{n}-c_A \quad \lambda_3=\mathbf{u}\cdot\mathbf{n}-c_s\\ &\lambda_4=\mathbf{u}\cdot\mathbf{n}\\ &\lambda_5=\mathbf{u}\cdot\mathbf{n}+c_s \quad \lambda_6=\mathbf{u}\cdot\mathbf{n}+c_A \quad \lambda_7=\mathbf{u}\cdot\mathbf{n}+c_f.\end{aligned}$$

Elles sont toutes réelles et positives. De plus, dans le cas où le champ magnétique $\mathbf{B}$ est non nul, ces valeurs sont distinctes et associées aux vecteurs propres.

$$
\begin{aligned}
r_1 &= (-\rho, c_f, \frac{c_f B^2}{b\left(1-c_f^2/c_A^2\right)}, \frac{c_f B^3}{b\left(1-c_f^2/c_A^2\right)}, -\gamma p, \frac{B^2}{c_A^2/c_f^2-1}, \frac{B^3}{c_A^2/c_f^2-1}) \\
r_2 &= (0, 0, B^3, -B^2, 0, \sqrt{\rho}B^3, -\sqrt{\rho}B^2), \\
r_3 &= (-\rho, c_s, \frac{c_s B^2}{b\left(1-c_s^2/c_A^2\right)}, \frac{c_s B^3}{b\left(1-c_s^2/c_A^2\right)}, -\gamma p, \frac{B^2}{c_A^2/c_s^2-1}, \frac{B^3}{c_A^2/c_s^2-1}), \\
r_4 &= (1, 0, 0, 0, 0, 0, 0), \\
r_5 &= (-\rho, -c_s, \frac{-c_s B^2}{b\left(1-c_s^2/c_A^2\right)}, \frac{-c_s B^3}{b\left(1-c_s^2/c_A^2\right)}, -\gamma p, \frac{B^2}{c_A^2/c_s^2-1}, \frac{B^3}{c_A^2/c_s^2-1}), \\
r_6 &= (0, 0, B^3, -B^2, 0, -\sqrt{\rho}B^3, \sqrt{\rho}B^2), \\
r_7 &= (-\rho, -c_f, \frac{-c_f B^2}{b\left(1-c_f^2/c_A^2\right)}, \frac{-c_f B^3}{b\left(1-c_f^2/c_A^2\right)}, -\gamma p, \frac{B^2}{c_A^2/c_f^2-1}, \frac{B^3}{c_A^2/c_f^2-1}).
\end{aligned}
$$

Ces vecteurs propres étant distincts, nous en concluons que la matrice du système est diagonalisable et que le problème est donc hyperbolique puisque le système est un système conservatif et que la condition d'hyperbolicité est vérifiée.

1.3.2 Dimension $\geqslant 2$

En dimension 1, nous avons vu que la condition (1.17 page 10) est vérifiée si le la composante du champ magnétique $\mathbf{B}$ dans la direction considérée est constante. Lorsque la dimension est supérieure à 1, cet argument n'est plus suffisant pour permettre de conserver la condition.

L'écriture du système MHD Powell (1.20 page 11) nous amène à considérer un terme source S dans l'équation (1.24 page 13) la transformant ainsi en un problème non conservatif que nous écrivons

$$\partial_t \mathbf{W} + \partial_i \mathbf{F}^i(\mathbf{W}) = \mathbf{S}.$$

où le terme source est donné par

$$\mathbf{S} = -\nabla \cdot \mathbf{B} \begin{pmatrix} 0 \\ \mathbf{B} \\ \mathbf{u} \cdot \mathbf{B} \\ \mathbf{u} \end{pmatrix}.$$

La matrice jacobienne $\mathbf{A}(\mathbf{Y}, \mathbf{n})$ est quant à elle remplacée par la matrice

$$\mathbf{A}'(\mathbf{Y}, \mathbf{n}) = \begin{pmatrix} \mathbf{u}\cdot\mathbf{n} & \rho n_1 & \rho n_2 & \rho n_3 & 0 & 0 & 0 & 0 \\ 0 & \mathbf{u}\cdot\mathbf{n} & 0 & 0 & \frac{n_1}{\rho} & -\frac{\mathbf{B}\cdot\mathbf{n}}{\rho} & \tau_{2,1} & \tau_{3,1} \\ 0 & 0 & \mathbf{u}\cdot\mathbf{n} & 0 & \frac{n_2}{\rho} & \tau_{1,2} & -\frac{\mathbf{B}\cdot\mathbf{n}}{\rho} & \tau_{3,2} \\ 0 & 0 & 0 & \mathbf{u}\cdot\mathbf{n} & \frac{n_3}{\rho} & \tau_{1,3} & \tau_{2,3} & -\frac{\mathbf{B}\cdot\mathbf{n}}{\rho} \\ 0 & \gamma p n_1 & \gamma p n_2 & \gamma p n_3 & \mathbf{u}\cdot\mathbf{n} & (\gamma-1)\mathbf{B}\cdot\mathbf{u} n_1 & (\gamma-1)\mathbf{B}\cdot\mathbf{u} n_2 & (\gamma-1)\mathbf{B}\cdot\mathbf{u} n_3 \\ 0 & \kappa_{2,3} & B^1 n_2 & B^1 n_3 & 0 & \mathbf{u}\cdot\mathbf{n} & -u^1 n_2 & -u^1 n_3 \\ 0 & B^2 n_1 & \kappa_{1,3} & B^2 n_3 & 0 & -u^2 n_1 & \mathbf{u}\cdot\mathbf{n} & -u^2 n_3 \\ 0 & B^3 n_1 & B^3 n_2 & \kappa_{1,2} & 0 & -u^3 n_1 & -u^3 n_2 & \mathbf{u}\cdot\mathbf{n} \end{pmatrix}$$

Suivant chaque direction une des lignes de la matrice $\mathbf{A}(\mathbf{Y}, \mathbf{n})$ parmi les trois dernières est nulle. De ce fait, la matrice est singulière. La matrice $\mathbf{A}'(\mathbf{Y}, \mathbf{n})$ donnée par les modifications de Powell pallie ce problème en apportant les modifications sur les équations du champ magnétique. De plus, ce système possède sept de ses valeurs propres et de ses vecteurs propres qui sont identiques aux sept valeurs propres et vecteurs propres du problème en dimension 1. Les détails concernant les calculs de la huitième valeur propre u et de son vecteur propre associé $(0, 0, 0, 0, 0, 1, 0, 0)$ sont donnés dans [54].

Dans le cas de la modification du système par la méthode de Dedner, nous définissons maintenant le vecteur de variables conservatives $\mathbf{W}$ par

$$\mathbf{W} = (\rho, \rho\mathbf{u}, Q, \mathbf{B}, \psi)^T$$

et nous pouvons donc réécrire l'équation (1.23 page 12) sous la forme d'une équation conservative

$$\partial_t \mathbf{W} + \partial_i \mathbf{F}^i(\mathbf{W}) = 0 \tag{1.29}$$

pour laquelle le flux numérique est maintenant donné par

$$\mathbf{F}(\mathbf{W},\mathbf{n}) = \begin{pmatrix} \rho\mathbf{u}\cdot\mathbf{n} \\ \rho(\mathbf{u}\cdot\mathbf{n})\mathbf{u} + (p + \frac{\mathbf{B}\cdot\mathbf{B}}{2})\mathbf{n} - (\mathbf{B}\cdot\mathbf{n})\mathbf{B} \\ (Q + p + \frac{\mathbf{B}\cdot\mathbf{B}}{2})\mathbf{u}\cdot\mathbf{n} - (\mathbf{B}\cdot\mathbf{u})(\mathbf{B}\cdot\mathbf{n}) \\ (\mathbf{u}\cdot\mathbf{n})\mathbf{B} - (\mathbf{B}\cdot\mathbf{n})\mathbf{u} + \psi\mathbf{n} \\ c_h^2\mathbf{B}\cdot\mathbf{n} \end{pmatrix}. \tag{1.30}$$

et la matrice jacobienne en variables primitives est donnée par

$$\mathbf{A}(\mathbf{Y},\mathbf{n}) = \begin{pmatrix} \mathbf{u}\cdot\mathbf{n} & \rho n_1 & \rho n_2 & \rho n_3 & 0 & 0 & 0 & 0 & 0 \\ 0 & \mathbf{u}\cdot\mathbf{n} & 0 & 0 & \frac{n_1}{\rho} & -\frac{\mathbf{B}\cdot\mathbf{n}}{\rho} & \tau_{2,1} & \tau_{3,1} & 0 \\ 0 & 0 & \mathbf{u}\cdot\mathbf{n} & 0 & \frac{n_2}{\rho} & \tau_{1,2} & -\frac{\mathbf{B}\cdot\mathbf{n}}{\rho} & \tau_{3,2} & 0 \\ 0 & 0 & 0 & \mathbf{u}\cdot\mathbf{n} & \frac{n_3}{\rho} & \tau_{1,3} & \tau_{2,3} & -\frac{\mathbf{B}\cdot\mathbf{n}}{\rho} & 0 \\ 0 & \gamma p n_1 & \gamma p n_2 & \gamma p n_3 & \mathbf{u}\cdot\mathbf{n} & (\gamma-1)\mathbf{B}\cdot\mathbf{u}n_1 & (\gamma-1)\mathbf{B}\cdot\mathbf{u}n_2 & (\gamma-1)\mathbf{B}\cdot\mathbf{u}n_3 & (1-\gamma)\mathbf{B}\cdot\mathbf{n} \\ 0 & \kappa_{2,3} & B^1 n_2 & B^1 n_3 & 0 & u^2 n_2 + u^3 n_3 & -u^1 n_2 & -u^1 n_3 & n_1 \\ 0 & B^2 n_1 & \kappa_{1,3} & B^2 n_3 & 0 & -u^2 n_1 & u^1 n_1 + u^3 n_3 & -u^2 n_3 & n_2 \\ 0 & B^3 n_1 & B^3 n_2 & \kappa_{1,2} & 0 & -u^3 n_1 & -u^3 n_2 & u^1 n_1 + u^2 n_2 & n_3 \\ 0 & 0 & 0 & 0 & 0 & c h^2 n_1 & c h^2 n_2 & c h^2 n_3 & 0 \end{pmatrix}$$

Afin de nous assurer de l'hyperbolicité de ce système, nous effectuons à nouveau l'étude dans la direction fixée $\mathbf{n}_x = (1,0,0)$. Nous rappelons que cela est possible puisque le système est invariant par rotation. Nous souhaitons donc étudier le système propre de la matrice

$$\mathbf{A}(\mathbf{Y},n_x) = \begin{pmatrix} u^1 & \rho & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & u^1 & 0 & 0 & \frac{1}{\rho} & -\frac{B^1}{\rho} & -\frac{B^2}{\rho} & -\frac{B^3}{\rho} & 0 \\ 0 & 0 & u^1 & 0 & 0 & -\frac{B^2}{\rho} & -\frac{B^1}{\rho} & 0 & 0 \\ 0 & 0 & 0 & u^1 & 0 & -\frac{B^3}{\rho} & 0 & -\frac{B^1}{\rho} & 0 \\ 0 & \gamma p & 0 & 0 & u^1 & (\gamma-1)\mathbf{B}\cdot\mathbf{u} & 0 & 0 & (1-\gamma)B^1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & B^2 & -B^1 & 0 & 0 & -u^2 & u^1 & 0 & 0 \\ 0 & B^3 & 0 & -B^1 & 0 & -u^3 & 0 & u^1 & 0 \\ 0 & 0 & 0 & 0 & 0 & c h^2 & 0 & 0 & 0 \end{pmatrix} \tag{1.31}$$

Nous remarquons que les variables B^1 et ψ sont découplé du système. Nous pouvons donc, en retirant les sixième et neuvième ligne et colonne de la matrice, retrouver le système de dimension sept dont les valeurs et vecteurs propres λ_i et r_i ont été donnés précédemment. Le système propre de la matrice (1.31) est donc composé de ces sept valeurs propres associées, respectivement, aux vecteurs propres

$$
\begin{aligned}
r'_1 &= (-\rho, c_f, \frac{c_f B^2}{b\left(1-c_f^2/c_A^2\right)}, \frac{c_f B^3}{b\left(1-c_f^2/c_A^2\right)}, -\gamma p, 0, \frac{B^2}{c_A^2/c_f^2-1}, \frac{B^3}{c_A^2/c_f^2-1}, 0)\\
r'_2 &= (0, 0, B^3, -B^2, 0, 0, \sqrt{\rho}B^3, -\sqrt{\rho}B^2, 0),\\
r'_3 &= (-\rho, c_s, \frac{c_s B^2}{b\left(1-c_s^2/c_A^2\right)}, \frac{c_s B^3}{b\left(1-c_s^2/c_A^2\right)}, -\gamma p, 0, \frac{B^2}{c_A^2/c_s^2-1}, \frac{B^3}{c_A^2/c_s^2-1}, 0),\\
r'_4 &= (1, 0, 0, 0, 0, 0, 0, 0, 0),\\
r'_5 &= (-\rho, -c_s, \frac{-c_s B^2}{b\left(1-c_s^2/c_A^2\right)}, \frac{-c_s B^3}{b\left(1-c_s^2/c_A^2\right)}, -\gamma p, 0, \frac{B^2}{c_A^2/c_s^2-1}, \frac{B^3}{c_A^2/c_s^2-1}, 0),\\
r'_6 &= (0, 0, B^3, -B^2, 0, 0, -\sqrt{\rho}B^3, \sqrt{\rho}B^2, 0),\\
r'_7 &= (-\rho, -c_f, \frac{-c_f B^2}{b\left(1-c_f^2/c_A^2\right)}, \frac{-c_f B^3}{b\left(1-c_f^2/c_A^2\right)}, -\gamma p, 0, \frac{B^2}{c_A^2/c_f^2-1}, \frac{B^3}{c_A^2/c_f^2-1}, 0).
\end{aligned}
$$

Les deux valeurs propres supplémentaires de ce système sont données par $\lambda_0 = -c_h$ et $\lambda_8 = c_h$. La constante c_h étant choisie suffisamment grande, ces deux nouvelles valeurs propres sont distinctes des précédentes. Le problème de la MHD avec correction de divergence (1.23 page 12) respecte donc toujours la condition d'hyperbolicité.

Déterminer les vecteurs propres r'_0 et $r'1$ associés à ces deux nouvelles valeurs propres peut s'avérer long et fastidieux. Nous proposons ici de déterminer ces vecteurs dans une autre base. Nous commençons par appliquer une permutation des lignes et colonne de la matrice $\mathbf{A}(\mathbf{Y}, n_x)$ afin de pouvoir l'écrire sous la forme de la matrice par blocs

$$
\mathbf{A}(\mathbf{Y}, n_x) = \begin{pmatrix} \mathbf{C} & \widetilde{\mathbf{C}} \\ 0 & \mathbf{D} \end{pmatrix}
$$

où les blocs sont donnés par

$$
\mathbf{C} = \begin{pmatrix}
u^1 & \rho & 0 & 0 & 0 & 0 & 0 \\
0 & u^1 & 0 & 0 & \frac{1}{\rho} & -\frac{B^2}{\rho} & -\frac{B^3}{\rho} \\
0 & 0 & u^1 & 0 & 0 & -\frac{B^1}{\rho} & 0 \\
0 & 0 & 0 & u^1 & 0 & 0 & -\frac{B^1}{\rho} \\
0 & \gamma p & 0 & 0 & u^1 & 0 & 0 \\
0 & B^2 & -B^1 & 0 & 0 & u^1 & 0 \\
0 & B^3 & 0 & -B^1 & 0 & 0 & u^1
\end{pmatrix},
$$

$$\widetilde{\mathbf{C}} = \begin{pmatrix} 0 & 0 \\ -\frac{B^1}{\rho} & 0 \\ -\frac{B^2}{\rho} & 0 \\ -\frac{B^3}{\rho} & 0 \\ (\gamma-1)\mathbf{B}\cdot\mathbf{u} & (1-\gamma)B^1 \\ -u^2 & 0 \\ -u^3 & 0 \end{pmatrix}$$

et

$$\mathbf{D} = \begin{pmatrix} 0 & 1 \\ c_h^2 & 0 \end{pmatrix}.$$

Les valeurs propres de la matrice $\mathbf{D}$ sont $\mu_1 = -c_h$ et $\mu_2 = c_h$ et sont associées aux vecteurs propres

$$\begin{aligned} s_1 &= (1, -c_h) \\ s_2 &= (1, c_h), \end{aligned}.$$

Nous utilisons la matrice de passage par bloc

$$\mathbf{P} = \begin{pmatrix} \mathbf{P}_1 & 0 \\ 0 & \mathbf{P}_2 \end{pmatrix},$$

où $\mathbf{P}_1$ et $\mathbf{P}_2$ sont respectivement composées des vecteurs propres associés aux matrices $\mathbf{C}$ et $\mathbf{D}$, pour écrire la matrice $\mathbf{A}(\mathbf{Y}, n_x)$ dans la nouvelle base. Nous avons alors la matrice

$$\mathbf{B}(\mathbf{Y}, n_x) = \begin{pmatrix} \mathbf{C}' & \widetilde{\mathbf{C}}' \\ 0 & \mathbf{D}' \end{pmatrix}.$$

Les matrices $\mathbf{C}'$ et $\mathbf{D}'$, sont des matrices diagonales composées des valeurs propres des matrices $\mathbf{C}$ et $\mathbf{D}$ et

$$\widetilde{\mathbf{C}}' = \begin{pmatrix} 0 & 0 \\ -\frac{B^1}{\rho} & -\frac{B^1}{\rho} \\ -\frac{B^2}{\rho} & -\frac{B^2}{\rho} \\ -\frac{B^3}{\rho} & -\frac{B^3}{\rho} \\ (\gamma-1)\mathbf{B}\cdot\mathbf{u} + c_h(1-\gamma)B^1 & (\gamma-1)\mathbf{B}\cdot\mathbf{u} - c_h(1-\gamma)B^1 \\ -u^2 & -u^2 \\ -u^3 & -u^3 \end{pmatrix}.$$

Dans cette nouvelle base, les vecteurs propres de $\mathbf{B}(\mathbf{Y}, n_x)$, sont déterminé en résolvant les systèmes linéaires

$$(\mathbf{B} + c_h\mathbf{I})\mathbf{x}_1 = 0 \quad \text{et} \quad (\mathbf{B} - c_h\mathbf{I})\mathbf{x}_2 = 0.$$

Nous obtenons les deux vecteurs

$$\mathbf{x}_1 = \begin{pmatrix} 0 \\ \frac{B^1}{\rho(\lambda_2 - c_h)} \\ \frac{B^2}{\rho(\lambda_3 - c_h)} \\ \frac{B^3}{\rho(\lambda_4 - c_h)} \\ \frac{(\gamma-1)\mathbf{B}\cdot\mathbf{u} + c_h(1-\gamma)B^1}{(\lambda_5 - c_h)} \\ \frac{u^2}{(\lambda_6 - c_h)} \\ \frac{u^2}{(\lambda_7 - c_h)} \\ 1 \\ 0 \end{pmatrix} \quad \text{et} \quad \mathbf{x}_2 = \begin{pmatrix} 0 \\ \frac{B^1}{\rho(\lambda_2 + c_h)} \\ \frac{B^2}{\rho(\lambda_3 + c_h)} \\ \frac{B^3}{\rho(\lambda_4 + c_h)} \\ \frac{(\gamma-1)\mathbf{B}\cdot\mathbf{u} - c_h(1-\gamma)B^1}{(\lambda_5 + c_h)} \\ \frac{u^2}{(\lambda_6 + c_h)} \\ \frac{u^3}{(\lambda_7 + c_h)} \\ 0 \\ 1 \end{pmatrix}.$$

Finalement, à l'aide des formules de changement de base, nous trouvons les vecteurs propres de $\mathbf{A}(\mathbf{Y}, n_x)$

$$r_0 = \sum_{i=1}^{7} x_{1,i}\mathbf{r}"_i + s"_1$$

et

$$r_8 = \sum_{i=1}^{7} x_{2,i}\mathbf{r}"_i + s"_2.$$

nous pouvons donc exprimer de façon presque analytique les vecteurs propres de la matrice $\mathbf{A}(\mathbf{Y}, n_x)$.

1.4 Conclusion

Dans ce premier chapitre, nous avons défini le problème de la magnétohydrodynamique. Pour cela, nous avons commencé par rappeler les deux systèmes que nous avons couplés : le système hydrodynamique et le système électromagnétique. Une des conditions du problème de la MHD consiste à conserver une divergence du champ magnétique nulle au cours de la simulation. Afin d'éviter l'apparition de valeurs numériques non nulles, nous avons présenté les méthodes proposées par Powell et par Dedner. La méthode de Powell consiste à ajouter un terme magnétique dans le second membre du problème. La méthode de Dedner consiste à nettoyer la divergence à l'aide d'une équation et d'une variable supplémentaires. Nous avons enfin démontré l'hyperbolicité du problème de la MHD avec ces deux méthodes de contrôle de la divergence.

Chapitre 2

Étude des équations en dimension 1

> To err is human, but to really foul things up you need a computer
>
> Paul R. Ehrlich

Sommaire

2.1 Schéma de volumes finis 25
2.1.1 Méthode Galerkin Discontinu (GD) 25
2.1.2 Formulation générale de la méthode des volumes finis 27
2.1.3 Convergence 29
2.1.4 Limiteur de pentes 30
2.2 Flux numérique 32
2.2.1 Flux de Rusanov 32
2.2.2 Flux VFRoe 33
2.2.3 Schéma décentré *upwind* 35
2.3 Ordre et coût 40
2.4 Résultats 40
2.5 Conclusion 43

2.1 Schéma de volumes finis

2.1.1 Méthode Galerkin Discontinu (GD)

Afin de définir la méthode de Galerkin Discontinue, nous considérons $\mathcal{T}$ un maillage de $\Omega \subset \mathbb{R}^d$ constitué d'un nombre fini de cellules $K_i \subset \mathcal{T}$ vérifiant les conditions suivantes

1. $\forall i, j \quad i \neq j, \quad K_i \cap K_j = \emptyset$
2. $\bigcup_i \overline{K_i} = \overline{\mathcal{T}}$.

La méthode Galerkin Discontinu est une méthode se situant entre la méthode des éléments finis et la méthode des volumes finis. Elle consiste à décomposer la solution discrète sur une base de fonctions φ_i^K dans chaque cellule K.

La solution de (1.24 page 13) est alors approchée dans une cellule K par

$$\mathbf{W}(\mathbf{x}, t) = \mathbf{W}_K(\mathbf{x}, t) = \sum_j \mathbf{W}_K^j(t)\varphi_j^K(\mathbf{x}), \quad \mathbf{x} \in K. \tag{2.1}$$

Le schéma d'approximation de Galerkin Discontinu pour notre problème s'obtient, pour chaque cellule K, en multipliant l'équation (1.24 page 13) par la fonction test φ_i^K et en intégrant sur K

$$\forall i, \quad \int_K \partial_t \mathbf{W}\varphi_i^K + \int_K \nabla \cdot \mathbf{F}(\mathbf{W})\varphi_i^K = \int_K \mathbf{S}\varphi_i^K. \tag{2.2}$$

La méthode de Galerkin-Discontinue tient son nom du fait qu'aucune continuité n'est imposée à la solution faible. Le flux du schéma est alors approché à l'aide d'un flux numérique que nous notons $\mathbf{F}(\mathbf{W}_L, \mathbf{W}_R, \mathbf{n}_{LR})$. Les champs $\mathbf{W}_L$ et $\mathbf{W}_R$ représentent les champs de part et d'autre d'une discontinuité. Par convention, nous considérons toujours la discontinuité orientée pour que le vecteur normal unitaire de celle-ci soit orienté de la cellule L à la cellule R. Ce vecteur est noté $\mathbf{n}_{LR}$ (voir la figure 2.1).

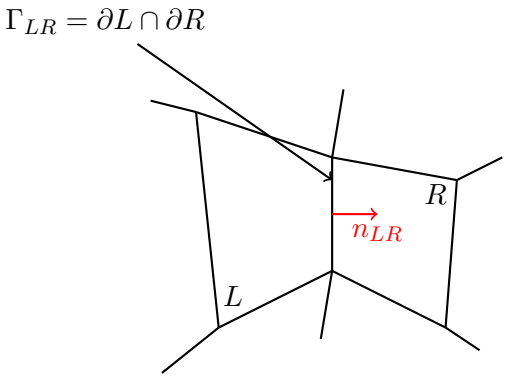

FIGURE 2.1 – Règle de nommage.

Afin de garantir le caractère conservatif, nous imposons à ce flux numérique de respecter l'égalité

$$\mathbf{F}(\mathbf{W}_L, \mathbf{W}_R, \mathbf{n}_{LR}) = -\mathbf{F}(\mathbf{W}_R, \mathbf{W}_L, \mathbf{n}_{RL}). \tag{2.3}$$

De plus, nous imposons également au schéma d'être consistant, c'est-à-dire de vérifier l'égalité

$$\mathbf{F}(\mathbf{W}, \mathbf{W}, \mathbf{n}) = \mathbf{F}(\mathbf{W}, \mathbf{n}). \tag{2.4}$$

En utilisant alors, la formule de Green pour l'intégration par partie sur l'équation (2.2 page précédente), nous obtenons

$$\forall K, \forall i, \quad \int_K \partial_t \mathbf{W} \varphi_i^K - \int_K \mathbf{F}(\mathbf{W}, \nabla \varphi_i^K) + \int_{\partial K} \mathbf{F}(\mathbf{W}_L, \mathbf{W}_R, \mathbf{n}_{LR}) \varphi_i^K = \int_K \mathbf{S} \varphi_i^K. \tag{2.5}$$

Nous pouvons alors introduire le développement (5.10 page 90) dans l'équation (5.11 page 90). L'approximation GD devient alors un système d'équations différentielles ordinaires satisfaites par les coefficients $\mathbf{W}_K^j(t)$ sur les bases locales. Dans ce chapitre, nous nous intéressons plus particulièrement à la méthode des volumes finis qui est un cas particulier de la méthode de Galerkin Discontinue. Les détails concernant la résolution numérique GD seront donnés dans le chapitre 5 page 85.

2.1.2 Formulation générale de la méthode des volumes finis

La méthode de Galerkin-Discontinue pour la résolution des équations de conservations approche la solution à l'aide d'une fonction polynomiale par morceaux. Dans le cas où ces polynômes sont de degrés 0, la solution est alors approchée par une fonction constante par morceaux, c'est la méthode des volumes finis.

La solution du problème (1.24 page 13) est approchée au temps t par

$$\mathbf{W}(\mathbf{x}, t) = \sum_{K \in \mathcal{T}} \mathbf{W}_K(\mathbf{x}, t) \mathbb{1}_K(\mathbf{x})$$

$$\mathcal{V}_h = \{\mathbf{W} |\ \forall K \in \mathcal{T},\ \mathbf{W}_K \in \mathbb{R}\}.$$

Le schéma de volumes finis est alors donné pour toute fonction test $\mathbf{V} \in \mathcal{V}_h$ par

$$\int_{\mathbf{x}\in\mathbb{R}^d} \partial_t \mathbf{W} \cdot \mathbf{V} + \sum_{K\in\mathcal{T}} \int_{\partial_K} \mathbf{F}(\mathbf{W}_L, \mathbf{W}_R, \mathbf{n}_{LR}) \cdot \nabla \mathbf{V} - \int_K \mathbf{F}^i(\mathbf{W}) \cdot \mathbf{V} = 0. \tag{2.6}$$

Dans ce chapitre, nous nous plaçons dans le cas simplifié de la dimension $d = 1$. De plus la fonction test $\mathbf{V}$ étant constante, nous avons

$$\sum_{K\in\mathcal{T}} \int_{\partial_K} \mathbf{F}(\mathbf{W}_L, \mathbf{W}_R, \mathbf{n}_{LR}) \cdot \nabla \mathbf{V} = 0.$$

De plus, nous considérons un maillage cartésien régulier de l'espace $\Omega =]x_{\min}, x_{\max}[$. On pose

$$\Delta x = \frac{x_{\max} - x_{\min}}{N_x}$$

où N_x représente le nombre de mailles du maillage dans la direction x. Nous pouvons ainsi définir pour tout $i \in \{1, \ldots, N_x\}$

$$x_i = x_{\min} + (i - 1/2)\Delta x.$$

le centre de la cellule

$$C_i =]x_{i-1/2}, x_{i+1/2}[.$$

et on note $t_n = n\Delta t$. La solution du problème de la MHD est alors approchée à l'instant t_n, en chaque cellule par

$$\mathbf{W}_i^n \simeq \mathbf{W}(\mathbf{x}_i, t_n)$$

Sur la grille régulière en dimension 1 sur laquelle nous nous sommes fixés, le schéma de volumes finis s'écrit de la forme

$$\mathbf{W}_i^{n+1} = \mathbf{W}_i^n - \frac{\Delta t}{\Delta x} \left(\mathbf{F}(\mathbf{W}_i^n, \mathbf{W}_{i+1}^n, \mathbf{n}_x) - \mathbf{F}(\mathbf{W}_{i-1}^n, \mathbf{W}_i^n, \mathbf{n}_x) \right) \tag{2.7}$$

2.1.3 Convergence

La méthode des volumes finis est ainsi déterminée par le choix du flux numérique. Nous verrons dans ce chapitre quelques exemples de flux numériques qui peuvent être utilisés. Cependant, avant de choisir un flux numérique, il est important de connaître quelles sont les conditions nécessaires qui doivent être respectées par le flux numérique afin que le schéma soit convergent, c'est-à-dire qu'il s'approche de la solution exacte du problème à mesure que le maillage se raffine. Dans un premier temps, le flux numérique doit être *consistant* (2.4 page 27). En dimension 1 la condition de consistance devient

$$\mathbf{F}(\mathbf{W}, \mathbf{W}, \mathbf{n}) = \mathbf{F}^i n_i. \tag{2.8}$$

Ensuite, le schéma doit être *stable*, c'est-à-dire que l'erreur faite au cours des pas de temps de la simulation ne peut pas augmenter de manière significative. Nous introduisons alors la condition CFL (Courant-Friedrichs-Lewy) qui doit nécessairement être respectée pour que le schéma soit stable. Afin de comprendre cette notion, plaçons-nous dans le cas d'une équation de transport en dimension 1

$$u_t + a u_x = 0$$

où $a > 0$ représente la vitesse de transport. D'après (2.7 page précédente), la solution dans une cellule C_i à un instant t ne dépend que de la solution à l'instant précédant dans les cellules directement voisines. Sur la figure 2.2, nous représentons le cas où l'information en un pas de temps est propagée à une distance inférieure à celle d'un pas de temps

$$a\Delta t < \Delta x.$$

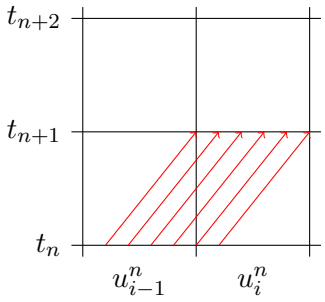

FIGURE 2.2 – Propagtion de l'information à une distance inférieure à Δt.

Dans ce cas, nous voyons que le calcul du flux numérique (2.7 page précédente) ne

dépendant que des cellules directement voisines est possible puisqu'en chaque cellule la solution à l'état précédent ne dépend que du voisinage direct. En revanche, si le pas de temps est tel que

$$a\Delta t > \Delta x.$$

nous sommes alors dans la configuration de la figure 2.3 où nous remarquons que la solution à l'instant t_{n+1} dépend également de l'état de cellules plus lointaines. Dans ces conditions, le flux numérique (2.7 page 28) ne peut pas être stable.

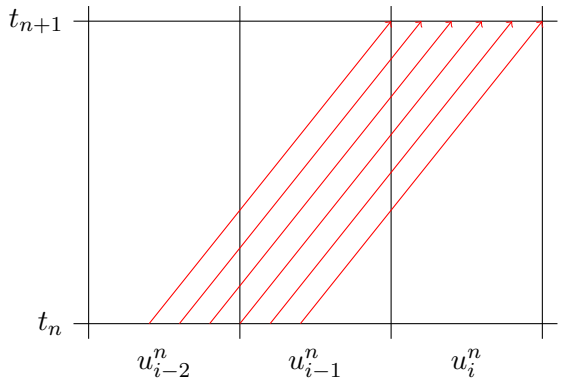

FIGURE 2.3 – Propagtion de l'information à une distance supérieure à Δt.

Dans la section 2.2.3 page 35, nous montrons que dans le cas de la MHD, cette condition de stabilité se traduit par

$$\Delta t \leqslant \frac{\Delta x}{\lambda_{\max}}. \tag{2.9}$$

où $\lambda_{\max}$ représente la valeur propre maximale du système

2.1.4 Limiteur de pentes

Le schéma (2.7 page 28) étant un schéma d'ordre 1, il ne permet donc pas d'obtenir une approximation précise de la solution. En effet, ces schémas provoquent une diffusion de la solution sur les chocs et dégradent ainsi les solutions. La solution est alors d'augmenter l'ordre de la méthode en utilisant les approximations des dérivés en espace et en temps de **W**

$$\boldsymbol{\zeta}_i^n \simeq \partial_x \mathbf{W}(\mathbf{x}_i, t_n), \quad \mathbf{r}_i^n \simeq \partial_t \mathbf{W}(\mathbf{x}_i, t_n).$$

Pour cela, nous utilisons la méthode MUSCL de Van Leer [64] qui propose d'approcher la dérivée spatiale à l'aide de la fonction *minmod*, aussi appelée limiteur de pentes, définit par

$$\text{minmod}(a,b,c) = \begin{cases} \min(a,b,c) & \text{si} \quad a,b,c > 0, \\ \max(a,b,c) & \text{si} \quad a,b,c < 0, \\ 0 & \text{sinon.} \end{cases}$$

Nous avons alors une approximation pour la dérivée en espace, valant

$$\boldsymbol{\zeta}_i^n = \text{minmod}\left(\frac{\mathbf{W}_{i+1}^n - \mathbf{W}_i^n}{\Delta x}, \frac{\mathbf{W}_{i+1}^n - \mathbf{W}_{i-1}^n}{2\Delta x}, \frac{\mathbf{W}_i^n - \mathbf{W}_{i-1}^n}{\Delta x}\right),$$

et une approximation pour la dérivée en temps donnée par

$$\mathbf{r}_i^n = -A(\mathbf{W}_i^n, \mathbf{n}_x)\boldsymbol{\zeta}_i^n.$$

Le schéma (2.7 page 28) à l'ordre 2 devient

$$\mathbf{W}_i^{n+1} = \mathbf{W}_i^n - \frac{\Delta t}{\Delta x}\left(\mathbf{F}(\mathbf{W}_{i+1/2-}^{n+1/2}, \mathbf{W}_{i+1/2+}^{n+1/2}, \mathbf{n}_x) - \mathbf{F}(\mathbf{W}_{i-1/2-}^{n+1/2}, \mathbf{W}_{i-1/2+}^{n+1/2}, \mathbf{n}_x)\right) \quad (2.10)$$

où les états gauche et droit du flux numérique dans la direction x sont calculés à l'aide des formules

$$\mathbf{W}_{i+1/2-}^{n+1/2} = \mathbf{W}_{i,j} + \boldsymbol{\zeta}_i^n \tfrac{\Delta x}{2} + r_i^n \tfrac{\Delta t}{2} \qquad \mathbf{W}_{i-1/2+}^{n+1/2} = \mathbf{W}_i - \boldsymbol{\zeta}_i^n \tfrac{\Delta x}{2} + r_i^n \tfrac{\Delta t}{2}$$

$$\mathbf{W}_{i+1/2+}^{n+1/2} = \mathbf{W}_{i+1} - \boldsymbol{\zeta}_{i+1}^n \tfrac{\Delta x}{2} + r_i^n \tfrac{\Delta t}{2} \qquad \mathbf{W}_{i-1/2-}^{n+1/2} = \mathbf{W}_{i-1} + \boldsymbol{\zeta}_{i-1}^n \tfrac{\Delta x}{2} + r_i^n \tfrac{\Delta t}{2}$$

L'utilisation de la fonction minmod fait sens puisque l'utilisation d'une même et unique approximation de la dérivée par exemple l'approximation décentrée *upwind*

$$\boldsymbol{\zeta}_i^n \simeq \frac{\mathbf{W}_i^n - \mathbf{W}_{i-1}^n}{\Delta x}$$

provoque l'apparition d'oscillations au niveau des chocs rendant alors le schéma instable. Une amélioration de ce limiteur [45, 55, 64] est possible en comparant par exemple, la moyenne centrale avec deux fois les pentes de gauche et de droite. Nous avons alors

$$\boldsymbol{\varsigma}_i^n = \text{minmod}\left(2\frac{\mathbf{W}_{i+1}^n - \mathbf{W}_i^n}{\Delta x}, \frac{\mathbf{W}_{i+1}^n - \mathbf{W}_{i-1}^n}{2\Delta x}, 2\frac{\mathbf{W}_i^n - \mathbf{W}_{i-1}^n}{\Delta x}\right),$$

2.2 Flux numérique

Il existe dans la littérature, de nombreux exemples de flux numérique [7, 9, 20, 53, 8]. Nous donnons trois exemples de flux numérique que nous comparons afin de trouver le meilleur rapport précision/temps d'exécution pour notre modèle MHD. Nous commençons par le Flux de Rusanov qui est un des plus simples à écrire, mais qui possède le désavantage d'être fortement diffusif. Suite à cela, nous étudions le flux VFRoe qui est lui en revanche, un flux précis, mais qui nécessite un grand nombre de calculs et donc possède un temps d'exécution plus long. Nous cherchons alors un compromis entre les deux en étudiant finalement un schéma décentré dit de type P2 [19].

2.2.1 Flux de Rusanov

Le flux numérique de Rusanov, entre les cellules L et R, est déterminé à l'aide de la formule

$$\mathbf{F}(\mathbf{W}_L, \mathbf{W}_R, \mathbf{n}) = \frac{1}{2}\left(\mathbf{F}(\mathbf{W}_L, \mathbf{n}) + \mathbf{F}(\mathbf{W}_R, \mathbf{n})\right) - \frac{\lambda_{\max}}{2}(\mathbf{W}_R - \mathbf{W}_L). \tag{2.11}$$

où $\lambda_{\max}$ est un majorant de la valeur propre maximale du système. Ce flux numérique vérifie les conditions de consistance et de conservation puisque nous avons

$$\begin{aligned}\mathbf{F}(\mathbf{W}, \mathbf{W}, \mathbf{n}) &= \tfrac{1}{2}\left(\mathbf{F}(\mathbf{W}, \mathbf{n}) + \mathbf{F}(\mathbf{W}, \mathbf{n})\right) - \tfrac{\lambda_{\max}}{2}(\mathbf{W} - \mathbf{W}) \\ &= F(\mathbf{W}, \mathbf{n})\end{aligned}$$

et

$$\begin{aligned}\mathbf{F}(\mathbf{W}_R, \mathbf{W}_L, \mathbf{n}_{RL}) &= \tfrac{1}{2}\left(\mathbf{F}(\mathbf{W}_R, \mathbf{n}_{RL}) + \mathbf{F}(\mathbf{W}_L, \mathbf{n}_{RL})\right) - \tfrac{\lambda_{\max}}{2}(\mathbf{W}_L - \mathbf{W}_R) \\ &= -\tfrac{1}{2}\left(\mathbf{F}(\mathbf{W}_L, \mathbf{n}_{LR}) + \mathbf{F}(\mathbf{W}_R, \mathbf{n}_{LR})\right) + \tfrac{\lambda_{\max}}{2}(\mathbf{W}_R - \mathbf{W}_L) \\ &= -\mathbf{F}(\mathbf{W}_L, \mathbf{W}_R, \mathbf{n}_{LR})\end{aligned}$$

2.2.2 Flux VFRoe

Un autre schéma numérique réputé pour être un schéma très précis est le schéma de VFRoe qui est donné par la relation

$$\mathbf{F}(\mathbf{W}_L, \mathbf{W}_R, \mathbf{n}) = \mathbf{F}(\mathcal{R}(\mathbf{W}_L, \mathbf{W}_R, 0, \mathbf{n})) \tag{2.12}$$

où $\mathcal{R}(\mathbf{W}_L, \mathbf{W}_R, \frac{x}{t})$ est solution d'un problème de Riemann linéarisé autour d'un état moyen

$$\mathbf{W}_{\text{mil}} = \frac{\mathbf{W}_L + \mathbf{W}_R}{2}.$$

Ce problème s'écrit

$$\begin{aligned}\partial_t \mathbf{W} + \mathbf{A}(\mathbf{W}_{\text{mil}}, \mathbf{n})\partial_x \mathbf{W} &= 0 \\ \mathbf{W}(x, 0) &= \mathbf{W}_L \text{ si } x < 0 \\ \mathbf{W}(x, 0) &= \mathbf{W}_R \text{ si } x > 0\end{aligned}$$

où $A(\mathbf{W}_{\text{mil}}, \mathbf{n})$ (1.27 page 15) représente la matrice jacobienne du système conservatif (1.24 page 13). La solution d'un tel problème est donnée par la formule

$$\mathcal{R}(\mathbf{W}_L, \mathbf{W}_R, \frac{x}{t}) = \mathbf{W}_{\text{mil}} - \frac{1}{2}\text{sgn}(\mathbf{A}(\mathbf{W}_{\text{mil}}) \cdot \mathbf{n} - \frac{x}{t}I)(\mathbf{W}_R - \mathbf{W}_L).$$

Ainsi pour déterminer la solution nous devons donc déterminer la valeur de la fonction sgn appliquée à la matrice

$$\mathbf{A}(\mathbf{W}_{\text{mil}}) \cdot \mathbf{n} - \frac{x}{t}I.$$

Dans le cas d'un nombre réel x, le signe est donné par :

$$\mathrm{sgn}(x) = \begin{cases} 1 \text{ si } x > 0 \\ 0 \text{ si } x = 0 \\ -1 \text{ si } x > 0 \end{cases} \tag{2.13}$$

en revanche pour une matrice, la définition d'une fonction d'une matrice est donnée par exemple dans [35]. Dans le cas où la matrice $\mathbf{A}$ peut être mise sous la forme de Jordan

$$\mathbf{A} = \mathbf{P}\mathbf{J}\mathbf{P}^{-1},$$

la fonction $f(A)$ est donnée par la relation

$$f(\mathbf{A}) = \mathbf{P}f(\mathbf{J})\mathbf{P}^{-1},$$

où $f(\mathbf{J}) = \mathrm{diag}(f(J_k)$ et où la fonction f appliquée à un bloc de Jordan J_k de taille m_k associée à une valeur propre λ_k vaut

$$f(\mathbf{J}_k) = \begin{pmatrix} f(\lambda_k) & f'(\lambda_k) & \cdots & \frac{f^{(m_k-1)}(\lambda_k)}{(m_k-1)!} \\ 0 & f(\lambda_k) & \ddots & \vdots \\ \vdots & \ddots & \ddots & f'(\lambda_k) \\ 0 & \cdots & 0 & f(\lambda_k) \end{pmatrix}$$

La matrice $\mathbf{B} = \mathbf{A}(\mathbf{W}_{\mathrm{mil}}) \cdot \mathbf{n} - \frac{x}{t} I$ est diagonalisable donc dans ce cas la définition de la fonction $f(\mathbf{B})$ est donnée par

$$\mathrm{sgn}(\mathbf{B}) = \mathbf{P}\mathrm{diag}(\mathrm{sgn}(\lambda_k))\mathbf{P}^{-1}.$$

Mais les vecteurs propres de la matrice sont parfois difficiles à déterminer (voir la section 1.3 page 12). Pour cela, nous utiliserons une autre définition de la fonction $f(\mathbf{A})$) basée sur le polynôme d'interpolation de $f^{(k)}(\lambda_k)$.

Finalement dans le cas du flux VFRoe nous souhaitons déterminer P, le polynôme d'interpolation du signe des valeurs propres du système, on a alors par définition

$$\mathrm{sgn}(\mathbf{A}) = P(\mathbf{A}).$$

Pour ce faire, nous utilisons un algorithme de différences divisées en utilisant les notations suivantes.

$$\text{sgn}[\lambda_i] := \text{sgn}(\lambda_i) \text{ et } \text{sgn}[\lambda_1 \cdots \lambda_{i+1}] := \frac{\text{sgn}[\lambda_2 \cdots \lambda_{i+1}] - \text{sgn}[\lambda_1 \cdots \lambda_i]}{\lambda_{i+1} - \lambda_i}$$

On a alors

$$P(\mathbf{A}) = \text{sgn}[\lambda_1] + \text{sgn}[\lambda_1\lambda_2](\mathbf{A} - \lambda_1\mathbf{I}) + \cdots + \text{sgn}[\lambda_1 \cdots \lambda_m](\mathbf{A} - \lambda_1\mathbf{I})(\mathbf{A} - \lambda_{m-1}\mathbf{I})$$

Bien que le flux VFRoe présente l'avantage d'être plus précis que d'autres flux qui sont plus diffusifs par exemple le flux de Rusanov, il peut provoquer l'apparition de chocs non physique lorsque les valeurs propres gauche et droite associées à la même famille $(\boldsymbol{\lambda}_L)$ et $(\boldsymbol{\lambda}_R)$ sont de signes contraires. Afin de pallier ce problème on ajoute une correction entropique décrite dans [31]. Le flux numérique (2.12 page 33) est alors remplacé par

$$\mathbf{F}(\mathbf{W}_L, \mathbf{W}_R, \mathbf{n}) = \mathbf{F}(\mathcal{R}(\mathbf{W}_R, \mathbf{W}_L, 0, \mathbf{n})) - \frac{\min(|\boldsymbol{\lambda}_L|, |\boldsymbol{\lambda}_R|)(\mathbf{W}_R - \mathbf{W}_L)}{2}. \tag{2.14}$$

2.2.3 Schéma décentré *upwind*

Le schéma de VFRoe consiste à linéariser le flux autour d'un état moyen $\mathbf{W}_{\text{mil}}$. Nous avons vu, dans le cas du flux VFRoe, que la matrice du linéarisé est donnée par $\mathbf{A}(\mathbf{W}_{\text{mil}}, \mathbf{n})$. Si le système hyperbolique de départ est lui aussi linéaire à coefficients constants, le flux est donné par

$$\mathbf{F}(\mathbf{W}_L, \mathbf{W}_R, \mathbf{n}) = \frac{1}{2}\left(\mathbf{F}(\mathbf{W}_L, \mathbf{n}) + \mathbf{F}(\mathbf{W}_R, \mathbf{n})\right) - \frac{1}{2} \mid \mathbf{A}(\mathbf{W}_{\text{mil}}) \mid (\mathbf{w}_R - \mathbf{W}_L).$$

Une approche proposée dans [19] consiste à appliquer cette formule, même si le système de départ est non-linéaire. Le calcul de la valeur absolue par un polynôme d'interpolation P peut-être coûteux. Afin de simplifier ce calcul, on envisage de remplacer P par une approximation V. Pour un tel polynôme d'interpolation noté

$$V(\mathbf{x}) \simeq |\mathbf{x}|\,.$$

Le flux décentré est alors définit par

$$\mathbf{F}(\mathbf{W}_L, \mathbf{W}_R, \mathbf{n}) = \frac{1}{2}\left(\mathbf{F}(\mathbf{W}_L, \mathbf{n}) + \mathbf{F}(\mathbf{W}_R, \mathbf{n})\right) - \frac{1}{2} V\left(\mathbf{A}(\mathbf{W}_{\text{mil}})\right)(\mathbf{w}_R - \mathbf{W}_L). \quad (2.15)$$

Proposition 2.2.1. *[19] Soit $\lambda_{\max}$ un majorant de la valeur propre maximale du système. Si*

$$|\ \mathbf{x}\ | \leqslant V(\mathbf{x}) \leqslant |\ \lambda_{\max}\ | \quad (2.16)$$

alors le schéma numérique (2.15) est linéairement stable sous la condition CFL

$$\Delta t \leqslant \frac{\Delta x}{\lambda_{\max}}. \quad (2.17)$$

Nous avons vu précédemment que dans le cas du système de la MHD idéale avec la correction de divergence de Dedner, la valeur c_h est choisie de sorte que l'on ait

$$c_h = \lambda_{\max}. \quad (2.18)$$

Par ailleurs nous pouvons supposer que

$$V(\mathbf{x}) = c_h R(\frac{\mathbf{x}}{c_h}).$$

Nous sommes ainsi ramenés au cas où $\lambda_{\max} = 1$.

Nous allons maintenant comparer les résultats obtenus avec différentes fonctions vérifiant la condition (2.16). Ces fonctions sont les suivantes

1. $R(\tau) = 1$.
2. $R(\tau) = \frac{\tau^2+1}{2}$. Cette approximation correspond à l'approximation utilisée dans [19].
3. $R(\tau) = \frac{1+3\tau^2}{3+\tau^2}$. L'approximation est ici faite par une approximation de Padé. On appelle donc le schéma correspondant le schéma *Padé-2*.
4. $R(\tau) = \frac{1+12\tau^2}{6+7\tau^2}$. Ce schéma est appelé le schéma *Other-Padé-2*.
5. $R(\tau) = \frac{5}{16} + \frac{15}{16}\tau^2 - \frac{5}{16}\tau^4 + \frac{1}{16}\tau^6$. L'approximation étant faite via un polynôme de degré 6 on appelle ce schéma le schéma $P6$.

La condition (2.17 page précédente), est équivalente, à dire que la fonction d'interpolation est contenue dans le triangle formé par la fonction valeur absolue et la fonction constante égale à $\lambda_{\max}$. Sur la figure 2.4, nous nous assurons de vérifier cette condition et nous comparons la précision de chaque fonction.

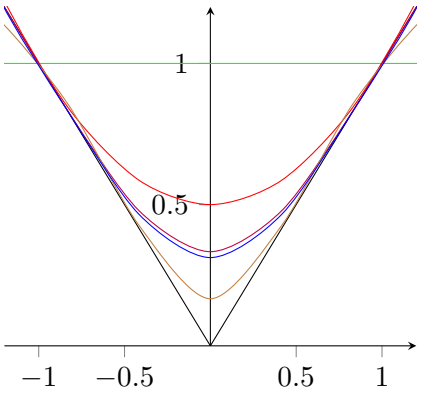

FIGURE 2.4 – Fonctions d'approximations de la valeur absolue.
Fonctions d'approximations de la valeur absolue. Le schéma est stable si, sous la condition (2.18 page précédente), la fonction d'approximation $R(x)$ vérifie $\mid x \mid \leqslant R(x) \leqslant 1$, pour tout $x \in [-1, 1]$.

Application numérique

Afin d'utiliser le flux (2.15 page ci-contre), nous commençons par déterminer la matrice $\mathbf{A}\left(\frac{\mathbf{W}_L+\mathbf{W}_R}{2c_h}\right)$. Ceci peut être fait de manière explicite puisque nous connaissons les termes de la matrice (voir chapitre 1 page 5). Après cela, nous pouvons ainsi déterminer le terme

$$\begin{aligned} V(\mathbf{A}(\frac{\mathbf{W}_L+\mathbf{W}_R}{2}))(\mathbf{W}_R-\mathbf{W}_L) &= c_h R(\mathbf{A}(\frac{\mathbf{W}_L+\mathbf{W}_R}{2c_h}))(\mathbf{W}_R-\mathbf{W}_L) \\ &\approx |\mathbf{A}(\frac{\mathbf{W}_L+\mathbf{W}_R}{2})|(\mathbf{W}_R-\mathbf{W}_L) \end{aligned}$$

Afin d'être le plus efficaces possible dans la détermination de ce terme, nous allons maintenant étudier quelles sont les stratégies d'optimisation pour un tel calcul, dans le cas où nous effectuons une approche polynomiale ou une approche par approximation de Padé.

Approximation par des fonctions polynomiales

Si nous choisissons d'approcher la valeur absolue à l'aide d'une approximation polynomiale par exemple avec le polynôme de degré 6

$$R(\mathbf{A}) = \frac{5}{16} + \frac{15}{16}\,\mathbf{A}^2 - \frac{5}{16}\,\mathbf{A}^4 + \tfrac{1}{16}\,\mathbf{A}^6$$

le calcul de la matrice $\mathbf{A}^6$ va être très coûteux puisque pour rappel, pour une matrice de dimension m, le produit matriciel est en $\mathcal{O}(m^3)$. Afin de réduire le nombre d'opérations et donc le temps de calcul, on utilise la méthode de Horner qui consiste à factoriser un polynôme

$$P(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_i x + a_0$$

en

$$P(x) = ((\cdots((a_n x + a_{n-1})x + a_{n-2})x + \cdots)x + a_1)x + a_0.$$

Dans notre cas, on peut donc calculer

$$\begin{aligned} V(\mathbf{A}(\frac{\mathbf{W}_L + \mathbf{W}_R}{2}))(\mathbf{W}_R - \mathbf{W}_L) &= (\frac{5}{16} + \frac{15}{16}\,\mathbf{A}^2 - \frac{5}{16}\,\mathbf{A}^4 + \tfrac{1}{16}\,\mathbf{A}^6)(\mathbf{W}_R - \mathbf{W}_L) \\ &= \frac{5}{16}(\mathbf{W}_R - \mathbf{W}_L) + \frac{15}{16}\,\mathbf{A}^2(\mathbf{W}_R - \mathbf{W}_L) \\ &\quad - \frac{5}{16}\,\mathbf{A}^4(\mathbf{W}_R - \mathbf{W}_L) + \mathbf{A}^6(\mathbf{W}_R - \mathbf{W}_L) \\ &= \frac{5}{16}(\mathbf{W}_R - \mathbf{W}_L) + \mathbf{A}(0 + \mathbf{A}(\frac{15}{16}(\mathbf{W}_R - \mathbf{W}_L) + \mathbf{A}(0 \\ &\quad + \mathbf{A}(-\frac{5}{16}(\mathbf{W}_R - \mathbf{A}_L) + \mathbf{A}(0 + \mathbf{A}(\mathbf{W}_R - \mathbf{W}_L)))))). \end{aligned}$$

Nous remarquons que de cette manière il ne persiste plus aucune multiplication matricielle puisqu'elles ont été remplacées par des multiplications matrice-vecteur qui elles sont en $\mathcal{O}(m^2)$.

Approximation par des fractions rationnelles

Dans le cas d'une approximation par une fonction de Padé, comme

$$R(\mathbf{A}) = \frac{1 + 3\mathbf{A}^2}{3 + \mathbf{A}^2}$$

Nous commençons par déterminer la matrice $\mathbf{A}^2$. Le passage par une multiplication matrice-matrice peut être évité en calculant de manière formelle la matrice. Puis nous déterminons ensuite la matrice

$$I + 3\mathbf{A}^2 \left(\frac{\mathbf{W}_L + \mathbf{W}_R}{2c_h} \right)$$

et le vecteur

$$\mathbf{B} = \left(I + 3\mathbf{A}^2 \left(\frac{\mathbf{W}_L + \mathbf{W}_R}{2c_h} \right) \right) (\mathbf{W}_R - \mathbf{W}_L).$$

Il ne reste finalement plus qu'à déterminer la matrice

$$\mathbf{M} = 3I + A^2 \left(\frac{\mathbf{W}_L + \mathbf{W}_R}{2c_h} \right)$$

et à résoudre l'équation linéaire

$$\mathbf{MX} = \mathbf{B}$$

puisqu'en effet nous avons les équivalences suivantes.

$$(3\mathbf{I}+\mathbf{A}^2(\frac{\mathbf{W}_L+\mathbf{W}_R}{2c_h}))\mathbf{X}=(\mathbf{I}+3\mathbf{A}^2(\frac{\mathbf{W}_L+\mathbf{W}_R}{2c_h}))(\mathbf{W}_R-\mathbf{W}_L)$$

$$\Leftrightarrow \mathbf{X}=(3\mathbf{I}+\mathbf{A}^2(\frac{\mathbf{W}_L+\mathbf{W}_R}{2c_h}))^{-1}$$
$$\times(\mathbf{I}+3\mathbf{A}^2(\frac{\mathbf{W}_L+\mathbf{W}_R}{2c_h}))(\mathbf{W}_R-\mathbf{W}_L)$$

$$\Leftrightarrow \mathbf{X}=R(\mathbf{A}(\frac{\mathbf{W}_L+\mathbf{W}_R}{2c_h}))(\mathbf{W}_R-\mathbf{W}_L)$$

$$\Leftrightarrow \mathbf{X}=\frac{1}{c_h}V(\mathbf{A}(\frac{\mathbf{W}_L+\mathbf{W}_R}{2}))(\mathbf{W}_R-\mathbf{W}_L).$$

Nous utilisons la même méthode pour le calcul de $R(\tau)=\frac{1+12\tau}{6+7\tau^2}$.

2.3 Ordre et coût

2.4 Résultats

Dans cette section, nous comparons les résultats donnés par les différents flux ainsi que les ordres de convergence que nous obtenons avec les méthodes d'ordre 1 et d'ordre 2. Concernant ces dernières, nous traiterons le cas du limiteur de pente minmod initial ainsi que le minmod amélioré que nous appelons minmod2. Pour ce faire, nous utiliserons le cas test 1D appelé *choc fort* [63]. Ce test est un problème de Riemann faisant intervenir à la fois des ondes de chocs et de détentes. Une solution de référence déterminée à l'aide d'un solveur de Riemann exact peut être calculée [1] [63]. Nous utiliserons ces solutions comme solutions de références pour nos tests. Pour ce cas test et pour la masse volumique, la solution à l'état initial et la solution exacte au temps $t = 1$ sont représentées sur la figure 2.5 page ci-contre.

Les données initiales discontinues du problème pour chaque variable sont données de façon plus précise dans le tableau 2.6 page suivante.

Sur les figures 2.7 à 2.12 pages 42–44 , nous présentons les résultats que nous obtenons pour deux des sept variables du système : la masse volumique ρ et la vitesse dans la direction y. Nous présentons les solutions obtenues à l'aide des différentes méthodes

1. *https ://web.mathcces.rwth-aachen.de/mhdsolver/*

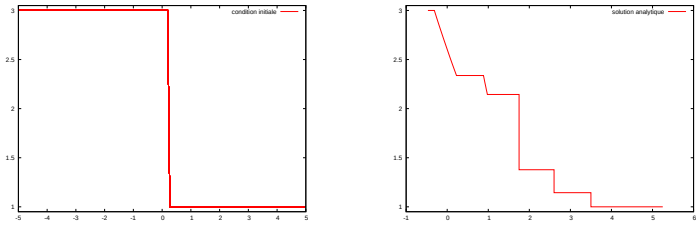

FIGURE 2.5 – Exemple de condition initiale ($t = 0$) et de solution analytique ($t = 1$). Masse volumique en fonction de x aux instants $t = 0$ et $t = 1$.

Variable	État Gauche	État droit
γ	5/3	5/3
ρ	3	1
u_x	1.3	1.3
u_y	0	0
u_z	0	0
$B_i x$	1.5	1.5
B_y	1	cos(1.5)
B_z	1	sin(1.5)
p	1	1/10

FIGURE 2.6 – Conditions initiales pour le cas test du choc fort

d'approximations de la valeur absolue (voir figure 2.4 page 37.) Nous effectuons cette comparaison pour la méthode à l'ordre 1 et les méthodes à l'ordre 2 obtenue à l'aide des fonctions minmod. Nous utilisons volontairement un maillage comportant peu de mailles (700) afin de pouvoir observer les différences entre les différentes méthodes.

Sur les figures 2.13 à 2.15 pages 45–46, nous effectuons un zoom sur une partie de la courbe de la densité. Nous observons, à l'ordre 1 que la précision des courbes est effectivement liée à la précision d'approximation de la valeur absolue dans le flux. Toutefois, à l'ordre 2 les différences deviennent moins significatives. Il est donc légitime de se demander si utiliser une approximation précise de la valeur absolue est raisonnable en fonction de son temps d'exécution.

C'est pourquoi dans le tableau 2.16 page 46, nous comparons les temps d'exécution des différentes méthodes. Il apparaît clairement que les méthodes d'approximation par fractions rationnelles, bien que plus précises pour le schéma d'ordre 1, sont très coûteuses.

Dans la suite de ce travail, nous retiendrons donc le flux P2 pour les tests 2D. Enfin afin de compléter notre analyse, nous étudions dans la figure 2.17 page 47 les ordres de convergences de ces trois différents schémas pour la norme L^1. Les ordres de convergence

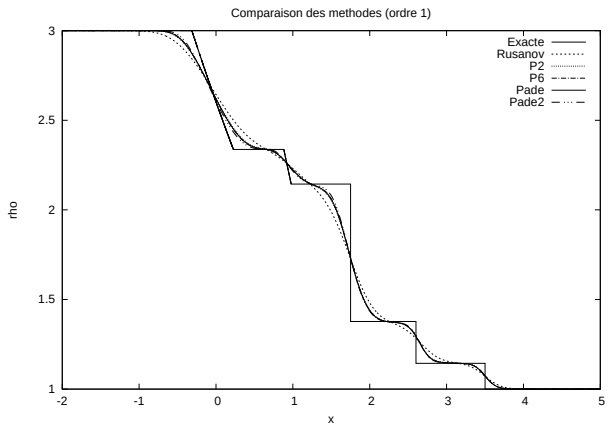

FIGURE 2.7 – Solution obtenue pour la densité à l'ordre 1.

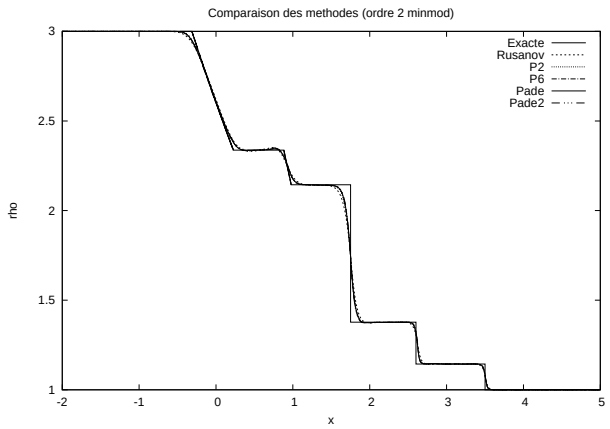

FIGURE 2.8 – Solution obtenue pour la densité à l'ordre 2 avec le limiteur de pentes minmod

expérimentaux sont compatibles avec les ordres attendus : 1/2 pour le schéma "d'ordre 1" et environ 2/3 pour le schéma "d'ordre 2" [57].

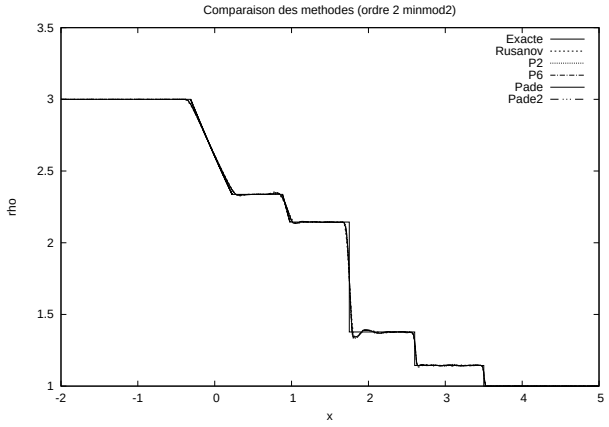

FIGURE 2.9 – Solution obtenue pour la densité à l'ordre 2 avec le limiteur de pentes minmod2.

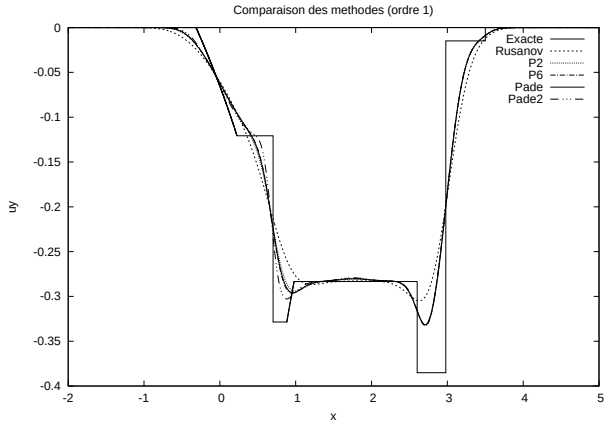

FIGURE 2.10 – Solution obtenue pour la vitesse dans la direction y à l'ordre 1.

2.5 Conclusion

Dans ce chapitre nous avons implémenté, une méthode de volumes finis pour la résolution du problème de la MHD. Après avoir vérifié la convergence d'un tel schéma, nous avons décrit les schémas en dimension en dimension 1 et 2. Dans le cas du schéma en dimension 2, nous avons utilisé une méthode Van Leer avec le limiteur de pente *minmod*. Nous avons également testé un limiteur de pente de type *minmod* prenant en compte

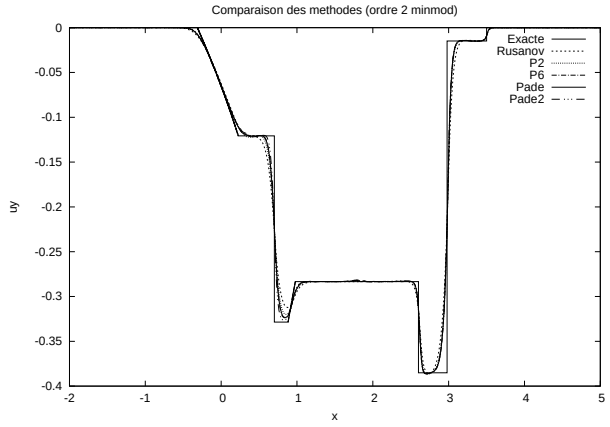

FIGURE 2.11 – Solution obtenue pour la vitesse dans la direction y à l'ordre 2 avec le limiteur de pentes minmod.

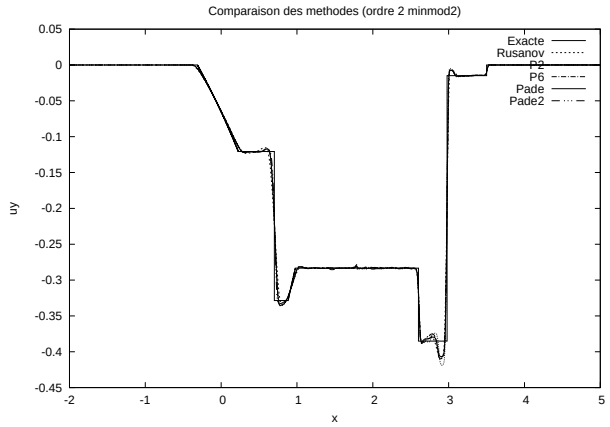

FIGURE 2.12 – Solution obtenue pour la vitesse dans la direction y à l'ordre 2 avec le limiteur de pentes minmod2.

deux fois les pentes gauche et droite. Lors de nos tests numériques, nous avons constaté l'apparition d'oscillation de la solution causée par l'utilisation d'un schéma d'ordre 1 en temps. Concernant les flux numériques de la méthode en volumes finis, nous avons utilisé un flux décentré *upwind* et comparé différentes approches selon la précision de l'approximation de la valeur absolue de la matrice jacobienne. Nos tests numériques nous ont permis de conclure que bien qu'à l'ordre 1 la précision de cette approximation ait

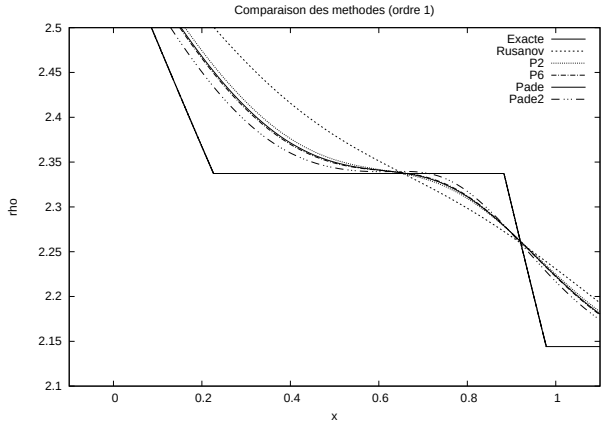

FIGURE 2.13 – Comparaison des différents schémas numériques à l'ordre 1

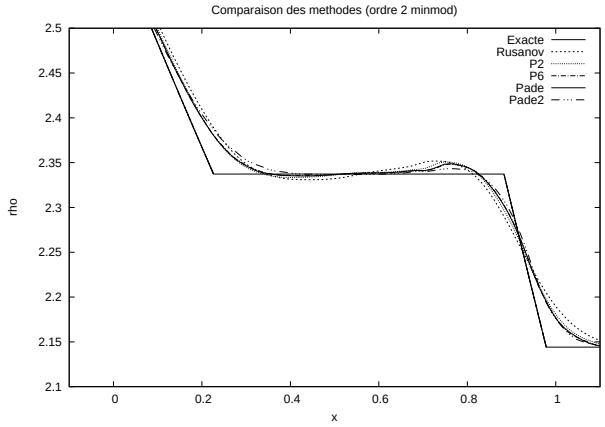

FIGURE 2.14 – Comparaison des différents schémas numériques à l'ordre 2 avec le limiteur de pentes minmod

une importance sur la précision de la solution, à l'ordre 2, quelle que soit cette précision, les solutions sont relativement identiques. L'utilisation d'un flux numérique de type $P2$ est donc le meilleur rapport précision/temps.

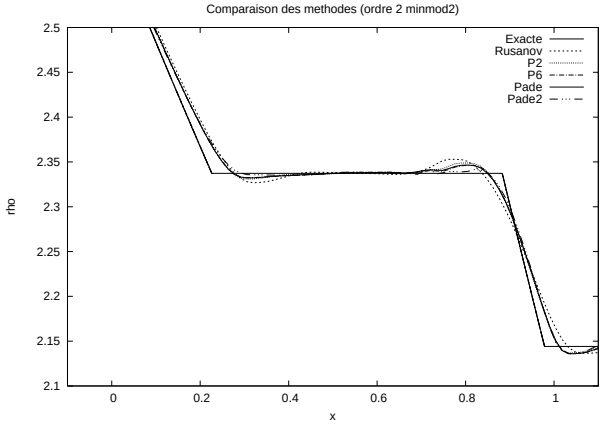

FIGURE 2.15 – Comparaison des différents schémas numériques à l'ordre 2 avec le limiteur de pentes minmod2

Schéma	Temps (sec)
Rusanov	10s
P2	18s
P6	24s
Padé	84s
Padé2	89s

FIGURE 2.16 – Comparaison des temps d'exécution des schémas

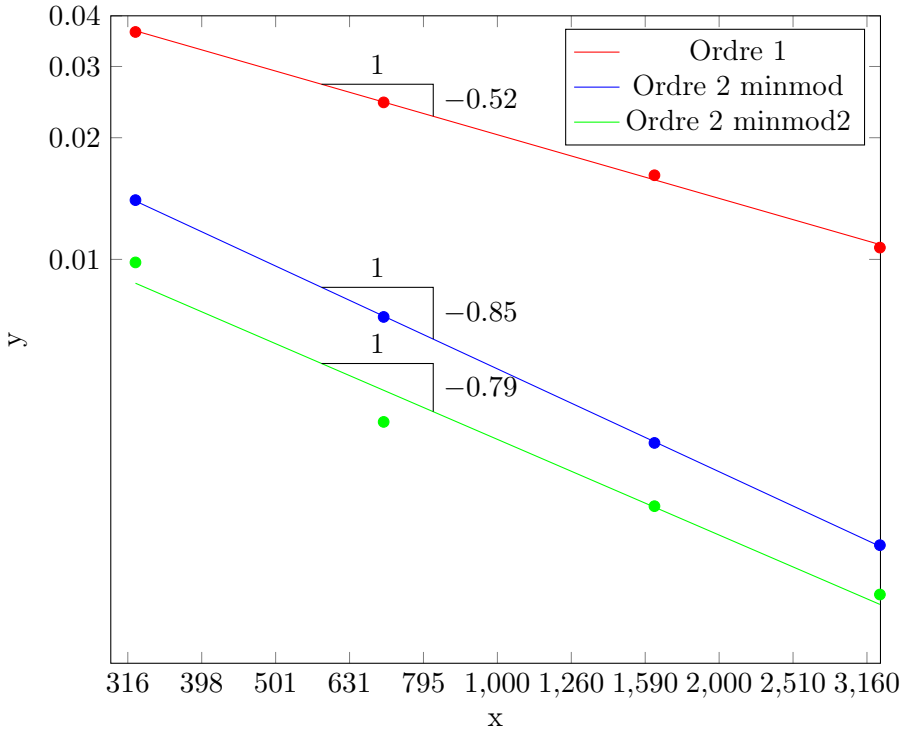

FIGURE 2.17 – Comparaison des ordres de convergence pour les méthodes d'ordre 1 et d'ordre 2.

Chapitre 3

Implémentation et passage en dimension 2

> Anything that is in the world when you're born is normal and ordinary. Anything that's invented between when you're fifteen and thirty-five is new and exciting. Anything invented after you're thirty-five is against the natural order of things.
>
> Douglas Adams

Sommaire

3.1 Implémentation 50
3.2 Optimisation et parallélisation automatique 51
3.2.1 Fonctionnement d'un processeur (CPU) 52
3.2.2 Optimisations du compilateur 56
3.2.3 Un premier code parallèle avec OpenMP 56
3.3 Optimisations manuelles 57
3.3.1 "Loop tiling" ou parcours par tuiles 57
3.4 Résultats 60
3.5 Conclusion 60

Dans le chapitre précédent, nous avons présenté un schéma de volumes finis sur maillage quelconque. Nous l'avons ensuite programmé et testé dans le cas de la dimension 1. Dans ce chapitre, nous étudions la programmation du schéma en dimension 2. Pour des raisons d'efficacité, nous considérons le cas d'un maillage structuré. Les cas des maillages et d'un ordre quelconque seront abordés dans les chapitres suivants. Nous expliciterons

les méthodes d'optimisation qui peuvent être appliquées à un tel programme. Le but est en effet d'obtenir de très bonnes performances sur un processeur multicoeur ou sur GPU.

Afin de faciliter la lecture, nous désignerons désormais par x, y et z les directions $(1,0,0)$, $(0,1,0)$ et $(0,0,1)$. De la même manière, les composantes du champ magnétique et du champ de vitesse seront désormais notées $\mathbf{B} = (B_x, B_y, B_z)$ et $\mathbf{u} = (u_x, u_y, u_z)$.

3.1 Implémentation

Toute implémentation d'un algorithme commence par une organisation des données. Dans notre cas, nous voulons simuler l'évolution de chaque variable conservative de notre système sur une grille régulière. De manière naturelle, nous avons donc choisi de ranger les variables dans un tableau à 3 dimensions. Les deux premières dimensions permettent de repérer les volumes finis (ou cellules) dans les directions x et y. La troisième dimension concerne l'indice des variables du système. De cette manière, on peut donc se représenter les données comme une succession de couches (figure 3.1)

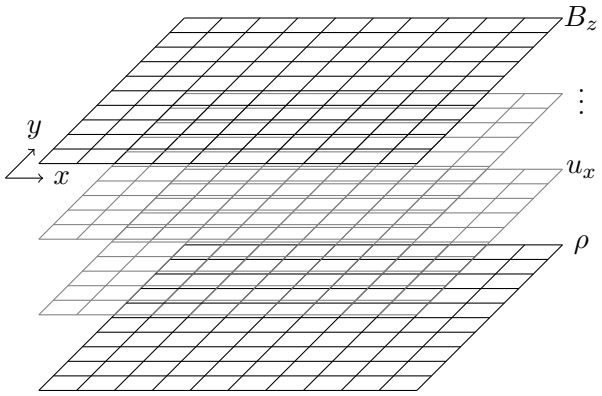

FIGURE 3.1 – Représentation schématique des données du problème.

Nous désignons par $\mathbf{W}_{i,j}^n$ le vecteur de variables conservatives correspondant à la cellule (i,j) et à l'itération temporelle n. La mise à jour de ces vecteurs est faite en y ajoutant la contribution de ses vecteurs voisins. Sur la figure 3.2 page ci-contre nous représentons la situation dans le cas d'un schéma à l'ordre 1 où le champ $\mathbf{W}_{i,j}^n$ est mis à jour à l'aide de l'équation

$$\begin{aligned}\mathbf{W}_{i,j}^{n+1} = \mathbf{W}_{i,j}^n &- \frac{\Delta t}{\Delta x}\left(\mathbf{F}(\mathbf{W}_{i,j}^n, \mathbf{W}_{i+1,j}^n, \mathbf{n}_x) - \mathbf{F}(\mathbf{W}_{i-1,j}^n, \mathbf{W}_{i,j}^n, \mathbf{n}_x)\right) \\ &- \frac{\Delta t}{\Delta y}\left(\mathbf{F}(\mathbf{W}_{i,j}^n, \mathbf{W}_{i,j+1}^n, \mathbf{n}_x) - \mathbf{F}(\mathbf{W}_{i,j-1}^n, \mathbf{W}_{i,j}^n, \mathbf{n}_x)\right).\end{aligned} \tag{3.1}$$

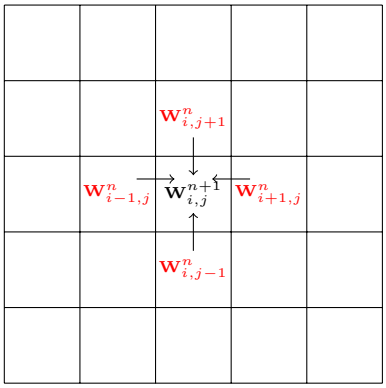

FIGURE 3.2 – Représentation des dépendances dans l'étape de mise à jour pour la méthode des volumes finis d'ordre 1.

Outre la résolution du système sur la grille, nous devons également mettre en place la gestion des conditions de bord. Il existe une multitude de conditions possible, cependant dans notre cas seul trois cas nous intéressent. Si dans la direction x, nous disposons de I cellules numérotées de 1 à I et indexées par i, nous avons alors les conditions aux bords

— $\mathbf{W}^n_{i+1,j} = \mathbf{W}^n_{1,j}$, qui sont les conditions aux limites périodiques.
— $\mathbf{W}^n_{i+1,j} = \mathbf{W}_\infty$, qui sont des conditions aux limites imposées par la valeur $\mathbf{W}_\infty$.
— $\mathbf{W}^n_{i+1,j} = \mathbf{W}^n_{i,j}$, qui sont des conditions libres.

Généralement, les conditions aux limites libres ne sont pas stables. Cependant, nous pouvons tout de même les utiliser afin de réaliser certains tests. La mise en place numérique de ces conditions se fait de la façon suivante. Lors d'une itération, lorsque toutes les cellules ont été mises à jour, nous copions les valeurs aux bords qui nous intéressent dans des cellules fantômes que nous utilisons spécifiquement pour les conditions aux limites. En effet, lors de la recherche des voisins, les cellules présentes sur les bords utiliseront ces cellules fantômes comme voisins directs. De cette manière, en définissant ces cellules à l'aide des valeurs concernant la condition aux limites choisie nous nous assurons de l'appliquer correctement. Sur la figure 3.3 page suivante, nous représentons le passage des conditions de bords périodiques dans la direction x. Les cellules hachurées représentent les cellules fantômes. Lorsque toutes les cellules internes ont été mises à jour, nous envoyons le bord gauche (en vert) et le bord droit (en bleu) dans les cellules fantômes du côté opposées.

3.2 Optimisation et parallélisation automatique

La résolution d'un tel système est simple, mais peut être extrêmement coûteuse sur maillage fin, c'est pourquoi une attention particulière doit être portée à l'optimisation de l'algorithme. En premier lieu, nous allons rappeler le fonctionnement d'un processeur

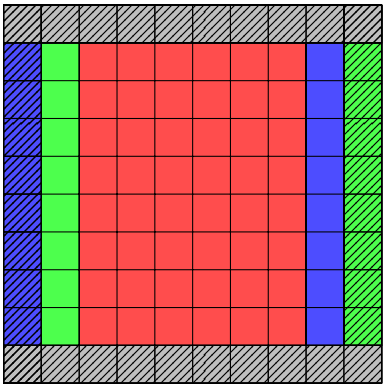

FIGURE 3.3 – Représentation du passage des conditions de bord périodique dans la direction x.

(CPU) afin de pouvoir comprendre comment optimiser au maximum l'exécution du code.

3.2.1 Fonctionnement d'un processeur (CPU)

Architecture

Un processeur est composé de plusieurs parties effectuant chacune un rôle différent (voir la figure 3.4).

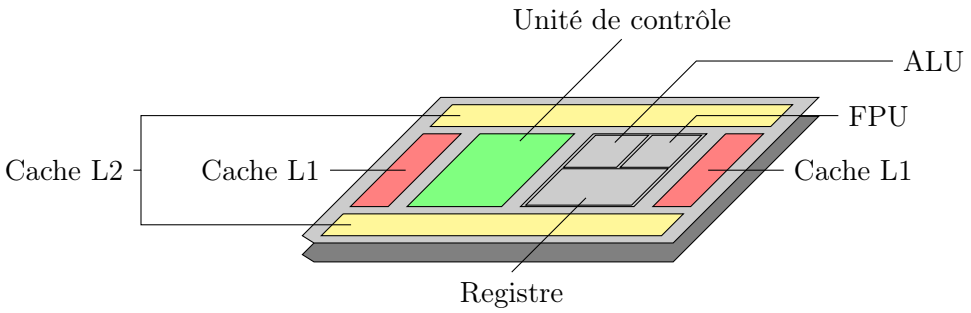

FIGURE 3.4 – Architecture simplifiée d'un CPU.

La plus commune, la mémoire cache (L1, L2) permet de conserver au plus proche du CPU des données ou des instructions. Cette proximité permet un accès plus rapide aux données nécessaires à une instruction. La partie effectuant les opérations est appelée, unité d'instruction. Elle comporte non seulement les unités de calculs et de logique (ALU : *Arithmetical and Logical Unit* et FPU : *Floating Point Unit*), mais également une zone de mémoire directe appelée le registre. C'est dans ce registre que sont stockées les données lors de l'exécution d'une instruction.

Pipeline

Le processeur fonctionne par cycle ou fréquence d'horloge. À chaque cycle, le CPU lance une partie d'instruction. On appelle CPI (*Cycle Per Instruction*) le nombre de cycles moyen pour lancer une instruction. La vitesse d'un code généralement calculé par le nombre d'opérations effectuées par seconde peut aussi être donnée par le nombre d'instructions par seconde et obtenue par le rapport de la fréquence du CPU sur le CPI. C'est bien plus difficile à déterminer puisqu'il ne suffit pas de compter le nombre d'opérations élémentaires, mais c'est en revanche, un critère plus précis. La fréquence d'un processeur étant constante, le seul moyen d'augmenter la performance d'un code est de réduire le CPI.

Classiquement, une instruction se déroule en cinq cycles

— LI : Lecture de l'instruction depuis le cache L1
— DI : Décodage de l'instruction dans le registre
— EX : Exécution de l'instruction (ALU, FPU)
— MEM : Écriture ou modification dans la mémoire
— ER : Écriture du résultat de l'instruction dans le registre

Tel que c'est présenté ici, il n'y a pas de possibilité d'effectuer une optimisation sur l'exécution d'une instruction et le CPI est constant égale à 5. Cependant, l'architecture du CPU étant scindée en différentes parties indépendantes les jeux d'instructions sont envoyés au travers d'un *pipeline*. De cette manière, l'exécution de plusieurs instructions successives se déroule comme sur la figure 3.5.

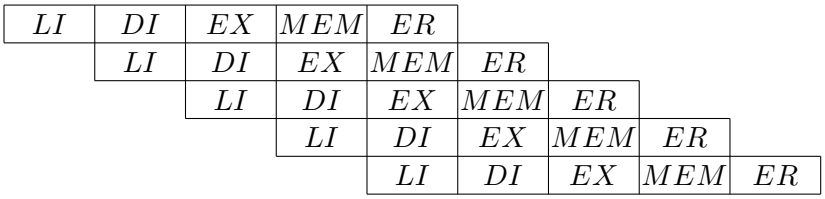

FIGURE 3.5 – Pipeline d'instructions.

De cette manière, après un temps de latence correspondant à cinq cycles donc après cinq instructions, à chaque cycle une instruction est complétée. Il arrive bien sûr qu'une instruction ait la nécessité d'attendre que l'instruction précédente soit arrivée à un certain point. Elle est alors mise en pause ce qui implique la perte de cycles. L'optimisation du *CPI* consiste travailler sur l'ordre d'exécution des opérations.

Exemple

Afin de mieux comprendre les mécanismes d'optimisation du *CPI* nous présentons l'exemple très simple d'une boucle de la forme

```
for(int i=0 ; i<1000 ; ++i){
  a[i] = b[i] + c[i];
}
```

qui se traduit en langage machine par le code assembleur suivant

```
Loop: LW R5, 0(R2)
LW R6, 0(R3)
ADD R7, R6, R5
SW 0(R1), R7
ADDI R1, R1, #4
ADDI R2, R2, #4
ADDI R3, R3, #4
SUBI R4, R4, #1
BNEZ R4, Loop
```

où $R1, R2, R3$ et $R4$ sont respectivement les adresses des tableaux a, b et c et d'un entier égal à 1000.

Sur le schéma (figure 3.6) du pipeline de ce code, on voit que la troisième instruction doit attendre que la donnée en $R6$ soit inscrite en mémoire avant de l'utiliser pour calculer $R7$. On utilise ainsi 13 cycles pour 8 instructions ce qui nous donne une *CPI* de 1.625.

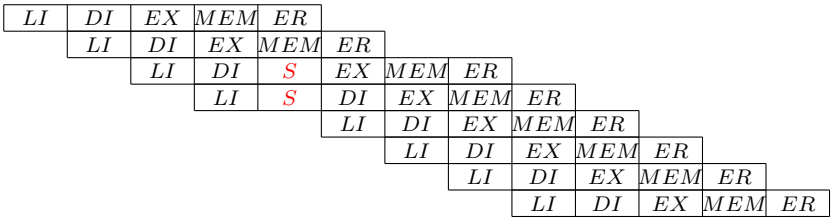

FIGURE 3.6 – Pipeline d'instructions pour l'addition de deux vecteurs.

En utilisant un simple réarrangement des instructions, on peut éviter la perte d'un cycle et ainsi obtenir un *CPI* de 1.5. Pour cela, on écrit

```
Loop: LW R5, 0(R2)
LW R6, 0(R3)
ADDI R2, R2, #4
ADD R7, R6, R5
```

```
ADDI R3, R3, #4
SW 0(R1), R7
ADDI R1, R1, #4
SUBI R4, R4, #1
BNEZ R4, Loop
```

Avec une simple boucle comme celle-ci, il existe un autre type d'optimisation qu'on appelle le *déroulage de boucle*. L'idée est très simple, au lieu de faire 1000 la mise à jour de $a[i]$ on fera plutôt 500 fois la mise à jour de $a[i]$ et $a[i+1]$

```
for(int i=0 ; i<1000 ; ++i){
  a[i]   = b[i]   + c[i];
  a[i+1] = b[i+1] + c[i+1];
}
```

Le code machine de cette boucle devient alors

```
Loop: LW R5, 0(R2)
LW R6, 0(R3)
ADD R7, R6, R5
LW R8, 4(R2)
LW R9, 4(R3)
ADD R10, R8, R9
SW 0(R1), R7
SW 4(R1), R10
ADDI R1, R1, #8
ADDI R2, R2, #8
ADDI R3, R3, #8
SUBI R4, R4, #2
BNEZ R4, Loop
```

et en appliquant un réarrangement des instructions identique à celui fait précédemment, il est possible d'effectuer ces 12 instructions sans perdre de cycle. On peut donc effectuer 12 instructions en 16 cycles ce qui fait descendre la *CPI* à 1.333. Dans cet exemple, nous avons ainsi augmenté de presque 20% la vitesse d'exécution de la boucle par un simple déroulage de boucle et un réarrangement des instructions. De plus, l'efficacité peut encore être augmentée en augmentant la taille du déroulage. Cette taille dépend du nombre de registres disponible dans le CPU.

3.2.2 Optimisations du compilateur

Comme nous l'avons vu, l'optimisation du code via la réduction du nombre de cycles par instruction est efficace, mais elle nécessite de passer par du code assembleur et une analyse complète du code, instruction par instruction. Heureusement pour nous, les compilateurs sont capables de faire une partie du travail de manière automatique via des options passées au moment de la compilation. Dans le tableau 3.1, nous rappelons quelques une de ces options pour le compilateur *gcc* à partir de la version 4.8.

Options	Optimisations
Ofast	nombreuse optimisation comme le calcul vectoriel. *Ofast* inclut *ffast-math* qui optimise certaines instructions de calcul.
march=native	Autorise le CPU à utilisé des optimisations spécifiques à son architecture.
mssse3	Ensemble de 32 optimisations concernant par exemple l'alignement des données dans le registre ou certaines opérations élémentaires.
unroll-loops	Déroule les boucles afin d'appliquer des optimisations identiques à celles de l'exemple ci-dessus.
funsafe-loop-optimizations	Contrairement à l'option précédente, celle-ci optimise même les boucles qui n'ont pas de fin définie à la compilation.
faggressive-loop-optimizations	D'autre optimisation sur les boucles.

TABLE 3.1 – Option d'optimisation pour la compilation avec *gcc*.

3.2.3 Un premier code parallèle avec OpenMP

L'interface OpenMP (Open Multi-Processing) permet d'utiliser le parallélisme sur un code séquentiel de façon presque transparente et automatique. Cette parallélisation peut être faite sur un processeur multicoeur ou sur une machine possédant plusieurs processeurs à condition que ces processeurs partagent la même mémoire.

Lorsque dans un code séquentiel nous avons réussi à isoler une partie qui peut être parallélisée par exemple la boucle de l'exemple précédent

```
for(int i=0 ; i<1000 ; ++i){
  a[i] = b[i] + c[i];
```

```
}
```

l'utilisation d'OpenMP pour la parallélisation de la boucle se fait en ajoutant simplement une ligne d'appel avant la boucle.

```
#pragma omp parallel for
for(int i=0 ; i<1000 ; ++i){
  a[i] = b[i] + c[i];
}
```

L'exécution parallèle ainsi que les éventuels transferts mémoires ou synchronisations à l'intérieur de la boucle sont effectués sans que nous ayons autre chose à effectuer. Le fonctionnement de la librairie est le suivant. Tant que le processeur maître ne rencontre pas la balise *#pragma* le programme est exécuté de manière séquentielle. Lorsque la balise est atteinte, le processeur maître crée une pile de tâches indépendantes à effectuer par lui et ses esclaves. Lorsque la pile est vide, le processeur maître continue l'exécution du code de manière séquentielle jusqu'à la prochaine balise. Le déroulement d'un code utilisant un tel parallélisme est souvent représenté tel que sur la figure 3.7.

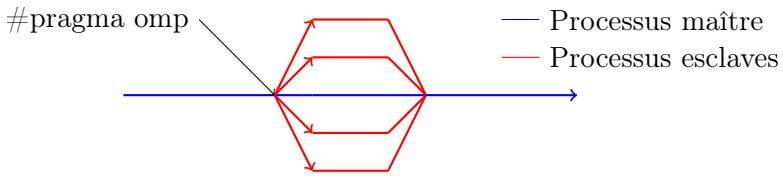

FIGURE 3.7 – Déroulement schématique d'un code utilisant la parallélisation à l'aide de OpenMP.

3.3 Optimisations manuelles

3.3.1 "Loop tiling" ou parcours par tuiles

Habituellement dans un code, quel que soit le langage, le parcours d'un tableau se fait de manière naturelle, c'est-à-dire que pour parcourir une grille en 2 dimensions on parcourt par exemple ligne par ligne (figure 3.8 page suivante) avec une boucle du type

```
for(int i=0 ; i<NX ; ++i){
  for(int j=0 ; j<NY ; ++j){
    table[i][j];
  }
}
```

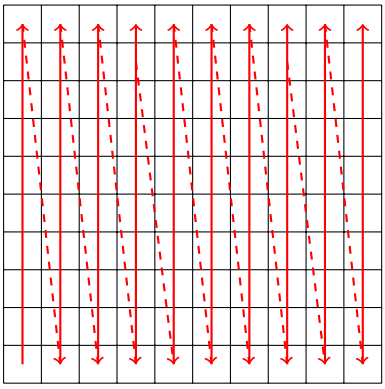

FIGURE 3.8 – Parcours traditionnel d'une grille

Nous avons vu dans la section 3.2 page 51 que lorsqu'une donnée est utilisée par une instruction, elle est copiée dans la mémoire cache. L'instruction est donc plus rapide si la valeur se trouve déjà dans cette mémoire, on appelle cela un *cache-hit*. Par opposition, un *cache-miss* se produit lorsque la copie d'une valeur à lire n'est pas présente dans la mémoire cache. Malheureusement, la taille de la mémoire cache n'est pas suffisante pour stocker un nombre élevé de valeurs. Lors du parcours du tableau tel que sur la figure 3.8, la mise à jour de la cellule courante nécessite la lecture des données dans les cellules directement voisines. De cette manière, les données présentes dans le cache forment une sorte de trainée en aval de la lecture (3.9). Le *cache-miss* est important puisque chaque cellule nécessite la lecture de trois valeurs hors du cache. Lors du passage à la ligne suivante, bien que ces valeurs, ainsi que celles présentes dans la ligne du dessous aient déjà été lues elles ont été écrasées et cela implique donc une lecture hors du cache.

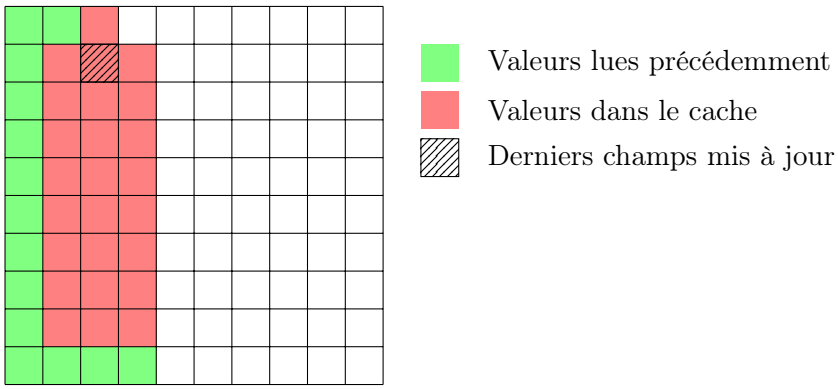

FIGURE 3.9 – Représentation des cellules dans la mémoire cache

Une première optimisation a donc été de parcourir la grille par blocs suffisamment petits pour entrer dans la mémoire cache (figure 3.10 page suivante). Afin d'obtenir les

meilleures performances, la taille des blocs est choisie de manière à contenir le plus de valeurs possible. La boucle est alors de la forme

```
#define TILE 16
for(int it=0 ; it<NX ; it+=TILE){
  for(int jt=0 ; jt<NY ; jt+=TILE){
    for(int i=it; i<it+TILE; ++i){
      for(int j=jt; j<jt+TILE; ++j){
        table[i][j];
      }
    }
  }
}
```

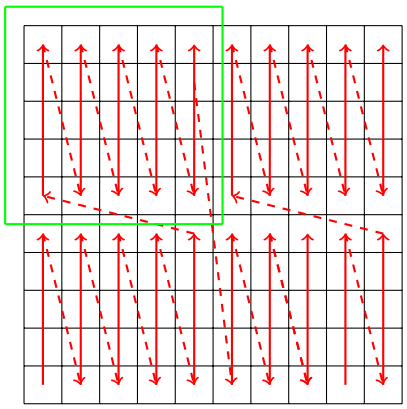

FIGURE 3.10 – Parcours par tuile d'une grille

De cette manière, les données présentes dans la mémoire cache ne représentent plus une trainée, mais restent localisées de manière optimale dans une zone restreinte (figure 3.11 page suivante). Lorsque nous arrivons à la fin d'une ligne de la tuile, le passage à la ligne suivante présente moins de *cache-miss* que dans le cas précédent, puisque sur les quatre valeurs à lire, trois l'on déjà été lors du traitement de la précédente ligne et sont toujours présentent dans le cache.

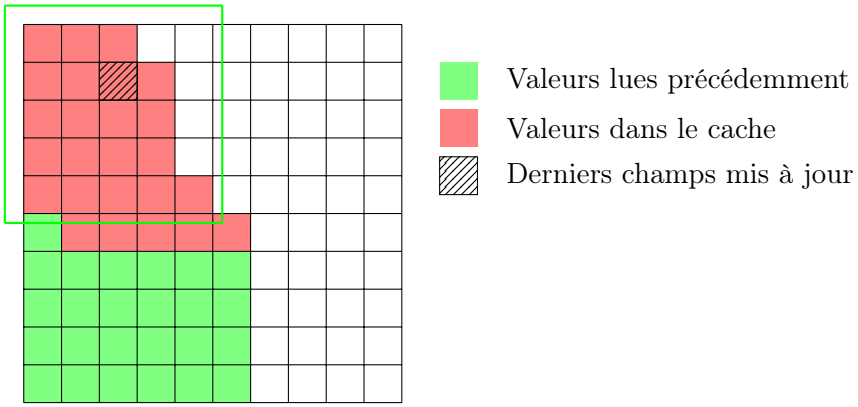

FIGURE 3.11 – Représentation des cellules dans la mémoire cache

3.4 Résultats

Dans le tableau ci-dessous, nous présentons les temps obtenus pour différentes versions du même code de résolution des équations. Les tests ont été réalisés sur un maillage 3000×3000 avec le flux $P2$ à l'ordre 2. Nous prenons comme temps de code de référence une version non optimisée et compilée sans option d'optimisation du compilateur. Nous appliquons ensuite les différentes méthodes d'optimisation vue dans ce chapitre. Nous désignons par *code séquentiel optimisé 1* le code ne comprenant que les optimisations automatiques du compilateur. Dans le *code séquentiel optimisé 2*, nous y ajoutons le parcours de la grille par tuile, ou *loop tiling*.

Architecture	Temps (sec)	Accelration
Code séquentiel non optimisé	2605629 (30j)	1
Code séquentiel optimisé 1	526110 (146h)	5
Code séquentiel optimisé 2	351663 (97h)	8
Code OpenMP (16 coeurs)	22398 (6.2h)	116

TABLE 3.2 – Comparaison des temps d'exécution en fonction des algorithmes

3.5 Conclusion

Nous avons commencé ce chapitre par la présentation d'une implémentation de l'algorithme de résolution du problème de la MHD en dimension deux. Puis la partie principale de ce chapitre a été consacrée à la présentation de différentes méthodes d'optimisation pour un code sur CPU. Nous avons vu qu'elles peuvent se faire à de nombreux niveaux

et peuvent être plus ou moins complexes à appliquer. De nombreuses optimisations liées à l'architecture du CPU sont possibles, mais difficilement réalisables manuellement. En revanche, les compilateurs actuels permettent de les effectuer de manière automatique. Nous avons montré sur un exemple que l'utilisation de ces options de compilation permet de gagner un facteur d'environ 5 sur l'exécution du code. L'autre optimisation séquentielle que nous avons présentée consistant à modifier le parcours des éléments en mémoire nous à permis de gagner encore un facteur de presque deux dans le gain de temps.

Enfin, nous avons introduit une parallélisation à l'aide de la librairie de fonctions OpenMP. Son utilisation sur 16 CPU nous a encore permis d'accélérer presque 15 fois l'exécution du code.

L'utilisation combinée de ces optimisations permet finalement une accélération d'un facteur 100 sur 16 coeurs de calcul. Nous allons maintenant comparer ces optimisations relativement classiques à celles qu'on peut obtenir grâce au calcul sur GPU.

Chapitre 4

Implémentation GPU/OpenCL

> Equipped with his five senses, man explores the universe around him and calls the adventure Science.
>
> Edwin Powell Hubble

Sommaire

4.1 Introduction à la programmation GPU 65
4.1.1 Architecture 65
4.1.2 Introduction à la programmation OpenCL 65
4.2 Programmation GPU du problème de la MHD 69
4.2.1 Splitting de Strang 69
4.2.2 Distribution des données en mémoire 70
4.2.3 Méthode de transposition 72
4.2.4 Distribution des tâches entre les processeurs 73
4.2.5 Déroulement d'un pas de temps 73
4.3 Performances 74
4.3.1 Modèle *roofline* 74
4.3.2 Temps mesurés 75
4.4 Multi-GPU 77
4.4.1 Décomposition de domaine 78
4.4.2 Temps mesurés 79
4.5 Résultats numériques 79
4.6 Conclusion 84

Dans le chapitre précédent, nous avons vu qu'une bonne gestion des données en mémoire, un bon ordonnancement des instructions et l'utilisation du parallélisme à l'inté-

rieur d'un CPU permettent d'obtenir des accélérations importantes dans l'exécution des algorithmes.

La mise en commun de plusieurs CPU permet théoriquement d'augmenter la puissance de calcul qui n'est alors limitée que par le nombre de noeuds de l'ordinateur utilisé. En 1976, le superordinateur *Cray I* possédait une puissance de 80 Mflop/s (à titre de comparaison. En 2015 un téléphone *Samsung Galaxy S6* possède une puissance de 35 Gflop/s). Le supercalculateur Chinois *Tianhe-2* posséde une puissance de 35 Pflop/s. Cepndant le prix de construction de ce dernier est d'environ 400 millions de dollars et sa consommation énergétique totale est de 24 MW. La course au nombre de coeurs est donc limitée par la consommation.

FIGURE 4.1 – À gauche le premier superordinateur *Cray-I*. À droite le superordinateur le plus puissant en juin 2015, *Tianhe-2*

À partir de 2001, certains précurseurs commencent à utiliser des cartes graphiques *Nvidia* pour le calcul scientifique en détournant des fonctions de la bibliothèque graphique OpenGL [43, 6, 37]. En 2007, *Nvidia* lance le premier kit de développement permettant un contrôle plus aisé des cartes dans un langage proche du *C*. La puissance des GPU est actuellement de l'ordre du Tflop/s (puissance équivalente aux superordinateurs de 2001) pour une consommation de seulement quelques centaines de Watt et pour un prix nettement plus faible (2000 euros).

Dans ce chapitre, nous commencerons par présenter les architectures GPU du point de vue matériel puis nous donnerons un bref exemple de type *Hello World* afin de familiariser le lecteur au langage OpenCL et à la programmation GPU. Nous détaillerons ensuite les stratégies que nous avons utilisées afin de résoudre le problème de la MHD sur ces architectures massivement parallèles et nous étudierons les améliorations que nous obtenons sur les temps de calcul. Nous verrons également que ces accélérations nous permettent alors de résoudre des problèmes sur des maillages très fins. Nous finirons ce chapitre avec une section consacrée à l'implémentation d'un algorithme hybride utilisant un couplage

entre la parallélisation GPU et la décomposition de domaine à l'aide de l'interface MPI permettant de résoudre le problème de la MHD sur une architecture multi-GPU.

4.1 Introduction à la programmation GPU

4.1.1 Architecture

De même que pour le CPU, il est important de bien comprendre le fonctionnement du GPU afin de pouvoir optimiser le code sur ce type de processeurs. Une carte graphique contient des milliers de coeurs de calcul optimisés pour travailler de manière parallèle. Ces coeurs de calcul (qui correspondent à des *work-item* dans la terminologie OpenCL et des *thread* dans celle de CUDA), sont regroupés dans des unités de calcul (correspondantes à des *work-group* OpenCL ou des *blocks* CUDA). Chacune de ces unités de calcul est donc comparable à un CPU : elles possèdent en effet toutes une mémoire locale, des registres, une unité de contrôle et plusieurs coeurs de calcul. À la différence d'un CPU, une unité de calcul de GPU possède cependant beaucoup plus de coeurs : typiquement quelques dizaines. De plus tous les coeurs partagent la même unité d'instruction : contrairement aux coeurs d'un CPU, les coeurs d'une même unité de calcul GPU doivent réaliser la même instruction en même temps (parallélisme SIMD : Single Instruction, Multiple Data). Chaque unité de calcul GPU possède une mémoire cache locale à accès rapide. Cependant, contrairement à celle d'un CPU multicoeur récent, qui présente une capacité de stockage de l'ordre du méga-octet, la mémoire locale dans un GPU est de l'ordre du kilo-octet. Les échanges de données entre les différentes unités de calcul se font au travers d'une mémoire partagée, appelée mémoire globale. Cette mémoire bien plus grande, de l'ordre du giga-octet, est en revanche plus lente que la précédente.

Une carte graphique ne peut pas fonctionner seule, elle nécessite obligatoirement le contrôle par le CPU qu'on appelle l'hôte. C'est lui qui transfère les données de la mémoire RAM à la mémoire globale du GPU. Ce transfert entre le CPU est le GPU est le plus long, il faut donc l'utiliser de manière efficace, l'idéal étant de ne faire qu'un transfert entre le GPU et le CPU en début et en fin de calcul, si la taille du problème le permet.

4.1.2 Introduction à la programmation OpenCL

Au cours de cette thèse, nous avons utilisé l'environnement OpenCL pour programmer le GPU. Dans la suite de ce chapitre, nous utiliserons donc la terminologie OpenCL. Actuellement géré par le groupe Khronos et inventé par Apple, qui en a libéré les droits, OpenCL est une bibliothèque de fonction permettant d'écrire des programmes parallèles. Un avantage important d'OpenCL sur son concurrent CUDA, développé par Nvidia est

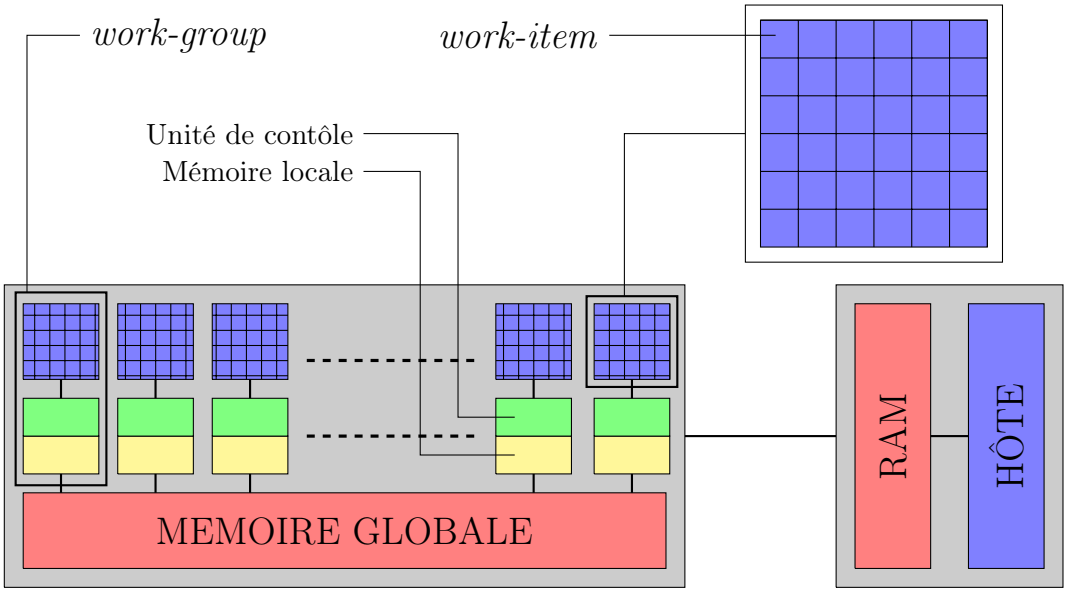

FIGURE 4.2 – Représentation schématique de l'architecture d'un GPU

que le même code OpenCL pour aussi tourner sur CPU multicoeur, sur GPU ou même encore sur les cartes Xeon Phi d'Intel. Dans la terminologie OpenCL, le GPU, le CPU ou la carte Xeon Phi sur lesquels sont réalisés les calculs parallèles est appelé accélérateur.

Le code principal est exécuté sur un coeur de CPU, appelé hôte. L'hôte envoie sur les accélérateurs disponibles les données et les programmes, appelés *kernel*, qui permettent de modifier ces données.

Afin d'illustrer simplement le fonctionnement d'un code OpenCL nous allons, dans cette partie donner les étapes principales d'écriture d'un programme calculant le carré des valeurs d'un tableau. En *C++*, cet algorithme s'écrirait :

```
void carre(int n, const float *a, float *c){
  for (int i=0; i<n; i++){
    c[i] = a[i] * a[i];
  }
}
```

Pour un tableau de taille n, un tel tableau effectuera donc n multiplications successives. Le kernel OpenCL consistant à effectuer cette tâche de manière parallèle sur tous les *work-item* du GPU s'écrit à la manière d'une fonction en langage *C* de la manière suivante

```
__kernel carre(const float *a, float *c){
  int id = get_global_id(0);
  c[id] = a[id] * a[id];
```

```
}
```

La fonction de type *__kernel* ne comporte aucune boucle et ne fait qu'une seule mise au carré de la valeur d'indice *id = get_global_id(0)*. Dans un GPU, chaque *work-item* est représenté par deux identifiants : un identifiant global unique propre à chaque *work-item* et un identifiant local permettant de différencier les *work-item* d'un même *work-group*.

Afin de pouvoir exécuter cette fonction sur un GPU, il faut dans un premier temps compiler cette fonction dans le GPU. Contrairement à la compilation d'un code classique, celle-ci se fait lors de l'exécution du programme hôte. Cette étape exécutée sur le CPU hôte s'effectue principalement à l'aide des trois fonctions présentées ci-dessous et d'un *context* OpenCL créé préalablement. Un contexte permet de gérer l'ensemble des objets ainsi que la mémoire et les accélérateurs sur lesquels sont lancés les kernels. La fonction à compiler est passée à la fonction *clCreateProgramWithSource* sous la forme d'une chaine de caractère nommé dans cet exemple *KernelSource*.

```
// Envoi de la fonction dans le GPU
program = clCreateProgramWithSource(
                                    context,
                                    1,
                                    (const char **) &KernelSource,
                                    NULL,
                                    NULL);

// Compilation de la fonction
clBuildProgram(program,
               0,
               NULL,
               NULL,
               NULL);

// Creation du noyau executable
kernel = clCreateKernel(program,
                        "carre",
                        NULL);
```

Lorsque tous les *kernel* sont compilés, nous devons ensuite, comme pour tout programme en *C*, allouer et initialiser dans le GPU les tableaux et variables nécessaires. Dans notre cas, nous avons un tableau de données en entrée et nous voulons un tableau contenant les résultats en sortie.

```
// Allocation de la memoire
donnees_GPU = clCreateBuffer(context,
```

```
                                CL_MEM_READ_ONLY,
                                sizeof(float) * n,
                                NULL,
                                NULL);
resultats_GPU = clCreateBuffer(context,
                                  CL_MEM_WRITE_ONLY,
                                  sizeof(float) * n,
                                  NULL,
                                  NULL);

// Initialisation des donn es
clEnqueueWriteBuffer(commands,
                     donnees_GPU,
                     CL_TRUE,
                     0, sizeof(float) * n,
                     donnees_CPU,
                     0,
                     NULL,
                     NULL);
```

À la suite de cela, le GPU possède tout les éléments dont il a besoin pour effectuer les opérations. Le CPU peut donc ordonner l'exécution du *kernel*. Cette exécution se fait en deux étapes. Premièrement, il faut indiquer au GPU quels sont les paramètres qui doivent être passé en argument dans la fonction, ensuite nous pouvons lancer la fonction en indiquant le nombre de *work-items* à utiliser ainsi que la répartition de ceux-ci à l'intérieur des *work-group*. Dans notre exemple, nous souhaitons simplement lancer la fonction sur n *work-item*. Nous verrons plus tard comment nous avons organisé cette répartition dans le cas du problème de la MHD.

```
// Passage des arguments
clSetKernelArg(kernel,
               0,
               sizeof(cl_mem),
               &donnees_GPU);
clSetKernelArg(kernel,
               1,
               sizeof(cl_mem),
               &resultats_GPU);

// Execution du kernel
clEnqueueNDRangeKernel(commands,
                       kernel,
                       1,
```

```
                        NULL,
                        &n,
                        NULL,
                        0,
                        NULL,
                        NULL);
```

Il ne nous reste finalement plus qu'à envoyer les résultats du GPU vers le CPU afin de la traiter.

```
// Envoie des resultats dans le CPU
clEnqueueReadBuffer(commands,
                    resultats_GPU,
                    CL_TRUE,
                    0,
                    sizeof(float) * n,
                    resultats_CPU,
                    0,
                    NULL,
                    NULL);
```

Dans l'annexe, nous donnons l'exemple complet commenté pour l'exemple ci-dessus.

4.2 Programmation GPU du problème de la MHD

La mémoire locale dans un GPU étant bien plus petite que la mémoire cache d'un CPU, la méthode de parcours par tuile présentée dans le chapitre précédent ne peut pas être appliquée de manière efficace sur une carte graphique puisque la taille des tuiles ne serait pas suffisante pour obtenir un gain. Dans cette section, nous allons décrire la méthode du *splitting de Strang* qui permet d'optimiser la mise à jour des champs en travaillant avec des données alignées en mémoire, on parle de données coalescentes.

4.2.1 Splitting de Strang

En dimension 2, le système d'équations de la MHD idéale s'écrit avec le flux $\mathbf{F}(\mathbf{W}, \mathbf{n})$ donné par l'équation (1.25 page 13),

$$\partial_t \mathbf{W} + \partial_x \mathbf{F}(\mathbf{W}, \mathbf{n}_x) + \partial_y \mathbf{F}(\mathbf{W}, \mathbf{n}_y) = 0. \tag{4.1}$$

Nous nous plaçons dans le cas d'une grille régulière, nous avons donc $\mathbf{n}_x = (1,0,0)^T$ et $\mathbf{n}_y = (0,1,0)^T$. De même, les variables conservatives sont données sur cette grille régulière, par commodité, nous adoptons la notation $\mathbf{W}(x,y) = \mathbf{W}$.

Afin de simplifier l'implémentation au maximum et pouvoir nous concentrer sur les optimisations OpenCL nous utiliserons la méthode du splitting directionnel [42, 51]. Pour chaque pas de temps, cette méthode consiste à résoudre, dans un premier temps, un problème unidimensionnel dans la direction x

$$\begin{array}{rcl} \partial_t \mathbf{V} + \partial_x \mathbf{F}(\mathbf{V}, \mathbf{n}_x) & = & 0, \\ \mathbf{V}(t=0) & = & \mathbf{W}_0, \end{array}$$

puis un deuxième problème unidirectionnel dans la direction y. Dans ce deuxième problème, l'état initial est donné par le résultat de la résolution précédente.

$$\begin{array}{rcl} \partial_t \mathbf{W} + \partial_y \mathbf{F}(\mathbf{W}, \mathbf{n}_y) & = & 0, \\ \mathbf{W}(t=0) & = & \mathbf{V}(t=\Delta t). \end{array}$$

4.2.2 Distribution des données en mémoire

Un des points importants de la programmation sur GPU est le transfert des données entre le GPU et le CPU. En effet ce transfert est lent et nous souhaitons par conséquent les éviter au maximum. Pour cela, nous choisissons d'envoyer une seule fois toutes les données du problème en début d'algorithme puis nous effectuons le transfert inverse seulement pour récupérer un résultat. La restriction qui s'impose donc immédiatement est celle de la place mémoire. En effet, la place disponible sur un GPU est plus faible que sur un calculateur standard et limite donc la taille du problème que nous pouvons traiter. Pour ces raisons, nous utiliserons dans la section suivante une décomposition de domaine afin de résoudre cette limitation, en utilisant plusieurs GPU.

Coalescence des données

L'obtention de transferts de données coalescents est une optimisation fondamentale dans la programmation GPU. En effet, comme nous le montrons sur la figure 4.3 page ci-contre, lorsque la carte graphique effectue une lecture ou une écriture en mémoire, elle traite en réalité un bloc de cellules mémoires. Sur la figure, nous prenons l'exemple où la lecture se fait sur cinq cellules. Si nous souhaitons lire cinq données dans un tableau nous voyons que si ces données ne sont pas voisines en mémoire, la lecture peut prendre jusqu'à cinq cycles au lieu d'un seul dans le cas contraire.

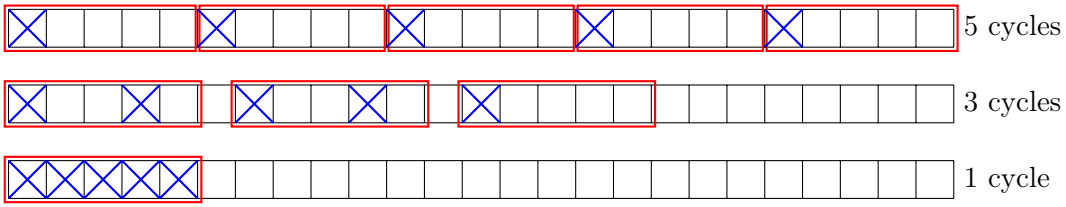

FIGURE 4.3 – Lecture de cinq valeurs en mémoire réparties de manières différentes dans la mémoire.

Organisation des données dans le cas de la MHD

Comme nous venons de le voir, la gestion des données en mémoire est un point essentiel dans un algorithme utilisant la parallélisation sur GPU. Sur un cas bidimensionnel, nous disposons de neuf variables pour chaque cellule sur une grille. Comme nous l'avons montré sur la figure 3.2 page 51, à chaque itération, chaque processeur a besoin de lire les cellules directement voisines dans la direction x puis dans la direction y. Afin de conserver les données le plus proches possible en mémoire, nous devons aligner les données différemment pour l'étape en x et pour l'étape en y. Le champ $\mathbf{W}_{i,j}^n$ s'écrit ainsi en fonction d'un seul indice, nous avons donc, si N_x représente le nombre de cellules dans la direction x,

$$\mathbf{W}_{i,j}^n = \mathbf{W}_{j\times N_x+i}^n$$

En nous référant à la figure 4.4 nous voyons que nous devons effectuer deux sauts de lecture en mémoire globale pour la mise à jour d'une cellule.

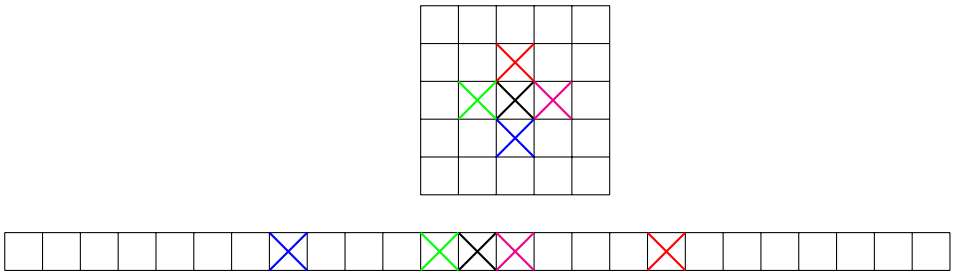

FIGURE 4.4 – Lecture des données dans une grille bi dimensionnelle et une grille linéarisée.

Dans la section précédente, nous avons défini une stratégie de programmation pour le cas bidimensionnel à l'aide du splitting de Strang. Cette méthode consistant à résoudre dans un premier temps un problème unidirectionnel en x puis un autre en y présente ici un avantage non négligeable. En effet, lors de la résolution dans la direction x, la mise à jour d'une valeur ne se fait qu'en prenant en compte les voisins dans la direction

x. De cette façon, les accès mémoires sont toujours coalescents. En revanche, lors de la résolution dans la direction y, ce n'est plus le cas. En effet, dans cette configuration, la lecture de ces données est la moins efficace possible puisqu'aucune des valeurs à lire ne sont voisines en mémoire. Afin d'améliorer ces accès mémoires, nous effectuons une transposition des données en mémoires entre les deux résolutions unidimensionnelles. De cette façon, les champs à lire présentés dans la figure 4.4 page précédente se trouvent maintenant dans la situation de la figure 4.5. Les lectures pour la résolution en y sont maintenant coalescentes et donc optimales.

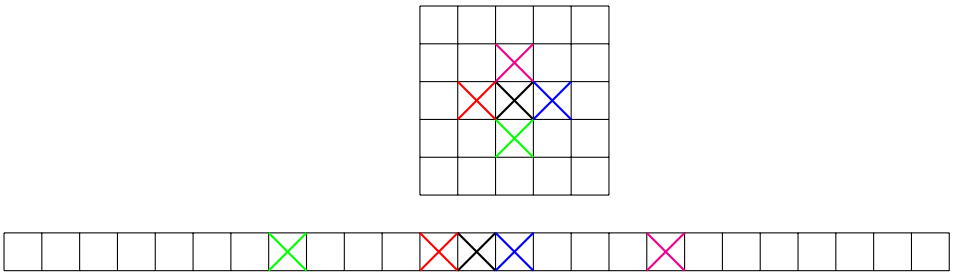

FIGURE 4.5 – Répartition des champs de la figure 4.4 après transposition de la grille.

La transposition d'une matrice de taille importante peut être très couteuse en termes de temps d'exécution à cause des nombreux accès en lecture et écriture. Nous utilisons donc un algorithme de transposition par blocs optimisé pour le calcul sur GPU [56].

4.2.3 Méthode de transposition

La transposition d'une matrice de grande taille peut être très coûteuse en temps de calcul si elle n'est pas réalisée correctement. Le tableau ne pouvant être stocké que dans la mémoire globale du GPU, les accès en lecture et en écriture sont relativement lents. La méthode de transposition que nous utilisons est représentée sur la figure 4.6 page suivante. Elle consiste à distribuer les données sous forme de blocs dans les différentes mémoires locales. Cette distribution est réalisée au moyen de transferts coalescents. [1] . Ensuite, la transposition de chaque sous-bloc se fait de manière parallèle dans les *work-group (WG)* [2] . Les accès mémoires sont rapides puisque les blocs sont à présent en mémoire locale. La dernière étape consiste enfin à transférer les blocs transposés en mémoire globale. Ce transfert peut lui aussi se faire de façon coalescente. [3] .

Le choix de la taille des sous-parties est un point clé dans cet algorithme. Outre l'accélération procurée par le travail en mémoire locale, cet algorithme profite également de la lecture par paquets de cellules voisines. La taille des blocs est alors choisie selon deux critères. De manière évidente, il faut que la sous-partie puisse entrer dans la mémoire locale. Le deuxième critère impose que la taille du bloc soit un multiple du nombre de valeurs contiguës lues par le GPU. En pratique nous utilisons des blocs de 32×32 valeurs.

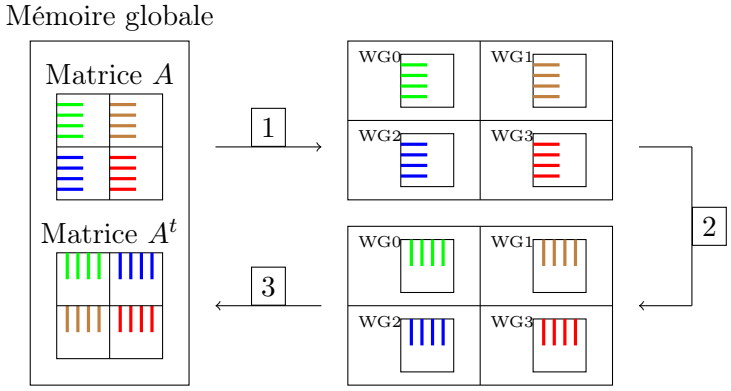

FIGURE 4.6 – Représentation des étapes effectuées lors de la transposition optimisée.

4.2.4 Distribution des tâches entre les processeurs

Après avoir alloué la mémoire et organisé la gestion de celle-ci, nous devons effectuer le partage des tâches entre les différents work-groups et work-items (WI). Cette étape est la deuxième étape essentielle permettant d'optimiser les performances du code. En effet, le travail à l'intérieur d'un work-group permet d'utiliser la mémoire locale rapide, mais cette mémoire est limitée. Dans le but d'éviter une lecture écriture intempestive en mémoire globale au cours du calcul, nous devons donc limiter la charge sur les work-group. A contrario, le nombre de groupes étant limité, une liste d'attente va se créer lors de l'exécution. De manière assez naturelle, nous souhaitons optimiser l'utilisation des groupes en choisissant de les faire travailler sur le maximum de leur capacité en mémoire.

Comme le work-group va copier en mémoire locale les données dont il a besoin, nous souhaitons que les copies soient coalescentes. Pour cela, nous avons choisi de confier la gestion d'une ligne de la matrice à chaque groupe. De cette manière, toutes les données qu'il cherche en mémoire sont voisines et le transfert est optimal. À l'intérieur d'un work-group, chaque work-item se voit attribuer le calcul sur une cellule (voir la figure 4.7 page suivante).

Pour certains gros cas de calcul, la taille de la matrice ne permet pas de stocker une ligne complète en mémoire locale. Dans ce cas-là, on ne passe alors qu'une partie de la ligne. Mais le principe général de l'algorithme reste inchangé.

4.2.5 Déroulement d'un pas de temps

Jusqu'à présent, nous avons, dans ce chapitre, décrit la méthodologie pour la mise en place de l'algorithme sur GPU. Nous avons également, dans les chapitres 2 page 25 et 3

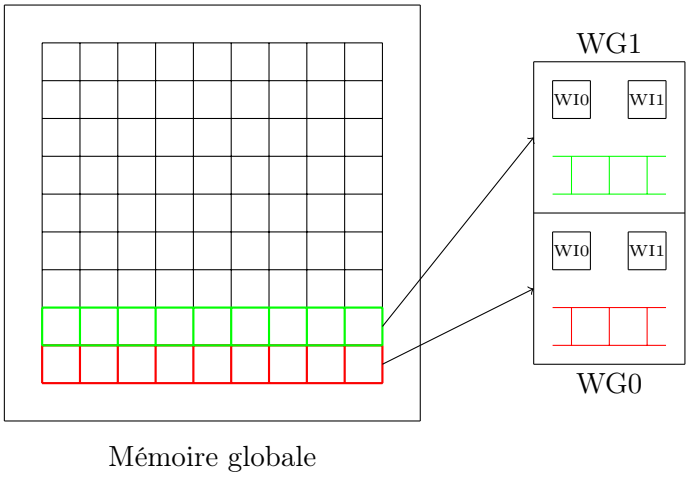

FIGURE 4.7 – Distribution des tâches dans le GPU

page 49 donné la méthode numérique à l'aide des volumes finis pour la résolution du système de la MHD. Pour terminer, nous donnons à présent un résumé du déroulement d'un pas de temps lors de la résolution.

1. **Résolution dans la direction** x. Chaque work-item charge en en mémoire locale les données d'une cellule. Une ligne de cellules est attribuée à chaque work-group.
2. Chaque work-item calcul le flux à gauche et le flux à droite.
3. Chaque work-item met à jour la valeur du champ de la cellule qui lui correspond dans la matrice globale puis copie le résultat en mémoire globale.
4. Application de la transposition de la matrice au moyen de l'algorithme présenté dans [56].
5. **Résolution dans la direction** y. On reprend les étapes 1, 2 et 3.
6. Application de la transposition. La matrice est maintenant prête pour le pas de temps suivant.

4.3 Performances

4.3.1 Modèle *roofline*

Quel que soit le matériel utilisé, l'efficacité d'un algorithme peut être limitée par deux facteurs indépendants. La première limitation est la puissance maximale de calcul du CPU ou GPU, qui est appelée la performance crête. La seconde limitation est imposée par la rapidité des transferts de données, c'est-à-dire la bande passante de la mémoire.

Le modèle de performance nommé le modèle *roofline* permet de déterminer l'efficacité d'un algorithme en prenant en compte ces deux paramètres.

L'intensité opérationnelle d'algorithme est le ratio entre le nombre d'opérations sur le nombre d'octets transférés.

$$IO = \frac{\text{nombre d'opération}}{\text{quantité de données lues/écrites}}$$

et son unité est le FLOP/octet. L'intensité opérationnelle d'un algorithme permet de mesurer son efficacité en termes de performance crête et de bande passante. En effet, si le nombre d'opérations est faible face au nombre de données lues ou écrites en mémoire, alors le facteur limitant sera la bande passante. Si par contre, le nombre d'opérations est dominant, c'est la performance crête qui domine. De cette manière, la performance maximale atteignable est donnée par

$$P_{\max} = \min(\text{bande passante} \times IO, \text{Performance crête}).$$

À l'aide de cela, nous pouvons, en calculant l'intensité opérationnelle de notre algorithme déterminer quelle est la performance maximale atteignable par celui-ci sur un accélérateur donné. Sur la figure 4.8 page suivante, nous représentons la performance maximale pour une carte AMD RADEON HD7970 ainsi que les intensités opérationnelles pour notre méthode de volumes finis à l'ordre 1 et à l'ordre 2. En déterminant manuellement le nombre d'opérations effectuées pour une variable, dans une cellule et pour une itération, nous pouvons déterminer le nombre total d'opérations effectuées. Nous constatons que ce nombre d'opérations domine dans ce modèle et que, sur ces cartes, nous sommes donc limités par la performance crête qui est de 2,8TFlop/s.

4.3.2 Temps mesurés

Afin de mesurer les performances de notre implémentation OpenCL, nous reprenons le test réalisé dans le chapitre 3 page 49, à savoir la résolution d'un test bidimensionnel sur une grille régulière 3000×3000. Pour comparer les temps obtenus avec la précédente implémentation, nous utilisons toujours le flux $P2$ à l'ordre 2. Dans le tableau 4.1 page suivante, nous ajoutons aux résultats précédents (tableau 3.2 page 60) les temps mesurés à l'aide de notre algorithme OpenCL.

Nous avons utilisé plusieurs architectures afin d'étudier le comportant d'une telle implémentation. Nous avons tout d'abord utilisé un simple ordinateur portable Dell M4500 composé d'un CPU Intel i7 M640 (2 coeurs) et d'un GPU NVidia Quatro FX 880M.

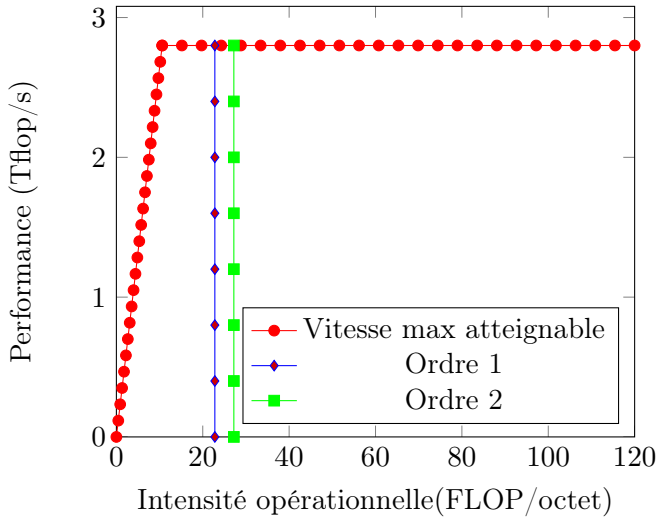

FIGURE 4.8 – Modèle roofline pour la méthode volumes finis à l'ordre 1 et 2. La vitesse maximale atteignable est celle d'une carte AMD RADEON HD7970.

Architecture	Temps (sec)	Accélération
Code séquentiel non optimisé	2605629 (30j)	1
Code séquentiel optimisé	526110 (146h)	5
Code séquentiel optimisé 2	351663 (97h)	8
Code OpenMP (16 coeurs)	22398 (6.2h)	116
(OCL) Intel i7 M640 (2 coeurs)	199368 (55h)	13
(OCL) Intel Xeon E5 2630 (6 coeurs)	65071 (18h)	40
(OCL) NVidia Quatro FX 880M	41535 (11h)	62
(OCL) NVidia Tesla K20	1204 (20 min)	2160
(OCL) AMD HD Radeon 7950	982 (16 min)	2650

TABLE 4.1 – Comparaison des temps d'exécution en fonction des algorithmes et des architectures

Bien que les temps de calcul ne rivalisent pas avec une l'implémentation OpenMP sur 16 coeurs. Pour un prix bien inférieur à celui de l'achat d'un tel CPU, l'implémentation OpenCL sur le CPU gagne tout de même un facteur 1.6 par rapport à l'implémentation séquentielle optimisée et un facteur 7.75 avec le GPU. L'implémentation sur un processeur plus performant, le Intel Xeon E5 2630 (6 coeurs) nous conduit à une conclusion équivalente. L'accélération obtenue avec ces 6 coeurs est d'un facteur 5, bien que le code soit plus complexe à développer, notre implémentation OpenCL atteint sur CPU, à nombre de coeurs équivalent, des performances proches de celle obtenue avec OpenMP.

Sur les deux dernières architectures GPU, bien plus performantes, notre implémentation se démarque significativement. Les accélérations obtenues par rapport à l'algorithme séquentiel optimisé sont de facteurs 270 et 330. Un résultat intéressant est à noter. Bien que le NVidia Tesla K20 soit plus performant que l' AMD HD Radeon 7950, il obtient de moins bonnes performances. Nous pouvons donc supposer que les drivers OpenCL développés par NVidia sont moins optimisés que ceux développés par AMD. Rappelons toutefois que la bibliothèque OpenCL est développée par AMD et que NVidia gère la bibliothèque CUDA.

4.4 Multi-GPU

Comme nous l'avons vu dans la section précédente, la programmation sur GPU nous permet d'obtenir de bonnes accélérations des calculs. Cependant, un des freins à l'utilisation de ce type de parallélisme est la taille réduite de la mémoire des cartes graphiques. En effet, les cartes actuelles possèdent une mémoire de l'ordre de 3Go. Afin d'avoir les meilleures performances, nous devons pouvoir charger l'intégralité des données dans la mémoire globale afin de limiter au maximum la perte de temps causée par les transferts entre le CPU et le GPU. Cette volonté n'est donc pas compatible avec la simulation d'un phénomène sur un maillage très fin. Nous souhaitons pouvoir traiter des maillages de plusieurs milliards de cellules en double précision. Pour le problème de la MHD supposons que nous n'avons besoin de garder que l'état du système à deux instants. Dans ces conditions, la taille mémoire nécessaire en octets est donc

$$NX \times NY \times 8 \times 9 \times 2$$

où NX et NY représentent respectivement le nombre de mailles dans les directions x et y. Les valeurs 8 représentent la taille en octet d'un réel en double précision, et le nombre de variables du système. Enfin, la valeur 2 désigne le nombre d'instants du système que nous conservons, c'est-à-dire le nombre de tableaux que nous devons conserver en mémoire.

On peut donc estimer qu'une carte graphique disposant de 3Go de mémoire ne nous permet pas de traiter un problème allant au-delà d'un maillage d'environ 4000×4000 mailles pour notre problème de MHD. Les machines de calcul récentes possèdent des noeuds de calcul composés de plusieurs cartes graphiques gérées par un même CPU hôte (voir figure 4.9 page suivante).

Dans la suite de ce chapitre, nous présentons une méthode basée sur la décomposition de domaine afin d'utiliser ce type d'architecture.

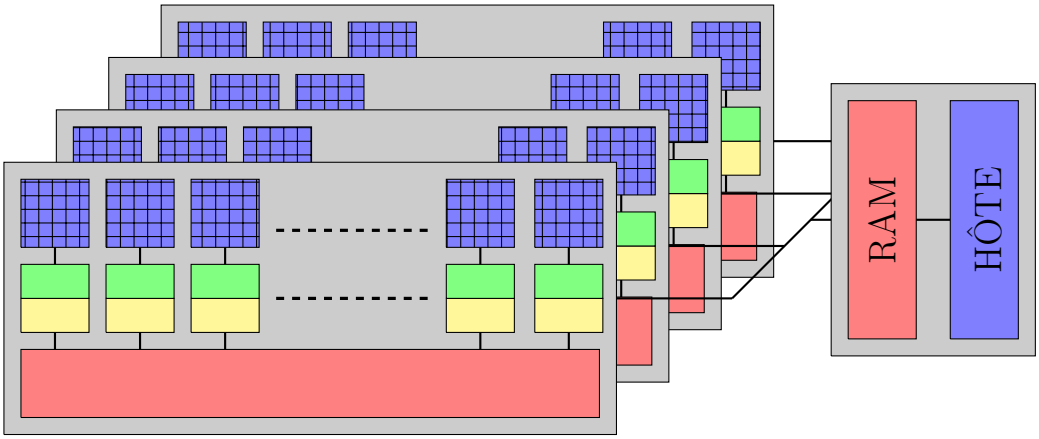

FIGURE 4.9 – Représentation schématique d'un ensemble de GPU

4.4.1 Décomposition de domaine

Afin de pouvoir traiter des maillages plus fins, nous utilisons une méthode de décomposition de domaine. Comme son nom l'indique, la méthode permet de faire un découpage en espace du maillage afin de traiter individuellement chaque partie. Notre but étant d'obtenir un algorithme de résolution rapide sur GPU, nous partageons les différents sous-domaines entre plusieurs GPU. Chacun effectuant une résolution parallèle telle qu'expliquée dans la section précédente.

Le découpage du maillage peut se faire de multiples façons différentes, cependant, pour des questions de simplicité et d'efficacité, nous avons décomposé le maillage que suivant l'axe en y (voir figure 4.10).

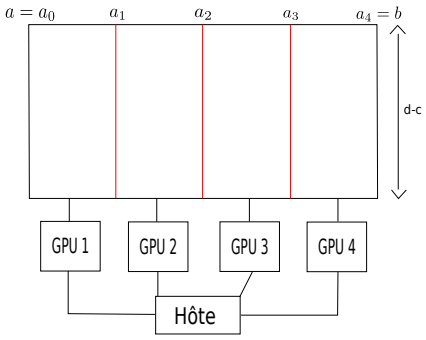

FIGURE 4.10 – Exemple du partage d'un maillage sur quatre cartes graphiques

Cette configuration nous permet de gérer plus simplement les conditions aux frontières. En effet, la méthode nous impose le maintien d'une zone tampon entre les sous-parties. Ce chevauchement permet à une sous-partie, par exemple la sous-partie la plus à gauche

de connaître les valeurs des champs de la sous-partie voisine. Les valeurs de ces cellules doivent être échangées à chaque itération en passant par le CPU qui joue le rôle d'hôte. Nous avons vu précédemment que le transfert entre le GPU et le CPU sont lents. Pour cela, nous devons limiter la zone de recouvrement. Le nombre de mailles nécessaire est identique à celui nécessaire pour les conditions de bord et dépend donc de l'ordre auquel nous travaillons. À l'ordre 1, nous aurons besoin de connaître les états du système sur une cellule voisine donc un recouvrement de deux cellules est suffisant. À l'ordre 2 en revanche, nous avons besoin de deux cellules voisines donc les recouvrements doivent être de quatre cellules. Sur la figure 4.11 nous présentons une représentation de la zone de recouvrement entre les sous-domaines.

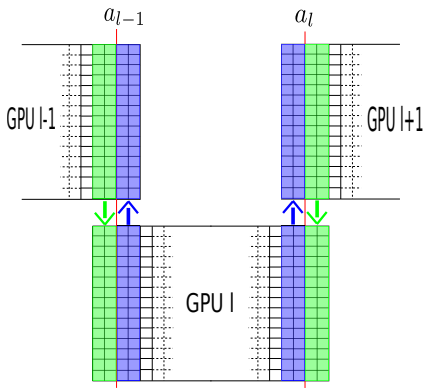

FIGURE 4.11 – Cellules de recouvrement entre les sous-parties à l'ordre 1

4.4.2 Temps mesurés

Dans le tableau 4.2 page suivante nous présentons de manière globale les accélérations que nous avons obtenues avec toutes les méthodes d'optimisation présentées précédemment. Nous pouvons constater que l'utilisation de plusieurs cartes graphiques permet non seulement d'effectuer des simulations sur des maillages de tailles plus importantes, mais également de réduire les temps de calcul de manière significative. En effet, nous montrons dans ce tableau que l'utilisation de 4 cartes de type Tesla K20 nous fournit des accélérations d'un facteur proche de 4.

4.5 Résultats numériques

Dans la fin de ce chapitre, nous donnons un ensemble de solutions obtenues à l'aide du modèle en volumes finis que nous avons présenté. Dans les figures 4.13 page 81, 4.14 page 81, 4.15 page 82 et 4.16 page 82 nous présentons les solutions au temps $0,5$, $1,5$

Architecture	Temps (sec)	Acceleration
Code séquentiel non optimisé	2605629 (30j)	1
Code séquentiel optimisé	526110 (146h)	5
Code séquentiel optimisé 2	351663 (97h)	8
Code OpenMP (16 coeurs)	22398 (6.2h)	116
(OCL) Intel i7 M640 (2 coeurs)	199368 (55h)	13
(OCL) Intel Xeon E5 2630 (6 coeurs)	65071 (18h)	40
(OCL) NVidia Quatro FX 880M	41535 (11h)	62
(OCL) NVidia Tesla K20	1204 (20 min)	2160
(OCL) AMD HD Radeon 7950	982 (16 min)	2650
(OCL+MPI) 4× Tesla K20	332s (5 min)	7848

TABLE 4.2 – Comparaison des temps d'exécution en fonction des algorithmes et des architectures

$2, 5$ et 3 pour le cas test de Orzsag et Tang [50]. Les conditions initiales du système sont données dans le tableau 4.12.

Variable	État initial
γ	$5/3$
ρ	γ^2
u_x	$-\sin(y)$
u_y	$\sin(x)$
u_z	0
$B_i x$	$-\sin(y)$
B_y	$-\sin(2x)$
B_z	0
p	γ

FIGURE 4.12 – Conditions initiales pour le cas test du vortex de Orzsag-Tang

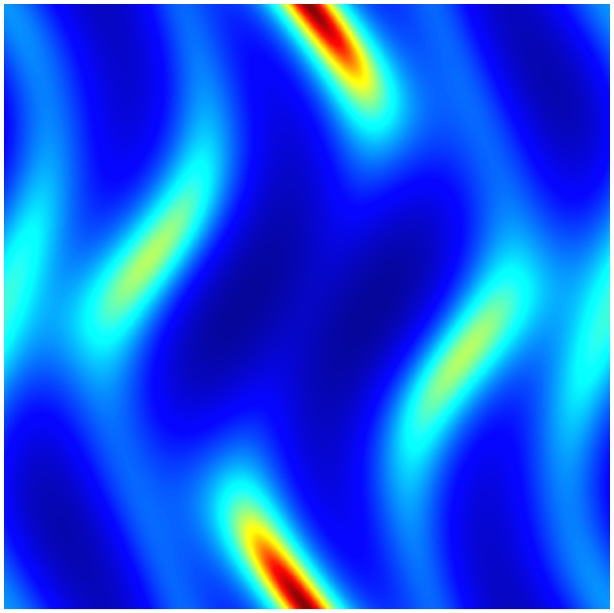

FIGURE 4.13 – Solution pour le cas test de Orzsag-Tang au temps $0,5$. Simulation effectuée à l'ordre 2 avec le flux $P2$ sur un maillage de taille 4000×4000

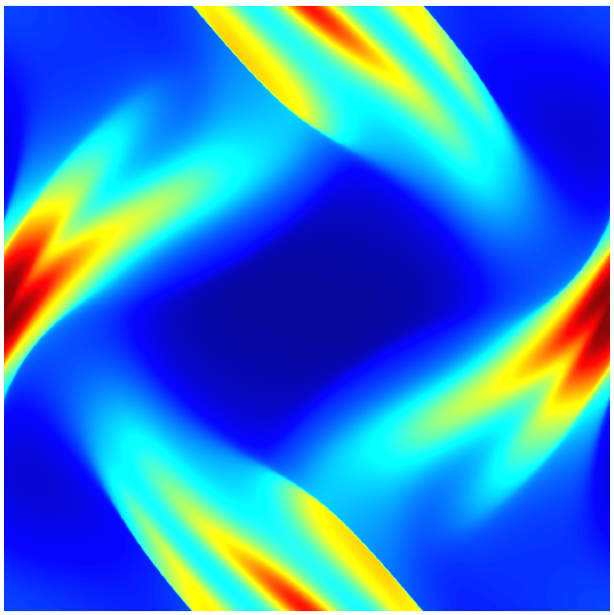

FIGURE 4.14 – Solution pour le cas test de Orzsag-Tang au temps $1,5$. Simulation effectuée à l'ordre 2 avec le flux $P2$ sur un maillage de taille 4000×4000

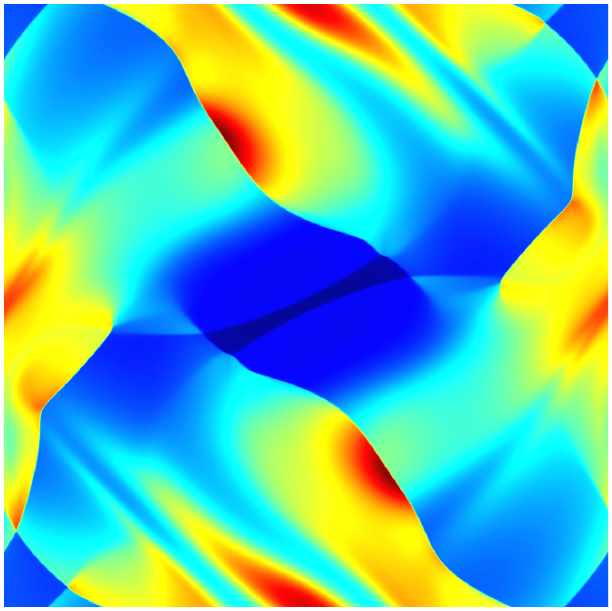

FIGURE 4.15 – Solution pour le cas test de Orzsag-Tang au temps $2,5$. Simulation effectuée à l'ordre 2 avec le flux $P2$ sur un maillage de taille 4000×4000

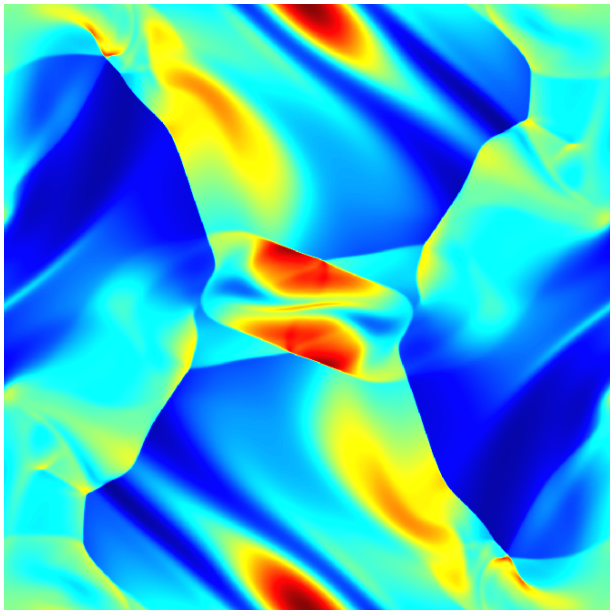

FIGURE 4.16 – Solution pour le cas test de Orzsag-Tang au temps 3. Simulation effectuée à l'ordre 2 avec le flux $P2$ sur un maillage de taille 4000×4000

Les simulations ont été réalisées sur une seule carte graphique avec un maillage de

4000×4000 cellules. Sur les solutions, nous pouvons remarquer que ce test est très turbulent et fait intervenir de nombreux chocs. Comme nous l'avions dit précédemment, nous pouvons augmenter la taille des maillages utilisés en utilisant la version multi-GPU de notre implémentation. Sur les figures 4.17 et 4.18 page suivante nous présentons des zones extraites de solutions obtenues aux temps 1 et 2 du cas test d'Orzsag-Tang. Les simulations ont été réalisées sur un ensemble de 4 cartes graphiques et sur un maillage de taille 15000×15000.

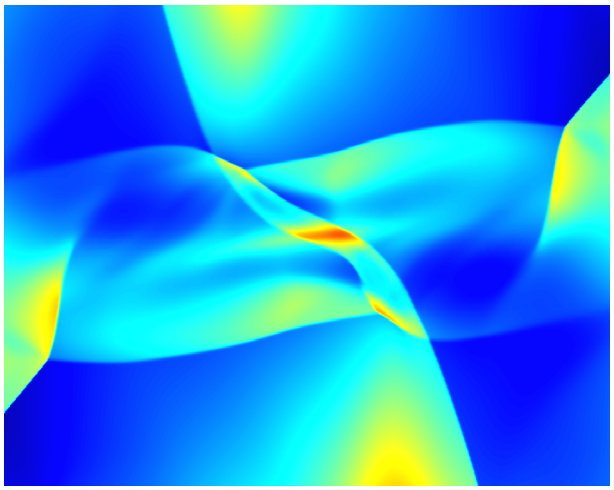

FIGURE 4.17 – Zoom sur une partie de la solution pour le cas test de Orzsag-Tang au temps 1. Simulation effectuée à l'ordre 2 avec le flux $P2$ sur un maillage de taille 15000×15000

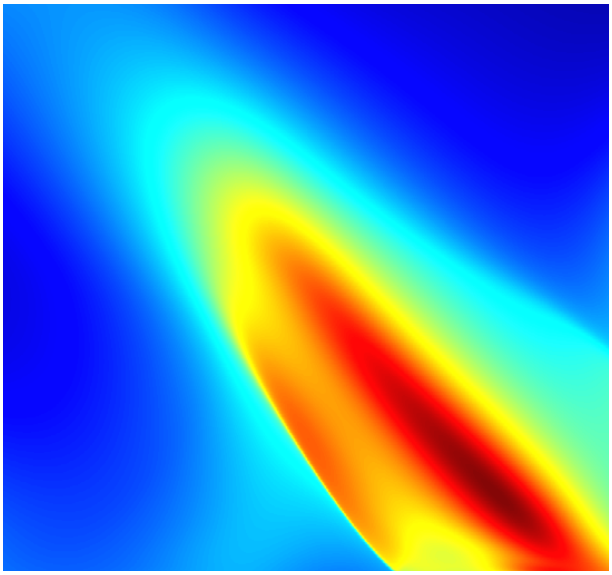

FIGURE 4.18 – Zoom sur une partie de la solution pour le cas test de Orzsag-Tang au temps 2. Simulation effectuée à l'ordre 2 avec le flux $P2$ sur un maillage de taille 15000×15000

4.6 Conclusion

Après une courte introduction sur la programmation sur carte graphique, nous avons commencé ce chapitre en présentant une méthode de splitting directionnel que nous utilisons pour la résolution en deux dimensions. Le splitting de Strang permet en effet de résoudre un problème en deux dimensions en ne résolvant que des problèmes en une dimension. Ce choix présente non seulement l'avantage d'être un moyen simple de passer sur un algorithme en dimension supérieur, mais également d'être une solution efficace pour apporter des optimisations supplémentaires sur GPU. En effet, la lecture/écriture des données par blocs est optimale pour des lectures de données voisines. En couplant ce splitting directionnel avec une méthode de transposition optimisée nous nous assurons d'avoir une coalescence maximale. La mesure de la performance de notre implémentation nous a permis de montrer que l'utilisation d'un code OpenCL sur CPU peut apporter des accélérations similaires à celles obtenues à l'aide de méthode automatique telle que la parallélisation avec OpenMP. Sur carte graphique, le nombre d'unités de calcul étant bien plus important que sur un CPU, nous avons pu obtenir des accélérations d'environ 300 sur un unique GPU. L'utilisation de 4 GPU simultanément accélère encore le code d'un facteur proche de 4. Le code obtenu nous a permis de réaliser des simulations MHD sur un maillage de taille 15000×15000.

Chapitre 5

SCHNAPS : Solveur Conservatif Hyperbolique Non-linéaire Appliqué aux PlaSmas

> If a man's wit be wandering, let him study the mathematics.
>
> Francis Bacon

Sommaire

5.1 Introduction 86
5.2 Méthode Galerkin Discontinu (GD) 86
5.2.1 Systèmes de lois de conservation 86
5.2.2 Flux numérique 87
5.2.3 Formalisme GD général 89
5.3 SCHNAPS : philosophie 91
5.3.1 StarPU 91
5.3.2 MPI 92
5.3.3 OpenCL 92
5.3.4 Modèles physiques 93
5.3.5 Sous-domaines 94
5.3.6 Macrocellules et champs 94
5.3.7 Simulation 95
5.3.8 Éléments géométriques 97
5.3.9 Interpolation 100
5.3.10 Interpolation de Gauss-Lobatto avec des sous-cellules 101

5.3.11 Schéma GD pour l'interpolation de Gauss-Lobatto sur des sous-cellules . . . 104
5.4 Programmation de la MHD dans SCHNAPS . . . 106
5.4.1 Commentaires sur la programmation . . . 106
5.4.2 Premières validations . . . 107
5.4.3 Tests de performances . . . 112
5.5 Conclusion . . . 118

5.1 Introduction

Dans ce chapitre, nous rappelons, avec plus de détail qu'au chapitre 2 page 25, le formalisme de la méthode Galerkin Discontinu (GD). Cette méthode est très générale puisqu'elle autorise des maillages non conformes ainsi que des degrés d'approximation non uniformes. Cependant, sa programmation dans un cadre trop général risque d'être inefficace sur un ordinateur hybride possédant plusieurs niveaux de mémoires et de parallélisme : les indirections mémoire, les trop grandes variations de paramètres, un rangement arbitraire des données en mémoire peuvent conduire à des performances décevantes. Nous allons donc présenter ici les choix que nous avons faits afin d'obtenir un bon compromis entre la généralité de la méthode et son efficacité. Le résultat de ces compromis est le logiciel SCHNAPS développé depuis deux ans à l'IRMA Strasbourg.

SCHNAPS est un acronyme de *Solveur Conservatif Hyperbolique Non-linéaire Appliqué aux PlasmaS*. Il s'agit d'une bibliothèque C99 pour résoudre numériquement des systèmes de lois de conservation sur des ordinateurs massivement parallèles, avec plusieurs niveaux de parallélisme, par exemple un gros cluster comprenant à la fois des CPU et des GPU.

5.2 Méthode Galerkin Discontinu (GD)

5.2.1 Systèmes de lois de conservation

SCHNAPS se veut assez général. Par conséquent, nous considérons un domaine ouvert borné $\Omega \subset \mathbb{R}^3$ de frontière $\partial\Omega$. Sur ce domaine, nous considérons un système de lois de conservation de la forme

$$\partial_t \mathbf{W} + \partial_i \mathbf{F}^i(\mathbf{W}) = \mathbf{S}(\mathbf{W}). \tag{5.1}$$

Dans cette équation, l'inconnue $\mathbf{W}(\mathbf{x}, t)$ est un vecteur de $\mathbb{R}^m$ qui dépend d'une variable d'espace $\mathbf{x} \in \Omega$, $\mathbf{x} = (x_1, x_2, x_3)$ et du temps $t \in [0, T]$. Le vecteur $\mathbf{S}$ représente les termes

sources du système. Nous utilisons la convention de sommation sur les indices répétés. Soit un vecteur $\mathbf{n} = (n_1, n_2, n_3) \in \mathbb{R}^3$, nous définissons le flux du système (5.1 page précédente) par

$$\mathbf{F}(\mathbf{W}, \mathbf{n}) = \mathbf{F}^i(\mathbf{W}) n_i. \tag{5.2}$$

Il faut adjoindre des conditions aux limites à (5.1 page ci-contre). Formellement, ces conditions aux limites sont données par un flux frontière $\mathbf{F}_b$ et s'écrivent

$$\mathbf{F}(\mathbf{W}, \mathbf{n}) = \mathbf{F}_b(\mathbf{W}, \mathbf{n}), \quad \mathbf{x} \in \partial\Omega, \tag{5.3}$$

où $\mathbf{n}$ désigne le vecteur normal sortant à Ω sur $\partial\Omega$. D'autre part, nous nous donnons aussi une condition initiale

$$\mathbf{W}(\mathbf{x}, 0) = \mathbf{W}_0(\mathbf{x}), \quad \mathbf{x} \in \Omega.$$

SCHNAPS permet de résoudre ce problème d'évolution par la méthode de Galerkin Discontinu (GD). Comme son nom l'indique, cette méthode consiste à construire une approximation de la solution au moyen d'éléments finis discontinus.

5.2.2 Flux numérique

La méthode GD est une généralisation de la méthode des éléments finis et de la méthode des volumes finis. Elle nécessite la définition d'un flux numérique sur les discontinuités de la solution discrète. Dans la section 2.2 page 32 nous avions présenté plusieurs exemples de flux que nous utilisons dans notre implémentation. Nous rappelons ici la notation utilisée à savoir

$$\mathbf{F}(\mathbf{W}_L, \mathbf{W}_R, \mathbf{n}_{LR}), \tag{5.4}$$

où $\mathbf{W}_L$ et $\mathbf{W}_R$ représentent les valeurs de $\mathbf{W}$ de part et d'autre de la discontinuité, et $\mathbf{n}_{LR}$ un vecteur normal à la discontinuité orienté du côté L vers le côté R. Cette notation, bien que légèrement abusive, ne peut pas être confondue avec celle du flux (5.2), car le flux numérique dépend de deux états, $\mathbf{W}_L$ et $\mathbf{W}_R$, au lieu d'un seul. Comme nous l'avions annoncé précédemment dans la section 2.1.1 page 25, pour que l'approximation GD soit consistante avec le système (5.1 page ci-contre), il faut que

$$\forall \mathbf{W}, \mathbf{n}, \quad \mathbf{F}(\mathbf{W}, \mathbf{W}, \mathbf{n}) = \mathbf{F}(\mathbf{W}, \mathbf{n}). \tag{5.5}$$

De plus, le flux numérique vérifie généralement une propriété de conservation

$$\forall \mathbf{W}_L, \mathbf{W}_R, \mathbf{n}, \quad \mathbf{F}(\mathbf{W}_L, \mathbf{W}_R, \mathbf{n}) = -\mathbf{F}(\mathbf{W}_R, \mathbf{W}_L, -\mathbf{n}), \tag{5.6}$$

mais ce n'est pas toujours le cas, et ce n'est pas une nécessité dans SCHNAPS.

À partir de la fonction flux, nous pouvons retrouver les composantes du flux. Nous introduisons le symbole de Kronecker

$$\delta_i^j = \begin{cases} 1 & \text{si} \quad i = j, \\ 0 & \text{si} \quad i \neq j. \end{cases}$$

Le vecteur $\boldsymbol{\delta}^j$ est un vecteur unitaire de $\mathbb{R}^3$ qui pointe dans la direction x_j. Alors

$$\mathbf{F}^j(\mathbf{W}) = \mathbf{F}(\mathbf{W}, \boldsymbol{\delta}^j). \tag{5.7}$$

Par exemple, les équations de Maxwell entrent dans ce formalisme. Elles s'écrivent

$$\partial_t \mathbf{E} - \nabla \times \mathbf{H} = -\mathbf{J}, \tag{5.8}$$
$$\partial_t \mathbf{H} + \nabla \times \mathbf{E} = 0, \tag{5.9}$$

où $\mathbf{E}$ est le champ électrique, $\mathbf{H}$ le champ magnétique et $\mathbf{J}$ le vecteur courant électrique. Nous posons

$$\mathbf{W} = (\mathbf{E}^T, \mathbf{H}^T)^T, \quad \mathbf{S} = (-\mathbf{J}^T, (0,0,0)^T)^T,$$

et

$$\mathbf{F}(\mathbf{W}, \mathbf{n}) = \begin{bmatrix} 0 & \mathbf{n}\times \\ -\mathbf{n}\times & 0 \end{bmatrix} \mathbf{W} = \mathbf{A}(\mathbf{n})\mathbf{W},$$

de telle façon que les équations de Maxwell sont un cas particulier de (5.1 page 86), avec $m = 6$.

Comme dans (5.7) nous pouvons définir les matrices symétriques 6×6

$$\mathbf{A}^i = \mathbf{A}(\delta^i), \quad i = 1 \cdots 3.$$

Alors les équations de Maxwell peuvent aussi être écrites sous la forme d'un système hyperbolique, aussi appelé système de Friedrichs

$$\partial_t \mathbf{W} + \mathbf{A}^i \partial_i \mathbf{W} = \mathbf{S}.$$

Dans le cas du modèle pour la MHD avec correction de la divergence proposée par Dedner [18], nous rappelons que le système hyperbolique est donné par (cf. chapitre 1 page 5)

$$\partial_t \begin{pmatrix} \rho \\ \rho\mathbf{u} \\ Q \\ \mathbf{B} \\ \psi \end{pmatrix} + \nabla \cdot \begin{pmatrix} \rho\mathbf{u} \\ \rho\mathbf{u} \otimes \mathbf{u} + (p + \frac{\mathbf{B}\cdot\mathbf{B}}{2})\mathbf{I} - \mathbf{B} \otimes \mathbf{B} \\ (Q + p + \frac{\mathbf{B}\cdot\mathbf{B}}{2})\mathbf{u} - (\mathbf{B}\cdot\mathbf{u})\mathbf{B} \\ \mathbf{u} \otimes \mathbf{B} - \mathbf{B} \otimes \mathbf{u} + \psi\mathbf{I} \\ c_h^2 \mathbf{B} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}.$$

Le flux numérique utilisé pour la MHD peut par exemple être le flux numérique décentré P2 (voir 1.25 et 2.15).

5.2.3 Formalisme GD général

Pour définir l'approximation GD, nous devons d'abord construire un maillage de Ω. Nous considérons un maillage constitué d'un nombre fini d'ensembles ouverts $L_k \subset \Omega$, $k = 1 \cdots N$, appelés *cellules* ou *éléments*, et satisfaisant les deux conditions :

1. $\forall k, l \quad k \neq l \Rightarrow L_k \cap L_l = \emptyset$.
2. $\overline{\cup_k L_k} = \overline{\Omega}$.

Soient L et R deux cellules voisines. La face commune à L et R est notée

$$L/R = \overline{L} \cap \overline{R}.$$

Nous notons aussi $\mathbf{n}_{LR}$ le vecteur normal unitaire sur L/R orienté L vers R. Donc, $\mathbf{n}_{LR} = -\mathbf{n}_{RL}$.

Dans chaque cellule L, nous construisons une base de fonctions scalaires φ_i^L, $i = 1 \cdots p_L$ dont le support est dans L. Il est possible d'avoir des niveaux d'approximation différents

p_L sur des cellules différentes. Dans la cellule L, la solution de (5.1*page* 86) est approchée par (avec somme sur les indices répétés)

$$\mathbf{W}(\mathbf{X},t) = \mathbf{W}_L(\mathbf{X},t) = \mathbf{W}_L^j(t)\varphi_j^L(\mathbf{X}) \quad \mathbf{X} \in L. \tag{5.10}$$

La solution numérique satisfait le schéma d'approximation GD

$$\forall L, \forall i \quad \int_L \partial_t \mathbf{W}\varphi_i^L - \int_L \mathbf{F}(\mathbf{W}, \nabla\varphi_i^L) + \int_{\partial L} \mathbf{F}(\mathbf{W}_L, \mathbf{W}_R, \mathbf{n}_{LR})\varphi_i^L = \int_L \mathbf{S}\varphi_i^L. \tag{5.11}$$

Pour plus de clarté, nous pouvons faire les remarques suivantes sur le schéma GD (5.11) :

1. La formulation (5.11) est obtenue formellement en multipliant (5.1 page 86) par une fonction de base φ_i^L et en intégrant par parties sur la cellule L.
2. Nous utilisons (de façon abusive) la même notation la solution exacte et la solution approchée. En général, à partir de maintenant $\mathbf{W}$ désignera l'approximation GD de la solution exacte.
3. Par R nous désignons une cellule générique qui touche la cellule L le long de sa frontière ∂L. Cette notation est justifiée par le fait que, à une rotation près, on peut toujours supposer que le vecteur normal $\mathbf{n}_{LR}$ est orienté de la cellule L à gauche (Left), vers la cellule R à droite (Right).
4. Comme $\mathbf{W}$ est discontinu sur le bord de la cellule, il n'est pas possible de définir $\mathbf{F}(\mathbf{W}, \mathbf{n}_{LR})$ sur ∂L. Par conséquent, comme dans la méthode des volumes finis, nous devons utiliser un flux numérique $\mathbf{F}(\mathbf{W}_L, \mathbf{W}_R, \mathbf{n}_{LR})$, introduit dans la section précédente. Le flux numérique satisfait souvent les deux conditions
   (a) Consistance : $\mathbf{F}(\mathbf{W}, \mathbf{W}, n) = \mathbf{F}(\mathbf{W}, \mathbf{n})$.
   (b) Conservation : $\mathbf{F}(\mathbf{W}_L, \mathbf{W}_R, \mathbf{n}_{LR}) = -\mathbf{F}(\mathbf{W}_R, \mathbf{W}_L, \mathbf{n}_{RL})$.
5. Dans notre cas, pour les simulations MHD, nous utiliserons en général le flux numérique P2.
6. Finalement, dans (5.11) nous devons être plus précis lorsque la cellule L touche le bord du domaine de calcul. En effet, sur $\partial L \cap \Omega$ le champ W_R n'est pas disponible. Nous remplaçons donc sur ces interfaces le flux numérique $\mathbf{F}(\mathbf{W}_L, \mathbf{W}_R, \mathbf{n}_{LR})$ par le flux frontière

$$\mathbf{F}_b(\mathbf{W}_L, \mathbf{n}_{LR}).$$

Nous pouvons alors introduire le développement (5.10) dans (5.11). L'approximation GD devient alors un système d'équations différentielles ordinaires satisfaites par les coefficients $\mathbf{W}_L^j(t)$ sur les bases locales. Nous résolvons ce système d'équations différentielles par un intégrateur Runge-Kutta du second ordre.

5.3 SCHNAPS : philosophie

Dans cette section, nous tentons de donner une vue d'ensemble synthétique de la conception du logiciel.

5.3.1 StarPU

Afin d'exploiter les architectures d'ordinateurs hybrides les plus récentes, il est indispensable de pouvoir soumettre de façon asynchrone des opérations de calcul ou de transferts (copies mémoire ou communications). Ces opérations, appelées tâches (*tasks* en anglais), dépendent en général les unes des autres. Ce sont les dépendances entre les tâches qui permettent de déterminer l'ordre dans lequel elles doivent être exécutées et si certaines d'entre elles peuvent être réalisées en parallèle.

Les dépendances entre tâches sont souvent représentées par un graphe. Les noeuds du graphe correspondent aux tâches. La présence d'un arc orienté de la tâche A vers la tâche B indique qu'il faut attendre la fin de A avant de lancer B. Voir Figure 5.2 page 96. En général, la construction d'un graphe des tâches ne nécessite que la connaissance des dépendances entre les données. Il suffit d'indiquer sur quelles données chaque tâche va opérer et si ces données sont accédées en lecture (R), écriture (W) ou lecture/écriture (RW). Il existe une littérature plus qu'abondante sur ces techniques algorithmiques. Ce n'est pas l'objet de cette thèse de rentrer trop dans les détails. Nous renvoyons par exemple à [13, 12].

La programmation efficace et générale de ce type d'algorithme est compliquée [61]. Nous avons donc décidé de déléguer la répartition des tâches sur les processeurs disponibles à une bibliothèque informatique spécialisée. Ce type de bibliothèque, aussi appelée *support d'exécution* (ou *runtime* en anglais), permet de décrire à la fois le code des tâches ainsi que le type d'accès aux données.

Nous avons choisi le support d'exécution StarPU `http://starpu.gforge.inria.fr/`. Il s'agit d'un logiciel développé depuis plus de 10 ans à l'Inria Bordeaux [3]. Au coeur de StarPU il y a les *codelettes* (ou *codelet* en anglais). Une codelette est une fonction écrite en C99 qui réalise l'implémentation d'une tâche. Une même tâche peut avoir plusieurs implémentations, donc plusieurs codelettes, différentes. Par exemple, pour une opération de calcul donnée, il est possible de programmer plusieurs codelettes avec des optimisations différentes, ainsi qu'une ou plusieurs codelettes pour divers types d'accélérateurs (GPU ou Xeon Phi par exemple). Le fait de pouvoir lancer les codelettes sur plusieurs types de matériel justifie le nom de StarPU que l'on peut aussi orthographier *PU : tous les types de processeurs (C ou G PU) sont accessibles. Une tâche est alors déterminée par ses codelettes ainsi que par la description du type d'accès aux données (R, W ou RW).

Une fois les codelettes et les tâches décrites, le programmeur n'a plus qu'à programmer son algorithme de manière séquentielle en soumettant les tâches à StarPU dans un ordre qui produirait le résultat correct. StarPU se charge alors de distribuer les calculs sur les processeurs et les accélérateurs disponibles sur le noeud de calcul. En fonction des dépendances, les tâches peuvent être réorganisées et exécutées en parallèle si possible. Par ailleurs StarPU dispose de plusieurs stratégies d'ordonnancement (*scheduling*) qui permettent de tenir compte des efficacités relatives de codelettes, des vitesses de transfert mémoire entre les accélérateurs, etc. afin d'offrir la meilleure exploitation possible des ressources de calcul.

5.3.2 MPI

StarPU est efficace pour répartir les tâches sur un noeud de calcul d'un supercalculateur. Typiquement, un tel noeud est aujourd'hui constitué d'un ou plusieurs CPU multicoeur et, de plus en plus souvent, d'un ou plusieurs accélérateurs de type GPU. Pour les communications entre noeuds, StarPU utilise l'environnement MPI (*Message Passing Interface*). Pour l'instant, StarPU n'est pas capable de distribuer les tâches entre les noeuds MPI. En revanche, il est capable de générer automatiquement les tâches de communications nécessaires au déroulement correct du graphe des tâches. Pour cela, l'utilisateur doit établir une distribution initiale des données et concevoir un logiciel dans lequel chaque noeud MPI possède une copie du graphe des tâches. Chaque noeud MPI doit alors (formellement) lancer toutes les tâches du graphe. Même si tout le graphe est parcouru, la perte d'efficacité est faible, car seules les tâches pour lesquelles les données sont disponibles seront effectivement réalisées. Si une tâche a besoin d'accéder en lecture à des données d'un noeud MPI voisin, StarPU génère automatiquement les dépendances et les transferts nécessaires. Dans cette thèse nous n'utiliserons pas la version MPI de StarPU, même si dans SCHNAPS certaines applications aux plasmas commencent à exploiter cette possibilité.

5.3.3 OpenCL

SCHNAPS est une bibliothèque, écrite en C, qui permet de résoudre numériquement un système de lois de conservation générique par la méthode GD sur un calculateur disposant d'un ou plusieurs accélérateurs de type GPU ou CPU multicoeur. Afin d'écrire les codelettes spécifiques pour le GPU, nous avons choisi, comme dans le chapitre 4 page 63, l'environnement OpenCL. OpenCL permet d'écrire et d'exécuter en parallèle des programmes (appelés *kernels*) sur tous les processeurs d'un accélérateur.

Une particularité d'OpenCL est qu'une partie du code est compilée à l'exécution. Cette particularité s'explique par la nécessité d'être compatible avec des accélérateurs d'architectures matérielles différentes.

SCHNAPS intègre donc un mécanisme afin de partager du code source de fonction avec OpenCL. Lorsqu'une portion de code source de SCHNAPS est incluse entre deux balises

```
#pragma start_opencl
  ...
#pragma end_opencl,
```

cette portion de code sera également compilée une deuxième fois à l'exécution pour des appels OpenCL. Les fonctions SCHNAPS se comportent donc comme des fonctions C traditionnelles, mais elles peuvent être appelées indifféremment depuis le programme hôte ou depuis un kernel OpenCL.

Par ailleurs, pour calculer sur des accélérateurs de type GPU, il est nécessaire de dupliquer des données entre le CPU et le GPU. Ces copies mémoire sont coûteuses et doivent être traitées avec précaution pour éviter notamment les conflits d'accès. Auparavant nous gérions ces transferts nous-mêmes grâce au mécanisme des évènements (*events*) OpenCL. Grâce aux évènements, OpenCL permet également l'implémentation d'un graphe des tâches et de ses dépendances. Cependant il faut décrire *à la main* les dépendances et il n'y a pas de mécanisme automatique pour les inférer à partir de l'accès aux données. Depuis l'intégration de StarPU à SCHNAPS nous avons donc préféré déléguer cette gestion au support d'exécution.

5.3.4 Modèles physiques

Dans SCHNAPS, nous utilisons une notion de *modèle physique* générique. Un modèle dans SCHNAPS (`Model`) contient les informations suivantes :

- dimension m du vecteur $\mathbf{W}$ des variables conservatives du système (5.1 page 86),
- flux numérique,
- condition initiale,
- flux ou conditions aux limites,
- termes sources,
- et éventuellement : solution de référence (si elle existe), caractéristiques des matériaux, *etc.*

Toutes ces informations sont données dans des fonctions SCHNAPS (entourées des balises `pragma start_opencl ... pragma end_opencl`), car elles doivent pouvoir être appelées depuis les kernels OpenCL.

5.3.5 Sous-domaines

Afin d'atteindre de bonnes performances en calcul parallèle, SCHNAPS repose sur plusieurs niveaux de parallélisme. Le premier niveau correspond à celui des sous-domaines. C'est un parallélisme à gros grain. Les transferts entre sous-domaines sont réalisés grâce à la bibliothèque MPI par StarPU.

Un niveau intermédiaire de parallélisme est piloté par StarPU qui distribue les tâches de calcul sur les accélérateurs du noeud MPI. Enfin, un troisième niveau de parallélisme à grain fin est piloté au niveau des accélérateurs OpenCL.

Les maillages de SCHNAPS sont organisés pour suivre cette hiérarchie. D'abord, le domaine Ω est découpé en *sous-domaines*. Chaque sous-domaine est associé à un noeud MPI. Un noeud MPI ne possède que les données de son sous-domaine. Mais il dispose aussi d'informations minimales sur le reste du maillage afin de pouvoir construire le graphe des tâches StarPU du calcul global.

S'il n'y a qu'un seul sous-domaine, SCHNAPS peut fonctionner sans la bibliothèque MPI.

5.3.6 Macrocellules et champs

Par ailleurs, chaque sous-domaine est lui-même découpé en macrocellules. Les macrocellules sont obtenues grâce à un diffémorphisme (transformation géométrique) qui envoie le cube de référence $[0,1]^3$ sur un hexaèdre courbé (voir figure 5.1). Il est possible d'envisager des transformations trilinéaires : hexaèdre à 8 noeuds de type H8 dans la terminologie élément fini ; ou des transformations quadratiques : hexaèdre à 20 noeuds de type H20.

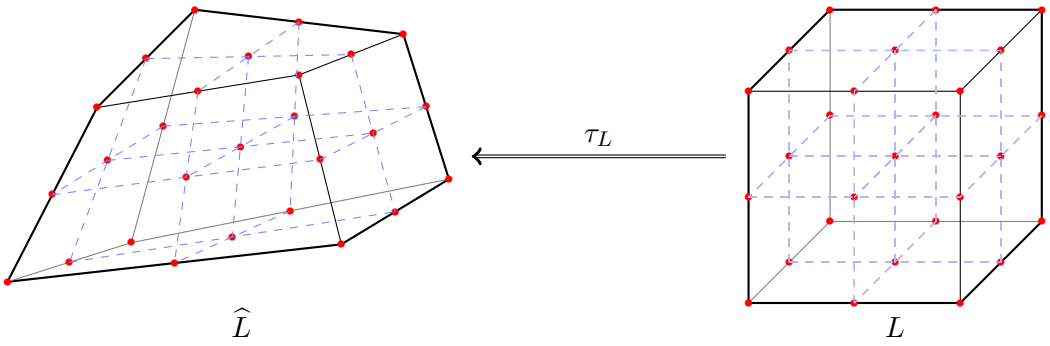

FIGURE 5.1 – Transformation géométrique

Les macro-cellules sont ensuite subdivisées dans chaque direction en sous-cellules partageant les mêmes caractéristiques (modélisation physique, interpolation). Notons que dans

chaque macrocellule, les sous-cellules sont organisées suivant une grille courbe, mais régulière et que les données géométriques de type H8 ou H20 sont partagées entre toutes les sous-cellules. Cette organisation a deux avantages. Elle permet d'abord de manipuler des structures de données géométriques légères, car seul un maillage grossier est nécessaire. C'est ce maillage grossier (macro-maillage), qui est partagé entre les noeuds MPI et qui permet de construire le graphe des tâches StarPU. D'autre part, l'organisation régulière des sous-cellules permet d'envisager des optimisations très efficaces notamment dans les calculs sur GPU. Les paramètres de découpage sont en général décrits dans deux tableaux `deg`[0 :2] et `raf`[0 :2]. `deg`[i] représente le degré d'interpolation dans la direction du maillage logique i et `raf`[i] le nombre de sous-cellules dans cette direction. Dans la direction i il y a donc (`deg`[i]+1)*`raf`[i] points d'interpolation. Il est possible d'avoir des paramètres de discrétisation différents suivant chaque direction, ce qui permet par exemple de réaliser des calculs 1D ou 2D, même si la structure du maillage est toujours 3D.

À chaque macro-cellule du macro-maillage est associé un champ. Cette structure contient les valeurs des variables conservatives sur les points d'interpolation à l'intérieur des sous-cellules.

Les échanges de données entre macrocellules et le calcul des conditions aux limites sont assurés par une structure d'interface. Cette structure est en particulier chargée d'extraire les données de bord des macrocellules dans des tampons. Ces tampons sont très utiles pour optimiser les dépendances de données et offrir plus de liberté à StarPU pour optimiser le parcours du graphe des tâches (voir figure 5.2 page suivante)

Nous réalisons les calculs sur les champs des macrocellules à partir de codelettes StarPU. Ces codelettes calculent les flux, les termes sources dans les macrocellules. Les codelettes calculent également les flux provenant des interfaces. Ils réalisent enfin les extractions de données volumiques vers les interfaces. Nous avons écrit à chaque fois deux versions des codelettes : une version C pour CPU et une version OpenCL qui peut s'exécuter sur un CPU multicoeur ou un GPU.

5.3.7 Simulation

Une *simulation* est une collection de champs sur des macrocellules. Cette classe est conçue pour contenir les algorithmes à appliquer sur les macrocellules. En particulier, c'est cette classe qui lance les tâches StarPU.

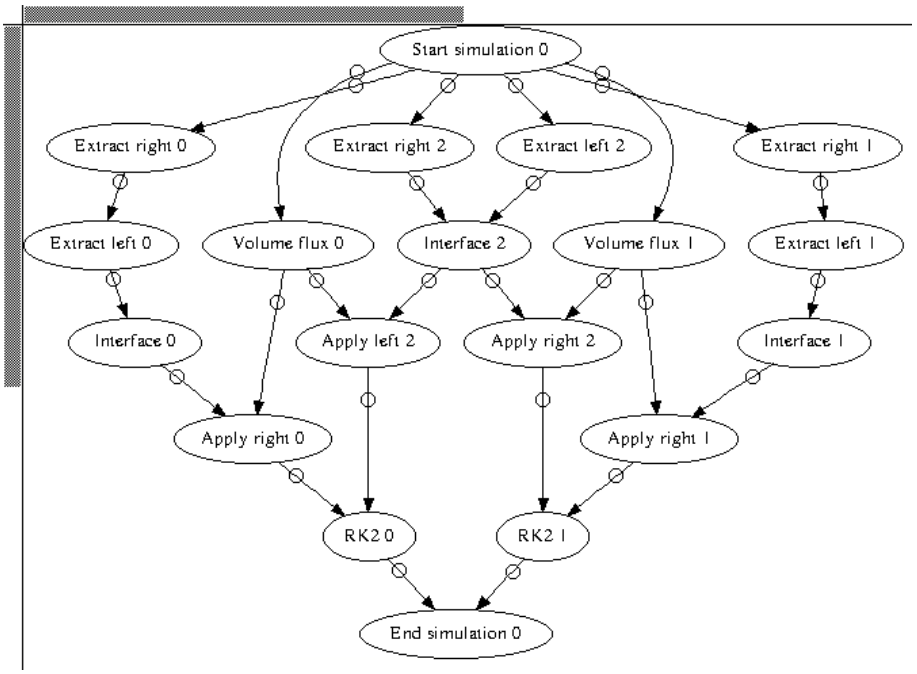

FIGURE 5.2 – Graphe des tâches pour un calcul dans un sous-domaine contenant 3 zones. Les tâches sont représentées par les noeuds du graphe et les flèches représentent les dépendances entre les tâches (une tâche doit attendre la fin des tâches dont elle dépend avant de démarrer).

5.3.8 Éléments géométriques

Dans cette section, nous nous plaçons au niveau d'un élément L. Suivant la zone dans laquelle il se trouve, cet élément peut avoir des caractéristiques géométriques différentes. Nous utilisons le formalisme classique des éléments finis. À chaque zone nous attribuons un élément de référence $\hat{L}$. Cet élément s'appuie sur des *noeuds géométriques*, notés $\hat{\mathbf{X}}_i$ et des *fonctions de base géométriques*, notées $\hat{\boldsymbol{\psi}}_i$ telles que

$$\hat{\boldsymbol{\psi}}_i(\hat{\mathbf{X}}_j) = \delta_{ij}.$$

Dans SCHNAPS, dans une zone donnée, le nombre de noeuds et de fonctions géométriques est noté `nb_nodes`. Pour un hexaèdre à huit noeuds (élément de type H8), `nb_nodes`=8. Les noeuds sont donnés dans le tableau `ref_node`. L'élément de référence possède également un certain nombre de faces, noté `nb_faces`. Le tableau `face2node` permet de retrouver les noeuds d'une face donnée. Le numéro du j-ième noeud de la face jf est donné par `face2node[nb_face_nodes*jf+j]`. L'entier `nb_face_nodes` est le nombre maximal de noeuds par face (par exemple `nb_face_nodes`=4 pour un hexaèdre à 8 noeuds).

Nous utiliserons ce formalisme pour toutes les familles d'éléments finis de SCHNAPS, qu'il s'agisse d'hexaèdres H8 ou H20. À l'avenir nous envisageons de traiter également des tétraèdres ou d'autres éléments courbes.

À titre d'exemple, nous montrons maintenant précisément comment nous pouvons définir l'interpolation pour des hexaèdres à 8 noeuds (éléments de type H8 dans la terminologie classique des éléments finis). En général, dans SCHNAPS nous utilisons des éléments quadratiques de type H20, mais la description du H8 est plus simple. Dans ce cas, l'élément de référence $\hat{L}$ est le cube unité

$$\hat{L} = [0,1]^3.$$

Soient $\hat{\mathbf{X}} = (\hat{x}, \hat{y}, \hat{z})$ les coordonnées dans l'élément de référence. Les noeuds de référence $\hat{\mathbf{X}}^i$, $i = 1 \cdots 8$ et les fonctions géométriques $\hat{\boldsymbol{\psi}}_i$ de l'élément de référence sont données par

i	$\hat{\mathbf{X}}^i$	$\hat{\boldsymbol{\psi}}_i$
1	$(0,0,0)$	$(1-\hat{x})(1-\hat{y})(1-\hat{z})$
2	$(1,0,0)$	$\hat{x}(1-\hat{y})(1-\hat{z})$
3	$(1,1,0)$	$\hat{x}\hat{y}(1-\hat{z})$
4	$(0,1,0)$	$(1-\hat{x})\hat{y}(1-\hat{z})$
5	$(0,0,1)$	$(1-\hat{x})(1-\hat{y})\hat{z}$
6	$(1,0,1)$	$\hat{x}(1-\hat{y})\hat{z}$
7	$(1,1,1)$	$\hat{x}\hat{y}\hat{z}$
8	$(0,1,1)$	$(1-\hat{x})\hat{y}\hat{z}$

Un hexaèdre H8 arbitraire est alors défini par huit noeuds X_L^i. La transformation géométrique qui envoie $\hat{L}$ sur L est définie par

$$\tau_L(\hat{\mathbf{X}}) = \hat{\boldsymbol{\psi}}_i(\hat{\mathbf{X}})\mathbf{X}_L^i. \tag{5.12}$$

Un hexaèdre H8 arbitraire est alors défini par huit noeuds X_L^i. La transformation géométrique qui envoie $\hat{L}$ sur L est définie par

Nous faisons l'hypothèse que les noeuds $\mathbf{X}_L^i$ sont choisis de telle sorte que τ_L est une transformation directe et inversible. En pratique, il faut s'assurer que le choix des noeuds ne conduit pas à des éléments mal orientés ou trop déformés. Comme les fonctions de base géométriques satisfont

$$\hat{\boldsymbol{\psi}}_i(\hat{\mathbf{X}}^j) = \delta_{ij}$$

nous déduisons que la transformation géométrique envoie les noeuds de référence sur les noeuds de l'élément L

$$\tau_L(\hat{\mathbf{X}}^i) = \mathbf{X}_L^i.$$

Pour la numérotation des faces, nous utilisons la convention suivante. Tout d'abord, pour le cube de référence, les faces sont numérotées de 0 à 5 selon le modèle de la figure 5.3 page ci-contre :

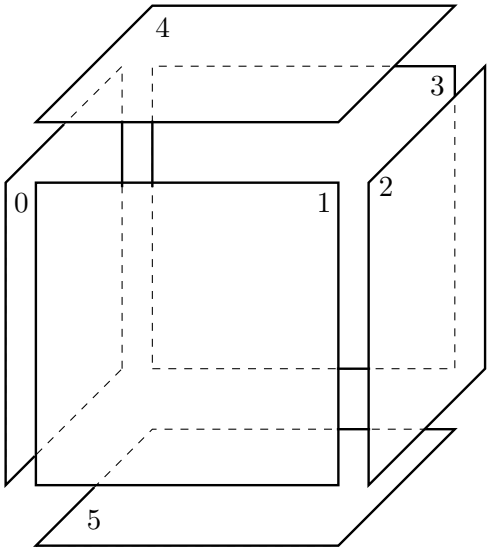

FIGURE 5.3 – Numérotation des faces

Nous définissons ensuite un tableau de permutation des axes des faces

$$\text{axis\_permut} = \begin{pmatrix} 0 & 2 & 1 & 0 \\ 1 & 2 & 0 & 1 \\ 2 & 0 & 1 & 1 \\ 2 & 1 & 0 & 0 \\ 0 & 1 & 2 & 1 \\ 1 & 0 & 2 & 0 \end{pmatrix}.$$

Chaque ligne correspond à une face du cube. Les trois premières colonnes code une permutation des axes selon la convention "0" pour x, "1" pour y et "2" pour z, les deux premiers axes étant dans le plan de la face et le troisième orienté suivant la normale sortante. La dernière colonne donne la valeur de la troisième coordonnée qui est constante sur la face. Par exemple, la troisième ligne correspond à la face 2 qui est dans le plan ("2","0") $= (z, x)$. La direction normale à la face est la direction "1" $= y$. Enfin sur cette face, on a bien $y = 1$.

Pour un numéro de face `ifa`, nous notons opposite_face(`ifa`) le numéro de face opposée :

$$\text{opposite\_face} = \begin{pmatrix} 2 \\ 3 \\ 0 \\ 1 \\ 5 \\ 4 \end{pmatrix}.$$

La géométrie des éléments est décrite dans la struture geometry. Cette classe contient des fonctions SCHNAPS (qui pourront donc être appelées depuis les kernel OpenCL) pour calculer la transformation géométrique τ en un point de référence, son gradient, son jacobien, son inverse, le vecteur normal dans le cas d'un élément surfacique, *etc.*

5.3.9 Interpolation

Une fois définie la géométrie d'un élément, on peut définir l'interpolation d'un champ scalaire. Cette information est décrite dans la structure `interpolation`.

Pour l'instant dans SCHNAPS, pour des raisons d'efficacité, nous utilisons une approximation nodale adaptée à la quadrature numérique (voir [11, 41]) : les champs sont définis aux points d'intégration de Gauss-Lobatto des éléments. Ce choix permet d'accéder directement aux champs pour l'intégration numérique dans le volume et sur les faces. Ce choix assure aussi que les matrices masses locales sont diagonales.

Pour commencer, nous décrivons par exemple comment nous interpolons les champs sur un élément de type H8 avec des polynômes de degré d. Pour simplifier, nous considérons d'abord le cas où la macrocellule correspondante ne contient qu'une seule sous-cellules. C'est à dire nous supposons que `raf`[0]=`raf`[1]=`raf`[2]=1 dans les trois directions.

Pour l'interpolation, nous fixons d'abord dans la cellule L un degré d. Nous considérons les $(d+1)$ points de Gauss-Lobatto $(\xi_i)_{i=0\cdots d}$ sur $[0,1]$, et les poids d'intégration correspondants ω_i.

Nous notons aussi I_k le $k^{\text{ième}}$ polynôme de Lagrange associé aux points $\boldsymbol{\xi}_i$. Rappelons que I_k est un polynôme de degré d et que

$$I_j(\boldsymbol{\xi}_i) = \delta_{ij}.$$

Nous construisons alors, sur l'élément de référence $\hat{L}$, les points de Gauss $\hat{\mathbf{Y}}_q$, les poids $\hat{\lambda}_q$ et les fonctions d'interpolation $\hat{\boldsymbol{\varphi}}^q(\hat{\mathbf{X}})$ à partir de produits tensoriels de quantités monodimensionnelles. Plus précisément, soient i, j et k trois entiers dans $\{0\cdots d\}$ et soit $q = (d+1)^2k + (d+1)j + i$ alors

$$\hat{\mathbf{Y}}_q = (\boldsymbol{\xi}_i, \boldsymbol{\xi}_j, \boldsymbol{\xi}_k), \quad \hat{\lambda}_q = \omega_i\omega_j\omega_k, \quad \hat{\boldsymbol{\varphi}}^q(\hat{\mathbf{X}}) = I_i(\hat{x})I_j(\hat{y})I_k(\hat{z}).$$

Finalement, nous obtenons les fonctions de base sur l'élément L en transportant les fonctions d'interpolation de référence avec la transformation géométrique.

$$\boldsymbol{\varphi}_L^i(\mathbf{X}) = \hat{\boldsymbol{\varphi}}^i(\hat{\mathbf{X}}) \text{ avec } \mathbf{X} = \tau_L(\hat{\mathbf{X}}).$$

La classe `interpolation` contient des fonctions permettant de calculer : la position des points de Gauss du volume ou des faces, les poids de Gauss, les valeurs des fonctions de base et de leurs gradients.

5.3.10 Interpolation de Gauss-Lobatto avec des sous-cellules

En fait, chaque macrocellule peut-être découpée en plusieurs sous-cellules. Pour l'interpolation nous fixons d'abord dans la macrocellule L un degré $d(\ell)$ pour chaque direction $\ell = 0, 1, 2$. Dans chaque direction, nous considérons les $(d(\ell)+1)$ points de Gauss-Lobatto $(\boldsymbol{\xi}_i)_{i=0\cdots d(\ell)}$ sur $[0, 1]$, et les poids d'intégration correspondants ω_i. Nous considérons de plus un raffinement $\mathrm{nraf}(\ell)$ pour $\ell = 0, 1, 2$. Nous notons aussi I_k le $k^{\text{ième}}$ polynôme de Lagrange associé aux points $\boldsymbol{\xi}_i$. Rappelons que I_k est un polynôme de degré d et que

$$I_j(\boldsymbol{\xi}_i) = \delta_{ij}.$$

Sur l'élément de référence $\hat{L}$, nous construisons alors les points de Gauss-Lobatto $\hat{\mathbf{Y}}_q$, les poids $\hat{\lambda}_q$ et les fonctions d'interpolation $\hat{\boldsymbol{\varphi}}^q(\hat{\mathbf{X}})$ à partir de produits tensoriels de quantités monodimensionnelles. Plus précisément, soient $i(\ell)_{\ell=0,1,2}$ trois entiers dans $\{0\cdots d(0)\}\times\{0\cdots d(1)\}\times\{0\cdots d(2)\}$ et $r(\ell)_{\ell=0,1,2}$ trois entiers dans $\{0\cdots \mathrm{nraf}(0)\}\times\{0\cdots \mathrm{nraf}(1)\}\times\{0\cdots \mathrm{nraf}(2)\}$ et soit

$$\begin{aligned} q &= i(0) \\ &+ i(1)(d(0)+1) \\ &+ i(2)(d(0)+1)(d(1)+1) \\ &+ r(0)(d(0)+1)(d(1)+1)(d(2)+1) \\ &+ r(1)(d(0)+1)(d(1)+1)(d(2)+1)\,\mathrm{nraf}(0) \\ &+ r(2)(d(0)+1)(d(1)+1)(d(2)+1)\,\mathrm{nraf}(0)\,\mathrm{nraf}(1), \end{aligned}$$

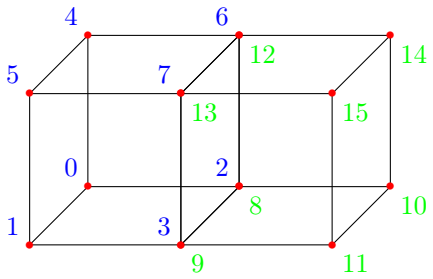

FIGURE 5.4 – Numérotation dans les sous-cellules

Nous voyons que la numérotation se fait sous-cellule par sous-cellule. En mémoire, les données de chaque sous-cellule seront groupées. Ce rangement assure un accès aux données efficace pour les calculs volumiques, qui représentent le plus d'opérations. En revanche, pour les calculs des interactions entre sous-cellules (flux aux interfaces), on observera forcément des défauts d'accès au cache.

La définition des points de référence, poids de référence et fonctions de base de référence s'écrit alors :

$$\begin{aligned}
\hat{\mathbf{Y}}_q &= (h(0)(r(0)+\boldsymbol{\xi}_{i(0)}), h(1)(r(1)+\boldsymbol{\xi}_{i(1)}), h(2)(r(2)+\boldsymbol{\xi}_{i(2)})), \\
\hat{\lambda}_q &= h(0)\omega_{i(0)} h(1)\omega_{i(1)} h(2)\omega_{i(2)}, \\
\hat{\boldsymbol{\varphi}}^q(\hat{\mathbf{X}}) &= I_{i(0)}(\frac{\hat{x}}{h(0)} - r(0)) I_{i(1)}(\frac{\hat{y}}{h(1)} - r(1)) I_{i(2)}(\frac{\hat{z}}{h(2)} - r(2))\, \chi^r(\hat{\mathbf{X}}),
\end{aligned}$$

où $h(\ell) = 1/\mathrm{nraf}(\ell)$ et

$$\chi^r(\hat{\mathbf{X}}) = \mathbb{1}_{[h(0)r(0),h(0)(r(0)+1)]}(\hat{x}) \mathbb{1}_{[h(1)r(1),h(1)(r(1)+1)]}(\hat{y}) \mathbb{1}_{[h(2)r(2),h(2)(r(2)+1)]}(\hat{z}).$$

En pratique, on calcule q grâce à la formule de Hörner

$$q = i(0)+(d(0)+1)(i(1)+(d(1)+1)(i(2)+(d(2)+1)(r(0)+\mathrm{nraf}(0)(r(1)+\mathrm{nraf}(1)r(2))))).$$

Dans la suite, nous utiliserons la notation q aussi pour le multi-indice

$$q = (i(0), i(1), i(2), r(0), r(1), r(2)) = (i, r).$$

Finalement, nous obtenons les fonctions de base sur l'élément L en transportant les fonctions d'interpolation de référence avec la transformation géométrique.

$$\boldsymbol{\varphi}_L^i(\mathbf{X}) = \hat{\boldsymbol{\varphi}}^i(\hat{\mathbf{X}}) \text{ avec } \mathbf{X} = \tau_L(\hat{\mathbf{X}}).$$

Nous définissons à présent l'indexation des points de Gauss-Lobatto sur les faces. Ces points sont repérés par un numéro de face $\texttt{ifa} \in \{0 \cdots 5\}$, deux indices de points $(i'(0), i'(1))$ et deux indices de sous-face $(r'(0), r'(1))$. Nous utilisons le tableau axis_permut introduit à la section 5.3.8 page 97 pour retrouver la position du point de Gauss-Lobatto dans le volume à l'aide des formules suivantes

$$\left\{\begin{array}{lcl} i(\text{axis\_permut}(\texttt{ifa}, 0)) & = & i'(0), \\ i(\text{axis\_permut}(\texttt{ifa}, 1)) & = & i'(1), \\ i(\text{axis\_permut}(\texttt{ifa}, 2)) & = & \text{axis\_permut}(\texttt{ifa}, 3)\, d(\text{axis\_permut}(\texttt{ifa}, 3)). \end{array}\right.$$

Les indices de la sous-cellule sont donnés par

$$\left\{\begin{array}{lcl} r(\text{axis\_permut}(\texttt{ifa}, 0)) & = & r'(0), \\ r(\text{axis\_permut}(\texttt{ifa}, 1)) & = & r'(1), \\ r(\text{axis\_permut}(\texttt{ifa}, 2)) & = & \text{axis\_permut}(\texttt{ifa}, 3)\, (\text{nraf}(\text{axis\_permut}(\texttt{ifa}, 3)) - 1). \end{array}\right.$$

Les degrés d'approximation permutés sont donnés par $d'(\text{axis\_permut}(\texttt{ifa}, \ell)) = d(\ell)$ et les raffinements permutés sont donnés par $\text{nraf}'(\text{axis\_permut}(\texttt{ifa}, \ell)) = \text{nraf}(\ell)$.

Dans la suite, il est nécessaire de pouvoir passer de l'indexation des points de Gauss-Lobatto sur les faces à leur indexation dans les volumes. Avec la convention choisie, le numéro $q' = (i', r')$ du point de Gauss-Lobatto sur la face est donné par

$$\begin{array}{rcl} q' & = & i'(0) \\ & + & i'(1)(d'(0)+1) \\ & + & r'(0)(d'(0)+1)(d'(1)+1) \\ & + & r'(1)(d'(0)+1)(d'(1)+1)\, \text{nraf}'(0). \end{array}$$

Nous notons Π la transformation qui fait passer de la numérotation q' sur le bord ∂L à la numérotation q à l'intérieur de la cellule

$$q = \Pi(\texttt{ifa}, q').$$

Enfin, nous avons aussi besoin d'une numérotation des points de Gauss des faces des sous-cellules. Soit un numéro de sous-cellule r, un numéro de face `ifa` et un numéro de point de Gauss de sous-cellule i'. L'indice correspondant à q dans la cellule L est donné par la transformation

$$q = \pi(r, \texttt{ifa}, i').$$

Dans le détail, on a $q = (i(0), i(1), i(2), r(0), r(1), r(2))$ avec

$$\left\{ \begin{array}{lcl} i(\text{axis\_permut}(\texttt{ifa}, 0)) & = & i'(0), \\ i(\text{axis\_permut}(\texttt{ifa}, 1)) & = & i'(1), \\ i(\text{axis\_permut}(\texttt{ifa}, 2)) & = & \text{axis\_permut}(\texttt{ifa}, 3)\, d(\text{axis\_permut}(\texttt{ifa}, 3)). \end{array} \right.$$

5.3.11 Schéma GD pour l'interpolation de Gauss-Lobatto sur des sous-cellules

Dans le cas de l'interpolation de Gauss-Lobatto sur des sous-cellules, nous pouvons exprimer l'algorithme de calcul GD sous une forme plus efficace, direction par direction. Nous commençons par remplacer les intégrales de volumes et de bord par leurs quadratures de Gauss-Lobatto. Pour une cellule L et une composante $q = (i, r)$ le schéma DG avec quadrature Gauss-Lobatto s'écrit

$$\omega_q^L \frac{d}{dt} \mathbf{W}_L^q - \mathcal{V} + \mathcal{D} + \mathcal{B} = \omega_q^L S(\mathbf{Y}_L^q),$$

où $\mathcal{V}$, $\mathcal{D}$ et $\mathcal{B}$ désignent respectivement les termes volumiques, les termes aux interfaces des sous-cellules et les termes de bord de la cellule L. ω_q^L désigne le poids de Gauss associé au point de Gauss-Lobatto q sur l'élément L. Le calcul de ce poids nécessite un appel au calcul de la jacobienne τ (voir 5.12). Enfin, dans cette formule nous n'utilisons pas la convention de sommation sur les indices répétés (cela provient du fait que la matrice de masse est diagonale, grâce aux propriétés des points de Gauss-Lobatto). Pour l'intégrale de volume, il faut considérer les contributions sur tous les points de Gauss-Lobatto $\mathbf{Y}_L^p$ avec $p = (j(0), j(1), j(2), s(0), s(1), s(2)) = (j, s)$, soit

$$\mathcal{V} = \sum_p \omega_p^L \mathbf{F}(\mathbf{W}_L^p) \cdot \nabla \boldsymbol{\varphi}_q^L(\mathbf{Y}_L^p).$$

D'autre part, si q correspond à un point de Gauss interne à la cellule L on a

$$\mathcal{B} = 0.$$

Si q correspond à un point du bord ∂L, il existe au moins un numéro de face ifa et un indice de bord q' tel que

$$q = \Pi(\text{ifa}, q')$$

et

$$\mathcal{B} = \sum_{q=\Pi(\text{ifa},q')} \omega_{q'}^{\partial L} \mathbf{F}(\mathbf{W}_L^q, \mathbf{W}_R(\mathbf{Y}_L^q), n_{LR}(\mathbf{Y}_L^q)).$$

Dans cette formule, $\omega_{q'}^{\partial L}$ désigne le poids d'intégration surfacique correspondant au point de Gauss-Lobatto. Il nécessite un calcul de la jacobienne de la transformation géométrique τ (voir 5.12).

Il reste à expliciter les termes de saut aux interfaces des sous-cellules.

Si q corresponds à un point de Gauss interne à une sous-cellule alors

$$\mathcal{D} = 0.$$

Si q correspond à un point de bord de la sous-cellule r alors il existe une ou plusieurs sous-faces `ifa` et un numéro i' de point de Gauss dans la sous-face tels que

$$q = \pi(r, \texttt{ifa}, i').$$

Le point q admet un unique vis-à-vis $\tilde{q}$ sur la face `ifa` dans une sous-cellule voisine $\tilde{r}$

$$\tilde{q} = \pi(\tilde{r}, \text{opposite\_face}(\texttt{ifa}), \tilde{i'}),$$

et les points de Gauss-Lobatto correspondants sont confondus

$$\mathbf{Y}_L^{\tilde{q}} = \mathbf{Y}_L^q.$$

Avec ces notations, les flux d'interfaces internes s'écrivent

$$\mathcal{D} = \sum_{q=\pi(r,\text{ifa},i')} \omega_{i'}^{\partial r} \mathbf{F}(\mathbf{W}_L^q, \mathbf{W}_L^{\tilde{q}}, n_{r\tilde{r}}(\mathbf{Y}_L^q)),$$

où $\omega_{i'}^{\partial r}$ désigne les poids de Gauss des points i' sur le bord de la sous-cellule r. La normale à l'interface entre les sous-cellules r et $\tilde{r}$, orientée de r vers $\tilde{r}$ est notée $n_{r\tilde{r}}$.

5.4 Programmation de la MHD dans SCHNAPS

5.4.1 Commentaires sur la programmation

La programmation d'une nouvelle application dans SCHNAPS se fait en écrivant les quatre fonctions suivantes

L'état initial du système

La programmation de l'état initial de l'application se fait en spécifiant les valeurs initiales des variables conservatives du système. En pratique, nous donnons les états initiaux des variables primitives et nous déterminons les variables primitives à l'aide des équations

$$\begin{aligned} W_1 &= Y_1 \\ W_2 &= Y_1 * Y_2 \\ W_3 &= Y_1 * Y_3 \\ W_4 &= Y_1 * Y_4 \end{aligned}$$

$$W_5 = \frac{Y_5}{\gamma - 1} + Y_1 \left(\frac{Y_2^2 + Y_3^2 + Y_4^2}{2} \right) + \left(\frac{Y_6^2 + Y_7^2 + Y_8^2}{2} \right)$$

$$\begin{aligned} W_6 &= Y_6 \\ W_7 &= Y_7 \\ W_8 &= Y_8 \end{aligned}$$

Le flux numérique dans le volume

La programmation des flux numériques se fait en implémentant les algorithmes qui ont été donnés dans la section 2.2 page 32. Il est possible, dans SCHNAPS d'écrire de nombreux flux numérique différents dans le même fichier source afin de pouvoir choisir le flux le plus adapté.

Le(s) flux numérique(s) aux bords

Enfin, l'implémentation des flux numériques aux bords se fait en déterminant le flux numérique à utiliser en fonction de la direction dans laquelle nous nous trouvons ainsi qu'en définissant qu'elles sont les valeurs des variables conservatives au-delà de la frontière.

5.4.2 Premières validations

Dimension 1

Dans cette partie, nous reprenons les tests effectués au cours de la section 2.4 page 40 afin de valider le solveur SCHNAPS. Comme le flux $P2$ est celui apportant le meilleur rapport précision/temps, nous utilisons exclusivement ce flux pour nos tests. La discrétisation en temps est quant à elle effectuée à l'aide d'une méthode Runge-Kutta d'ordre 2 (RK2). Sur les figures 5.5 page suivante, 5.6 page suivante et 5.7 page 109 nous comparons les solutions obtenues à l'aide de la méthode GD pour des degrés 1, 2 et 3. Pour les trois cas, nous utilisons un maillage constitué d'une seule macro-cellule subdivisé en 200 sous-cellules.

Sur les trois solutions, nous pouvons observer l'apparition d'un phénomène de Gibbs sur les discontinuités [4] causées par l'absence de limiteur de pentes.

Nous observons aussi l'apparition caractéristique d'un choc stationnaire non entropique. Ce phénomène est bien sûr lié à l'absence d'une correction entropique dans la détente ayant un point sonique. Pour supprimer ce phénomène, il faudrait augmenter la diffusion du schéma dans cette détente. Par ailleurs si le choc stationnaire n'était pas apparu cela aurait été l'indice d'une erreur de programmation, car le schéma GD est censé être très peu diffusif à l'ordre élevé.

Afin de pouvoir calculer les ordres de convergence du modèle, nous avons décalé la détente stationnaire afin de faire disparaitre ces chocs non entropiques. Sur la figure 5.8 page 109 nous avons représenté le taux de convergence du schéma GD pour plusieurs ordres d'approximation et pour ce cas test 1D. Nous constatons que le taux de conver-

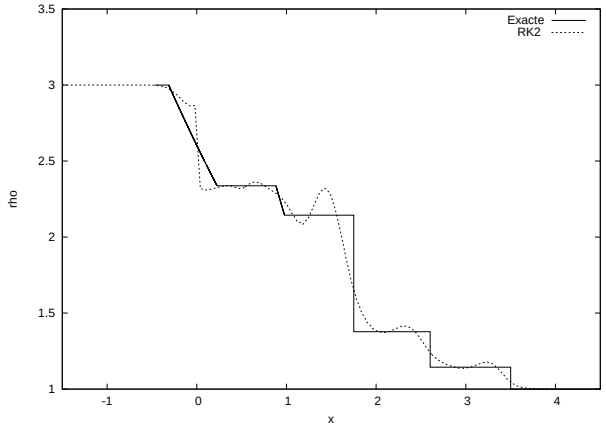

FIGURE 5.5 – Solutions obtenues par SCHNAPS pour le cas test du choc fort au degré 1.

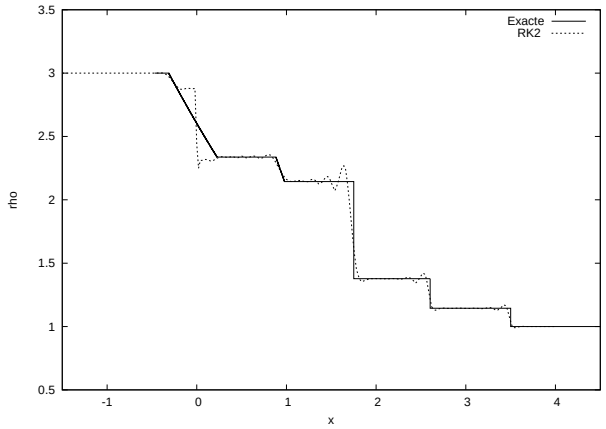

FIGURE 5.6 – Solutions obtenues par SCHNAPS pour le cas test du choc fort au degré 2.

gence augmente avec l'ordre d'approximation, sans néanmoins dépasser 1, car la solution exacte contient des discontinuités.

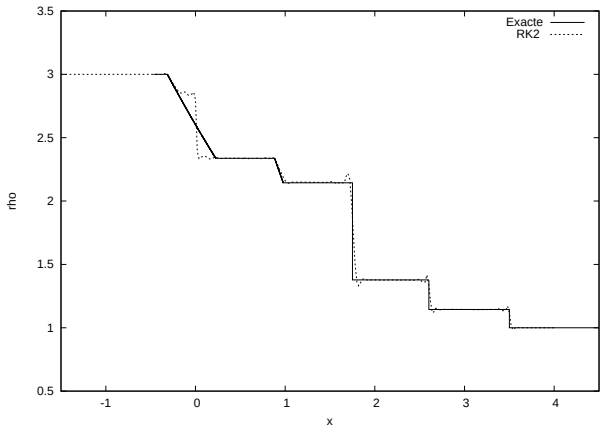

FIGURE 5.7 – Solutions obtenues par SCHNAPS pour le cas test du choc fort au degré 3.

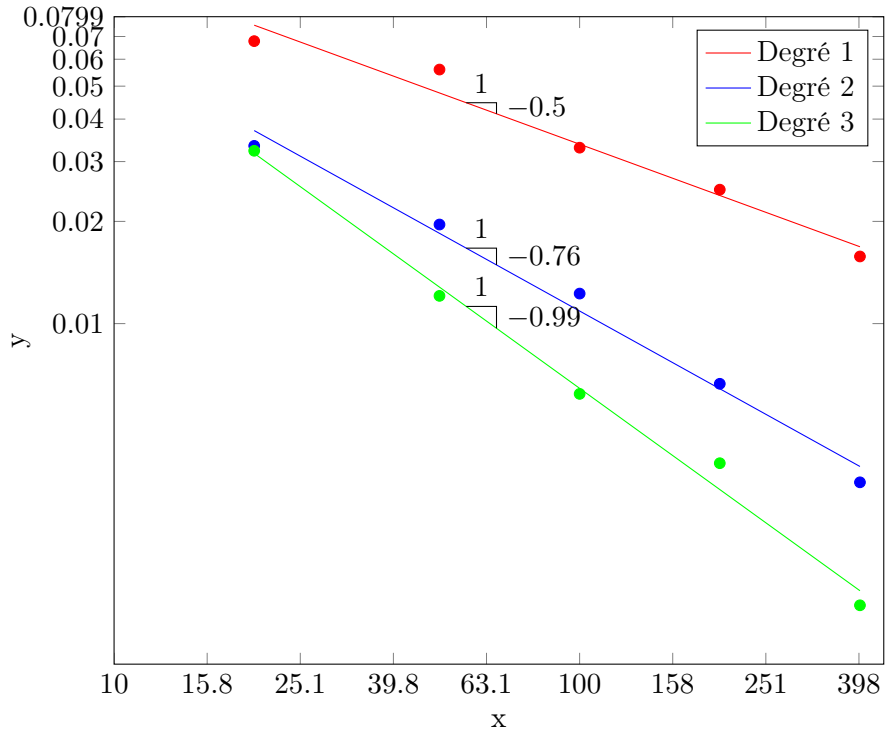

FIGURE 5.8 – Comparaison des ordres de convergence pour les méthodes d'ordre 1 et d'ordre 2.

Dimension 2

Afin de tester SCHNAPS pour la résolution d'un problème en deux dimensions, nous utilisons dans un premier temps le cas test d'Orszag-Tang dont les paramètres initiaux sont donnés dans le tableau 4.12 page 80. Nous traçons la solution aux instants 1 et 2.

Sur la figure 5.9 nous affichons la solution obtenue sur une macromaille subdivisée en 75 sous-maille dans les directions x et y. Pour ce test nous avons utilisé une méthode de Galerkin Discontinu de degré 3. Ce calcul correspond donc, en termes de mémoire à un maillage de 300 cellules volumes finis. Nous constatons sur la figure 5.9, à l'instant 1, que les résultats sont similaires à ceux obtenus sur un maillage volumes finis beaucoup plus fin. Sur la figure 5.10 page suivante nous constatons également une similarité entre la solution volumes finis et la solution GD, mais comme dans le cas en unidimensionnel, nous pouvons observer l'amplification du phénomène de Gibs dans la solution GD. Ces oscillations sont également visibles au niveau des chocs en observant de plus près la solution au temps 1 (figure 5.11 page ci-contre).

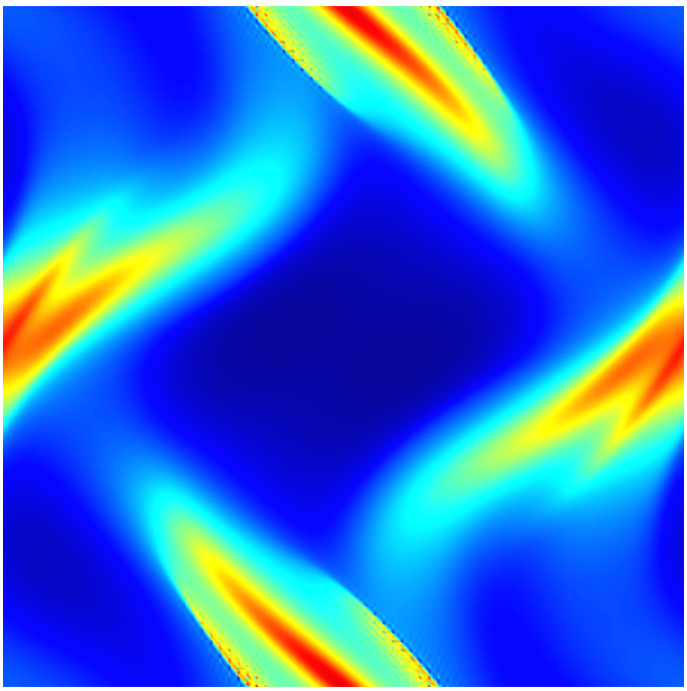

FIGURE 5.9 – Solution au temps $t = 1$ pour le cas test de Orzsag-Tang. La simulation a été effectuée avec SCHNAPS au degré 3 sur une grille 75×75 avec le flux numérique $P2$

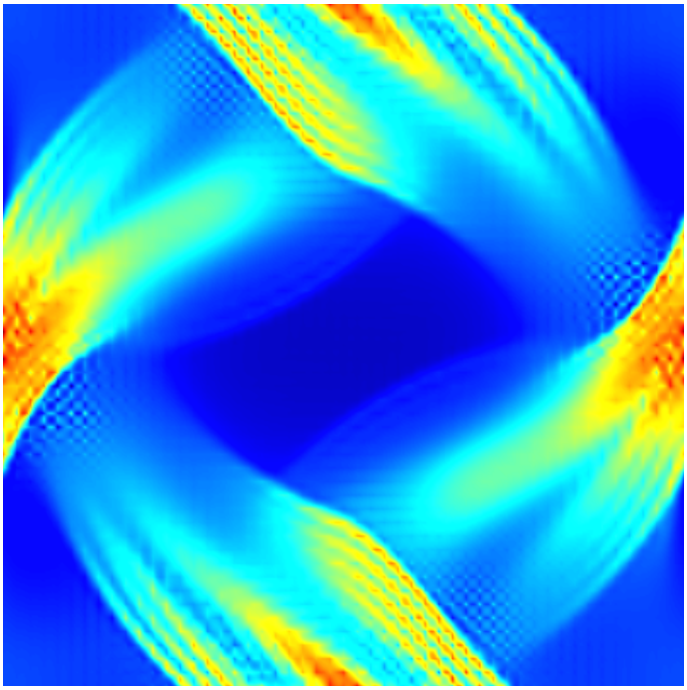

FIGURE 5.10 – Solution au temps $t = 1.5$ pour le cas test de Orzsag-Tang. La simulation a été effectuée au degré 3 sur une grille 75×75 avec le flux numérique $P2$

Nous n'avons pas poursuivi le calcul GD au-delà du temps 1.5. Il faudrait pour cela mettre en place une stratégie de limitation des oscillations. Ce n'est pas l'objectif prioritaire dans le développement de SCHNAPS, dont le but est avant tout de réaliser des calculs MHD faible Mach, donc sans choc.

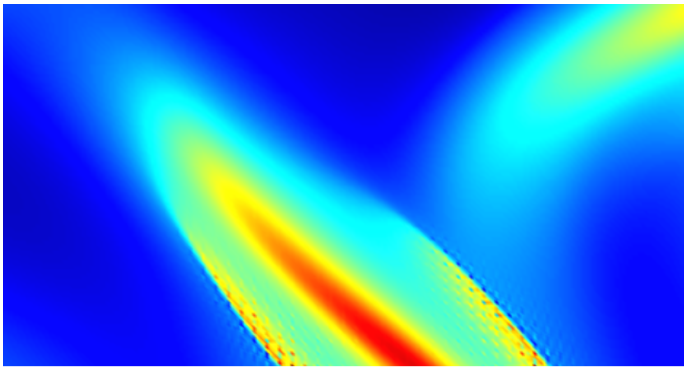

FIGURE 5.11 – Zoom sur une partie de la solution au temps $t = 1$ pour le cas test de Orzsag-Tang. La simulation a été effectuée au degré 2 sur une grille 75×75 avec le flux numérique $P2$

5.4.3 Tests de performances

Implémentations C et OpenCL

SCHNAPS est une bibliothèque permettant d'exécuter une application en utilisant au choix une optimisation exclusivement à l'aide d'une implémentation OpenCL permettant ainsi d'exécuter les calculs sur les accélérateurs GPU ou une optimisation hybride utilisant StarPU. Dans un premier temps, nous mesurons les performances de SCHNAPS dans le cas où nous utilisons uniquement la librairie OpenCL. Pour ces tests, nous utilisons le cas test 2D d'Orszag-Tang présenté dans le chapitre 4 page 63. Le maillage est une grille cartésienne composée d'une seule macrocellule subdivisée en 50 sous-cellules dans chaque direction. Nous effectuons ensuite les mesures du temps de calcul en utilisant le flux P2 (voir le chapitre 2 page 25) pour différents degrés d'interpolation.

Il est possible d'utiliser SCHNAPS de manière séquentiel sur une machine ne disposant pas des librairies nécessaires à l'utilisation de la programmation OpenCL des accélérateurs (CPU ou GPU). Dans ce cas, seules les codelettes C sont activées.

Cependant, nous montrons dans la figure 5.12 que l'utilisation de la version OpenCL de SCHNAPS sur un seul coeur de calcul CPU est plus efficace que la version C séquentielle, ce qui prouve que la programmation des accès mémoire dans la codelette OpenCL a été faite de façon correcte. Nous pouvons également voir sur la même figure que l'écart entre les temps de calcul des deux méthodes se creuse avec l'augmentation du degré de la méthode GD.

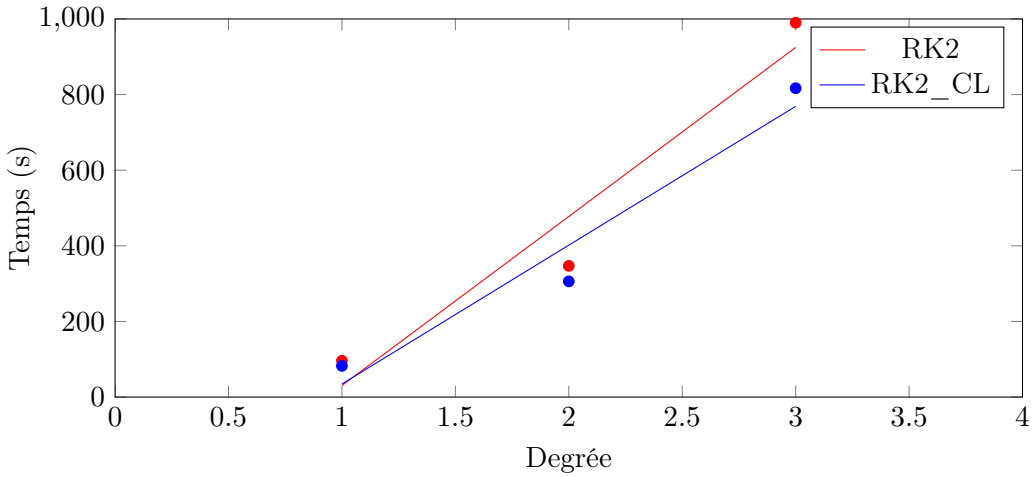

FIGURE 5.12 – Comparaison des temps d'exécution entre la version C et la version OpenCL de SCHNAPS sur un seul coeur de calcul

Comme nous pouvons le voir sur la figure 5.13 page ci-contre, l'utilisation d'un accélé-

rateur graphique n'est pas nécessaire. Comme nous l'avons montré dans le chapitre 4 page 63, l'utilisation de la librairie OpenCL, sur plusieurs coeurs CPU est également possible et permet un gain de temps important. Une nouvelle fois, nous avons comparé les temps pour les degrés 1, 2 et 3. Pour ce test, nous avons utilisé un maximum de 48 threads de calcul d'un ordinateur parallèle à mémoire partagée. La machine est composée de 2 CPU Intel Xeon E5 2630 v3 possédant chacun 12 coeurs (24 threads).

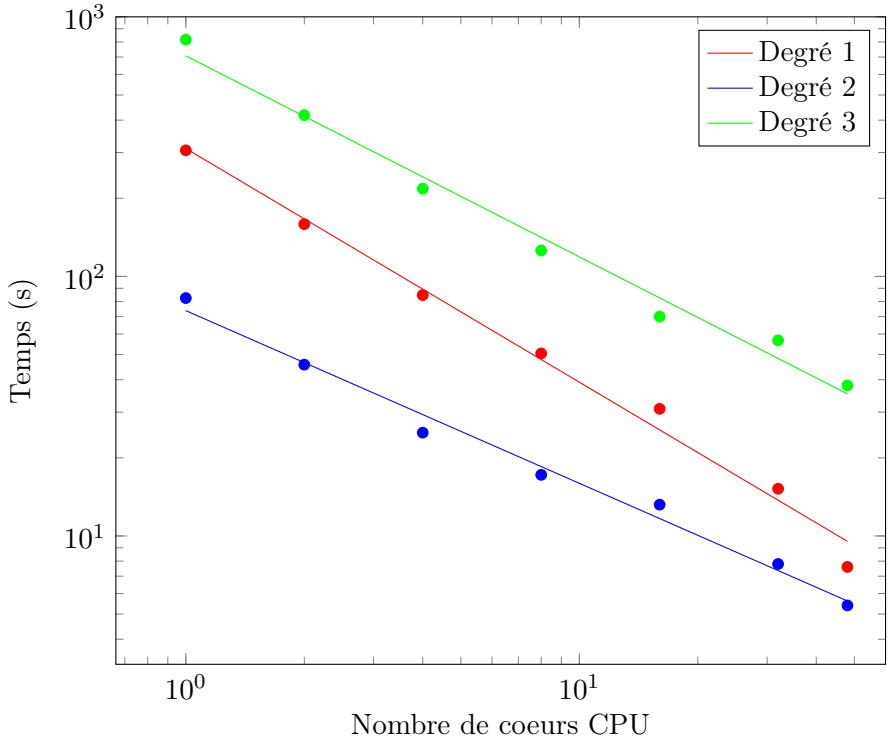

FIGURE 5.13 – Comparaison des temps d'exécution de la méthode RK2_CL pour les degrés 1, 2 et 3 de la méthode GD.

Pour finir l'évaluation des performances de l'implémentation OpenCL de SCHNAPS, nous regroupons les précédentes mesures dans le tableau 5.1 page suivante dans lequel nous ajoutons également les Temps obtenus avec l'utilisation d'une carte graphique NVidia Tesla K80. Nous en concluons que l'implémentation OpenCL de SCHNAPS est très efficace en termes de temps de calcul. De plus, nous constatons une nouvelle fois que l'accélération obtenue augmente avec la taille du problème.

Implémentations StarPU

Nous souhaitons à présent tester l'implémentation StarPU de SCHNAPS. Cette implémentation permet théoriquement, en utilisant au choix des codelettes C, OpenCL ou les

Degré	Architecture	Temps (sec)	Acceleration
1	RK2 (1 coeur)	96	1
	RK2_CL (1 coeur)	82.5	1.1
	2 Intel Xeon E5-2680 (48 coeurs)	5.4	17.8
	NVidia Tesla K80	4.6	21
2	RK2 (1 coeur)	347	1
	RK2_CL (1 coeur)	306	1.1
	2 Intel Xeon E5-2680 (48 coeurs)	15.2	22.8
	NVidia Tesla K80	7.6	45.6
3	RK2 (1 coeur)	990	1
	RK2_CL (1 coeur)	816	1.2
	2 Intel Xeon E5-2680 (48 coeurs)	38	26
	NVidia Tesla K80	11.1	90

TABLE 5.1 – Comparaison des temps d'exécution en fonction des algorithmes et des architectures

deux de lancer les taches de manière optimale. Dans cette partie nous faisons une étude des temps de calcul en utilisant encore une fois un maillage de taille 50×50 mais nous utilisons des découpages différents pour les macromaillages. Les macromaillages utilisés ainsi que les subdivisions correspondantes sont donnés dans le tableau 5.2.

Macromaillage	subdivision
1×1	50×50
2×2	25×25
3×3	17×17
4×4	12×12

TABLE 5.2 – Configurations des différents maillages utilisés

Pour les simulations, nous utilisons un noeud de calcul composé d'un CPU Intel Xeon E5-2680 (12 coeurs) ainsi que de quatre GPU NVidia Tesla K80. Sur la figure 5.14 page ci-contre nous constatons, en exécutant la codelette C sur un seul CPU que le nombre de macromailles joue un rôle important dans l'optimisation de la version StarPU de SCHNAPS.

Sur la figure 5.15 page suivante, nous étudions l'évolution des temps de calcul en fonction du nombre de coeurs de CPU utilisés. Nous remarquons qu'à l'exception du macromaillage 4×4, dans tous les cas le minimum est atteint lorsque le nombre de macromailles est égal au nombre de coeurs. Dans le cas où nous avons 16 macromailles, le minimum est atteint pour 12 coeurs. StarPU n'utilise pas l'hyperthreading, car les développeurs ont supposé que les kernels sont optimisés pour l'utilisation d'un coeur physique et que son utilisation pourrait également fournir des résultats non optimaux dans le partage des

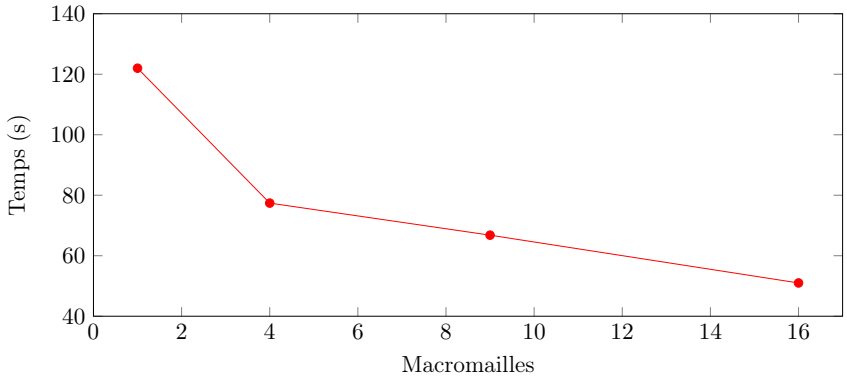

FIGURE 5.14 – Comparaison des temps d'exécution entre la version C et la version OpenCL de SCHNAPS sur un seul coeur de calcul

tâches. Nous disposons donc de 12 coeurs sur le CPU d'où le résultat.

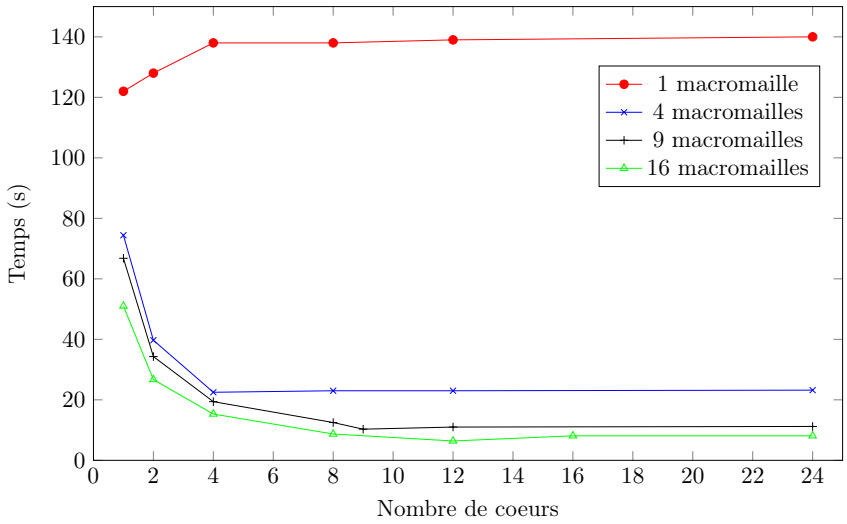

FIGURE 5.15 – Comparaison des temps d'exécution de la version StarPU de SCHNAPS pour les codelets C pour différents nombres de coeurs

A l'aide du module FxT, il est possible de générer la trace d'exécution du programme et ainsi de suivre l'exécution des différentes fonctions sur les coeurs de calcul. Sur la figure 5.16 page suivante nous représentons ces traces dans le cas de la simulation sur un macromaillage composé de 9 macromailles et pour 4, 9 et 12 CPU. les parties vertes correspondent à l'exécution d'une fonction alors que les parties rouge correspondent aux moments où le coeur est en attente. Nous avions vu sur la figure 5.15 que pour cette configuration, le meilleur temps était obtenu avec 9 CPU. Sur les traces nous pouvons voir que pour 9 coeurs, le parallélisme est important puisque les fonctions s'exécutent de manière simultanée et complète sur chaque coeur. ans le cas où nous utilisons seulement 4

coeurs, la distribution des tâches entre les coeurs n'est pas optimal puisque les processeurs exécutent de très nombreuses tâches courtes.

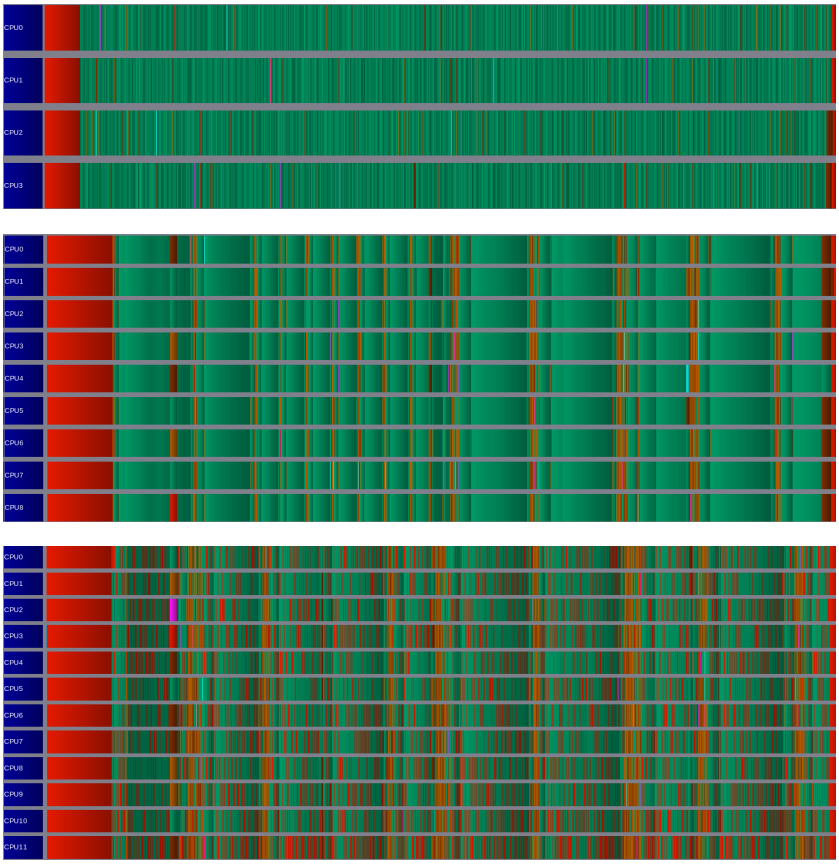

FIGURE 5.16 – Trace de l'exécution du programme pour un macromaillage 3×3 sur 4, 9 et 12 CPU.

Enfin, si nous utilisons 12 coeurs, nous remarquons qu'il y a un nombre plus important de moments où les processeurs sont en attente ce qui engendre la perte de performance malgré l'augmentation du nombre d'unités de calcul. Sur la figure 5.17 page ci-contre, nous représentons l'activité de quatre des coeurs de calcul dans le cas où nous en utilisons 9 et 12. La partie verte (partie basse) correspond à la proportion de temps utilisé par le processeur pour effectuer la résolution. La partie rouge (partie haute) représente quant à elle la proportion de temps utilisé par StarPU pour la distribution des tâches. Cette figure nous montre donc que l'utilisation d'un nombre de coeurs supérieur au nombre de macromailles entraine une perte de temps causée par les fonctions de *scheduling* de StarPU.

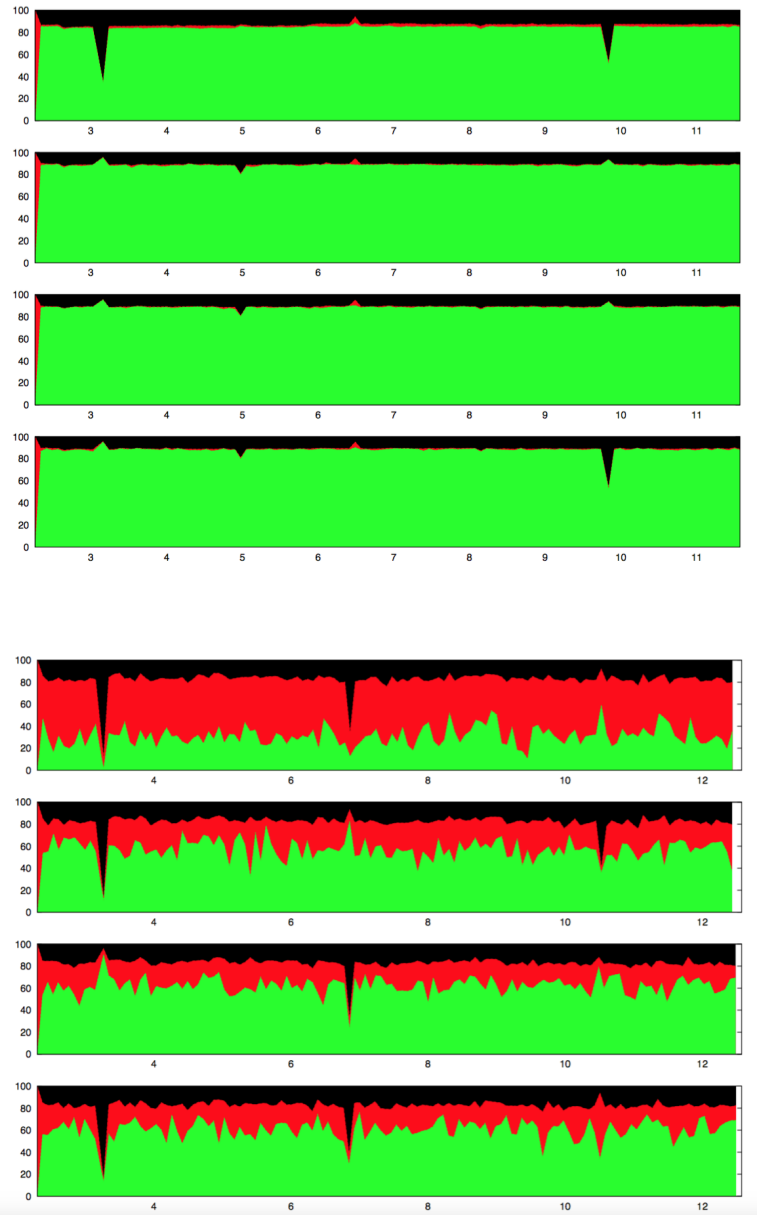

FIGURE 5.17 – Trace de l'exécution du programme pour un macromaillage 3×3 sur 4, 9 et 12 CPU.

Performances "multi-devices"

Comme nous l'avons dit précédemment, l'objectif d'une implémentation StarPU est l'utilisation simultanée de l'ensemble des plateformes de calcul disponibles. Nous terminons ce chapitre par une présentation des temps de calcul que nous avons obtenus en uti-

lisant plusieurs GPU combinés au CPU. Cette fois encore, nous avons utilisé le test 2D d'Orszag-Tang. Nous présentons dans le tableau 5.3 les temps de calcul mesurés en fonction de la taille de la grille et de l'architecture utilisée.

	1 GPU	2 GPU	4 GPU	4 GPU + 1 CPU
128^2	14.2	8.8	6.1	4.8
256^2	19.7	12.6	9.8	10.4
512^2	57	28.8	23.5	24.3
1024^2	208	96	60	49
2048^2	740	380	250	207

TABLE 5.3 – Comparaison des temps de calcul (en secondes) en fonction de la taille du maillage et de l'architecture utilisée.

Nous pouvons constater que l'utilisation de plusieurs accélérateurs en parallèle nous permet d'obtenir une réduction des temps de calcul. Toutefois, bien que les accélérations obtenues tendent à augmenter avec la taille du maillage, nous n'obtenons pas une bonne scalabilité au sens fort du terme. En revanche, en étudiant cette scalabilité au sens faible, c'est-à-dire, en comparant les temps de calcul obtenus en utilisant le même nombre de cellules dans chaque GPU, nous obtenons des résultats plus satisfaisants. Dans le tableau 5.4, nous présentons ces résultats.

	1 GPU	4 GPU
256^2	19.7	23.7
512^2	57	60
1024^2	208	250
2048^2	740	760

TABLE 5.4 – Comparaison des temps de calcul (en secondes) en fonction du nombre de cellules par GPU et du nombre de GPU utilisés.

Pour ce type d'évaluation, la scalabilité est assurée dans le cas où les temps de calcul sont constants en fonction du nombre de GPU utilisés. Dans notre cas, nous obtenons des ratios compris entre 0.83 et 0.97 en fonction du nombre de cellules du maillage. Nous pouvons donc en conclure que bien que la scalabilité au sens fort ne soit pas vérifiée, notre implémentation respecte ce critère au sens faible.

5.5 Conclusion

Ce dernier chapitre a été consacré à la présentation de la librairie SCHNAPS. Dans ce but, nous avons commencé par un rappel général de la méthode de Galerkin Discontinu.

Une partie importante du chapitre a ensuite été dédiée à la philosophie de SCHNAPS. Nous avons ainsi exposé les multiples possibilités d'optimisations proposées par la librairie. SCHNAPS travaillant sur un maillage à plusieurs niveaux, nous avons traité dans ce chapitre le cas de la méthode de Galerkin Discontinu sur des sous-cellules. Comme nous l'avions fait pour les chapitres précédents, nous terminons par une étude des temps d'exécution de l'algorithme sur différentes architectures. Nous comparons dans cette partie les implémentations C, OpenCL et StraPU. Sur le plan qualitatif, nous avons vu que dans le cas du problème de la MHD, l'absence de limiteur de pente implique l'apparition de phénomènes de Gibs au niveau des chocs. Sur le plan quantitatif, nous avons montré que les implémentations parallèles de la bibliothèque SCHNAPS nous fournissent de bonnes performances ainsi qu'une bonne scalabilité au sens faible.

Chapitre 6

Applications de la MHD

Sommaire

6.1 Instabilité de Kelvin-Helmoltz 121
6.2 Nappes de courant oscillantes 124

Afin d'appliquer les travaux effectués au cours de cette thèse, nous terminons ce manuscrit par un chapitre dédié à la présentation de solutions de notre modèle MHD pour différents cas tests issus de l'astrophysique.

6.1 Instabilité de Kelvin-Helmoltz

La première application consiste à étudier l'évolution d'un champ magnétique cisaillé instable. Nous effectuons le test dans le cas où le nombre d'Alfvèn est égal à 3.33. De cette manière, à l'état stationnaire, les flux magnétohydrodynamiques se détendent pour créer un écoulement laminaire stable. Dans notre cas, le faible champ magnétique entraine l'apparition de petits vortex prédits par les lois de la dynamique des gaz. Cependant, les contraintes magnétiques deviennent alors localement fortes ce qui provoque la destruction de ces vortex. Le champ magnétique provoque alors une reconnexion et un alignement dynamique entre les champs magnétiques et les vitesses [5].

Les conditions initiales du système sont données dans le tableau 6.1 page suivante. Le domaine est une grille carrée $[-1,1]\times[-1,1]$ et les conditions aux bords sont périodiques dans la direction x et libres dans la direction y.

Sur les figures 6.2 à 6.5 pages 122–124 nous représentons le champ de densité et le champ magnétique dans la direction x dans le cas d'une instabilité de Kelvin-Helmoltz. Nous

Variable	État initial
γ	$5/3$
ρ	1
u_x	$\frac{1.29}{2}\tanh(\frac{y}{0.05})$
u_y	$0.01\sin(2\pi x)\exp^{-(\frac{y}{0.2})^2}$
u_z	0
B_x	$\frac{1.29}{3.33}$
B_y	0
B_z	0
p	1

FIGURE 6.1 – Conditions initiales pour le cas test des instabilités de Kelvin-Helmoltz

présentons les solutions aux temps 1.5 et 2. La simulation a été faite sur un maillage 4000×4000 à l'ordre 2 et avec le flux numérique $P2$

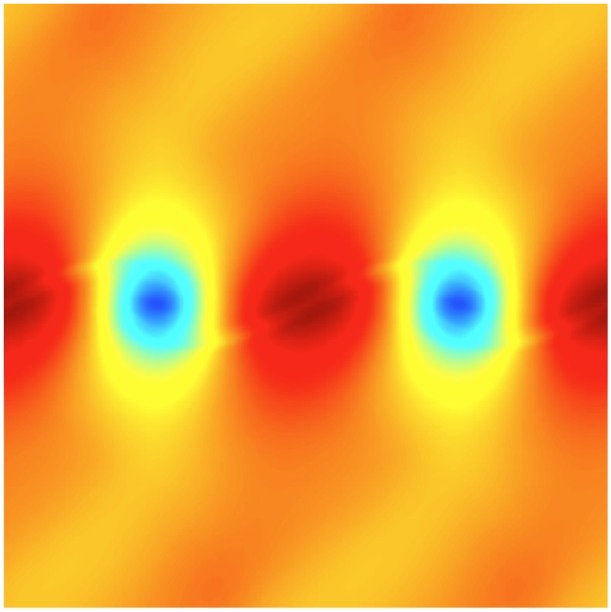

FIGURE 6.2 – Densité pour le cas test des instabilités de Kelvin-Helmoltz au temps 1.5. Simulation effectuée à l'ordre 2 avec le flux $P2$ sur un maillage de taille 4000×4000

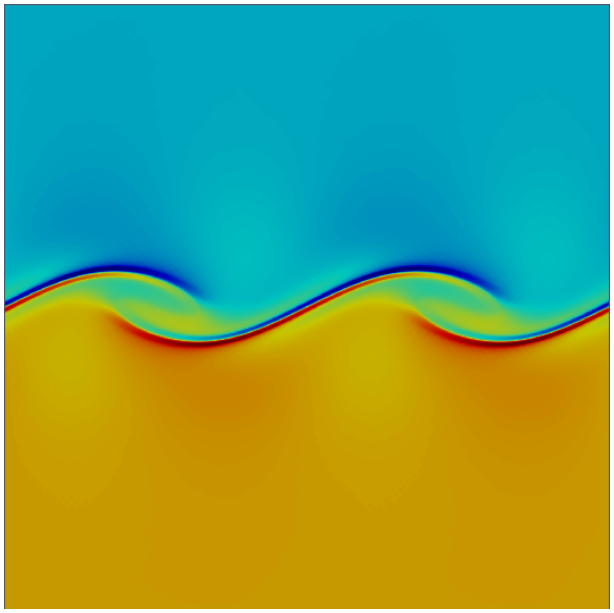

FIGURE 6.3 – Champ magnétique dans la direction x pour le cas test des instabilités de Kelvin-Helmoltz au temps 1.5. Simulation effectuée à l'ordre 2 avec le flux $P2$ sur un maillage de taille 4000×4000

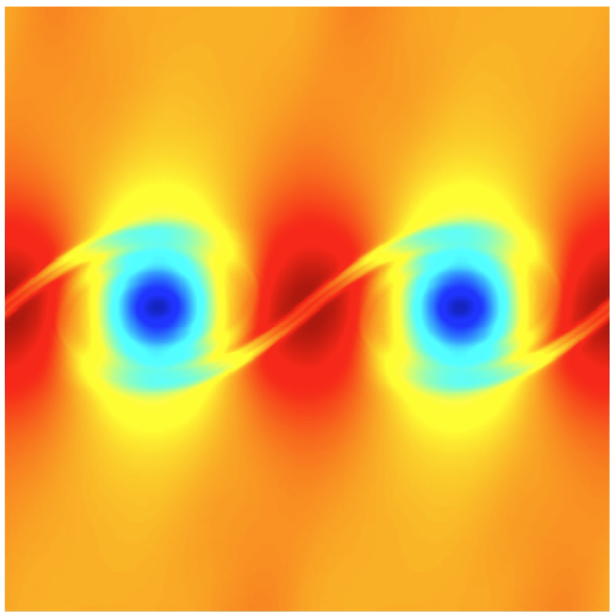

FIGURE 6.4 – Densité pour le cas test des instabilités de Kelvin-Helmoltz au temps 2. Simulation effectuée à l'ordre 2 avec le flux $P2$ sur un maillage de taille 4000×4000

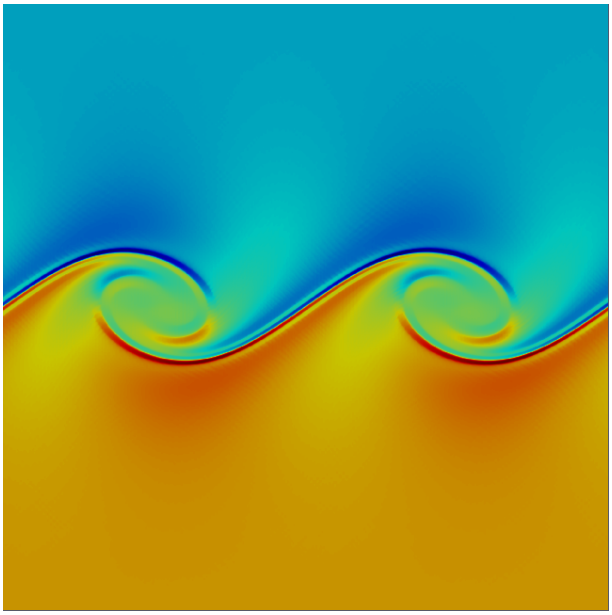

FIGURE 6.5 – Densité pour le cas test des instabilités de Kelvin-Helmoltz au temps 2. Simulation effectuée à l'ordre 2 avec le flux $P2$ sur un maillage de taille 4000×4000

6.2 Nappes de courant oscillantes

Ce test bidimensionnel consiste à étudier l'évolution de deux nappes de courant créées par un champ magnétique discontinu [24, 25]. Comme le modèle ne contient pas de dissipation, la simulation est dominée par la résistivité numérique du schéma. Une reconnexion magnétique a lieu au niveau des discontinuités du champ ce qui provoque l'apparition d'ilots le long de cette discontinuité. Ces ilots se déplacent le long des nappes de courant magnétique et viennent fusionner deux à deux afin de former un plus gros ilot. En allant plus loin dans la simulation, nous pouvons constater que de nouveaux petits ilots sont alors créés et se déplacent de la même manière le long des discontinuités pour aller à leur tout nourrir les plus gros ilots.

Nous travaillons sur un domaine carré $[0, 2] \times [0, 2]$ avec des conditions doublement périodiques. Les données initiales du problème sont données dans le tableau 6.6 page ci-contre.

Sur les figures 6.7 à 6.10 pages 125–127 nous donnons la solution que nous obtenons sur un maillage 2500×2500 à l'ordre 2 avec le flux numérique $P2$. Le champ représenté est le champ magnétique dans la direction y. Nous présentons les solutions aux temps 0.5, 1.5, 2 et 2.5

Variable	État initial
γ	$5/3$
ρ	1
u_x	$0.1 * \sin(\pi y)$
u_y	0
u_z	0
B_x	1 si $\frac{1}{3} < x \frac{3}{2}$
	-1 si $0 < x < \frac{1}{3}$ ou $\frac{3}{2} < x < 2$
B_y	0
B_z	0
p	0.1

FIGURE 6.6 – Conditions initiales pour le cas test des nappes de courants oscillantes

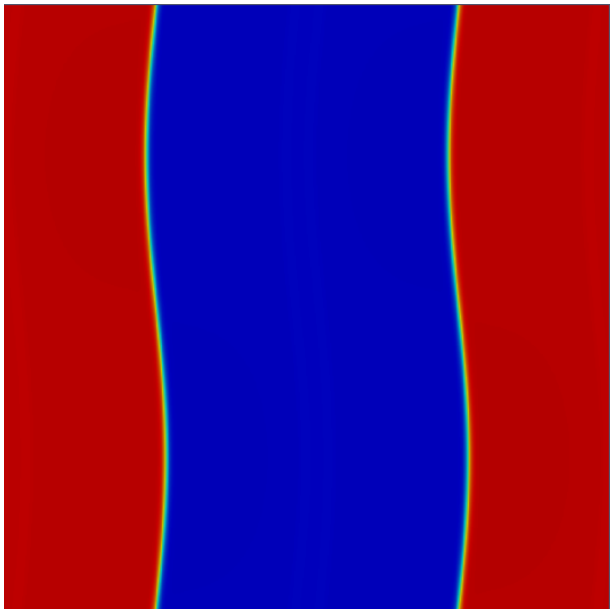

FIGURE 6.7 – Champ magnétique dans la direction y pour le cas test des nappes de courant oscillantes au temps 0.5. Simulation effectuée à l'ordre 2 avec le flux $P2$ sur un maillage de taille 2500×2500

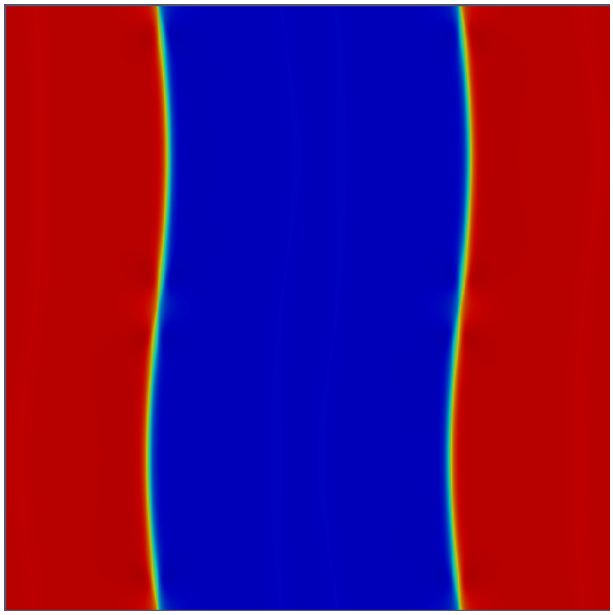

FIGURE 6.8 – Champ magnétique dans la direction y pour le cas test des nappes de courant oscillantes au temps 1.5. Simulation effectuée à l'ordre 2 avec le flux $P2$ sur un maillage de taille 2500×2500

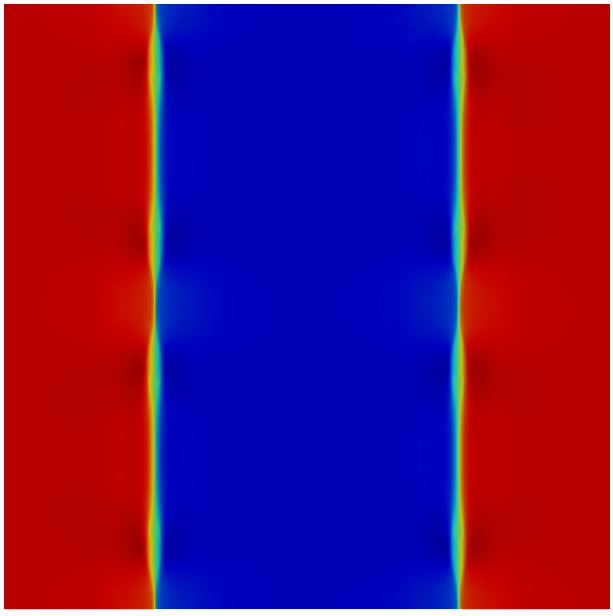

FIGURE 6.9 – Champ magnétique dans la direction y pour le cas test des nappes de courant oscillantes au temps 2. Simulation effectuée à l'ordre 2 avec le flux $P2$ sur un maillage de taille 2500×2500

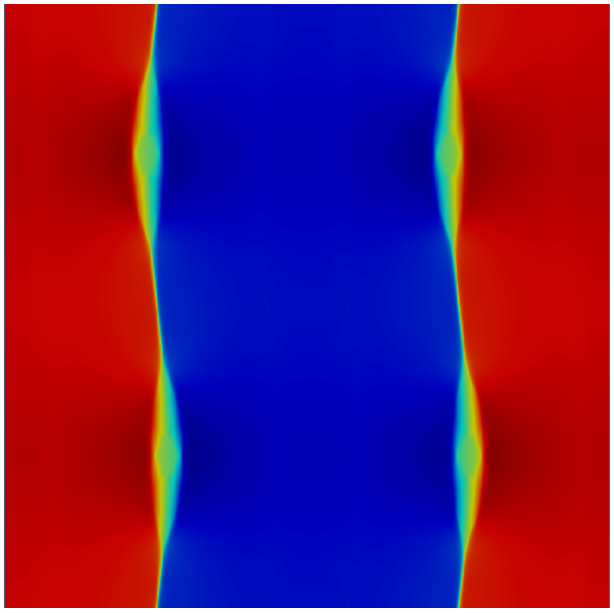

FIGURE 6.10 – Champ magnétique dans la direction y pour le cas test des nappes de courant oscillantes au temps 2.5. Simulation effectuée à l'ordre 2 avec le flux $P2$ sur un maillage de taille 2500×2500

Afin de mieux identifier l'apparition des ilots, nous représentons sur les figures 6.11 à 6.13 pages 128–129 les solutions au temps 1.5, 2 et 2.5 pour le champ de pression.

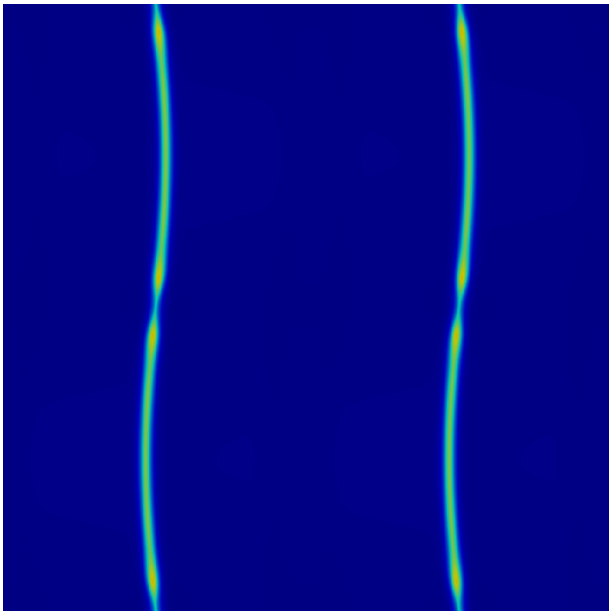

FIGURE 6.11 – Pression pour le cas test des nappes de courant oscillantes au temps 1.5. Simulation effectuée à l'ordre 2 avec le flux $P2$ sur un maillage de taille 2500×2500

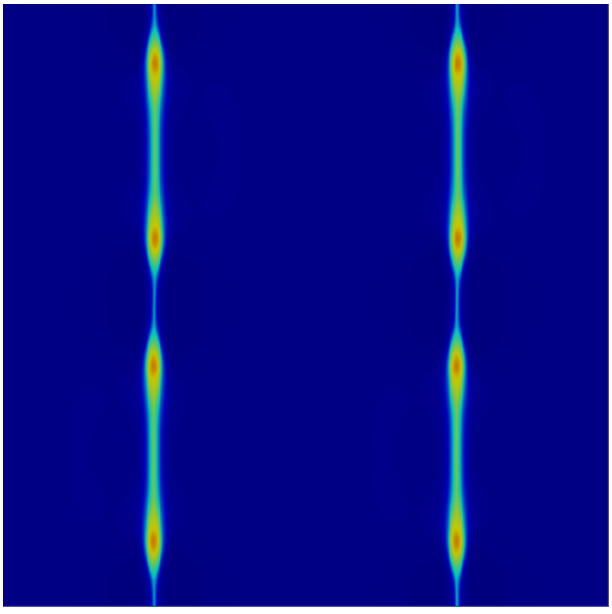

FIGURE 6.12 – Pression pour le cas test des nappes de courant oscillantes au temps 2. Simulation effectuée à l'ordre 2 avec le flux $P2$ sur un maillage de taille 2500×2500

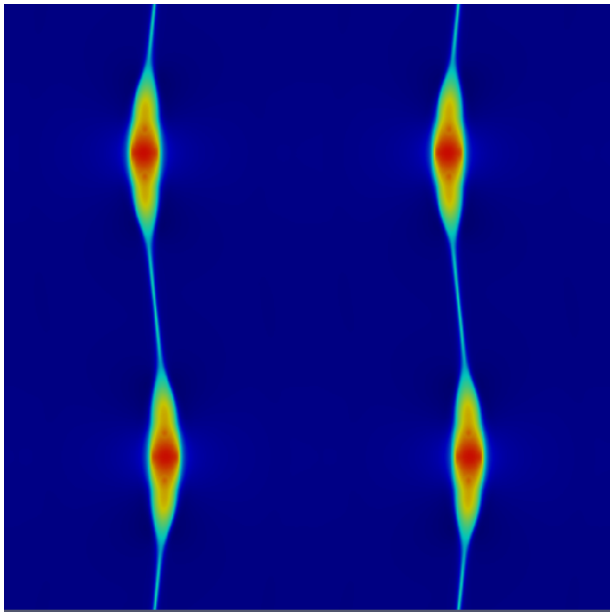

FIGURE 6.13 – Pression pour le cas test des nappes de courant oscillantes au temps 2.5. Simulation effectuée à l'ordre 2 avec le flux $P2$ sur un maillage de taille 2500×2500

Conclusion

Dans cette thèse, nous avons développé un code de volumes finis pour la résolution du problème de la Magnéto-Hydro-Dynamique. La description du problème à l'aide d'un couplage des équations de l'hydrodynamique et de l'électromagnétique permet de mettre en évidence la condition de divergence nulle du champ magnétique. Nous avons vu que, d'un point de vue théorique, cette condition est respectée si les conditions initiales sont telles que $\nabla \cdot \mathbf{B} = 0$. En revanche d'un point de vue numérique, de légères variations peuvent intervenir. Ces valeurs sont alors conservées et la divergence non nulle va se propager et potentiellement entrainer une interruption de la simulation. Nous avons alors étudié les méthodes de correction de la divergence proposées par Powell et Dedner proposant respectivement d'ajouter un terme magnétique dans les équations et une nouvelle équation dans le problème. Dans les deux cas, nous avons montré que ces deux corrections préservent l'hyperbolicité des modèles, mais seule la méthode de Dedner préserve la conservation.

Suite à cela, nous avons, après en avoir étudié la convergence, implémenté un schéma de volumes finis pour la résolution du problème de la MHD en dimension 1. Le passage à l'ordre 2 s'est fait en utilisant une méthode MUSCL de Van Leer qui propose d'approcher la dérivée spatiale à l'aide d'un limiteur de pente *minmod* afin d'éviter l'apparition d'oscillations non physique au niveau des chocs. Le choix du flux numérique étant un point essentiel de la méthode de volumes finis, la deuxième partie du chapitre 2 a été consacrée à la comparaison de différents flux décentrés *upwind* caractérisés par la précision d'approximation de la valeur absolue de la matrice jacobienne du système. Nous avons pu constater qu'a l'ordre 1, la précision dans l'approximation de cette valeur absolue joue un rôle important dans la précision de la solution obtenue. En revanche, pour un schéma d'ordre 2 en espace, le coût supplémentaire en temps de calcul n'est plus justifié. Nous avons conclu de ces comparaisons que le flux numérique $P2$ fournit le meilleur rapport précision/coût. Enfin, nous avons également constaté que l'utilisation de limiteur de pente tel que le *minmod* amélioré, proposé par LeVeque, entraine des oscillations de la solution avec un schéma en temps d'ordre 1.

Le problème de la MHD comporte un nombre important de variables et peut fournir des solutions très turbulentes nous imposant de travailler sur des maillages relativement fins.

De ce fait, le passage en dimension 2 est très coûteux en temps de calcul c'est pourquoi nous avons étudié dans le chapitre 3 des méthodes d'optimisation. Dans un premier temps, nous nous sommes concentrés sur des méthodes d'optimisation automatique. Nous avons pu constater que la simple utilisation de certaines options lors de la compilation permet d'obtenir des accélérations d'un facteur 5. Toujours concernant l'optimisation automatique, l'utilisation de la librairie OpenMP pour la parallélisation est elle aussi une ressource importante pour l'accélération d'un code. Sur notre exemple, l'utilisation de 16 coeurs de calcul nous a permis d'atteindre une accélération de 14. Dans le même chapitre, nous avons également mis en évidence l'augmentation de la performance de l'algorithme en utilisant un parcours efficace des tableaux. En effet, un parcours traditionnel en ligne ou en colonne ne permet pas de bénéficier des effets de cache du processeur c'est pourquoi nous avons présenté dans cette thèse un parcours par tuile réduisant au maximum les *cache-miss* du processeur.

Afin d'utiliser des moyens techniques plus récents et d'obtenir un gain de temps plus important, nous avons ensuite présenté une implémentation massivement parallèle utilisant l'architecture des processeurs graphiques et le langage OpenCL. À l'aide d'un splitting directionnel de Strang et d'une transposition optimisée pour ces accélérateurs, nous nous sommes assuré une coalescence maximale des données. L'analyse du modèle *roofline* nous a permis de constater que notre modèle est limité par la performance crête de la carte graphique et non par la bande passante, ce qui nous assure un bon ratio quantité de calcul/données transférées. Les résultats de nos mesures de temps nous ont permis de constater que l'utilisation de notre implémentation OpenCL fournit de bonnes performances mêmes sur CPU puisque nous obtenons des accélérations respectives de 1.6 et 5 sur des processeurs 2 et 6 coeurs. Cet algorithme étant optimisé pour une utilisation sur GPU, nous obtenons les meilleures performances sur ces architectures où nous avons des accélérations de l'ordre de 300 sur des cartes Nvidia K20 et AMD 7950. La limitation de mémoire des cartes pouvant être un problème, nous avons également implémenté à l'aide de la librairie MPI une version multi GPU permettant de distribuer le volume de calcul sur plusieurs GPU.

Dans la dernière partie de cette thèse, nous présentons la librairie SCHNAPS en développement au sein de l'IRMA de Strasbourg. Ce solveur permet d'utiliser au choix une implémentation sur CPU, une implémentation sur GPU à l'aide de la programmation OpenCL ou une implémentation hybride à l'aide du logiciel StarPU. D'une manière générale, nous avons montré que la librairie fournit de bonnes performances concernant les temps de calcul. Nous avons également montré que la version OpenCL de SCHNAPS est entièrement scalable puisque l'accélération est linéaire en fonction du nombre de coeurs de calcul ainsi que de la taille du problème. De bonnes performances ont également été mises en évidence avec la version StarPU où nous avons également montré l'efficacité de l'utilisation d'un maillage à plusieurs niveaux de finesse.

Chapitre A

Exemple complet d'un code OpenCL

Dans cette annexe nous donnons l'intégralité du code OpenCL permettant d'effectuer la fonction carrée à l'intégralité d'un tableau.

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <OpenCL/opencl.h>

#define INP_SIZE (1024)

// Simple compute kernel which computes the cube of an input
// array

const char *KernelSource = "\n"\
"__kernel␣void␣square(__global␣float*␣input,\n"\
"␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣__global␣float*␣output,\n"\
"␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣const␣unsigned␣int␣count){\n"\
"int␣i␣=␣get_global_id(0);\n"\
"if(i<count)\n"\
"output[i]=input[i]*input[i]*input[i];\n"\
"}\n";
```

```
int main(int argc, char** argv) {

  int err;                    // error code
  float data[INP_SIZE];       // original input data set
  float results[INP_SIZE]; // results returned from device
  unsigned int correct;       // no. of correct results returned

  size_t global;              // global domain size
  size_t local;               // local domain size

  cl_device_id device_id;  // compute device id
  cl_context context;      // compute context
  cl_command_queue commands; // compute command queue
  cl_program program;      // compute program
  cl_kernel kernel;        // compute kernel
  cl_mem input;            // memory used for the input array
  cl_mem output;           // memory used for the output array

  // Fill our data set with random values
  int i = 0;
  unsigned int count = INP_SIZE;

  for(i = 0; i < count; i++)
    data[i] = rand() / 50.00;


  // Connect to a compute device
  // If want to run your kernel on CPU then
  // replace the parameter CL_DEVICE_TYPE_GPU
  // with CL_DEVICE_TYPE_CPU

  err = clGetDeviceIDs(NULL,
                       CL_DEVICE_TYPE_GPU,
                       1,
                       &device_id,
                       NULL);

  if(err != CL_SUCCESS) {
    printf("Error: Failed to create a device group!\n");
    return EXIT_FAILURE;
  }
```

```
// Create a compute context
// Contexts are responsible for managing objects such
// as command-queues, memory, program and kernel objects
// and for executing kernels on one or more devices
// specified in the context.

context = clCreateContext(0,
                          1,
                          &device_id,
                          NULL,
                          NULL,
                          &err);

if(!context) {
  printf("Error: Failed to create a compute context!\n");
  return EXIT_FAILURE;
}

// Create a command commands
commands = clCreateCommandQueue(context,
                                device_id,
                                0,
                                &err);
if(!commands) {
  printf("Error: Failed to create a command commands!\n");
  return EXIT_FAILURE;
}

// Create the compute program from the source buffer
program = clCreateProgramWithSource(
                                context,
                                1,
                                (const char**) &KernelSource,
                                NULL,
                                &err);
if(!program) {
  printf("Error: Failed to create compute program!\n");
  return EXIT_FAILURE;
}

// Build the program executable
err = clBuildProgram(program,
                     0,
```

```
                        NULL,
                        NULL,
                        NULL,
                        NULL);
  if(err != CL_SUCCESS) {
    size_t len;
    char buffer[2048];
    printf("Error: Failed to build program executable!\n");
    clGetProgramBuildInfo(program,
                          device_id,
                          CL_PROGRAM_BUILD_LOG,
                          sizeof(buffer),
                          buffer,
                          &len);
    printf("%s\n", buffer);
    exit(1);
  }

  // Create the compute kernel in the program we wish to run
  kernel = clCreateKernel(program,
                          "square",
                          &err);
  if(!kernel || err != CL_SUCCESS) {
     printf("Error: Failed to create compute kernel!\n");
     exit(1);
  }

  // Create the input and output arrays in device memory
  // for our calculation
  input = clCreateBuffer(context,
                         CL_MEM_READ_ONLY,
                         sizeof(float) *count,
                         NULL,
                         NULL);
  output = clCreateBuffer(context,
                          CL_MEM_WRITE_ONLY,
                          sizeof(float) *count,
                          NULL,
                          NULL);

  if(!input || !output) {
     printf("Error: Failed to allocate device memory!\n");
     exit(1);
```

```
}

// Write our data set into the input array in device memory
err = clEnqueueWriteBuffer(commands,
                           input,
                           CL_TRUE,
                           0,
                           sizeof(float) *count,
                           data,
                           0,
                           NULL,
                           NULL);
if(err != CL_SUCCESS) {
  printf("Error: Failed to write to source array!\n");
  exit(1);
}

// Set the arguments to our compute kernel
err = 0;
err = clSetKernelArg(kernel,
                     0,
                     sizeof(cl_mem),
                     &input);
err |= clSetKernelArg(kernel,
                      1,
                      sizeof(cl_mem),
                      &output);
err |= clSetKernelArg(kernel,
                      2,
                      sizeof(unsigned int),
                      &count);

if(err != CL_SUCCESS) {
  printf("Error: Failed to set kernel arguments! %d\n",err);
  exit(1);
}

// Get the maximum work group size for executing the kernel
// on the device
err = clGetKernelWorkGroupInfo(kernel,
                               device_id,
                               CL_KERNEL_WORK_GROUP_SIZE,
                               sizeof(local),
```

```
                                    &local,
                                    NULL);
  if(err != CL_SUCCESS) {
    printf("Error: Failed to retrieve kernel work
␣␣␣␣␣␣␣␣␣␣␣␣␣␣group info! %d\n", err);
    exit(1);
  }

  // Execute the kernel over the entire range of our 1d
  // input data set using the maximum number of work
  // group items for this device
  global = count;
  err = clEnqueueNDRangeKernel(commands,
                               kernel,
                               1,
                               NULL,
                               &global,
                               &local,
                               0,
                               NULL,
                               NULL);
  if (err) {
    printf("Error: Failed to execute kernel!\n");
    return EXIT_FAILURE;
  }

  // Wait for the command commands to get serviced before
  // reading back results
  clFinish(commands);

  // Read back the results from the device to check the output
  err = clEnqueueReadBuffer(commands,
                            output,
                            CL_TRUE,
                            0,
                            sizeof(float) *count,
                            results,
                            0,
                            NULL,
                            NULL );
  if(err != CL_SUCCESS) {
    printf("Error: Failed to read output array! %d\n", err);
    exit(1);
```

```
  }

   // Print obtained results from OpenCL kernel
  for(i=0; i<count); i++) {
    printf("result[%d] = %f", i, result[i]);
  }

  // Cleaning up
  clReleaseMemObject(input);
  clReleaseMemObject(output);
  clReleaseProgram(program);
  clReleaseKernel(kernel);
  clReleaseCommandQueue(commands);
  clReleaseContext(context);

 return 0;
}
```

Bibliographie

[1] H. Alfvén. On the cosmogony of the solar system iii. *Stockholms Observatoriums Annaler*, 14 :9, 1946.

[2] C. Altmann, T. Belat, M. Gutnic, P. Helluy, H. Mathis, E. Sonnendrucker, W. Angulo, and J.-M. Herrard. A local time-stepping discontinuous galerkin algorithm for the mhd system. cemracs 2008–modeling and numerical simulation of complex fluids. *ESAIM Proc.*, 28 :33–54, 2009.

[3] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. Starpu : a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation : Practice and Experience*, 23(2) :187–198, 2011.

[4] Maxime B. Introduction to the theory of fourier's series. *Annals of Mathematics*, 7(3) :81–152, 1906.

[5] H. Baty, R. Keppens, and P. Comte. The two-dimensional magnetohydrodynamic kelvin–helmholtz instability : Compressibility and large-scale coalescence effects. *Physics of Plasmas (1994-present)*, 10(12) :4661–4674, 2003.

[6] J. Bolz, I. Farmer, E. Grinspun, and P. Schröoder. Sparse matrix solvers on the gpu : Conjugate gradients and multigrid. *ACM Trans. Graph.*, 22(3) :917–924, July 2003.

[7] F. Bouchut, C. Klingenberg, and C. Waagen. A multiwave approximate riemann solver for ideal mhd based on relaxation ii. *Numer. Math*, 115(4) :647–679, 2010.

[8] M. Brio and C.-C. Wu. An upwind differencing scheme for the equations of ideal magnetohydrodynamics. *Journal of computational physics*, 75(2) :400–422, 1988.

[9] P. Cargo and G. Gallice. Roe matrices for ideal mhd and systematic construction of roe matrices for systems of conservation laws. *Journal of Computational Physics*, 136(2) :446–466, 1997.

[10] A. J. Chorin and J. E. Marsden. *A Mathematical Introduction to Fluid Mechanics*. Springer-Verlag, 1993.

[11] G. Cohen, X. Ferrieres, and S. Pernet. A spatial high-order hexahedral discontinuous galerkin method to solve maxwell's equations in time domain. *Journal of Computational Physics*, 217(2) :340–363, 2006.

[12] M. Cosnard and E. Jeannot. Compact dag representation and its dynamic scheduling. *Journal of Parallel and Distributed Computing*, 58(3) :487–514, 1999.

[13] M. Cosnard and M. Loi. Automatic task graph generation techniques. In *System Sciences, 1995. Proceedings of the Twenty-Eighth Hawaii International Conference on*, volume 2, pages 113–122. IEEE, 1995.

[14] A. Crestetto. *Optimisation de méthodes numériques pour la physique des plasmas : application aux faisceaux de particules chargées.* PhD thesis, Université de Strasbourg, 2012.

[15] A. Crestetto and P. Helluy. An opencl tutorial. *http://www-irma.u-strasbg.fr/~helluy/OPENCL/tut-opencl.html*, 2010.

[16] W. Dai and P.-R. Woodward. An approximate riemann solver for ideal magnetohydrodynamics. *J. Comput. Phys.*, 111(2) :354–372, April 1994.

[17] F. De Vuyst and F. Salvarani. Gpu-accelerated numerical simulations of the knudsen gas on time-dependent domains. *Computer Physics Communications*, 184(3) :532–536, 2013.

[18] A. Dedner, F. Kemm, D. Kröner, C.-D. Munz, T. Schnitzer, and M. Wesenberg. Hyperbolic divergence cleaning for the mhd equations. *Journal of Computational Physics*, 175(2) :645–673, 2002.

[19] P. Degond(F-TOUL3-IP), P.-F. Peyrard(F-ONCE), G. Russo(I-LAQL), and Villedieu(F-ONCE) P. Polynomial upwind schemes for hyperbolic systems. *C. R. Acad. Sci Paris Ser. I Math*, 328(6) :479–483, 1999.

[20] B. Einfeldt, C.-D. Munz, P.-L. Roe, and B. Sjögreen. On godunov-type methods near low densities. *Journal of computational physics*, 92(2) :273–295, 1991.

[21] C. R. Evans and J. F Hawley. Simulation of magnetohydrodynamic flows-a constrained transport method. *The Astrophysical Journal*, 332 :659–677, 1988.

[22] L. C. Evans. *Entropy and partial differential equations.* American Math Society, 2004.

[23] R. Eymard, T. Gallouët, and R. Herbin. Finite volume methods. *Handbook of numerical analysis*, 7 :713–1018, 2000.

[24] S. Fromang, P. Hennebelle, and R. Teyssier. A high order godunov scheme with constrained transport and adaptive mesh refinement for astrophysical magnetohydrodynamics. *Astronomy & Astrophysics*, 457(2) :371–384, 2006.

[25] T. A. Gardiner and J. M. Stone. An unsplit godunov method for ideal {MHD} via constrained transport. *Journal of Computational Physics*, 205(2) :509 – 539, 2005.

[26] E. Godlewski and P.-A. Raviart. *Numerical approximation of hyperbolic systems of conservation laws.* Applied Mathematical Sciences 118. Springer-Verlag, New York, 1996.

[27] Khronos Group. Opencl online documentation. http://www.khronos.org/opencl/.

[28] A. Harten, P. Lax, and B. Van Leer. On upsteam differencing and godunov-type schemes for hyperbolic conservation laws. *SIAM Rev.*, 25(1) :35–61, 1983.

[29] A. Harten, P. D. Lax, and P. Van Leer. On upstream differencing and godunov-type schemes for hyperbolic conservation laws. *SIAM Rev.*, 25(1) :35–61, 1983.

[30] P. Helluy. *Numerical resolution of the harmonic Maxwell equations by a discontinuous Galerkin finite element method.* PhD thesis, Ecole nationale superieure de l'aeronautique et de l'espace, 1994.

[31] P. Helluy, J.-M. Herrard, H. Mathis, and S. Müller. A simple parameter-free entropy correction for approximate riemann solvers. *Compte rendu Mécanique*, 2010.

[32] P. Helluy and J. Jung. Opencl numerical simulation of two-fluid compressible flows with a 2d random choice method. *International Journal on Finite Volume*, 10 :1–38, 2013.

[33] P. Helluy, M. Massaro, L. Navoret, N. Pham, and T. Strub. Reduced vlasov-maxwell modeling. pages 2628–2632, 2014.

[34] P. Helluy, T. Strub, M. Massaro, and M. Roberts. Asynchronous opencl/mpi numerical simulations of conservation laws. In *Proceedings of the 3rd International Workshop on OpenCL*, page 4. ACM, 2015.

[35] N.-J. Higham. *Functions of matrices : theory and computation.* Siam, 2008.

[36] INRIA. Starpu handbook. http://starpu.gforge.inria.fr/doc/html/index.html.

[37] D. H. Jesse, A. C. Nathan, and C. H. John. Cache and Bandwidth Aware Matrix Multiplication on the GPU. 2003.

[38] S. Jund. *Méthodes d'éléments finis d'ordre élevé pour la simulation numérique de la propagation d'ondes.* PhD thesis, Université Louis Pasteur, Strasbourg, 2007.

[39] J. Jung. *Méthodes numériques d´ écoulements multiphasiques bidimensionnels sur GPU.* PhD thesis, University of Strasbourg, 2013.

[40] V. V. Kindratenko, J. Enos, M. T. Showerman, G. W. Arnold, J. E. Stone, J. C. Phillips, and W.-M. Hwu. Gpu clusters for high- performance computing. *Proc. Workshop on Parallel Programming on Accelerator Clusters, IEEE Cluster 2009*, pages 1–8, 2009.

[41] A. Klöckner, T. Warburton, J. Bridge, and J.-S. Hesthaven. Nodal discontinuous galerkin methods on graphics processors. *Journal of Computational Physics*, 228(21) :7863–7882, 2009.

[42] D. Lanser and J.-G. Verwer. Analysis of operator splitting for advection–diffusion–reaction problems from air pollution modelling. *Journal of computational and applied mathematics*, 111(1) :201–216, 1999.

[43] E. S. Larsen and D. McAllister. Fast matrix multiplies using graphics hardware. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, SC '01, pages 55–55, New York, NY, USA, 2001. ACM.

[44] R.-J. LeVeque. *Numerical methods for conservation laws.* Lectures in Mathematics ETH Zürich. Birkhäuser Verlag, Basel, second edition, 1992.

[45] R. J LeVeque. *Finite volume methods for hyperbolic problems*, volume 31. Cambridge university press, 2002.

[46] V. Loechner, B. Meister, and P. Clauss. Precise data locality optimization of nested loops. *The Journal of Supercomputing*, 21(1) :37–76, January 2002.

[47] M. Massaro, P. Helluy, and V. Loechner. Numerical simulation for the mhd system in 2d using opencl. *ESAIM : PROCEEDINGS AND SURVEYS*, 45 :485–492, 2014.

[48] H. Mathis. *Étude théorique et numérique des écoulements avec transition de phase.* PhD thesis, Université de Strasbourg, 2010.

[49] D. Montgomery. Non-linear wave propagation. with applications to physics and magnetohydrodynamics. a. jeffrey and t. taniuti. academic press, new york, 1964. x + 369 pp. illus. 12. *Science*, 146(3643) :512–513, 1964.

[50] S. A Orszag and C.-M. Tang. Small-scale structure of two-dimensional magnetohydrodynamic turbulence. *Journal of Fluid Mechanics*, 90(01) :129–143, 1979.

[51] C. Palenzuela, L. Lehner, O. Reula, and L. Rezzolla. Beyond ideal mhd : towards a more realistic modelling of relativistic astrophysical plasmas. *Monthly Notices of the Royal Astronomical Society*, 394(4) :1727–1740, 2009.

[52] L. Pebernet, X. Ferrieres, S. Pernet, B.-L. Michielsen, F. Rogier, and P. Degond. Discontinuous galerkin method applied to electromagnetic compatibility problems : introduction of thin wire and thin resistive material models. *IET Science, Measurement & Technology*, 2(6) :395–401, 2008.

[53] P.F Peyrard and P. Villedieu. A roe scheme for ideal {MHD} equations on 2d adaptively refined triangular grids. *Journal of Computational Physics*, 150(2) :373 – 393, 1999.

[54] K. G. Powell. An approximate riemann solver for magnetohydrodynamics. 1994.

[55] P. L. Roe. Some contributions to the modelling of discontinuous flows. In *Large-scale computations in fluid mechanics*, volume 1, pages 163–193, 1985.

[56] G. Ruetsch and P. Micikevicius. Optimizing matrix transpose in cuda. *NVidia GPU Computing SDK*, pages 1–24, 2009.

[57] N. Seguin. *Modélisation et simulation numérique des écoulements diphasiques.* PhD thesis, PhD thesis, Université de Provence, 2015.

[58] F. Shakeri and M. Dehghan. A finite volume spectral element method for solving magnetohydrodynamic (mhd) equations. *Applied Numerical Mathematics*, 61(1) :1–23, 2011.

[59] M. Snir. *MPI–the Complete Reference : The MPI core*, volume 1. MIT press, 1998.

[60] G. Strang. On the construction and comparison of difference schemes. *SIAM J. Numer. Anal.*, 5 :506,517, 1968.

[61] T. Strub. *Résolution numérique des équations de Maxwell tridimensionnelles instationnaires sur architecture massivement multicoeur.* PhD thesis, PhD thesis, University of Strasbourg, 2015.

[62] T. Strub. *Résolution des équations de Maxwell tridimensionnelles instationnaires sur architecture massivement multicœur*. PhD thesis, Université de Strasbourg, 2015.

[63] M. Torrilhon. Uniqueness conditions for riemann problems of ideal magnetohydrodynamics. *Journal of plasma physics*, 69(03) :253–276, 2003.

[64] B. Van Leer. Toward the ultimate conservative difference scheme. a second order sequel to the godunov's method. *Journal of Computational Physics*, 32 :101–136, 1979.

[65] Wikipedia. Pipeline (architecture des processeurs). https://fr.wikipedia.org/wiki/Pipeline_(architecture_des_processeurs).

[66] H.-C. Wong, U.-H. Wong, X. Feng, and Z. Tang. Efficient magnetohydrodynamic simulations on graphics processing units with cuda. *Comput. Phys. Comm.*, 182(10) :2132–2160, 2011.

Michel MASSARO

Méthodes numériques pour les plasmas sur architectures multicœurs

Résumé

Cette thèse traite de la résolution du système de la Magnéto-Hydro-Dynamique (MHD) sur architectures massivement parallèles. Ce système est un système hyperbolique de lois de conservation. Pour des raisons de coût en termes de temps et d'espace, nous utilisons la méthode des volumes finis. Ces critères sont particulièrement importants dans le cas de la MHD, car les solutions obtenues peuvent présenter de nombreuses ondes de choc et être très turbulentes. L'approche d'un phénomène physique nécessite par conséquent de travailler sur un maillage fin entrainant une grande quantité de calcul. Afin de réduire les temps d'exécution des algorithmes proposés, nous proposons des méthodes d'optimisations pour l'exécution sur CPU telles que l'utilisation d'OpenMP pour une parallélisation automatique ou le parcours optimisé afin de bénéficier des effets de cache. Une implémentation sur architecture GPU à l'aide de la librairie OpenCL est également proposée. Dans le but de conserver une coalescence maximale des données en mémoire, nous proposons une méthode utilisant un splitting directionnel associé à une méthode de transposition optimisée pour les implémentations parallèle. Dans la dernière partie, nous présentons la librairie SCHNAPS. Ce solveur utilisant la méthode Galerkin Disontinu (GD) utilise des implémentations OpenCL et StarPUafin de profiter au maximum des avantages de la programmation hybride.

Mots-clé : Magnéto-Hydro-Dynamique, Volumes finis, Galerkin Discontinu, Parallélisation, OpenCL, StarPU.

Abstract

This thesis deals with the resolution of the Magneto-Hydro-Dynamic (MHD) system on massively parallel architectures. This problem is an hyperbolic system of conservation laws. For cost reasons in terms of time and space, we use the finite volume method. These criteria are particularly important in the case of MHD because the solutions obtained may have many shock waves and be very turbulent. The approach of a physical phenomenon requires working on a fine mesh which involves a large quantity of computations. In order to reduce the execution time of the proposed algorithms, we present several optimization methods for CPU execution such as the use of OpenMP for an automatic parallelization or an optimized way to browse a grid in order to benefit from cache effects. An implementation on GPU architecture using the OpenCL library is also available. To maintain a maximal coalescence of the data in memory, we propose a method using a directional splitting associated with an optimized transposition method for parallel implementations. In the last part, we present the SCHNAPS library. This solver using the Galerkin Disontinu (GD) method uses OpenCL and StarPU implementations in order to maximize the benefits of hybrid programming.

Keywords : Magneto-Hydro-Dynamic, Finite volume, Discontinuous Galerkin, Parallelization, OpenCL, StarPU.